Formal Derivation of A Class of Computers

Li-Guo Wang

Doctor of Philosophy

University of Edinburgh
November 1994

Abstract

The aim of this thesis is to investigate how to use logic-based specification, construction, and proof methods to formally derive a class of computers.

Differing from the traditional concepts of specification, verification, and synthesis, the emphasis will be on the *formal design process* and the notion of derivation. A derivation is a stepwise specification, construction, and proof process. The *specification* of derivation is a set of specifications which is described by abstract syntax capturing the common structure of the specifications in the set. The *construction* of derivation is a mapping from the set of specifications to the power set of specifications. The *proof* of derivation is a guarantee that each construction is total and each constructed specification is correct.

Computers are among the most complex hardware devices. Formal verification researchers have often used computers as their goals or case studies. But there are few reports of the formal construction and proof of a computer. There is no report is about the formal derivation of a class of computers.

In this thesis we present the concepts and methods of derivation, which includes, mainly, abstract-syntax-based specification schemes for hierarchical specifications, formal and nearly mechanical construction, and abstract and general verification for a class of computers. The central concept is the *unified derivation model,* developed from the predicate model, used to describe the whole derivation process. The central method is *logic structural refinement* which, complements behavioral, data, temporal, and (spatial) structural abstraction. It concentrates on describing the main aspect of construction: logic decomposition. The key point is the concept of an *entry predicate* which reveals the basic relationship in logic and time between a specification and its sub-specifications.

We use state-transition-based abstract syntax as the behavior specification scheme to describe a class of computers at the machine instruction level. Starting from the scheme we take 9 steps of constructions and proofs to derive and obtain structural specifications at register transfer level. Each individual step, and thereby the whole construction, is proved to be total and correct.

Our work develops the basic view point that the hierarchical architecture of a computer can be organized through a dynamic formal construction and proof process, in a three dimensional structure of logic, space and time. The aspect of relation and construction in logic is fundamental and novel. In addition to the traditional understanding of microprograms our work brings up three new levels in microprogramming: global loop, abstract microprogram structure, and linear microprogram. Also, there is a sequence of constructions of time layers in which the usual verification based on two time layers is only a special case. As a corollary we discuss no-clock, variable clock, and constant clock machines.

Gordon's computer is taken as a case study to illustrate our concepts and methods. We derive three different implementations and compare them with the original one of Gordon's paper.

Acknowledgments

I would like to thank first of all my supervisor, Professor Mike Fourman, for his encouragement and hints in the development of this thesis, especially, I am grateful to him for his kind help for the chance to come Edinburgh University for my PhD study. I would also like to thank to my second supervisor, Professor Rod Burstall, for his kind help for my scholarship and for his careful checking of parts of this thesis.

I am grateful to Siemens AG Munich for providing scholarship and SERC for further financial support under grant GR/F 35890.

I would like most sincerely to thank Dr Michael Mendler for his sincere encouragement and for proof reading my whole thesis (first version, near 300 pages) word by word, pointing out some mistakes, and improving my English. This is a quite hard work and has taken a lot of his time. Without his kind help I could not have the present thesis.

I would like to thank to Dr Zhaohui Luo for his warmhearted encouragement and kind help during my whole PhD study in Edinburgh and to Dr Liang Chen for his sincere helps.

I own special thanks to Dr Imre Kayhan for his warmhearted help for me to use Macintosh, to Dr Kees G. W. Goossens for his help to check of parts of this thesis, to Miss Eleanor Kerse and Mrs Lorraine Edgar for their many kind helps.

I own sincere thanks to the National Science Foundation of China, Professor Edsger W. Dijkstra, Professor C.A.R. Hoare, Professor Jieh Hsiang, Professor Jifeng He for their helps and invitations which finally lead to my PhD study in LFCS, Department of Computer Science of Edinburgh University.

I own special thanks to Dr Mike Gordon for his providing HOL system, to Dr Ranga Vemuri and Dr Phillip J. Windley for their providing relevant reference materials.

Declaration

This thesis has been composed by myself. The work reported herein, except where indicated in the text, is my own, and has not been presented for any university degree before.

Li-Guo Wang

Table of Contents

Part I

Introduction and Concepts

In this part we introduce the concept of derivation, and describe its three aspects: structure, construction and proof. As a foundation for the formal derivation of a class of computers we present a derivation model for formal hardware construction.

Chapter 1

Introduction

In this chapter we introduce the concept of derivation. We discuss two aspects: motivation, and the contribution of this work.

1.1 Motivation

Formal hardware verification is becoming more and more accepted as a means of increasong confidence in the correctness of hardware design. Two aspects are notable: formal methods and experimental case studies, especially, the verification of microprocessors. What we obtain is, in essence, a clearer understanding of the relation between two extremes, specification and implementation, through applying rigorous mathematics.

The inherent complexity of hardware systems stimulates synthesis research, as post-design verification is difficult to deal with [26], [36]. However, synthesis is a somewhat ambiguous concept. In fact, there are several different understandings for synthesis, for example, synthesis as algorithmic optimization [60]; synthesis as a design methodology [27]; synthesis as a logic of design [36]; synthesis as case study [6], [73].

In this thesis instead of the word "synthesis" we use the word "derivation".

Definition 1.1.1. [derivation]:

Derivation is a formal methodology to deal with the specification, construction, and proof of a class of objects. □

More concretely, by *formal derivation of a class of computers,* we mean: using structure (abstract syntax) to describe a computer, using formal construction to describe the design of a computer, and using generic proof to guarantee the totalness and correctness of the construction. We emphasize the application of a formal method to solve a realistic, complex problem.

1.2 The Contribution of This Work

We present an overall specification, construction, and proof process to cover the formal design and verification of a class of computers, from behavioral specification at the machine instruction level, to structural implementation at the register transfer level. The general process is illustrated through a case study of Gordon's computer.

We present a new, five-level hierarchy for derivation of a microprogram through a dynamic, top-down process of construction and proof:

1. Global loop

2. Abstract microprogram structure

3. Linear microprogram

4. Microinstruction

5. Microcode

We present proof-based computer architecture as a new understanding for computer architecture.

Our main conceptual and methododological contribution is the derivation model. This establishes a foundation for the construction of hardware, and provides a concise and clear picture of the complicated process of computer design. The entire derivation of a class of computers is based on the model. Furthermore based on the model we present a new understanding for the logical, spatial, and temporal aspects of formal hardware construction.

1.3 Structure of the Thesis

The thesis consists of three parts: Part 1, introduction and concepts, Part 2, formal derivation of a class of computers and Part 3, case study and further discussion.

Part 1 consists of three chapters, of which the first is just this introduction. In Chapter 2 we describe the main concepts and methods of the thesis. They include structure, construction, and proof. In Chapter 3 we present the concepts of derivation model and entry predicate. These concepts are the basis of our derivation methods.

Part 2 is the main part of this thesis and studies the formal derivation process of a class of computers. It consists of the seven chapters 4 - 10, which are summarized below.

In Chapter 4 we define exactly the behavior specification scheme for describing a class of computers at machine instruction level.

In Chapter 5 we discuss time refinement. This includes two construction steps: making time computable, and changing time layers. These two construction steps are general, and in fact they are suitable for any level of time and time transformation.

In Chapter 6 we discuss the construction involving selecting the global structure at the behavioral specification level. These selections, especially ones for parallel and sequential execution, are important steps of the design. They will deeply influence the following design.

In Chapter 7 we have an important step of formal derivation: decomposition from the behavioral specification level to the term transition level. We establish, through exact construction and proof, a term transition structure as a general, high-level computer architecture which corresponds to an abstract microprogram structure.

In Chapter 8 we further decompose the term transition structure to function unit and connection level. We present the concept of linear chain and a general proof-based construction process which can act as a framework covering a wide range of allocation and scheduling strategies. An optimization algorithm can be represented within the framework through the selections used in the construction.

In Chapter 9 we discuss a generic process to introduce Bus, Register and Connection to get datapath and microprogram. Meanwhile we prove the zero-time entry predicate $E\ 0$ can be eliminated.

In Chapter 10 we discuss structural specification schemes.

Part 3 includes five chapters. In Chapter 11, we have a case study. the formal derivation of Gordon's computer from its machine instruction specification to three different structural implementations at the register transfer level. We discuss and compare the derived machine and [50]'s verified machine. The case study nicely illustrates our concepts and methods of formal derivation. In Chapter 12 we present a new concept of proof-based computer architecture and discuss its basic structure. In Chapter 13 we present and discuss the fundamental concepts: the logic, space and time of hardware construction. We demonstrate that, based on our construction model, we can set up a complete new theoretical system for formal construction of hardware. In Chapter 14 we describe related work about computer verification and formal synthesis. In Chapter 15 we give the conclusions of the research work of the Thesis and discuss some further research work.

Chapter 2

Structure, Construction and Proof

In this chapter we discuss the concepts of structure, construction and proof on which our formal derivation is based.

2.1 Structure

Structure is "a combination of parts" (Arthur L. Hayward and John J. Sparkes, The Concise English Dictionary, Fourth edition, twelfth impression 1990, page 1123).

In this thesis we use "structure" to mean a collection of objects. For the objects the emphasis is on the relations among their parts while any further detail that might be known of these parts is ignored. We take "structure" as a means to describe the common points of a class of objects and their essential aspects .

Jeff Joyce verified a concrete computer, Tamarack-3, and generalized his method [53] by using the concept of "uninterpreted data types and uninterpreted primitives" for a generic specification. He emphasizes that "generic specification plays a dominant role" in his dissertation. We deal with a more general case: a class of computers versus a concrete computer, and derivation (construction + verification) versus verification. Instead of a generic specification, we deal with a class of objects, and use structure to describe the class.

For the computer construction we call the structure the specification scheme, to emphasize that it, as a *scheme*, can be instantiated to get a set of specifications. We adopt the set-theoretic abstract syntax approach, BNF, [67] to describe specification schemes. Let us illustrate BNF through an example.

We define a term using BNF as below:

Specification Scheme:	**Abbreviation:**
$tm ::= \ c$	tm : term
$\mid v$	v : variable
$\mid f\ (tm, \cdots, tm)$	f : function
	c : constant

In BNF the symbol "::=" is read as "may be composed of". The symbol "|" is read as "or". In the example, under "specification scheme" we write the BNF rule that a term "tm" may be a constant "c", a variable "v" or a function "f" whose arguments recursively are terms "tm". Under "abbreviation" we write syntax domain, for example, "v : *variable*" that v is a variable. A symbol string is nonterminal if it has an appearance in the left of ::=, otherwise, terminal. For example, tm is a nonterminal and c, v and f are the terminals.

In this thesis we use a convenience that the terminals and nonterminals of a specification scheme can be inherited by another specification scheme. Therefore, it is possible that a terminal (a nonterminal) in a specification scheme may become nonterminal (terminal) in the other specification scheme. For example, term tm can be inherited by another scheme where it becomes terminal as we do not repeat its definition. The constant c may become nonterminal if in another scheme it is defined is the boolean constant T or F.

2.2 Construction

Construction is a transformation from a specification to a specification. The example below is a construction Ψ that the specification body sb of a specification scheme is transformed to the state transition structure sts of another specification scheme.

Specification body sb**:**

$$
\begin{aligned}
sb &::= \; os = iss \\
iss &::= \; is \\
&\quad\; \mid \text{let } A \text{ in } iss \\
&\quad\; \mid P \Rightarrow iss \downarrow iss
\end{aligned}
$$

Abbreviation

sb : specification body

iss : input state structure

: let_in structure

: condition structure

os : output state

is : input state

A : assignment

P : predicate

Construction:

2.2.1. $\Psi\ (os = is) \mathrel{\widehat{=}} os = is$

2.2.2. $\Psi\ (os = (\text{let } A \text{ in } iss)) \mathrel{\widehat{=}} \text{let } A \text{ in } \Psi\ (os = iss)$

2.2.3. $\Psi\ (os = (P \Rightarrow \alpha_1 \downarrow \alpha_2)) \mathrel{\widehat{=}} P \Rightarrow \Psi\ (os = \alpha_1) \downarrow \Psi\ (os = \alpha_2)$

State transition structure sts**:**

$$
\begin{aligned}
sts &::= \; st \\
&\quad\; \mid \text{let } A \text{ in } sts \\
&\quad\; \mid P \Rightarrow sts \downarrow sts \\
st &::= \; os = is
\end{aligned}
$$

Abbreviation:

sts : state transition structure

st : state transition

By the construction equation 2.2.1 the $os = is$ of sb is transformed to the same one of sts. By the construction equation 2.2.2 the $os =$ (let A in iss) of sb is transformed to the let A in Ψ $(os = iss)$ of sts, and similarly for $P \Rightarrow \alpha_1 \downarrow \alpha_2$.

We write construction with the Prolog style [12] by two aspects as followings.

Definition 2.2.1. [match of construction]:

Given a construction equation Ψ $(x_1, \cdots x_n) \hat{=} y$ and a object $< z_1, \cdots z_n >$ of the same tuple $(n \geq 1)$. we say they are matchable if for each pair x_i and z_i the x_i is either z_i or is a variable which is detailed by a definition 6.1.1.1, (see page 39).

□

Definition 2.2.2. [order of construction]:

The construction equations of construction are taken as an ordered sequence from 1 to n

$$eq_1,\ eq_2,\ \cdots,\ eq_n$$

To construct an object $< z_1, \cdots z_n >$ is always to check against the order to find a matchable construction equation and then to deal with produced new objects. (As our construction is total such a equation can be always found). □

For example, in the construction Ψ each construction equation matches a case of sb and the construction is done along the order from 1 to 3.

2.3 Proof

The domain of a construction is defined by a scheme. A construction is required to be total, namely, it can cover each instance specification matching this scheme. Also a construction is required to be correct, namely, each constructed child specification certainly satisfies its parent specification.

Definition 2.3.1. [totalness of construction]:

A construction Ψ of scheme S is total if we can prove

$$\vdash \forall s : \mathcal{S}.\ \Psi\ (s) \in \mathcal{S}' \qquad \square$$

Here and elsewhere, we write $\mathcal{S}$ to represent the set of all specifications which can be produced by the specification scheme S and we add prime to express S' is the child scheme of the scheme S and similarly $\mathcal{S}'$.

Definition 2.3.2. [correctness of construction]:

A construction Ψ of scheme S is correct if we can prove

$$\vdash \forall s : \mathcal{S}.\ \Psi\ (s) \Rightarrow s \qquad \square$$

The main proof method is structural induction [9] as a specification scheme is an abstract syntax structure.

Chapter 3

Derivation Model

In this chapter we present a derivation model which is the foundation of the formal derivation of a class of computers.

In formal hardware verification, a widely accepted and successfully used hardware model [32], which we call the *verification model*, can be characterized by three features:

1. Component and ports are represented by predicates and free variables.

2. Internal port connection is realized by existential quantification.

3. Composition of components is achieved by logical conjunction.

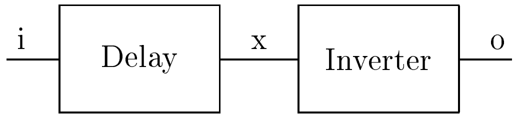
For example, the delay, inverter and delay-inverter circuits are defined as

Delay $(i, x) \equiv \forall t.\ x\ (t + 1) = i\ t$

inverter $(x, o) \equiv \forall t.\ o\ (t + 1) = \neg(x\ t)$

Delay-inverter $(i, o) \equiv \forall t.\ o\ (t + 2) = \neg(i\ t)$

The structure of the delay-inverter circuit below



is described as

Delay-Inverter $(i, o) \equiv \exists x.$ Delay $(i, x) \wedge$ Inverter (x, o)

The verification model concerns the structural abstraction of hardware as verification is a process starting from a given implementation. This is as expressed in [57] (pp. 269): "The type of abstraction most fundamental to hardware verification is *structural* abstraction – the suppression of information about a device's internal structure." The verification model is less suitable for the construction of hardware, particularly, for the construction of microprogrammed computers for this we present a new hardware model, the *derivation model*.

3.1 Derivation Model

In contrast to verification, derivation is a process starting from an abstract behavioral specification to construct the structural implementation. In this process the main aspect is logical, structural refinement.

Derivation Model of Hardware 3.1.1:

Our framework for specifying hardware behavior is a predicate of the shape

$$R\ (\vec{i},\ \vec{o})\ [E_i,\ E_o] \equiv \forall x.\exists y.\ E_i\ x \Rightarrow R\ (\vec{i}\ x,\ \vec{o}\ y) \wedge E_o\ y$$

Here, E_i and E_o are free predicates variables with a time argument. E_i is called entry predicate and E_o is called leaving predicate. $R\ (\vec{i},\ \vec{o})$ is a predicate with free variable argument vectors $\vec{i}$ and $\vec{o}$ for a hardware device and its input and output ports. We write $\vec{i}\ x$ to represent $i_1\ x, \cdots, i_n\ x$ and similarly for $\vec{o}\ y$. Corresponding to E_i and E_o we call x and y entry and leaving time points respectively. $\square$

As a convenience we call the predicate R or $R\ (\vec{i},\ \vec{o})$ the specification body and call $[E_i,\ E_o]$ logical boundary. For convenience in this thesis we often, instead of $R\ (\vec{i},\ \vec{o})\ [E_i,\ E_o]$, write $R\ (\vec{i}\ x,\ \vec{o}\ y)\ [E_i,\ E_o]$ as the latter form is easier to express.

With the model an inverter is written as

$$\text{Inverter}\ (i,\ o)\ [E_i,\ E_o] \equiv \forall x.\ \exists y.\ E_i\ x \Rightarrow o\ y = \neg(i\ x) \wedge E_o\ y$$

In this model, a hardware device is viewed as a process from a time point x to another time point y. We use entry and leaving predicates E_i and E_o to express the controls to enter and leave the process, therefore, the entry predicate also becomes the logical condition to activate a hardware device. In the later chapters we will see that for a microprogrammed computer entry predicates will be instantiated to become predicates about the entry addresses of its microprogram. We existentially quantify the y to leave it to be determined by construction.

Let us take an example. Suppose in a construction process we have a sub-specification:

$$\forall x.\ \exists y.\ E_i\ x \Rightarrow \text{receiver}\ y = \text{ADD}\ (\text{sender}_1\ x, \text{sender}_2\ x) \wedge E_o\ y. \tag{1}$$

This requires us, in the time interval $< x, y >$ and under the condition E_i, to implement an add function and thereafter to ensure the condition E_o. To construct (1) following its structure *naturally* we introduce two new entry predicates E_a and E_b and three new ports $input_1$, $input_2$ and $output$.

$$(\forall x.\ \exists y.\ E_i\ x \Rightarrow input_1\ y = \text{sender}_1\ x \wedge input_2\ y = \text{sender}_2\ x \wedge E_a\ y) \tag{2.1}$$

$$\wedge$$

$$(\forall x.\ \exists y.\ E_a\ x \Rightarrow output\ y = \text{ADD}\ (input_1\ x, input_2\ x) \wedge E_b\ y) \tag{2.2}$$

$$\wedge$$

$$(\forall x.\ \exists y.\ E_b\ x \Rightarrow \text{receiver}\ y = output\ x \wedge E_o\ y) \tag{2.3}$$

Then, introduce two kinds of device, Register and ALU:

$$\text{Register}\ (c, i, o) \equiv \forall t.\ o\ (t+1) = (c\ t \Rightarrow i\ t \mid o\ t)$$

$$\text{ALU}\ (c, i_1, i_2, o) \equiv \forall t.\ o\ (t+1) = (c\ t \Rightarrow \text{ADD}\ (i_1\ t, i_2\ t) \mid i_1\ t)$$

we can further get an abstract microprogram

$$(\forall t.\ E_i\ t \Rightarrow \mathrm{c}_{s1}\ t \wedge \mathrm{c}_{s2}\ t \wedge E_a\ (t+1)) \tag{3.1}$$

$$\wedge$$

$$(\forall t.\ E_a\ t \Rightarrow \mathrm{c}_a\ t \wedge E_b\ (t+1)) \tag{3.2}$$
$$\wedge$$
$$(\forall t.\ E_b\ t \Rightarrow \mathrm{c}_r\ t \wedge E_o\ (t+1)) \tag{3.3}$$

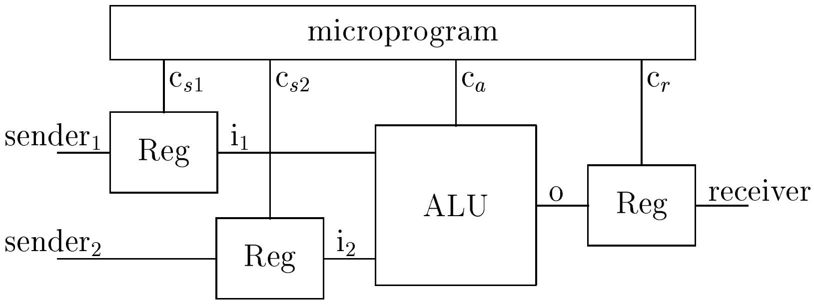
and a datapath

$$\text{Register}\ (\mathrm{c}_{s1}, \text{sender}_1, \mathrm{i}_1) \wedge \text{Register}\ (\mathrm{c}_{s2}, \text{sender}_2, \mathrm{i}_2) \wedge$$
$$\text{ALU}\ (\mathrm{c}_a, \mathrm{i}_1, \mathrm{i}_2, \mathrm{o}) \wedge \text{Register}\ (\mathrm{c}_r, \mathrm{o}, \text{receiver})$$

We can further instantiate the entry predicates of the abstract microprogram to get part of a concrete microprogram.

$$(\forall t.\ \text{address}\ t = 12 \Rightarrow \mathrm{c}_{s1}\ t \wedge \mathrm{c}_{s2}\ t \wedge \text{address}\ (t+1) = 13)$$
$$\wedge$$
$$(\forall t.\ \text{address}\ t = 13 \Rightarrow \mathrm{c}_a\ t \wedge \text{address}\ (t+1) = 14)$$
$$\wedge$$
$$(\forall t.\ \text{address}\ t = 14 \Rightarrow \mathrm{c}_r\ t \wedge \text{address}\ (t+1) = 15)$$

Finally, we get an implementation as the figure below.



We take the specification (1) for our example because a similar form is generally used in the derivation process, and the form is representative.

From this example we can comprehend that the main means of construction is to introduce new entry predicates and new ports. We have an assumption:

Assumption 3.1.1. [existential variable as new one]:

The new entry predicates and new signals (as ports) introduced in construction process are always existential variables. □

Existential variables give us more freedom for further construction until at the last step, to get a concrete implementation, we instantiate these variables.

Part II

Formal Derivation of a Class of Computer

This part is the main content of this thesis. In it we discuss a logic-based process of specification, construction, and verification for a class of computers, from behavioral specification at machine instruction level to structural specification (as implementation) at register transfer level. The process includes five stages:

1. refining time,
2. selecting global structure,
3. decomposing to term transition,
4. decomposing to functional unit and connection,
5. introducing clock.

In conclusion, we present and discuss a concept of proof-based computer architecture.

Chapter 4

Behavior Specification Scheme

In this chapter we define the generic form of behavioral specification for a class of computers, at machine instruction level. This form is the starting point of the whole derivation. We describe three aspects: syntax, semantics, and constraints. An example of a behavioral specification scheme for Gordon's computer can be found in part 3, the case study.

4.1 Syntax

In this section we present the syntax of behavioral specification schemes for a class of computers.

To describe a particular computer at machine instruction level we use state transitions as the semantics of its machine instructions. To describe a class of computers we use abstract syntax to describe a general structure of the state transitions. This is an abstract structure because its terminal symbols are uninterpreted, however, the relationship of the symbols is clearly defined. This is a scheme for a class of computers because the construction and verification of any concrete computer specified by instantiating the scheme will follow the construction and verification of the scheme.

Specification Scheme:			**Abbreviation:**	
$spsc$	$::=$	$\forall u.\ sb$	$spsc$	: specification scheme
sb	$::=$	$os = iss$	sb	: specification body
os	$::=$	$(sos_1, \cdots, sos_n)$	os	: output state
sos_i	$::=$	$sg\ (u+1)$	sos_i	: the i^{th} sub output state $(1 \leq i \leq n)$
iss	$::=$	is	iss	: input state structure
		$\mid$ let A in iss		: let_in structure
		$\mid P \Rightarrow iss \downarrow iss$		: condition structure
is	$::=$	$(sis_1, \cdots, sis_n)$	is	: input state
sis_i	$::=$	tm	sis_i	: the i^{th} sub input state $(1 \leq i \leq n)$
A	$::=$	$v = tm$	A	: assignment
P	$::=$	$p\ (tm, \cdots, tm)$	P	: quantifier free formula
		$\mid \neg P$		
		$\mid P \circ P$		
tm	$::=$	c	tm	: term
		$\mid v$	v	: variable
		$\mid sg\ u$	sg	: state signal
		$\mid f\ (tm, \cdots, tm)$	f	: function
$\circ$	$::=$	$\wedge$	$\circ$	: binary relation
		$\mid \vee$		
		$\mid \Rightarrow$		
		$\mid \Leftrightarrow$		
			u	: time point
			p	: predicate
			c	: constant

It is somewhat interesting but less practical that we could define and discuss a more generic and abstract computer. It is a tradeoff between abstract structure and practical computer that we select the behavior specification scheme above. The important point is that such a scheme must be *constructible*. In fact, the work of the thesis can be understood as a proof that this quite generic behavior specification scheme is constructible. And we will construct it in this thesis.

4.2 Semantics

A behavioral specification scheme, of a computer, describes transitions to an output state os from different input states is; the transition can be taken at any time point u. Each transition corresponds to a machine instruction and expresses the semantics of the instruction.

The output and input states have the same number n of sub-output states sos_i and sub-input sis_i. The sub-output state sos_i has a simple structure: it is the value of a signal sg at a time point $u+1$. By the time point $u+1$ we mean a moment at which a new state of the computer has been produced, namely, the execution of an instruction has been completed. This also means the 1 of $u+1$ is a time unit for the execution of a machine instruction.

In contrast, a sub-input state sis_i has a more complicated structure. In fact, it may be any term tm. A term tm can be a constant c, a variable v or the value of a signal sg at a time point u. Moreover, a term tm can be also constructed by applying a function f to terms, so that we can obtain arbitrarily complex terms through recursion. The time point u is the moment at which a new state has been established and an instruction starts its state transition from the state.

All input states are organized to form an input state structure iss. The iss can recursively lead to any complicated structure. Its base case is just an input state is. Its step cases are constructed through the let_in structure "let A in iss" and the condition structure "$P \Rightarrow iss \downarrow iss$". In the let_in structure we have an assignment $A ::= v = tm$ which declares a simple variable v to replace a complicated term

tm used in iss. By the condition structure "$P \Rightarrow iss_1 \downarrow iss_2$" we mean that if P is true then iss_1 else iss_2. Here the predicate part P is a quantifier-free formula whose base case is an atomic predicate $p\ (tm, \cdots, tm)$ and, through step cases, P can be any complex predicate formula in the logic connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$.

Throughout the thesis we call a signal in the behavior specification of a computer a state signal of the computer. Instead of $P \Rightarrow R \mid S$ we write the condition structure as $P \Rightarrow R \downarrow S$, namely, we use "$\downarrow$" to replace "$|$". This is to avoid confusion with the "$|$" of BNF.

4.3 Constraint

To make the construction for let-in structure feasible, and to get an effective implementation, we have constraints for a let-in structure, let $v = tm$ in S, in the behavior specification scheme.

Constraint 4.3.1. [let-in structure]:

- The variable v must have two or more appearances in its sub-structure S. Otherwise the let-in structure will be unnecessary.

- All variables of let-in structures are distinct from each other and each variable is only used once. So, a redundant structure such as

  let $v = 5$ in let $v = 9$ in S

  can be excluded.

- The variable v only appears in the sub-structure S, which we call well-ordered. By the constraint, recursive assignments, for example below, can be avoided

  let $u = v$ in let $v = u$ in S

as $v = u$ is not well-ordered. □

Based on the constraints we have a theorem about the structure of sequential assignment let-in structure, which follows. Before discussing the theorem we introduce three definitions.

Definition 4.3.1. [sequential assignment let-in structure]:

A sequential assignment let-in structure is formed as

$$\text{let } v_1 = tm_1 \text{ in let } v_2 = tm_2 \text{ in } \cdots \text{ let } v_n = tm_n \text{ in } S$$ □

Definition 4.3.2. [independent assignments]:

A group of assignments formed as $v_j = tm_j \ \ (1 \leq j \leq k)$ is called independent if in the set of variables of the group any two variables are different from each other and no variable appears in any tm_j. □

We can understand that the assignments in such an independent group can be executed in parallel as they are completely independent from each other.

Definition 4.3.3. [independent group sequence let-in structure]:

An independent group sequence let-in structure is formed as

$$G_1 \ G_2 \cdots G_m \ S$$

Here, each $G_i \ \ (1 \leq i \leq m)$ is an independent assignment group and S is a substructure of let-in structure. □

Theorem 4.3.1. [independent group sequence let-in structure]:

Any let-in structure formed under the constraints 4.3.1 can be certainly grouped to get an independent group sequence let-in structure.

Proof:

We sketch an algorithm to sort out the groups. The basic step of the algorithm is, along the assignment sequence of given let_in structure from left to right, to find the sequential separating points as the boundary of groups. Each of such separating points is located in the left of the first assignment $v = tm$ whose tm contain one or more variables which appear as the variables in the assignments located in the left of the separating point. If there is not such a separating point then all assignments are grouped to one group. $\square$

Chapter 5

Refining Time

In this chapter we discuss two steps for time refinement: making time computable and changing time layer.

We introduce some new ideas about time for hardware construction, however to make the ideas easy to understand, in this chapter we basically follow the approach of verification. In the chapter 13.1, after reader has sufficient understanding of our derivation model, we will present and discuss new concepts: how to understand and how to construct time layers in a formal hardware design process.

5.1 Time in Construction

The construction process of a hardware device also constructs the timing of the hardware device. Based on the derivation model we can have a concise and clear concept about time construction.

5.1.1 Constructing Time

Construction raises a concept of time which differs from the one of verification.

For verification both implementation and specification are given the task of verification is to prove that the implementation satisfies the specification. For construction only specification is given the task of construction is to construct an implementation which satisfies the specification.

This leads to a difference in the treatment of time, for verification and construction. Verification makes a temporal abstraction from a given low level time of implementation to the given high level time of specification. Construction makes a temporal refinement from a given high level time of specification to the *unknown* low level time of implementation.

Verification:

[**low_level_time, temporal_abstraction, high_level_time**]

Construction:

[**high_level_time, temporal_refinement,** ***low_level_time***]

Let us emphasize that the low level time of implementation in construction is constructed, not given.

5.1.2 Time Based on Derivation Model

Intuitively, for a hardware device we take the execution periods of its components as the time unit and take the synchronisation of the execution process of two components as time point. Separating a device to its components is relative, namely we can have different coarse level components, therefore we have different time layers.

Based on the derivation model the basic means of construction is to introduce a new middle entry predicate to decompose a derivation model to two new models. We take the derivation model as a foundation to measure time. For derivation model

$$\forall x.\ \exists y.\ E_i\ x \Rightarrow R\ (\vec{i}\ x,\ \vec{o}\ y) \wedge E_o\ y$$

we take entry predicate to measure *time point*, e.g. by making $E_i\ x$ be true the x is taken as a time point. And we take the specification entity $R\ (\vec{i}\ x,\ \vec{o}\ y)$ to measure *time interval* $< x, y >$ as to implement the R consumes a certain time. Therefore by construction when we get a specification, mainly through decomposition, we use all the derivation models in the specification to determine the time points of the specification and all these time points establish the time layer of the specification. In the time layer we take the execution time of each derivation model as a time unit, namely we take the time unit as individual. Of course, this is only relative to the time layer of the specification.

5.2 Making Time Computable

We start from the most abstract form of the behavior specification scheme. In this form we only give the specification body sb by $S\ (u, u + 1)$, namely, we do not consider the internal structure of sb.

Specification Scheme 0: $\forall u : \text{time}.\ S\ (u, u + 1)$

The specification merely is a timed predicate S having two successive time point arguments u and $u + 1$. The time is a set of natural number for the time point of specification of a computer at machine instruction level. We use $u + 1$ merely to express a successive time point which follows the time point u and it is better if we write "succ u" instead of "$u + 1$" to emphasize that only successive relation is required by our specification. The specification tells us that a computer should work well at any time. By construction 1 we make the time point of the scheme 1 become computable.

Construction 1:

5.2.1. $\Psi_1\ (\forall u.\ S\ (u, u + 1)) \mathrel{\hat{=}} E\ 0 \wedge \forall u.\ E\ u \Rightarrow S\ (u, u + 1) \wedge E\ (u + 1)$

In the construction we introduce an entry predicate E to make an inductively computable time point. This is a loop: starting from 0 and goes along 0, 1, 2, $\cdots$.

We let the E be a existential variable to leave it to further constructions.

Specification Scheme 1: $E\ 0 \wedge \forall u.\ E\ u \Rightarrow S\ (u, u+1) \wedge E\ (u+1)$

Even at the such an abstract level as the specification scheme 1 we have obtained, by construction, a clear, but abstract, structure: that a computer should certainly have a loop structure for its time. Having $E\ 0$ as start point and repeatedly making E be true to get any further time point.

The totalness and correctness of construction are obvious.

5.3 Changing Time Layer

It is natural that a certain level of specification is always described in a certain time layer, and that different level specifications are described in the different time layers.

For example, the behavioral specification of a computer at machine instruction level is described in the machine instruction time layer which defines a time unit corresponding to the complete execution of a machine instruction. In contrast, the structural specification of the computer at register transfer level is described in the microinstruction time layer which defines a time unit as the interval representing the execution of a microinstruction.

To construct implementation from specification we must change time layer. Furthermore as time underlies hardware specification, the change of time layer will influence all things which are set up based on time.

In contrast to verification in this stage of construction what we know is only the high time layer, machine instruction time layer; we do not yet know what low time layer is. What we establish in this construction step is the relationship of two time layers. Namely, the low time layer must be obtained by construction. To this end we introduce the temporal abstraction function [50] to set up the relationship between two, arbitrary, successive time layers.

Definition 5.3.1. [temporal abstraction function]:

$\vdash$ abs $mark\ 0 = \varepsilon\ t.\ mark\ t \wedge (\forall t'.\ t' < t \Rightarrow \neg mark\ t')$

$\vdash$ abs $mark\ (u + 1) =$

$\quad\varepsilon\ t.$

$\quad\quad mark\ t \wedge$

$\quad\quad(\text{abs}\ mark\ u) < t \wedge$

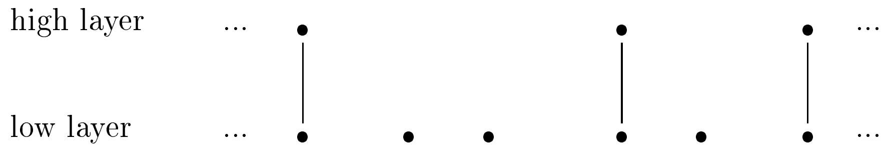
$\quad\quad(\forall t'.\ (\text{abs}\ mark\ u) < t' < t \Rightarrow \neg mark\ t')$

Here, "abs" is a function of type: (num$\rightarrow$ bool) $\rightarrow$ num $\rightarrow$ num, "num" is the set of natural numbers as the type of discrete time points and "*mark*" is a function of type: num $\rightarrow$ bool making (abs *mark*) a function of type: num $\rightarrow$ num. ε is the symbol used in the HOL system [33] for Hilbert's epsilon operator which may be read as 'the t such that ...'. As pp. 271 - 272 [57] if P[x] is a predicate involving a variable x of type ty then $\varepsilon\ x$. P[x] denotes some value of type ty such that P is true of that value. If there is no such value (i.e. P[v] is false for each value v of type ty) then $\varepsilon\ x$. P[x] denote some fixed but arbitrarily chosen value of type ty.

$mark$ is a boolean valued signal in the low time layer. "abs $mark\ u$" denotes the point of low time layer at which $mark$ is true for the uth time. Therefore we can take "abs $mark$" as a function from the time point in high time layer to the time point in low time layer.

What the temporal abstraction function establishes is that a time point in the high layer corresponds to a time point in the low layer and one time unit in the high layer can correspond to a time interval in the low layer, which can include one or more time units in the low time layer.

Verification:



5.3.1 Construction 2 and Specification Scheme 2

We use the construction 2 to construct the specification scheme 2 from the specification scheme 1.

Construction 2:

5.3.1.1. $\Psi_2\ (E\ 0 \wedge \forall u.\ E\ u \Rightarrow S\ (u, u+1) \wedge E\ (u+1)) \mathrel{\widehat{=}}$

$$E'\ 0 \wedge \Psi_{2a}\ (\forall u.\ E\ u \Rightarrow S\ (u, u+1) \wedge E\ (u+1))$$

5.3.1.2. $\Psi_{2a}\ (\forall u.\ E\ u \Rightarrow S\ (u, u+1) \wedge E\ (u+1)) \mathrel{\widehat{=}}$

$$\forall x.\ \exists y.\ E'\ x \Rightarrow \Psi_{2b}\ (S\ (u, u+1)) \wedge E'\ y$$

5.3.1.3. $\Psi_{2b}\ (S\ (u, u+1)) \mathrel{\widehat{=}} S'\ (x, y) \wedge (\forall t.\ x < t < y \Rightarrow \neg E\ t\ ) \wedge x < y$

By construction 2 we get the scheme 2 at low time layer from the scheme 1 at high time layer. We know that all signals, functions and predicate in specification are established based on the time. Therefore when we change time layer we must, for each component in specification scheme 2, reestablish its new form in the new time layer. For this in the scheme 2 we use S' to note that it is got by replacing each component c in S of the scheme 1, signal, function and predicate, by c'. Here, we write c as a component in the high time layer and c' as a component in the low time layer, which corresponds to c. For example, E' which is in the low time layer corresponds to E which is in the high time layer.

To guarantee that such a change leads to a correct construction we need to introduce some assumptions which are based on the temporal abstraction function.

Assumption 5.3.1.1. [correspondence through signal value]:

For each signal s in the S of the scheme 1 we assume that we can construct a signal s' in the low time layer such that

$s\ u = s'\ (abs\ E'\ u)$ $\square$

Note: selecting E' as the *mark* of temporal abstract function is not fortuitous. We will discuss this in the chapter 13.1.

We can have such an assumption because that we have an observation for the behavior specification:

Observation 5.3.1.1.

In the behavior specification a time point u always appears as an argument of a signal and vice versa a signal sg always appears in application to a time point. Namely, they always appear together with the form $sg\ u$. $\square$

However, by the assumption 5.3.1.1 we only get the s' having definition at some low layer time points that can correspond to the high layer time points. To establish these s' are total for all time points in low time layer we need the second assumption.

Assumption 5.3.1.2. [signal is total for low time layer]:

We extend all signal s' of assumption 5.3.1.1 to have definition at all low time points and leave the concrete form of the definition to be established in the further construction process (if the definition is not required by the further construction process then we have a trivial case that we can simply assume that they have an arbitrary value of the same type as those having defined time point). $\square$

We need to further consider other high level structure such as function and predicate to make clear their significant in specification scheme 2. For this we have the third assumption:

Assumption 5.3.1.3. [extending higher level structures]:

We extend all higher level structures, we say higher level structure to distinguish it (functions or predicates) from signals (the function of time), in specification scheme 1 to have definitions in scheme 2 for their arguments which include time points, the extended signals by the assumption 5.3.1.2 and the higher level structures which have been extended by this assumption. Similar to the assumption 5.3.1.2, we leave the concrete form of these definitions to further construction process or simply have a trivial definition if they are irrelevant to construction. □

Notice 5.3.1.1. [omitting]:

Only in the construction 2 we use prime "′" to distinguish the same component but in two different time layers. After the construction 2 we will omit the prime.

□

Specification Scheme 2:

$$
\begin{array}{ll}
E\ 0 \wedge & \\
\forall x.\ \exists y.\ E\ x \Rightarrow S\ (x, y) \wedge & \\
& (\forall t.\ x < t < y \Rightarrow \neg E\ t) \wedge \\
& x < y \wedge \\
& E\ y
\end{array}
$$

The construction 2 and the specification scheme 2 show that

1. The further construction from the specification scheme 2 can be in a new finer time layer;

2. The time point x and y in the scheme 2 corresponds to the time point u and $u + 1$ in the scheme 1.

3. the y is existentially quantified such that it is, following arbitrarily given time point x, constructible and we constrain $x < y$ to make sure that y follows x as that $u + 1$ is the successor of u;

4. the scheme 2 describes such a constraint that a new introduced finer time point t should correspond to an entry predicate differing from E. This is

done by the predicate:

$$\forall t.\ x < t < y \Rightarrow \neg E\ t.$$

Convenience 5.3.1.1. [omitting]:

In the further constructions from the scheme 2, for a concise writing form, we will omit the constraint $x < y$ as it will certainly be satisfied by at least a introduction of two time point a and b between x and y having relation $a < b$ or we have $x < y$ directly. We can also omit the constraint $(\forall t.\ x < t < y \Rightarrow \neg E\ t)$ as all further introduced new entry predicates are certainly different from E therefore the constrain is satisfied automatically. □

5.3.2 Totalness and Correctness of Construction $2'$ and 2

In this section if necessary we will use subscripts u and t to distinguish the time point in the high and low layer. The totalness of the construction 2 is obvious.

Theorem 5.3.2.1. [correctness of construction 2]:

For any $s \in \mathcal{S}_1$ if $\vdash \Psi_2\ (s)$ then $\vdash s$.

Proof:

The statement of the theorem is equal to that

if

$$\begin{aligned}\vdash\ & E'\ 0\ \wedge \\ & \forall x.\ \exists y.\ E'\ x \Rightarrow S'\ (x, y)\ \wedge \\ & \qquad\qquad (\forall t.\ x < t < y \Rightarrow \neg E'\ t) \wedge x < y\ \wedge \\ & \qquad\qquad E'\ y\end{aligned}$$

then

$$\begin{aligned}\vdash\ & E\ 0\ \wedge \\ & \forall u.\ E\ u \Rightarrow S\ (u, u+1)\ \wedge \\ & \qquad\qquad E\ (u+1).\end{aligned}$$

We have two steps to prove:

(1). By assumption 5.3.1.1 we have

$\vdash E\ 0_u = E'\ (\text{abs}\ E'\ 0_u)$.

By the definition of abs and E' we can suppose abs $E'\ 0_u = 0_t$,

therefore, we have $\vdash E\ 0_u = E'\ 0_t$. Omitting subscript we have $\vdash E\ 0 = E'\ 0$.

As $E'\ 0$ has been given we have $\vdash E\ 0$.

(2). Given $E\ u$, here, u is an arbitrary time point in the high time layer.

By the assumption 5.3.1.1 we have

$\vdash E\ u = E'\ (\text{abs}\ E'\ u)$.

Suppose $t_a = \text{abs}\ E'\ u$ we have

$\vdash E'\ t_a$.

From

$$\begin{aligned}
\vdash\ & E'\ 0\ \wedge \\
& \forall x.\ \exists\ y.\ E'\ x \Rightarrow S'\ (x, y)\ \wedge \\
& \qquad\qquad (\forall t.\ x < t < y \Rightarrow \neg E'\ t) \wedge x < y\ \wedge \\
& \qquad\qquad E'\ y
\end{aligned}$$

we can get

$$\begin{aligned}
\vdash\ & \exists y.\ E'\ t_a \Rightarrow S'\ (t_a, y)\ \wedge \\
& \qquad\qquad (\forall t.\ t_a < t < y \Rightarrow \neg E'\ t) \wedge t_a < y\ \wedge \\
& \qquad\qquad E'\ y.
\end{aligned}$$

Then taking t_b as the selected existential variable y of $\exists y$, we have

$$\begin{aligned}
\vdash\ & E'\ t_a \Rightarrow S'\ (t_a, t_b)\ \wedge \\
& \qquad\qquad (\forall t.\ t_a < t < t_b \Rightarrow \neg E'\ t) \wedge t_a < t_b\ \wedge \\
& \qquad\qquad E'\ t_b.
\end{aligned}$$

As we have $\vdash E'\ t_a$ we have

$\vdash S'\ (t_a, t_b)\ \wedge$

$(\forall t.\ t_a < t < t_b \Rightarrow \neg E'\ t) \wedge t_a < t_b \wedge$

$E'\ t_b.$

By the definition 5.3.1 we have

$\vdash \text{abs } E'\ (u+1) =$

$\varepsilon\ t_b.$

$E'\ t_b \wedge$

$(\text{abs } E'\ u) < t_b \wedge$

$(\forall t.\ (\text{abs } E'\ u) < t < t_b \Rightarrow \neg E'\ t).$

As we have supposed $\vdash t_a = \text{abs } E'\ u$ we have

$\vdash \text{abs } E'\ (u+1) =$

$\varepsilon\ t_b.$

$E'\ t_b \wedge$

$t_a < t_b \wedge$

$(\forall t.\ t_a < t < t_b \Rightarrow \neg E'\ t),$

therefore, we can get

$\vdash \text{abs } E'\ (u+1) = t_b.$

By the assumption 5.3.1.1 we have

$\vdash E\ (u+1) = E'\ (\text{abs } E'\ (u+1)).$

Hence we have

$\vdash E\ (u+1) = E'\ t_b.$

We have

$\vdash E'\ t_b.$

So we can get

$\vdash E\ (u+1).$

By the assumption 5.3.1.1, 2 and 3, and from $\vdash S'\ (t_a, t_b)$ we get

$\vdash S\ (u, u+1).$

This is that from a given $E\ u$ of an arbitrary u we can get $S\ (u, u+1)$ and $E\ (u+1)$.

By above (1) and (2) we get

$\vdash E\ 0 \wedge \forall u.\ E\ u \Rightarrow S\ (u, u+1) \wedge E\ (u+1).$ $\square$

5.3.3 Discussion: Time Refinement and Predicate E

In Gordon's computer [50] a special predicate "ready" is introduced as the *mark* of the temporal abstraction function. However in this section we have proved that the "ready" is redundant and can be replaced by the entry predicate E. This is a interesting example as it illustrates some differences between derivation and verification: derivation is an active construction and proof for an unknown object whereas verification is a passive proof for a known object. So, derivation has more freedom to create a new thing.

It is a compromise that we basically take the viewpoint of verification to deal with time layer change. It is to view the high and low two time layers as two *complete different* layers and their relation is the *corresponding* of time points. After reader becomes be familiar with the derivation model we will describe a new concept: low time layer includes high time layer and their relation is construction. The description for time layer change in this chapter is quite wordy. However, by the new concept all these are unnecessary.

Chapter 6

Selecting Global Structure

Construction as formal design includes two aspects: selection and decomposition. By selection we choice a desired one from many possible structures at a certain specification level. By decomposition we divide a structure at the high specification level to the several structures at the low specification level. It is important to note that a selection at the earlier stage of a design has a big influence on the later stages of the design. And a basic fact is that the structure of a specification often implies the structure of the implementation of the specification:

- A let_in structure "let A in S" implies the selection that first is to execute A and then is to execute S.

- A condition structure $P \Rightarrow S_1 \downarrow S_2$ means a necessary situation that first is to execute P then is to execute S_1 if the execution of P gets boolean value true or to execute S_2 if gets false.

By the selecting global structure we mainly do two things: selecting parallel/sequential execution and selecting common block. However, before this we have a technical step: setting state transition.

6.1 Setting State Transition

The specification scheme 2, like as the behavior specification scheme, has the output state and the input states of computer written separately. To get an individual state transition for each machine instruction we need to distribute output state to each input state across the assignment of let_in structure and the condition predicate of condition structure. For example, from

$$\begin{aligned} output\_state = \ & \text{let } A_1 \text{ in} \\ & \text{let } A_2 \text{ in} \\ & P_1 \Rightarrow input\_state_1 \downarrow \\ & (P_2 \Rightarrow input\_state_2 \downarrow input\_state_3) \end{aligned}$$

transform to

$$\begin{aligned} & \text{let } A_1 \text{ in} \\ & \text{let } A_2 \text{ in} \\ & P_1 \Rightarrow (output\_state = input\_state_1) \downarrow \\ & (P_2 \Rightarrow (output\_state = input\_state_2) \downarrow (output\_state = input\_state_3)) \end{aligned}$$

Each $output\_state = input\_state_i \ \ (1 \leq i \leq 3)$ is a state transition for a machine instruction.

6.1.1 Construction 3 and Specification $2'$ and 3

In this step we give more details to the $S\ (x, y)$ of the scheme 2 and we rewrite 2 to $2'$.

Specification Scheme $2'$:

$E\ 0 \wedge \forall x.\ \exists y.\ E\ x \Rightarrow sb \wedge E\ y$

Specification body sb:		**Abbreviation**	
$sb ::=$	$os = iss$	sb	: specification body
$iss ::=$	is	iss	: input state structure
	$\mid$ let A in iss		: let_in structure
	$\mid P \Rightarrow iss \downarrow iss$		: condition structure
		os	: output state
		is	: input state
		A	: assignment
		P	: predicate

For this construction we need not the further detail of the output state os, input state is, assignment A and predicate P.

The construction 3 only change the specification body sb of the scheme $2'$.

Construction 3:

6.1.1.1. $\Psi_3\ (os = is) \mathrel{\widehat{=}} os = is$

6.1.1.2. $\Psi_3\ (os = (\text{let } A \text{ in } iss)) \mathrel{\widehat{=}} \text{let } A \text{ in } \Psi_3\ (os = iss)$

6.1.1.3. $\Psi_3\ (os = (P \Rightarrow \alpha_1 \downarrow \alpha_2)) \mathrel{\widehat{=}} P \Rightarrow \Psi_3\ (os = \alpha_1) \downarrow \Psi_3\ (os = \alpha_2)$

In the construction equation 6.1.1 we use variables α_1 and α_2 to represent two different input state structure iss. Generally we have

Definition 6.1.1.1. [variable in construction]:

In a construction equation Ψ we can use variable to represent any part of the argument of the Ψ. Variable is written by Greek letter. In the match of construction (the definition 2.2.1) a variable can always match the concrete part of an object to be constructed. $\square$

Usually, we use variable to represent a complex part of an object and use different variables to distinguish different parts of an object as these parts sometimes can be written as the same in BNF.

Use the construction 3 to the scheme $2'$ we get

Specification Scheme 3:

$E\ 0 \wedge \forall x.\ \exists y.\ E\ x \Rightarrow sts \wedge E\ y$

State transition structure sts**:** **Abbreviation:**

$$\begin{aligned} sts &::= st \\ &\quad \mid \text{let } A \text{ in } sts \\ &\quad \mid P \Rightarrow sts \downarrow sts \\ st &::= os = is \end{aligned}$$

sts : state transition structure

st : state transition

6.1.2 Totalness and Correctness of Construction 3

Theorem 6.1.2.1. [totalness of construction 3]:

$\forall s : \mathcal{S}_{2'}.\ \Psi_3\ (s) \in \mathcal{S}_3$

Proof: Use structural induction on the structure of the input state structure iss of the scheme $2'$. □

Lemma 6.1.2.1. [distributing let_in over $\Rightarrow$]:

If $\vdash P \Rightarrow Q$ then $\vdash$ let A in $P \Rightarrow$ let A in Q

Proof: Supposing $A \equiv (v = tm)$, by the constraint 4.3.1 (see page 22) when we write "let A in P" we certainly have the v as a variable in P. Therefore from $\vdash P \Rightarrow Q$ we can get $\vdash P[tm/v] \Rightarrow Q[tm/v]$.
Then we get $\vdash$ let A in $P \Rightarrow$ let A in Q. □

Here and elsewhere, we use substitution defined as below.

Definition 6.1.2.1. [substitution]:

We write $\alpha[y/x]$ to denote the result of replacing each x in α by y. More precise definition of substitution, for example about avoiding the capture of a free variable, follows [64] Page 26-27. □

Lemma 6.1.2.2. [distributing condition over $\Rightarrow$]:

If $\vdash Q \Rightarrow Q'$ and $\vdash R \Rightarrow R'$ then $\vdash (P \Rightarrow Q \downarrow R) \Rightarrow (P \Rightarrow Q' \downarrow R')$

Proof:

From $\vdash Q \Rightarrow Q'$ we have $\vdash (P \Rightarrow Q) \Rightarrow (P \Rightarrow Q')$

and

from $\vdash R \Rightarrow R'$ we have $\vdash (\neg P \Rightarrow R) \Rightarrow (\neg P \Rightarrow R')$.

Therefore we have

$\vdash (P \Rightarrow Q) \wedge (\neg P \Rightarrow R) \Rightarrow (P \Rightarrow Q') \wedge (\neg P \Rightarrow R')$.

Finally we get

$\vdash (P \Rightarrow Q \downarrow R) \Rightarrow (P \Rightarrow Q' \downarrow R')$. □

Lemma 6.1.2.3. [correctness of Ψ_3]:

$\vdash \forall sb : \mathcal{S}_{2'}.\ \Psi_3\ (sb) \Rightarrow sb$

Proof:

Use structural induction on the iss of sb of the scheme $2'$.

Base case: $iss\ ::=\ is$. It is obvious.

Step case: There are two cases: s1 and s2.

Case s1: $iss\ ::=$ let A in iss'.

By the induction hypothesis we have

$\vdash \Psi_3\ (os = iss') \Rightarrow os = iss'$.

Then by lemma 6.1.2.1 we have

$\vdash$ let A in $\Psi_3\ (os = iss') \Rightarrow$ let A in $(os = iss')$.

As there is no variable in os (by the behavior specification scheme) we have

$\vdash$ let A in $\Psi_3\ (os = iss') \Rightarrow os =$ let A in iss'.

By Ψ_3 we have

$\Psi_3\ (os = (\text{let } A \text{ in } iss')) = \text{let } A \text{ in } \Psi_3\ (os = iss')$,

therefore we get

$\vdash\ \Psi_3\ (os = (\text{let } A \text{ in } iss')) \Rightarrow os = \text{let } A \text{ in } iss'$

Case s2: $iss\ ::=\ P \Rightarrow iss' \downarrow iss''$.

Similar with the above case 1 but use the lemma 6.1.2.2. □

Theorem 6.1.2.2. [correctness of Ψ_3]:

$\vdash \forall s : \mathcal{S}_{2'}.\ \Psi_3\ (s) \Rightarrow s$.

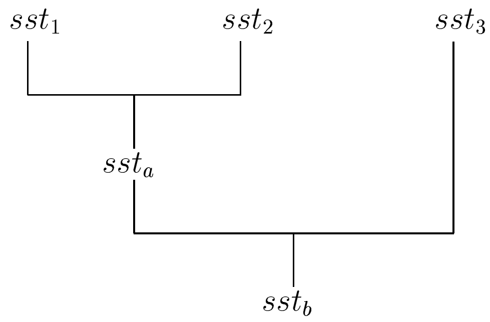
Proof: By the fact that only sb of $S_{2'}$ is changed and use the lemma 6.1.2.3. $\square$

6.2 Selecting Common Sub_State Transition

Using let_in sentence we can pull out common structure in a term. This will lead to a corresponding implementation structure. However, for the construction of computer and as a complement, we can also select some common sub-structure of state transitions and the selection will also lead to its corresponding implementation structure. For example, for

$$State\_Trans_1 = Sub\_ST_1 \wedge [Sub\_ST_a] \wedge [Sub\_ST_b]$$
$$State\_Trans_2 = Sub\_ST_2 \wedge [Sub\_ST_a] \wedge [Sub\_ST_b]$$
$$State\_Trans_3 = Sub\_ST_3 \wedge [Sub\_ST_b]$$

we select sub-state transition Sub_ST_a as the common block of the state transition 1, $State\_Trans_1$, and 2, $State\_Trans_2$, and Sub_ST_b as the common block of the state transition 1, 2 and 3. We bracket them to mark our selection. This will finally lead to the microprogram structure of an implementation formed as below (Here, we write sst for $State\_ST$)



Definition 6.2.1. [common block]:

In a state transition st some of its sub-state transitions, if they are shared by two or more state transitions, can be selected as common block. We use brackets "[" and "]" to mark selected block, for example $[cb]$, to express that cb is selected as a common block. $\square$

Introducing common blocks in a specification can make the corresponding implementation with less hardware resources but, possibly, taking more computation time.

We can extend let_in structure to include common block. However, for the construction of computer we use the concept of common block to express a selection for the execution order, which is as the picture above shown:

Assumption 6.2.1. [common block in implementation]:

A common block as sub-state transition is always to be put to the rear part of the execution process of the state transition. □

The brackets of common block are purely syntactic and should not influence the logic meaning of specification.

Assumption 6.2.2. [bracket non-relating to logic meaning]:

We extend specification scheme to including the syntactic form of common block and suppose that for any common block $[\alpha]$ to have $\vdash [\alpha] \Leftrightarrow \alpha$. □

6.2.1 Construction 4 and Specification Scheme $3'$ and 4

For the construction 4 we rewrite the scheme 3 to the scheme $3'$ by rewriting the state transition st of the scheme 3 to the conjunction of its sub-state transitions. The rewriting is according to the behavior specification scheme.

Specification Scheme $3'$:

$E\ 0 \wedge \forall x.\ \exists y.\ E\ x \Rightarrow sts \wedge E\ y$

State transition structure sts:	**Abbreviation:**
$sts ::= st$	sts : state transition structure
$\quad\quad \mid \text{let } A \text{ in } sts$	
$\quad\quad \mid P \Rightarrow sts \downarrow sts$	
$st \ ::= sst$	st : state transition
$\quad\quad \mid sst \wedge st$	

$sst ::= \quad sos_i = sis_i$

sst : sub-state transition

sos_i: the i^{th} sub output state

sis_i: the i^{th} sub input state

$(1 \leq i \leq n)$

Construction 4 only selects whether or not a sub-state transition sst as a common block.

Construction 4:

6.2.1.1. $\Psi_4\ (st) \widehat{=} \mathcal{A}_4\ (\Psi_{4a}\ (st))$

6.2.1.2. $\Psi_{4a}\ (sst) \widehat{=} \Psi_{4b}\ (sst)$

6.2.1.3. $\Psi_{4a}\ (sst \wedge st) \widehat{=} \Psi_{4b}\ (sst) \wedge \Psi_{4a}\ (st)$

6.2.1.4. $\Psi_{4b}\ (sst) \widehat{=}\ sst \mid [sst]$

In the construction equation 6.2.1.4 we introduce symbol "|" which means "or". The construction equation says that by construction Ψ_{4b} sst can be transformed to sst or $[sst]$. Generally, we have

Definition 6.2.1.1. [nondeterministic construction]:

An object x can be constructed by Ψ to get y_1 or y_2 or ... y_n. We call this nondeterministic construction and write:

$$\Psi\ (x) \widehat{=} y_1 \mid y_2 \mid \cdots \mid y_n \qquad \square$$

In the construction 4 we introduce an abstract algorithm $\mathcal{A}_4$ to sort the common blocks in a state transition. Generally we have

Definition 6.2.1.2. [abstract algorithm]:

Abstract algorithm is a construction which is borrowed from from other area of computer science, therefore we only require its specification. For an object x, through an abstract algorithm $\mathcal{A}$, is constructed to get y we write

$$\mathcal{A}\ (x) \widehat{=} y. \qquad \square$$

We introduce abstract algorithm as a means to interface our derivation to other methods, especially to the optimization algorithm in the high level synthesis field

[60]. We call abstract algorithm as we only describe its specification.

Abstract Algorithm 6.2.1.1. [$\mathcal{A}_4$: sorting common blocks]:
$\mathcal{A}_4$ manipulates Ψ_{4a} (st).

1. All sst which are not selected as common blocks are put together to get sub-state transition set $ssts$ as the private part pp of the state transition.

2. All $[sst]$ that are shared by the same state transitions will be put together to get a common sub-state transition set $[ssts]$.

3. Then, all $[ssts]$ are sorted as below to get the common part cp of scheme 5:

   - All the $[ssts]$ in a state transition are sorted from left to right in the order of increasing number of shared state transitions st. For example, suppose a state transition st includes three common blocks cb_1, cb_2 and cb_3, where cb_1 is shared by 5 state transitions, cb_2 is shared by 2 state transitions, and cb_3 is shared by 3 state transitions. Then we have $\mathcal{A}_5$ $(cb_1 \wedge cb_2 \wedge cb_3) = cb_2 \wedge cb_3 \wedge cb_1$. As the private part pp of a state transition is always "shared" by itself only the pp always is put the most left. For the example above we get $st = pp \wedge cb_2 \wedge cb_3 \wedge cb_1$.

4. For each state transition st_i check the sequences of sub-state transition set $ssts$ in the st_i from left to right such that for arbitrary two state transition, for example st_1 and st_2, if they have the same common block then all other $ssts$ on the right (if any) of the common block must be the same and have the same order.

   - Where this is not satisfied change some common block back to the private part of st to satisfy the requirement above.

     Such checking and changing is to guarantee a tree form structure. For example, the case

     $st_1 = pp_1 \wedge [ssts_a] \wedge [ssts_b]$

$st_2 = pp_2 \wedge [ssts_a]$

$st_3 = pp_3 \wedge [ssts_b]$

cannot lead to a tree form implementation structure and we need to change to either

$st_1 = (pp_1 \wedge ssts_a) \wedge [ssts_b]$

$st_2 = (pp_2 \wedge ssts_a)$

$st_3 = pp_3 \wedge [ssts_b]$

in which case $ssts_a$ is moved back to the private parts, or

$st_1 = (pp_1 \wedge ssts_b) \wedge [ssts_a]$

$st_2 = pp_2 \wedge [ssts_a]$

$st_3 = (pp_3 \wedge ssts_b)$

in which case $ssts_b$ is moved back to the private parts. □

After the construction 4 we get the specification scheme 4.

Specification Scheme 4:

$E\ 0 \wedge \forall x.\ \exists y.\ E\ x \Rightarrow sts \wedge E\ y$

State transition structure sts**:** **Abbreviation:**

$$
\begin{array}{lll}
sts & ::= & st \\
 & | & \text{let } A \text{ in } sts \\
 & | & P \Rightarrow sts \downarrow sts \\
st & ::= & pp \wedge cp \\
pp & ::= & \mathrm{T} \\
 & | & ssts \\
cp & ::= & \mathrm{T} \\
 & | & [ssts] \wedge cp \\
ssts & ::= & sst \\
 & | & sst \wedge ssts
\end{array}
$$

sts : state transition structure

st : state transition

pp : private part

cp : common part

$ssts$: sub-state transition set

T : boolean value true

sst : sub-state transition

In the scheme 4 a state transition st includes two parts: the private part pp and the common part cp. However, each part can be empty which is expressed by the boolean constant T.

The totalness and correctness of the construction 4 is obvious.

6.3 Selecting Parallel Term Transition

We can put two function units ADD and SUB into the same time interval to execute if they are implemented by two hardware devices. We call this as the parallel execution. The parallelism is at function unit level which is traditionally called as the register transfer level.

In the construction process we find that we can introduce parallelism at a higher level which we call the term transition level in the earlier stage of design.

Definition 6.3.1. [term transition]:

A term transition is formed as $sg\ y = tm$. Here, sg is a signal and tm is a term which is defined in the behavior specification scheme, however here, is extended to include predicate p and predicate connective $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$ as function f. □

For the construction of computer there are two cases where we can select whether or not to take the parallel term transition. For example, for a let_in structure

let $v_1 = tm_1$ in let $v_2 = tm_2$ in S,

suppose that it satisfies the constraint 4.3.1 (page 22) and their two assignments are independent from each other. We can construct two term transitions for the two assignments

$$s_1\ y = tm_1 \qquad (1)$$

$$s_2\ y = tm_2 \qquad (2)$$

and replace S by $S\,[s_1\ y/v_1,\ s_2\ y/v_2]$.

Then we have two selections:

(a). parallel execution: in the same time interval to execute both (1) and (2).

(b). sequential execution: taking two time intervals: first to execute (1) and then to execute (2)

For the case (a) we write:

$$\{_1 \text{ let } v_1 = tm_1 \text{ in let } v_2 = tm_2 \text{ in } \{_2\ S\}\} \tag{l1}$$

and for the case (b) we write:

$$\{_3 \text{ let } v_1 = tm_1 \text{ in } \{_4 \text{ let } v_2 = tm_2 \text{ in } \{_5\ S\}\}\} \tag{l2}$$

We have

Assumption 6.3.1. [parallel selection for let_in structure]:

The assignments between two successive braces "{" are selected for parallel execution. □

For example, in (l1) both $v_1 = tm_1$ and $v_2 = tm_2$ are put into two successive braces $\{_1$ and $\{_2$. And in (l2) only one $v_1 = tm_1$ is put between $\{_3$ and $\{_4$ and another $v_2 = tm_2$ is put between $\{_4$ and $\{_5$. Note, here we put subscript to braces "{" for conveniently distinguishing them.

However we have a constraint for the selection.

Constraint 6.3.1. [parallel selection for let_in structure]:

Only the assignments in the same group of independent assignment (the definition 4.3.2, page 23) can be selected for parallel execution. □

We have the constraint as only under it a parallel execution is feasible or, at least, better parallel.

Similarly, for a condition structure

$P_1 \Rightarrow S_1 \downarrow (P_2 \Rightarrow S_2 \downarrow S_3)$

we introduce two term transitions for the executions of two predicates P_1 and P_2

$$b_1\ y = P_1 \tag{3}$$

$$b_2\ y = P_2 \tag{4}$$

Here b_1 is a boolean valued signal and (3) expresses a process to calculate predicate P_1 such that at y b_1 has the same boolean value as of the P_1.

Similarly, we have selection for that parallel and sequential executions for (3) and (4). After the execution of (3) and (4) the execution of the condition structure becomes to execute

$$b_1\ x \Rightarrow S_1 \downarrow (b_2\ x \Rightarrow S_2 \downarrow S_3)$$

For the parallel one we write the condition structure

$$\{_1 P_1 \Rightarrow \{_2 S_1\} \downarrow (P_2 \Rightarrow \{_3 S_2\} \downarrow \{_4 S_3\})\} \qquad \text{(p1)}$$

and for the sequential one

$$\{_5 P_1 \Rightarrow \{_6 S_1\} \downarrow \{_7 P_2 \Rightarrow \{_8 S_2\} \downarrow \{_9 S_3\}\}\} \qquad (p2)$$

We also have

Assumption 6.3.2. [parallel selection for condition structure]:

The predicates between the same brace "{" and a group of successive braces "{" are selected for parallel execution. □

For example, in (p1) P_1 and P_2 are put into a pair of $\{_1$ and a group of $\{_2$ and $\{_3$. And in (p2) P_1 is put into a pair of $\{_5$ and $\{_6$ and P_2 is put into another pair of $\{_7$ and $\{_8$ separately.

The selection for parallelism in the earlier stage of design has a bigger influence on the structure of implementation than the one at function unit level. This is because the term transition is implemented by a sequence of function unit and the selection at function unit level is one in the inside of a term transition. Therefore, they are high and low two level selections. The high one influences the the whole structure of microprogram of computer and the low one only influences the local structure of microprogram (linear part). In the chapter 8 we will see more details about the low level.

We need a selection for two possibilities of parallel one or sequential one. This means that it is not case that a parallel is *always* better than the sequential one. For example for

$$\{_1P_1 \Rightarrow \{_2S_1\} \downarrow (P_2 \Rightarrow \{_3S_2\} \downarrow \{_4S_3\}) \tag{c1}$$

$$\{_5P_1 \Rightarrow \{_6S_1\} \downarrow \{_7P_2 \Rightarrow \{_8S_2\} \downarrow \{_9S_3\}\}\} \tag{$d1$}$$

Let us suppose that the execution of (3) above ($b_1\ y = P_1$) takes 2 time units, the one of (4) above ($b_2\ y = P_2$) takes 100 time unit and the execution of S_1 takes 3 time units and we have an application where P_1 is more often calculated to be true. Then in the case the sequential selection (d1) will lead to a faster execution.

The each sub-state transition *sst* is naturally a term transition. For the execution of the state transition *st*, traditionally, all the *sst* in the private part *pp* of a *st* are selected for parallel execution and so are all the *sst* in a common block of a *st*. However, it is also natural the *pp* and common blocks, if all they are taken as an execution unit, are selected for sequential execution because the common block will be shared by two or more state transitions. For example if a $st = pp \wedge cb_1 \wedge cb_2$ then the execution of the *st* takes three time intervals: the first is for *pp*, the second is for cb_1 and the third is for cb_2.

The brace used for the selections for parallel or sequential executions are purely syntactic and should not influence the logic meaning of specification. We have similar assumption like as one for common block.

Assumption 6.3.3. [brace non-relating to logic meaning]:

We extend specification scheme to including braces for expressing selection for parallel or sequential executions and suppose that for any part formed as $\{\alpha\}$ to have $\vdash \{\alpha\} \Leftrightarrow \alpha$. □

6.3.1 Construction 5 and Specification Scheme $4'$ and 5

For the construction 5 we only need a shorted form of the specification scheme 4 as the state transition st of the scheme 4 can be taken as a terminal.

Specification Scheme $4'$:

$E\ 0 \wedge \forall x.\ \exists y.\ E\ x \Rightarrow sts \wedge E\ y$

State transition structure sts: **Abbreviation:**

$$
\begin{aligned}
sts\ ::=\ & st \\
& \mid \text{let } A \text{ in } sts \\
& \mid P \Rightarrow sts \downarrow sts
\end{aligned}
$$

sts : state transition structure

For the construction 5 and the scheme 5 we take a convenience.

Convenience 6.3.1.1. [omitting the most outside braces]:

The most outside a pair braces "{" and "}" of the state transition structure sts are omitted. □

Only the sts of the scheme $4'$ is dealt with by the construction 5.

Construction 5:

6.3.1.1. $\Psi_5\ (st) \mathrel{\widehat{=}} st$

6.3.1.2. $\Psi_5\ (\text{let } A \text{ in } sts) \mathrel{\widehat{=}} \text{let } A \text{ in } \Psi_l\ (sts)$

6.3.1.3. $\Psi_5\ (P \Rightarrow \alpha_1 \downarrow \alpha_2) \mathrel{\widehat{=}} P \Rightarrow \Psi_p\ (\alpha_1) \downarrow \Psi_p\ (\alpha_2)$

6.3.1.4. $\Psi_l\ (st) \mathrel{\widehat{=}} \{st\}$

6.3.1.5. $\Psi_l\ (\text{let } A \text{ in } sts) \mathrel{\widehat{=}} \text{let } A \text{ in } \Psi_l\ (sts)$
$\qquad\qquad \mid \{\text{let } A \text{ in } \Psi_l\ (sts)\}$

6.3.1.6. $\Psi_l\ (P \Rightarrow \beta_1 \downarrow \beta_2) \mathrel{\widehat{=}} \{P \Rightarrow \Psi_p\ (\beta_1) \downarrow \Psi_p\ (\beta_2)\}$

6.3.1.7. $\Psi_p\ (st) \mathrel{\widehat{=}} \{st\}$

6.3.1.8. $\Psi_p\ (\text{let } A \text{ in } sts) \mathrel{\widehat{=}} \{\text{let } A \text{ in } \Psi_l\ (sts)\}$

6.3.1.9. $\Psi_p\ (P \Rightarrow \gamma_1 \downarrow \gamma_2) \mathrel{\widehat{=}} P \Rightarrow \Psi_p\ (\gamma_1) \downarrow \Psi_p\ (\gamma_2)$
$\qquad\qquad \mid \{P \Rightarrow \Psi_p\ (\gamma_1) \downarrow \Psi_p\ (\gamma_2)\}$

In the construction 5 the equations 6.3.1.1, 2 and 3 deal with the sts. The equations 6.3.1.4, 5 and 6 deal with let_in structure and through the equation 6.3.1.5 we have selection that whether or not to take a parallel execution. Similarly the equation 6.3.1.7, 8 and 9 deal with condition structure and the equation 6.3.1 is for parallel selection.

Specification Scheme 5:

$E\ 0 \wedge \forall x.\ \exists y.\ E\ x \Rightarrow sts \wedge E\ y$

State transition structure sts**:** **Abbreviation:**

sts	$::=$	st	sts	: state transition structure
		$\mid$ let A in ls		
		$\mid P \Rightarrow cs \downarrow cs$		
ls	$::=$	let A in ls	ls	: let_in structure
		$\mid \{sts\}$		
cs	$::=$	$P \Rightarrow cs \downarrow cs$	cs	: condition structure
		$\mid \{sts\}$		

The totalness and correctness of the construction 5 are obvious.

Chapter 7

Decomposing to Term Transition

Decomposition, by which a complex object is divided into its simple components, is the main means of construction. In this chapter we discuss how to decompose a specification at the scheme 5 level to a specification whose basic component is the term transition (the definition 6.3.1 in page 47)

We take term transition as a stage of the construction of computer because:

- The term transition is a middle level structure of specification of computer. A sub-state transition by itself is a term transition. The assignment $v = tm$ of a let_in structure is implemented by a term transition $s\ y = tm$. The predicate P of a condition structure is implemented by a term transition $b\ y = P$. The term transition nearly is the unique component of a specification at the term transition level, except the branch structure. These will be clearer in the specification scheme 6.

- When taking term transition as a basic component of the specification of computer, namely at this level we do not further consider the inside structure of term transition, we can have some arbitrary selections for the parallel and sequential executions to organize all the term transitions in a specification of computer. This is as the discussions of preceding chapter.

- The organization of term transitions will form a high level structure of a computer. The structure will directly correspond to an abstract microprogram of the implementation of computer, which is formed as a closed tree:
  - Its each arc corresponds to a term transition or a group of parallel term transitions and the arc is implemented through the control of a corresponding leaner microprogram.
  - Its each node entries an arc and corresponds to the entry address of a linear microprogram which the arc corresponds to.
  - Its cross node that an arc is connected to two or more arcs corresponds to a branch structure: under different conditions an arc is connected to different arcs.
  - The tree is closed as its every leaf arc will certainly go back to the root of the tree. The root is the global entry point of the whole microprogram of computer.

Let us give two examples to help reader to understand the description above and what a decomposition is. We only describe what two pictures before and after the decomposition are, and leave the decomposition process to the construction 6 where we give a generic and exact description for the construction process.

Example 7.1

The example is a piece of specification picked from the specification of Gordon's computer.

Specification 1:

(let $v = \mathrm{Fetch}_{13}\ (memory\ x)\ (pc\ x)$ in

$(\mathrm{Val}_3\ (\mathrm{Opcode}\ v) = 0) \Rightarrow state\_transition_4 \mid state\_transition_5$

) $[E_i,\ E_o]$

Here,

$state\_transition_4 \equiv (pc\ y = pc\ x\ \ \wedge idle\ y = \mathrm{T})$

$state\_transition_5 \equiv (pc\ y = \mathrm{Cut}_{13}\ v\ \ \wedge idle\ y = \mathrm{F})$

pc (program counter) and $idle$ are the state signals of the computer.

Fetch_{13}, Opcode, Val_3 and Cut_{13} are constant functions.

v is an assignment variable. T and F are boolean constants.

By decomposition we get

Specification 2:

$$(v\ y = \mathrm{Fetch}_{13}\ (memory\ x)\ (pc\ x) \wedge pc\ y = pc\ x \wedge idle\ y = idle\ x)\ [E_i,\ E_1] \quad (1)$$

$$\wedge$$

$$(\forall x.\ E_1\ x \Rightarrow (\mathrm{Val}_3\ (\mathrm{Opcode}\ (v\ x)) = 0 \Rightarrow E_2\ x \downarrow E_3\ x)) \quad (2)$$

$$\wedge$$

$$state\_transition_4\ [E_2,\ E_o] \quad (3)$$

$$\wedge$$

$$state\_transition_{5.1}\ [E_3,\ E_o] \quad (4)$$

Here,

$state\_transition_{5.1} \equiv (pc\ y = \mathrm{Cut}_{13}\ (v\ x)\ \ \wedge idle\ y = \mathrm{F})$

Thus specification 1 is decomposed into the four parts: (1), (2), (3) and (4). The decomposition follows the natural structure of the specification 1: (1) corresponds to its let_in sentence; (2) corresponds to its conditional sentence; (3) and (4) correspond to its two state transitions. Each part becomes an independent sub-specification which has the uniform form of the derivation model: (1), (3) and (4) follow the derivation model 3.1.1 and (2) follows the branch structure model 7.1 as below. If we use the definitions of models to unfold the specifications 1 and 2 then it is obvious that specification 2 satisfies specification 1, namely,

$\vdash$ Specification 2 $\Rightarrow$ Specification 1.

In the (1) above we introduce $pc\ y = pc\ x$ and $idle\ y = idle\ x$. They are used to keep both pc and $idle$ unchanged in the time intervals of (1).

Branch Structure Model 7.1:

Syntax: Branch Model $::= [E_i,\ \mathcal{E}_x]$

$\mathcal{E}_x ::= E_o \mid b\ (\mathcal{E}_x,\ \mathcal{E}_x)$

Semantics: $\mathcal{S}_m\ ([E_i,\ \mathcal{E}_x]) = \forall x.\ E_i\ x \Rightarrow \mathcal{S}_m\ (\mathcal{E}_x)$

$\mathcal{S}_m\ (E_o) = E_o\ x$

$\mathcal{S}_m\ (b\ (\mathcal{E}_{x1},\ \mathcal{E}_{x2})) = (b\ x \Rightarrow \mathcal{S}_m\ (\mathcal{E}_{x1}) \downarrow \mathcal{S}_m\ (\mathcal{E}_{x2}))$

Here, b is a boolean valued signal;

E_i is an entry predicate;

E_o is a leaving predicate.

$\mathcal{E}_x$ is called leaving predicate structure.

□

For instance, we write

$[E_2,\ b_1\ (E_3,\ b_2\ (E_4,\ E_5))]$

to express

$\forall x.\ E_2\ x \Rightarrow (b_1\ x \Rightarrow E_3\ x \downarrow (b_2\ x \Rightarrow E_4\ x \downarrow E_5\ x))$

For easier to write a branch structure we can use the convenient form that we can write $b\ x$ to replace b, even write a formula to replace b if the formula is considered to be executable at a given impelmentation level.

Example 7.2

The example is a simple and abstract but somewhat complete one to give us a general picture about what is after decomposition.

Specification 1:

$\forall x.\ \exists y.\ E\ x \Rightarrow sts \wedge E\ y$

$sts \equiv \{$ let $v_1 = tm_1$ in

let $v_2 = tm_2$ in

$\{\ P_1 \Rightarrow \{\ pp_1 \wedge [cb]\ \} \downarrow$

$\{\ P_2 \Rightarrow \{\ pp_2 \wedge [cb]\ \} \downarrow \{\ pp_3\ \}\ \}$

$\}$

}

In the specification the two assignments $v_1 = tm_1$ and $v_2 = tm_2$ of the let_in structure has been selected for parallel execution and so do the two predicates P_1 and P_2 of the condition structure. There are three state transitions: $pp_1 \wedge [cb]$, $pp_2 \wedge [cb]$ and pp_3. $[cb]$ marks that the cb is selected as a common block.

Using the derivation model we rewrite the specification to

Specification $1'$:

$$
\begin{array}{l}
\{ \text{ let } v_1 = tm_1 \text{ in} \\
\quad \text{let } v_2 = tm_2 \text{ in} \\
\qquad \{ \ P_1 \Rightarrow \{ \ pp_1 \wedge [cb] \ \} \downarrow \\
\qquad\qquad\qquad \{ \ P_2 \Rightarrow \{ \ pp_2 \wedge [cb] \ \} \downarrow \{ \ pp_3 \ \} \ \} \\
\qquad \} \\
\} \ [E, \ E]
\end{array}
$$

By decomposition from the specification $1'$ we get:

Specification 2:

$$(\ s_1 \ y = tm_1 \wedge s_2 \ y = tm_2 \wedge K_1 \ ) \ [E, \ E_1] \tag{1}$$

$\wedge$

$$(\ b_1 \ y = P_1 \wedge b_2 \ y = P_2 \wedge K_2 \ ) \ [E_1 \ E_2] \tag{2}$$

$\wedge$

$$[E_2, \ b_1 \ (E_3, \ b_2 \ (E_4, \ E_5))] \tag{3}$$

$\wedge$

$$(pp_1 \ [s_1 \ x/v_1, \ s_2 \ x/v_2] \wedge K_3) \ [E_3, \ E_6] \tag{4}$$

$\wedge$

$$(pp_2 \ [s_1 \ x/v_1, \ s_2 \ x/v_2] \wedge K_4) \ [E_4, \ E_6] \tag{5}$$

$\wedge$

$$(cb \ [s_1 \ x/v_1, \ s_2 \ x/v_2] \wedge K_5) \ [E_6, \ E] \tag{6}$$

$\wedge$

$$(pp_3\ [s_1\ x/v_1,\ s_2\ x/v_2])\ [E_5,\ E] \tag{7}$$

Here,

$K_1 = P_1^i \wedge P_2^i \wedge pp_1^i \wedge pp_2^i \wedge pp_3^i \wedge cb^i$

$K_2 = pp_1^i \wedge pp_2^i \wedge pp_3^i \wedge cb^i$

$K_3 = K_4 = cb^i$

$K_5 = pp_1^o \wedge pp_2^o$

Here and elsewhere, we use the abbreviation S^i and S^o:

Definition 7.1. [abbreviation for input and output signal protection]:

$S^i \mathrel{\widehat{=}} (r_1\ y = r_1\ x) \wedge \cdots \wedge (r_n\ y = r_n\ x)$

Here, $r_i\ \ (1 \leq i \leq n)$ are all the input signals in S.

Similarly,

$S^o \mathrel{\widehat{=}} (s_1\ y = s_1\ x) \wedge \cdots \wedge (s_m\ y = s_m\ x)$

Here, $s_i\ (1 \leq i \leq m)$ are all the output signals in S. We also take the boolean constant T as a special S^i or S^o. □

And following the branch model 7.1

$[E_2,\ b_1\ (E_3,\ b_2\ (E_4,\ E_5))]$

$= \forall x.\ E_2\ x \Rightarrow (b_1\ x \Rightarrow E_3\ x \downarrow (b_2\ x \Rightarrow E_4\ x \downarrow E_5\ x))$

In the specification 2

1. (1) is a process to calculate tm_1 and tm_2 parallel and then assign the results to the signals s_1 and s_2 respectively at some time point y.
   Meanwhile in the time interval of calculation all the input signals in the $P_1,\ P_2,\ pp_1,\ pp_2,\ pp_3$ and cb are kept unchanged.

2. (2) is a process to calculate P_1 and P_2 in parallel manner and then assign the boolean values to the signals b_1 and b_2 respectively at some time point y.
   Meanwhile in the time interval of calculation all the input signals in the $pp_1,\ pp_2,\ pp_3$ and cb are kept unchanged.

3. (3) is a branch structure which starts from E_2 at a time point x if b_1 at x is true then makes E_3 at x be true, otherwise if b_2 at x is true then makes E_4 at x be true, otherwise makes E_5 at x be true.

4. (4) is a process to calculate $pp_1\ [s_1\ x/v_1,\ s_2\ x/v_2]$ (for short we write it pp_1' in the figure 7.1.) The v_1 in pp_1 has been replaced by $s_1\ x$ through which the result of the calculation of tm_1 is passed to, and similarly for the v_2. Meanwhile in the time interval of the calculation all the input signals in the cb are kept unchanged.

5. (5) is similar to (4).

6. (6) is to calculate $cb\ [s_1\ x/v_1,\ s_2\ x/v_2]$.
   Meanwhile in the time interval of calculation all the output signals in the pp_1 and pp_2 are kept unchanged.

7. (7) is a process to calculate $pp_3\ [s_1\ x/v_1,\ s_2\ x/v_2]$.

The above is as the figure 7.1.

From the two examples we can imagine that the decomposition is a process to introduce new entry predicates, for example the $E_i\ \ (1 \leq i \leq 6)$ in the specification 2 of the example 2, to divide a specification to a set of term transitions along the natural structure of the specification: let_in structure, condition structure and state transition.

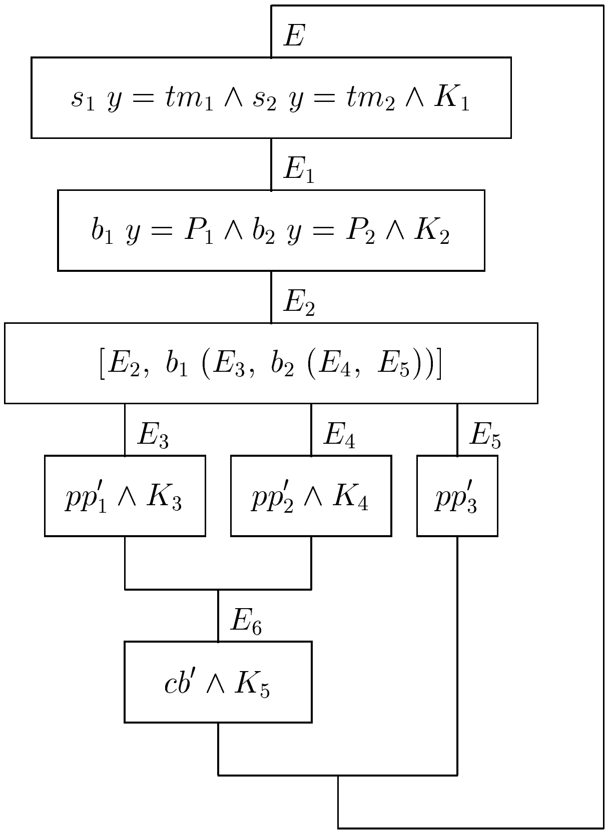


Figure 7.1 Flowchart of specification 2

The flowchart shows us a high level structure of the computer whose specification is the specification 1 of the example 2. Each frame-box corresponds to a group of parallel term transitions.

7.1 Construction 6, Abstract Algorithm 6 and Specification Scheme $5'$ and 6

The construction will deal with the decomposition to term transition level. We take the description of state transition *st* of the specification scheme 4 to the specification scheme 5 to get the specification scheme $5'$ which shows not only the parallel selections for let_in structure and condition structure but also the selection for the common block of state transition.

Specification Scheme $5'$:

$E\ 0 \wedge (sts)\ [E,\ E]$

State transition structure sts: **Abbreviation:**

$$
\begin{array}{lcl}
sts & ::= & st \\
 & & \mid \text{let } A \text{ in } ls \\
 & & \mid P \Rightarrow cs \downarrow cs \\
ls & ::= & \text{let } A \text{ in } ls \\
 & & \mid \{sts\} \\
cs & ::= & P \Rightarrow cs \downarrow cs \\
 & & \mid \{sts\} \\
st & ::= & pp \wedge cp \\
pp & ::= & \mathrm{T} \\
 & & \mid ssts \\
cp & ::= & \mathrm{T} \\
 & & \mid [ssts] \wedge cp
\end{array}
$$

sts : state transition structure

ls : let_in structure

cs : condition structure

st : state transition

pp : private part

cp : common part

The construction 6 manipulates the scheme $5'$ to produce the specification scheme 6. It deals with $(sts)\ [E,\ E]$, however, in the construction Ψ_6, instead of $[E,\ E]$, we write $[E_i,\ E_o]$ because the E_i and E_o can be used to represent any entry predicate. This is also because that the Ψ_6 can be recursively used to any level sub-structure of a sts.

Construction 6:

7.1.1. $\Psi_6\ (st\ [E_i,\ E_o]) \mathrel{\widehat{=}} \Psi_{6s}\ (((\mathrm{T}) \wedge st)\ [E_i,\ E_o])$

7.1.2. $\Psi_6\ ((\text{let } A \text{ in } \alpha)\ [E_i,\ E_o]) \mathrel{\widehat{=}} (\Psi_{6l1}\ (\text{let } A \text{ in } \alpha))\ [E_i,\ E_m] \wedge$
$\Psi_6\ ((\Psi_{6l2}\ (\text{let } A \text{ in } \alpha,\ [\ ]))\ [E_m,\ E_o])$

7.1.3. $\Psi_6\ ((P \Rightarrow \alpha_1 \downarrow \alpha_2)\ [E_i,\ E_o]) \mathrel{\widehat{=}}\ (\Psi_{6p1}\ (P \Rightarrow \alpha_1 \downarrow \alpha_2))\ [E_i,\ E_m] \wedge$
$[E_m,\ \Psi_{6p2}\ (P \Rightarrow \alpha_1 \downarrow \alpha_2)] \wedge$
$\Psi_{6p3}\ (P \Rightarrow \alpha_1 \downarrow \alpha_2,\ E_o)$

7.1.4. $\Psi_{6s}\ (((K) \wedge \alpha \wedge [\beta])\ [E_i,\ E_o]) \mathrel{\widehat{=}} \Psi_{6s}\ (((\beta^i \wedge K) \wedge \alpha)\ [E_i,\ E_m]) \wedge$
$((\alpha^o \wedge K) \wedge \beta)\ [E_m,\ E_o]$

Here, $E_m = E_{(\beta, E_o)}$.

7.1.5. $\Psi_{6s}\ (((K) \wedge \gamma)\ [E_i,\ E_o]) \mathrel{\widehat{=}} (K \wedge \gamma')\ [E_i,\ E_o]$

Here, if $\gamma = [\alpha]$ then $\gamma' = \alpha$ otherwise $\gamma' = \gamma$

7.1.6. Ψ_{6l1} (let $v = tm$ in α) $\mathrel{\widehat{=}} s_v\ y = tm \wedge \Psi_{6l1}(\alpha)$

7.1.7. $\Psi_{6l1}\ (\{\beta\}) \mathrel{\widehat{=}} \beta^i$

7.1.8. Ψ_{6l2} (let $v = tm$ in $\alpha,\ L$) $\mathrel{\widehat{=}} \Psi_{6l2}\ (\alpha,\ [s_v\ x/v] :: L)$

7.1.9. $\Psi_{6l2}\ (\{\beta\},\ L) \mathrel{\widehat{=}} \beta\ L$

7.1.10. $\Psi_{6p1}\ (P \Rightarrow \alpha_1 \downarrow \alpha_2) \mathrel{\widehat{=}} b_p\ y = P \wedge \Psi_{6p1}\ (\alpha_1) \wedge \Psi_{6p1}\ (\alpha_2)$

7.1.11. $\Psi_{6p1}\ (\{\beta\}) \mathrel{\widehat{=}} \beta^i$

7.1.12. $\Psi_{6p2}\ (P \Rightarrow \alpha_1 \downarrow \alpha_2) \mathrel{\widehat{=}} b_p\ (\Psi_{6p2}\ (\alpha_1),\ \Psi_{6p2}\ (\alpha_2))$

7.1.13. $\Psi_{6p2}\ (\{\beta\}) \mathrel{\widehat{=}} E_\beta$

7.1.14. $\Psi_{6p3}\ (P \Rightarrow \alpha_1 \downarrow \alpha_2,\ E_o) \mathrel{\widehat{=}} \Psi_{6p3}\ (\alpha_1,\ E_o) \wedge \Psi_{6p3}\ (\alpha_2,\ E_o)$

7.1.15. $\Psi_{6p3}\ (\{\beta\},\ E_o) \mathrel{\widehat{=}} \Psi_6\ ((\beta)\ [E_\beta,\ E_o])$

Let us explain the construction 6.

The equation 7.1.1, 2 and 3 distinguish the three cases: state transition "*st*", let_in structure "let A in α" and condition structure "$P \Rightarrow \alpha_1 \downarrow \alpha_2$", and pass them to the constructions Ψ_{6s}, Ψ_{6li} $(1 \leq i \leq 2)$ and Ψ_{6pi} $(1 \leq i \leq 3)$ respectively to deal with further.

Construction of State Transition:

The construction Ψ_{6s} deals with state transition which has form:

$$pp \wedge [cb_1] \wedge [cb_2] \wedge \cdots \wedge [cb_n] \tag{11}$$

Here, pp is the private part and others are the sorted common blocks.

However, the (l1) will be treated along direction from right to left, namely along the reverse of the sequence (l1):

$$[cb_n],\ [cb_{n-1}], \cdots, [cb_1], pp$$

(This is for correctly introducing new entry predicate to get a tree-shape implementation (see page 42))
Therefore in the construction equation 7.1.4 when we write

$$\alpha \wedge [\beta]$$

where,

$[\beta]$ corresponds to the $[cb_n]$ of (l1),

α corresponds to the $pp \wedge [cb_1] \wedge [cb_2] \wedge \cdots \wedge [cb_{n-1}]$ of (l1).

(The match follows logic programming Prolog: we use the bracket "[" and "]" to identify the last common block)

In the Ψ_{6s} we also introduce variable K to accommodate all signal protections (the definition 7.1 in page 58). To distinguish the K from the state transition $\alpha \wedge [\beta]$ of the construction equation 7.1.4 and from the sub-state transition γ of the construction equation 7.1.5 we bracketed it as (K). The K is initiated by the boolean constant T in the construction equation 7.1.1

In the construction equation 7.1.4 we introduce a new entry predicate E_m whose detailed form is $E_{(\beta,E_o)}$ where the subscript (β, E_o) means that $E_{(\beta,E_o)}$ is determined uniquely following β and E_o. Through the E_m we break down $\alpha \wedge [\beta]$ to two parts α and $[\beta]$. This is that the original process (derivation model)

$$((K) \wedge \alpha \wedge \beta)\ [E_i,\ E_o]$$

is divided to two new processes:

$$((\beta^i \wedge K) \wedge \alpha)\ [E_i,\ E_m] \tag{l2}$$

$$((\alpha^o \wedge K) \wedge \beta)\ [E_m,\ E_o] \tag{l3}$$

In the (l3) by α^o all output signals in α are protected and in the (l2) by β^i all input signals in β are protected. All these protections are added to K.

The construction equation 7.1.5 deals with the most left sub-state transition γ. In this case the brackets of K is taken off as where they have become unnecessary.

In the construction equation 7.1.4 and 5 all the brackets "[" and "]" of common blocks are also taken off as their tasks have completed.

Example 7.1.1:

$\Psi_{6s}\ ((pp \wedge [cb1] \wedge [cb2])\ [E_i,\ E_o]) =$

$$\Psi_{6s}\ (((cb_2^i) \wedge pp \wedge [cb_1])\ [E_i,\ E_b]) \qquad \text{(e1)}$$

$\{$ Here, $E_b \equiv E_{(cb_2, E_o)}$ $\}$

$\wedge$

$$(((pp \wedge cb_1)^o) \wedge cb_2)\ [E_b,\ E_o] \qquad \text{(e2)}$$

– By the construction equation 7.1.4

(e1) =

$$\Psi_{6s}\ (((cb_1^i \wedge cb_2^i) \wedge pp)\ [E_i,\ E_a]) \qquad \text{(e3)}$$

$\{$ Here, $E_a \equiv E_{(cb_1, E_b)}$ $\}$

$\wedge$

$$((pp^o \wedge cb_2^i) \wedge cb_1)\ [E_a,\ E_b] \qquad \text{(e4)}$$

– By the construction equation 7.1.4

(e3) =

$$((cb_1^i \wedge cb_2^i) \wedge pp)\ [E_i,\ E_a] \qquad \text{(e5)}$$

– By the construction equation 7.1.5

Therefore we get

$\Psi_{6s}\ ((pp \wedge [cb1] \wedge [cb2])\ [E_i,\ E_o]) =$

$$((cb_1^i \wedge cb_2^i) \wedge pp)\ [E_i,\ E_a] \qquad \text{(e5)}$$

$\wedge$

$$((pp^o \wedge cb_2^i) \wedge cb_1)\ [E_a,\ E_b] \qquad \text{(e4)}$$

$\wedge$

$$(((pp \wedge cb_1)^o) \wedge cb_2)\ [E_b,\ E_o] \tag{e2}$$

We have

$(pp \wedge cb_1)^o = pp^o \wedge cb_1^o$

Finally, we can sort up the result above to get:

$\Psi_{6s}\ ((pp \wedge [cb1] \wedge [cb2])\ [E_i,\ E_o]) =$

$$(pp \wedge cb_1^i \wedge cb_2^i)\ [E_i,\ E_a] \tag{e5$'$}$$

$$\wedge$$

$$(pp^o \wedge cb_1 \wedge cb_2^i)\ [E_a,\ E_b] \tag{e4$'$}$$

$$\wedge$$

$$(pp^o \wedge cb_1^o \wedge cb_2)\ [E_b,\ E_o] \tag{e2$'$}$$

□

A picture below can help reader to understand our result.

$$\begin{array}{ccccccc}
E_i & \longrightarrow & E_a & \longrightarrow & E_b & \longrightarrow & E_o \\
 & pp & & pp^o & & pp^o & \\
 & cb_1^i & & cb_1 & & cb_1^o & \\
 & cb_2^i & & cb_2^i & & cb_2 &
\end{array}$$

From the picture we can see that the original model having logic boundary $[E_i,\ E_o]$ is divided to three pieces by introducing two middle entry predicates E_a and E_b. In the model having logic boundary $[E_i,\ E_a]$, pp is implemented and meanwhile the input signals of cb_1 and cb_2 are protected. In the model having logic boundary $[E_a,\ E_b]$, cb_1 is implemented and meanwhile the output signals of pp and the input signals of cb_2 are protected. In the model having logic boundary $[E_b,\ E_o]$, cb_2 is implemented and meanwhile the output signals of pp and cb_1 are protected.

Construction of Let_In Structure:

To deal with a let_in structure the construction 6 uses two constructions Ψ_{6l1} and Ψ_{6l2} and each one of the two constructions takes the same copy of the let_in structure but for different proposals. The Ψ_{6l1} is used to produce a group of term transitions for the assignments selected for parallel execution. The Ψ_{6l2} deals with the sub-structure of let_in structure. The sub-structure is bracketed by two braces "{" and "}". Moreover, the Ψ_{6l2} replaces each variable v in the sub-structure by a corresponding signal s_v applied to a time point x. (v as variable is used for description, however, s_v as signal in $s_v\ x$ is used for implementation). Two constructions Ψ_{6l1} and Ψ_{6l2} are put into two successive time intervals and this is expressed through the $[E_i,\ E_m]$ and $[E_m,\ E_o]$, namely a new entry predicate E_m is introduced to divide the $[E_i,\ E_o]$ of the let_in structure.

In the construction equation 7.1.6 each assignment formed as

$$v = tm$$

in a group of assignments selected for parallel execution will lead to a term transition

$$s_v\ y = tm$$

and all these term transitions are conjuncted in construction equation 7.1.6 and then is put into the same derivation model (process) as shown in the construction equation 7.1.2.

In the construction equation 7.1.7 when a bracketed sub-structure $\{\beta\}$ is met, it means that the treatment for parallel assignment of a let_in structure have been completed. In this case an additional work is to protect all the input signals in β.

In the construction equation 7.1.8 for the variable v of each assignment "$v = tm$" we constructs a substitution "$s_v\ x/v$" and append the substitution to a list L which is initiated by a empty list [ ] in the construction equation 7.1.2. We use operation "::" for append operation [12]

Let us use examples to help reader:

$$[a,\ b,\ c]\ ::\ [d,\ e] = [a,\ b,\ c,\ d,\ e]$$

$$[a]\ ::\ [b,\ c] = [a,\ b,\ c]$$

$$[\,]\ ::\ [a,\ b] = [a,\ b]$$

$[a,\ b]\ ::\ [\,] = [a,\ b]$

In the construction equation 7.1.9 the substitution list L is used to the substructure β and we write $\beta\, L$ to express that each variable formed as v in β has been replaced by a corresponding signal value $s_v\ x$. The brackets of $\{\beta\}$ are taken off as their task have completed.

Example 7.1.2:

$\Psi_6\ ((\text{let } v_1 = tm_1 \text{ in let } v_2 = tm_2 \text{ in } \{S\})\ [E_i,\ E_o]) =$

$$(\Psi_{6l1}\ (\text{let } v_1 = tm_1 \text{ in let } v_2 = tm_2 \text{ in } \{S\}))\ [E_i,\ E_m] \tag{11}$$

$$\wedge$$

$$\Psi_6\ ((\Psi_{6l2}\ (\text{let } v_1 = tm_1 \text{ in let } v_2 = tm_2 \text{ in } \{S\},\ [\,]))\ [E_m,\ E_o]) \tag{12}$$

– By the construction equation 7.1.2

$\Psi_{6l1}\ (\text{let } v_1 = tm_1 \text{ in let } v_2 = tm_2 \text{ in } \{S\}) =$

$$s_1\ y = tm_1 \tag{13}$$

$$\wedge$$

$$\Psi_{6l1}\ (\text{let } v_2 = tm_2 \text{ in } \{S\}) \tag{14}$$

– By the construction equation 7.1.6

$(14) =$

$$s_2\ y = tm_2 \tag{15}$$

$$\wedge$$

$$\Psi_{6l1}\ (\{S\}) \tag{16}$$

– By the construction equation 7.1.6

$(16) =$

$$S^i \tag{17}$$

– By the construction equation 7.1.7

Therefore we get

$(11) =$

$$(s_1\ y = tm_1 \wedge s_2\ y = tm_2 \wedge S^i)\ [E_i,\ E_m] \tag{18}$$

$\Psi_{6l2}\ (\text{let } v_1 = tm_1 \text{ in let } v_2 = tm_2 \text{ in } \{S\},\ [\,])$

$= \Psi_{6l2}$ (let $v_2 = tm_2$ $in\{S\}$, $[s_1\ x/v_1]$) – By the construction equation 7.1.8

$= \Psi_{6l2}$ $(\{S\}\ [s_2\ x/v_2,\ s_1\ x/v_1])$ – By the construction equation 7.1.8

$= S\ [s_2\ x/v_2,\ s_1\ x/v_1]$ – By the construction equation 7.1.9

Therefore,

$(l2) = \Psi_6\ ((S\ [s_2\ x/v_2,\ s_1\ x/v_1])\ [E_m,\ E_o])$

Finally, we get

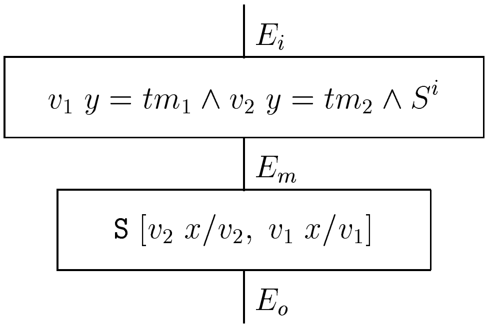
Ψ_6 (let $v_1 = tm_1$ in let $v_2 = tm_2$ in $\{S\}$) $[E_i,\ E_o]) =$

$(v_1\ y = tm_1 \wedge v_2\ y = tm_2 \wedge S^i)\ [E_i,\ E_m]$

$\wedge$

$\Psi_6\ ((S\ [s_2\ x/v_2,\ s_1\ x/v_1])\ [E_m,\ E_o])$ $\square$

A picture below can help reader to understand our result.



Note, in the second frame box of the picture above we write `S` to mark that the box represents

$\Psi_6\ ((S\ [s_2\ x/v_2,\ s_1\ x/v_1])\ [E_m,\ E_o])$.

It will be similar in the example 7.1.3 for the construction of condition structure.

Construction of Condition Structure:

To deal with a condition structure the construction 6 uses the three constructions Ψ_{6p1}, Ψ_{6p2} and Ψ_{6p3}. Each one of the three constructions takes the same copy of the condition structure but does different thing. The Ψ_{6p1} produces the term transitions for all the predicates which are selected for the parallel execution. The Ψ_{6p2} produces a branch structure. And the Ψ_{6p3} introduces the new derivation models for every sub-structure which is bracketed and passes them to the construction Ψ_6. Differing from the constructions Ψ_{6l1} and Ψ_{6l2} for let_in structure where only a new entry predicate E_m is introduced to divide the let_in structure, the construction for condition structure introduces a branch structure to divide the condition structure and it produces a group of term transition and several (two or two more) sub-structures to be further constructed.

In the construction equation 7.1.10 each predicate P in the group of predicates selected for parallel execution leads to constructing a corresponding term transition

$$b_p\ y = P$$

Here, the subscript p of b_p represent the b_p is uniquely corresponding to the predicate P. And all these term transitions are conjuncted in the construction equation 7.1.10 and then are put into the same model of $[E_i,\ E_m]$ in the construction equation 7.1.3.

In the construction equation 7.1.11 when a bracketed sub-structure $\{\beta\}$ is met, it means that the treatment for the parallel predicates has been completed so we can take off the brackets of the sub-structure and write β^i to express that all the input signals in the β are protected.

In the construction equation 7.1.12 for a condition structure we construct a branch structure by taking the boolean valued signal formed as b_p constructed in the construction equation 7.1.10. The construction is recursive along the structure of the condition structure until meeting a bracketed sub-structure $\{\beta\}$.

The sub-structure is dealt with by the construction equation 7.1.13 which introduces a new entry predicate E_β. The subscript β marks that the E_β is unique for the sub-structure β. The E_β is the leaving predicate of the branch structure and is also the entry predicate for the derivation model (process) of β.

In the construction equation 7.1.14 and 15 Ψ_{6p3} recursively passes the predicates of the condition structure until meeting a bracketed sub-structure $\{\beta\}$ where it introduce a new model (β) $[E_\beta,\ E_o]$ and then the new model is passed to Ψ_6 to deal with.

Example 7.1.3.

$$\Psi_6\ (P \Rightarrow \{S_1\} \downarrow (Q \Rightarrow \{S_2\} \downarrow \{S_3\})\ [E_i,\ E_o]) =$$

$$(\Psi_{6p1}\ (P \Rightarrow \{S_1\} \downarrow (Q \Rightarrow \{S_2\} \downarrow \{S_3\})))\ [E_i,\ E_m] \quad \text{(p1)}$$

$$\wedge$$

$$[\ E_m,\ \Psi_{6p2}\ (P \Rightarrow \{S_1\} \downarrow (Q \Rightarrow \{S_2\} \downarrow \{S_3\}))\ ] \quad \text{(p2)}$$

$$\wedge$$

$$\Psi_{6p3}\ (P \Rightarrow \{S_1\} \downarrow (Q \Rightarrow \{S_2\} \downarrow \{S_3\}),\ E_o) \quad \text{(p3)}$$

– By the construction equation 7.1.3

$$\Psi_{6p1}\ (P \Rightarrow \{S_1\} \downarrow (Q \Rightarrow \{S_2\} \downarrow \{S_3\}))$$
$$= (b_p\ y = P) \wedge \Psi_{6p1}\ (\{S_1\}) \wedge \Psi_{6p1}\ (Q \Rightarrow \{S_2\} \downarrow \{S_3\})$$

– By the construction equation 7.1.10

$$= (b_p\ y = P) \wedge S_1^i \wedge (b_q\ y = Q) \wedge \Psi_{6p1}\ (\{S_2\}) \wedge \Psi_{6p1}\ (\{S_3\})$$

– By the construction equation 7.1.11 and 10

$$= (b_p\ y = P) \wedge S_1^i \wedge (b_q\ y = Q) \wedge S_2^i \wedge S_3^i$$

– By the construction equation 7.1.11

$$= (b_p\ y = P) \wedge (b_q\ y = Q) \wedge S_1^i \wedge S_2^i \wedge S_3^i$$

Therefore,

(p1) $= (b_p\ y = P \wedge b_q\ y = Q \wedge S_1^i \wedge S_2^i \wedge S_3^i)\ [E_i,\ E_m]$.

$$\Psi_{6p2}\ (P \Rightarrow \{S_1\} \downarrow (Q \Rightarrow \{S_2\} \downarrow \{S_3\}))$$
$$= b_p\ (\Psi_{6p2}\ (\{S_1\}), \Psi_{6p2}\ (Q \Rightarrow \{S_2\} \downarrow \{S_3\}))$$

– By the construction equation 7.1.12

$$= b_p\ (E_{S_1}, b_q\ (\Psi_{6p2}\ (\{S_2\}),\ \Psi_{6p2}\ (\{S_3\})))$$

– By the construction equation 7.1.13 and 12

$$= b_p\ (E_{S_1}, b_q\ (E_{S_2},\ E_{S_3}))$$

– By the construction equation 7.1.13

Therefore, we have

(p2) $= [\ E_m,\ b_p\ (E_{S_1}, b_q\ (E_{S_2},\ E_{S_3}))\ ]$.

(p3)

$= \Psi_{6p3}\ (\{S_1\},\ E_o) \wedge \Psi_{6p3}\ (Q \Rightarrow \{S_2\} \downarrow \{S_3\},\ E_o)$

– By the construction equation 7.1.14

$= \Psi_6\ ((S_1)\ [E_{S_1},\ E_o]) \wedge \Psi_{6p3}\ (\{S_2\},\ E_o) \wedge \Psi_{6p3}\ (\{S_3\},\ E_o)$

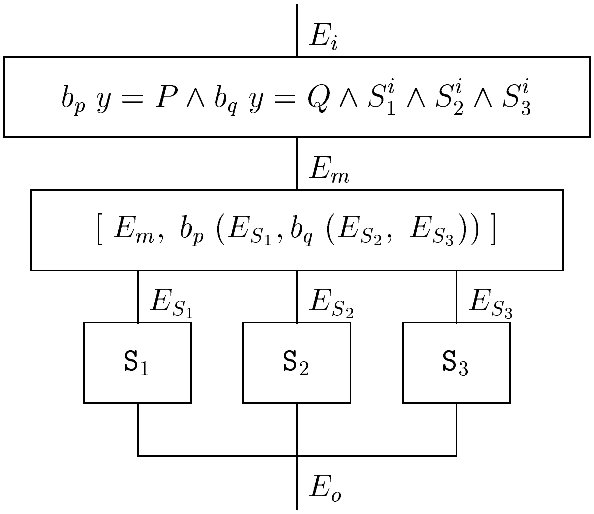
– By the construction equation 7.1.15 and 14

$= \Psi_6\ ((S_1)\ [E_{S_1},\ E_o]) \wedge \Psi_6\ ((S_2)\ [E_{S_2},\ E_o]) \wedge \Psi_6\ ((S_3)\ [E_{S_3},\ E_o])$

– By the construction equation 7.1.15

□

A picture below can help reader to understand our result.



In the example 7.2 (page 58) we can see that the output signal protection K_5 of the common block "*cb*" includes both "pp_1^o" and "pp_2^o" as the state transition "$pp_1 \wedge [cb]$" and "$pp_2 \wedge [cb]$" share the "*cb*". Generally speaking, a common block should protect all the output signals which comes from the state transitions sharing the common block (of course, these signals are ones in the private part or other common blocks which are at the left of the common block). However, the construction 6 only can separately construct the protection for each state transition, therefore after the construction 6 we need an abstract algorithm to combinate all the output signal protections of a common block.

Abstract Algorithm 7.1.1. [$\mathcal{A}_6$: merging signal protection in common block]:

For the specification scheme which is produced by the construction 6 all the output signal protection of each common block are put together (through conjunction) as the output signal protection of the common block. □

For example, from $K_1^o \wedge [cb]$ and $K_2^o \wedge [cb]$ by the algorithm to get $K_1^o \wedge K_2^o \wedge [cb]$.

Specification Scheme 6:

$E\ 0 \wedge ttms$

Term transition model set $ttms$**:**

$$
\begin{array}{lcl}
ttms & ::= & ttm \\
 & & \mid ttm \wedge ttms \\
ttm & ::= & (tts)\ [E_i,\ E_o] \\
 & & \mid brs \\
tts & ::= & tt \\
 & & \mid tt \wedge tts \\
tt & ::= & s\ y = tm \\
brs & ::= & [E_i,\ E_x] \\
E_x & ::= & E_o \\
 & & \mid bs\ (E_x,\ E_x)
\end{array}
$$

Abbreviation:

$ttms$	: term transition model set
ttm	: term transition model
tts	: term transition set
tt	: term transition
brs	: branch structure model
E_x	: leaving predicate structure
E_i	: entry predicate
E_o	: leaving predicate
tm	: term
s	: signal
bs	: boolean valued signal

The specification scheme 6 includes two kinds of models: the term transition model and branch structure model. Note, signal protection such as $s\ y = s\ x$ is included in the term transition tt.

Theorem 7.1.1. [logic boundary of term transition model]:

In the scheme 6, there are not two term transition models which have the same entry predicate. Moreover, only the common block can lead its corresponding term transition models having the same leaving predicate.

Proof:

By the construction 6 a new entry predicates is always introduced to divide a state transition structure sts of the scheme $5'$, therefore, the new one always becomes a middle one between the given entry and leaving predicates of the sts. Moreover, the new introduced predicate is always taken as the leaving predicate of a model and meanwhile as the entry predicate of another model.

The term transition models introduced by decomposing the let_in structure ls and condition structure cs always have different entry and different leaving predicates.

The term transition models introduced by decomposing the state transition st, by the construction 6 and the abstract algorithm $\mathcal{A}_6$, can be the same, therefore they are viewed and treated as the same model, or having the different entry predicate and the same leaving predicate or having both entry and leaving predicates different.

Therefore in any case we have term transition models of the scheme 6 as the theorem described. □

Corollary 7.1.1. [uniqueness of entry time point and time interval]:

In the scheme 6 there are not two term transition models having the same entry time point, therefore, any two term transition models always are executed in the two different time intervals.

Proof:

The time point of a model is decided by the corresponding entry predicate of the model. And further by the lemma 7.1.1 at any time point there is only one entry predicate which can be made to be true. □

7.2 Totalness and Correctness of Construction 6

Theorem 7.2.1. [totalness of construction 6]:

$\forall s : \mathcal{S}_{5'} .\ \mathcal{A}_6\ (\Psi_6\ (s)) \in \mathcal{S}_6$

Proof: Use structural induction on the structure of the state transition structure *sts* of the specification scheme $5'$ and aware that the base case of *sts* is state transition *st* and step case includes the let_in structure and the condition structure. In all cases the term transition models are produced by the construction 6 and the abstract algorithm 6, but the only exception is the branch structure models which is produced in the construction of the condition structure. □

Lemma 7.2.1. [correctness of Ψ_{6s} for st]:

$\forall st : \mathcal{S}_{5'} .\ \Psi_{6s}\ (((K) \wedge st)\ [E_i,\ E_o]) \Rightarrow (K \wedge st)\ [E_i,\ E_o]$

Proof:

Let us use induction on the number of sub-state transition set *ssts* of the *st*. By the scheme $5'$ we know that a *ssts* can be a private part *pp* or a common block.

Base case: There is only a sub-state transition set γ.

In this case by the construction equation 7.1.5 we have

$$\Psi_{6s}\ (((K) \wedge \gamma)\ [E_i,\ E_o]) = (K \wedge \gamma')\ [E_i,\ E_o]$$

where, if $\gamma = [\alpha]$ then $\gamma' = \alpha$ otherwise $\gamma' = \gamma$

Namely, the brackets are taken off as the brackets are only used to mark for common block.

Therefore we can get

$$\Psi_{6s}\ (((K) \wedge \gamma)\ [E_i,\ E_o]) \Rightarrow (K \wedge \gamma)\ [E_i,\ E_o]$$

Step case: Suppose $st = \alpha \wedge [\beta]$

where, α contains one or more conjuncted *ssts* and β is a *ssts*. By the construction equation 7.1.4 we have

$$\Psi_{6s}\left(\left(\left(K\right)\wedge\alpha\wedge[\beta]\right)[E_i,\ E_o]\right)\mathrel{\widehat{=}}\Psi_{6s}\left(\left(\left(\beta^i\wedge K\right)\wedge\alpha\right)[E_i,\ E_m]\right)$$
$$\wedge$$
$$\left(\left(\alpha^o\wedge K\right)\wedge\beta\right)[E_m,\ E_o]$$

By induction hypothesis we have

$$\Psi_{6s}\left(\left(\left(\beta^i\wedge K\right)\wedge\alpha\right)[E_i,\ E_m]\right)\Rightarrow\left(\left(\beta^i\wedge K\right)\wedge\alpha\right)[E_i,\ E_m]$$

Hence our proof is reduced to

$$\left(\left(\beta^i\wedge K\right)\wedge\alpha\right)[E_i,\ E_m]$$
$$\wedge$$
$$\left(\left(\alpha^o\wedge K\right)\wedge\beta\right)[E_m,\ E_o]$$
$$\Rightarrow$$
$$\left(K\wedge\alpha\wedge[\beta]\right)[E_i,\ E_o]$$

This is obviously tenable. □

The specification for let_in structure in the scheme $5'$ is described as

$$sts ::= \text{let } A \text{ in } ls \qquad (l1)$$
$$ls ::= \{sts\} \qquad (l2)$$
$$\mid \text{let } A \text{ in } ls \qquad (l3)$$

By structural induction to prove the correctness of the construction for the let_in structure of (l1) we can take the hypothesis that the constructions for the (l2) and (l3) are correct. However, in the proof of the lemma below we need not to distinguish the two case but only consider them as ls.

Lemma 7.2.2. [correctness of construction for let_in structure]:

$$\Psi_6\left((\text{let } v = tm \text{ in } ls)\,[E_i,\ E_o]\right)\Rightarrow(\text{let } v = tm \text{ in } ls)\,[E_i,\ E_o]$$

if

$$\Psi_6\left(ls\,[E_x,\ E_y]\right)\Rightarrow ls\,[E_x,\ E_y]$$

Note: all entry predicates in the lemma can be arbitrarily given because the structural induction merely concerns the structure of sts. For example, here we use $[E_x, E_y]$.

Proof:

$$\Psi_6\ ((\text{let } v = tm \text{ in } ls)\ [E_i,\ E_o])$$
$$= \Psi_6\ ((ls\ [tm/v])\ [E_i,\ E_o])$$
$$\Rightarrow (ls\ [tm/v])\ [E_i,\ E_o] \qquad \text{– By the hypothesis of the lemma}$$
$$= (\text{let } v = tm \text{ in } ls)\ [E_i,\ E_o] \qquad \text{– By the definition of let\_in}$$

Therefore we get

$$\Psi_6\ ((\text{let } v = tm \text{ in } ls)\ [E_i,\ E_o]) \Rightarrow (\text{let } v = tm \text{ in } ls)\ [E_i,\ E_o] \qquad \square$$

The specification for condition structure in the scheme $5'$ is described as

$$sts ::= P \Rightarrow cs \downarrow cs \qquad (\text{p1})$$
$$cs ::= \{sts\} \qquad (\text{p2})$$
$$\mid P \Rightarrow cs \downarrow cs \qquad (\text{p3})$$

By structural induction to prove the correctness of the construction for the condition structure of (p1) we can take the hypothesis that the constructions for the (p2) and (p3) are correct.

Lemma 7.2.3. [correctness of construction for condition structure]:

$$\Psi_6\ ((P \Rightarrow R \downarrow S)\ [E_i,\ E_o]) \Rightarrow (P \Rightarrow R \downarrow S)\ [E_i,\ E_o]$$

if

$$\Psi_6\ (R\ [E_x,\ E_y]) \Rightarrow R\ [E_x,\ E_y]$$

and

$$\Psi_6\ (S\ [E_u,\ E_v]) \Rightarrow S\ [E_u,\ E_v]$$

Proof:

For each of R and S we should discuss two cases according to (p2) and (p3), therefore there are four combinations. Here, for short we only discuss two combinations (the other two are similar):

(1). Both R and S are of case (p2).

(2). R is of case (p2) and S is of case (p3)

In (1) and (2) below let us follow such a proof structure:

Suppose $\Psi_6\ ((P \Rightarrow R \downarrow S)\ [E_i,\ E_o])$

To get $(P \Rightarrow R \downarrow S)\ [E_i,\ E_o]$

(1). In this case by the construction 6 we have

$$\Psi_6\ ((P \Rightarrow \{S_1\} \downarrow \{S_2\})\ [E_i,\ E_o])$$

$$= (b_p\ y = P \wedge S_1^i \wedge S_2^i)\ [E_i,\ E_m] \tag{L3.1}$$

$$\wedge$$

$$[E_m,\ b_p\ (E_1,\ E_2)] \tag{L3.2}$$

$$\wedge$$

$$\Psi_6\ ((S_1)\ [E_1,\ E_o])$$

$$\wedge$$

$$\Psi_6\ ((S_2)\ [E_2,\ E_o])$$

By the hypothesis of the lemma we have

$$\Psi_6\ ((S_1)\ [E_1,\ E_o]) \Rightarrow (S_1)\ [E_1,\ E_o]$$

and

$$\Psi_6\ ((S_2)\ [E_2,\ E_o]) \Rightarrow (S_2)\ [E_2,\ E_o]$$

Therefore we can get

$$(L3.1) \wedge (L3.2) \wedge (S_1)\ [E_1,\ E_o] \wedge (S_2)\ [E_2,\ E_o] \tag{L3.3}$$

Given $E_i\ t_1$ for an arbitrary time point t_1 and suppose that P is true at t_1, by (L3.1) we get a time point t_2 such that

$$b_p\ t_2 = P \text{ and } E_m\ t_2$$

hold, therefore $b_p\ t_2$ gets true, and S_1^i and S_2^i hold for t_1 and t_2, namely, all the input signals in S_1 and S_2 are protected for two time points t_1 and t_2

By (L3.2) and $E_m\ t_2$ and $b_p\ t_2$ we get E_1 to be true at t_2. Then by $(S_1)\ [E_1,\ E_o]$ we have a time point t_3 such that S_1 holds for t_2 and t_3 and E_o is true at t_3. By S_1^i we further have S_1 holds for t_1 and t_3.

Therefore given $E_i\ t_1$ of the arbitrary t_1 and suppose P to be true at t_1 we get a t_3 such that S_1 holds for t_1 and t_3 and E_o holds at t_3.

It is similar for the case where we start from another assumption that P to be false to get S_2.

Therefore, we get

$(P \Rightarrow S_1 \downarrow S_2)\ [E_i,\ E_o]$

(2). In this case by the construction 6 we have

$$\Psi_6\ ((P \Rightarrow \{S\} \downarrow (Q \Rightarrow \alpha \downarrow \beta))\ [E_i,\ E_o]) \tag{L3.4}$$

$$= (b_p\ y = P \wedge b_q\ y = Q \wedge S^i \wedge$$

$$b_1\ y = P_1 \wedge \cdots \wedge b_k\ y = P_k \wedge S_1^i \wedge \cdots \wedge S_n^i)\ [E_i,\ E_m] \tag{L3.5}$$

$$\wedge$$

$$[E_m,\ b_p\ (E_s,\ b_q\ (\mathcal{E}_\alpha,\ \mathcal{E}_\beta))] \tag{L3.6}$$

$$\wedge$$

$$\Psi_6\ ((S)\ [E_s,\ E_o])$$

$$\wedge$$

$$\Psi_6\ ((S_1)\ [E_1,\ E_o])$$

$$\wedge \cdots \wedge$$

$$\Psi_6\ ((S_n)\ [E_n,\ E_o])$$

Here, we suppose α and β contain k predicates $P_1, \cdots, P_k$ and and n bracketed sub-structures $\{S_1\}, \cdots, \{S_n\}$. Further, for (L3.6) by the branch structure model 7.1 (page 56) we have

$$[E_m,\ b_p\ (E_s,\ b_q\ (\mathcal{E}_\alpha,\ \mathcal{E}_\beta))]$$

$$= \forall x.\ E_m\ x \Rightarrow (b_p\ x \Rightarrow E_s\ x \downarrow (b_q\ x \Rightarrow \mathcal{S}_m\ (\mathcal{E}_\alpha) \downarrow \mathcal{S}_m\ (\mathcal{E}_\beta)))$$

$$= \forall x.\ E_m\ x \Rightarrow (b_p\ x \Rightarrow E_s\ x) \wedge (\neg b_p\ x \Rightarrow (b_q\ x \Rightarrow \mathcal{S}_m\ (\mathcal{E}_\alpha) \downarrow \mathcal{S}_m\ (\mathcal{E}_\beta))) \tag{L3.7}$$

Suppose given $E_i\ t_1$ for an arbitrary time point t_1 and suppose P has false value at t_1. From (L3.5) we can have a time point t_2 such that $E_m\ t_2$ is true and $b_1\ t_2$ is false.

Then by (L3.7) for any time point x and suppose $b_p\ x$ false we can get

$\forall x.\ E_m\ x \Rightarrow (b_q\ x \Rightarrow \mathcal{S}_m\ (\mathcal{E}_\alpha) \downarrow \mathcal{S}_m\ (\mathcal{E}_\beta))$

This is

$$[E_m,\ b_q\ (\mathcal{E}_\alpha,\ \mathcal{E}_\beta)] \tag{L3.8}$$

From (L3.5) we can get

$$(b_q\ y = Q \wedge b_1\ y = P_1 \wedge \cdots \wedge b_k\ y = P_k \wedge S_1^i \wedge \cdots \wedge S_n^i)\ [E_i,\ E_m] \tag{L3.9}$$

And from (L3.4) we have

$$\Psi_6\ ((S_1)\ [E_1,\ E_o]) \wedge \cdots \wedge \Psi_6\ ((S_n)\ [E_n,\ E_o]) \tag{L3.10}$$

Hence from (L3.4) we can get

(L3.9) $\wedge$ (L3.8) $\wedge$ (L3.10)

By the construction 6 this is just

$\Psi_6\ ((Q \Rightarrow \alpha \downarrow \beta)\ [E_i,\ E_o])$

Then by the hypothesis of the lemma we have

$$(Q \Rightarrow \alpha \downarrow \beta)\ [E_i,\ E_o] \tag{L3.11}$$

Therefore we have that

from a given $E_i\ t_1$ of the arbitrary t_1 and by (L3.11)
we can get a time point t_3 such that

$Q \Rightarrow \alpha \downarrow \beta$

holds for t_1 and t_3 and

$E_o\ t_3$

holds.

As (L3.11) is got from (L3.4) therefore we get that

(A). suppose given $E_i\ t_1$ of the arbitrary t_1 and suppose P is false at t_1
from (L3.4) we can get a time point t_3 such that

$Q \Rightarrow \alpha \downarrow \beta$

holds for t_1 and t_3 and

$E_o\, t_3$

holds.

It is similar to the preceding (1) of this lemma that we have that

(B). suppose given $E_i\, t_1$ of the arbitrary t_1 and suppose P is true at t_1

from (L3.4) we can get a time point t_3 such that

S

holds for t_1 and t_3 and

$E_o\, t_3$

holds.

From the two conclusions (A) and (B) above we can finally get

$$(P \Rightarrow \{S\} \downarrow (Q \Rightarrow \alpha \downarrow \beta))\; [E_i,\; E_o]. \qquad \square$$

Theorem 7.2.2. [correctness of construction 6]:

$$\forall sts : \mathcal{S}_{5'} .\; \mathcal{A}_6\; (\Psi_6\; ((sts)\; [E_i,\; E_o])) \Rightarrow (sts)\; [E_i,\; E_o]$$

Proof: Let us use structural induction on the structure of sts.

Base case: $sts = st$. In the case

$\Psi_6\; (st\; [E_i,\; E_o])$

$= \Psi_6\; (((\mathrm{T}) \wedge st)\; [E_i,\; E_o])$ – By the construction equation 7.1.1

$\Rightarrow ((\mathrm{T}) \wedge st)\; [E_i,\; E_o]$ – By the lemma 7.2.1

$\Leftrightarrow (st)\; [E_i,\; E_o]$ – as $(\mathrm{T} \wedge st) \Leftrightarrow st$

Step case: There are further two case (1) and (2).

(1). $sts = \text{let } A \text{ in } ls$

In the case taking the induction hypothesis:

$$\Psi_6\; (ls\; [E_x,\; E_y]) \Rightarrow ls\; [E_x,\; E_y]$$

and using the lemma 7.2.2 we get:

$$\Psi_6\ ((\text{let } A \text{ in } ls)\ [E_i,\ E_o]) \Rightarrow (\text{let } A \text{ in } ls)\ [E_i,\ E_o]$$

(2). $sts = P \Rightarrow R \downarrow S$

In this case taking the induction hypothesis:

$$\Psi_6\ (R\ [E_x,\ E_y]) \Rightarrow R\ [E_x,\ E_y]$$

and

$$\Psi_6\ (S\ [E_u,\ E_v]) \Rightarrow S\ [E_u,\ E_v]$$

and using the lemma 7.2.3 we get:

$$\Psi_6\ ((P \Rightarrow R \downarrow S)\ [E_i,\ E_o]) \Rightarrow (P \Rightarrow R \downarrow S)\ [E_i,\ E_o]$$

By the base and step cases above we get:

$$\Psi_6\ ((sts)\ [E_i,\ E_o]) \Rightarrow (sts)\ [E_i,\ E_o]$$

By the abstract algorithm 6 $\mathcal{A}_6$ we have

$$\mathcal{A}_6\ (\Psi_6\ ((sts)\ [E_i,\ E_o])) \Rightarrow \Psi_6\ ((sts)\ [E_i,\ E_o])$$

Finally, we get

$$\mathcal{A}_6\ (\Psi_6\ ((sts)\ [E_i,\ E_o])) \Rightarrow (sts)\ [E_i,\ E_o] \qquad \square$$

Chapter 8

Decomposing to Function Unit and Connection

In this chapter we present a generic process to decompose a term transition to function unit and connection. By function unit and connection we mean the elementary specification pieces which can be directly implemented by hardware devices at register transfer level which is characterized by using register and bus to "glue" up function units. We will demonstrate how to use derivation model to express and treat the decomposition process.

We have an assumption that each function in the term transition is implementable at register transfer level. If not then we further assume that through data refinement it is always to be able to transformed to such functions.

Let us use an example below to illustrate the basic decomposition process.

Specification 1:

$\forall x.\exists y.\ E_i\ x \Rightarrow s_1\ y = f\ (g\ (r_1\ x)) \wedge s_2\ y = h\ (r_2\ x) \wedge E_o\ y$

For short we rewrite it:

$(s_1\ y = f\ (g\ (r_1\ x)) \wedge s_2\ y = h\ (r_2\ x))\ [E_i,\ E_o]$

We start decomposition with two sub-specifications below:

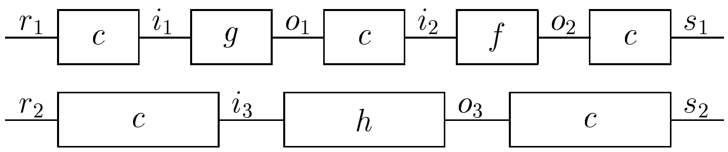
$s_1\ y = f\ (g\ (r_1\ x))\ [E_i,\ E_o]$ (a1)

$$s_2\ y = h\ (r_2\ x)\ [E_i,\ E_o] \tag{a2}$$

Decomposing to function unit and connection level:

$$(i_1\ y = r_1\ x)\ [E_i,\ E_1] \wedge (o_1\ y = g\ (i_1\ x))\ [E_1,\ E_2] \wedge (i_2\ y = o_1\ x)\ [E_2,\ E_3] \wedge$$
$$(o_2\ y = f\ (i_2\ x))\ [E_3,\ E_4] \wedge\ (s_1\ y = o_2\ x)\ [E_4,\ E_o] \tag{b1}$$

$$(i_3\ y = r_2\ x)\ [E_i,\ E_a] \wedge (o_3\ y = h\ (i_3\ x))\ [E_a,\ E_b] \wedge$$
$$(s_2\ y = o_3\ x)\ [E_b,\ E_o] \tag{b2}$$

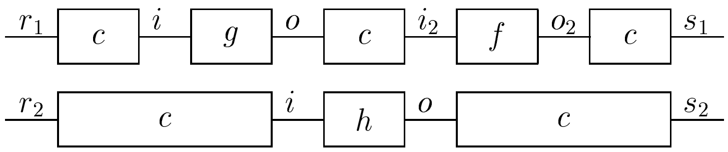


Here we write "*c*" to represent the specification body is a connection.

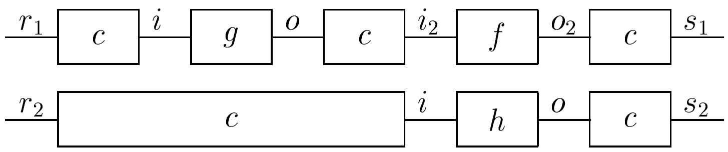
We select a device to implement the function f. And we select another device to implement both the function units g and h, therefore we unified their inputs and outputs to be the same ones: i and o.

$$(i\ y = r_1\ x)\ [E_i,\ E_1] \wedge (o\ y = g\ (i\ x))\ [E_1,\ E_2] \wedge (i_2\ y = o\ x)\ [E_2,\ E_3] \wedge$$
$$(o_2\ y = f\ (i_2\ x))\ [E_3,\ E_4] \wedge (s_1\ y = o_2\ x)\ [E_4,\ E_o] \tag{c1}$$

$$(i\ y = r_2\ x)\ [E_i,\ E_a] \wedge (o\ y = h\ (i\ x))\ [E_a,\ E_b] \wedge$$
$$(s_2\ y = o\ x)\ [E_b,\ E_o] \tag{c2}$$



We can put f and h into a model as they are selected to implement by two different devices.



We divide the model

$(i\ y = r_2\ x)\ [E_i,\ E_a]$

into three models which match the models

$(i\ y = r_1\ x)\ [E_i,\ E_1] \wedge (o\ y = g\ (i\ x))\ [E_1,\ E_2] \wedge (i_2\ y = o\ x)\ [E_2,\ E_3]$

Then compose each pair of corresponding models to get:

$$(i\ y = r_1\ x \wedge r_2\ y = r_2\ x)\ [E_i,\ E_1] \tag{d1}$$

$\wedge$

$$(o\ y = g\ (i\ x) \wedge r_2\ y = r_2\ x)\ [E_1,\ E_2] \tag{d2}$$

$\wedge$

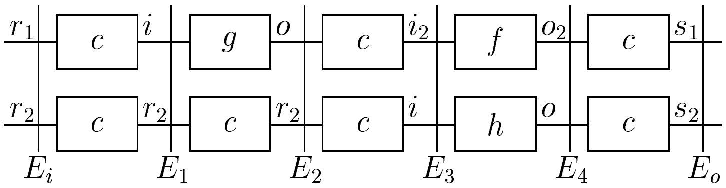
$$(i_2\ y = o\ x \wedge i\ y = r_2\ x)\ [E_2,\ E_3] \tag{d3}$$

$\wedge$

$$(o_2\ y = f\ (i_2\ x) \wedge o\ y = h\ (i\ x))\ [E_3,\ E_4] \tag{d4}$$

$\wedge$

$$(s_1\ y = o_2\ x \wedge s_2\ y = o\ x)\ [E_4,\ E_o] \tag{d5}$$



8.1 Construction 7 and Specification Scheme $6'$ and 7

8.1.1 Specification $6'$

As the extension of the scheme 6 in the scheme $6'$ we give the details of term tm which will be dealt with by construction 7.

Specification Scheme $6'$:

$E\ 0 \wedge ttms$

Term transition model set $ttms$**:**

$$
\begin{array}{lcl}
ttms & ::= & ttm \\
 & & \mid ttm \wedge ttms \\
ttm & ::= & (tts)\ [E_i,\ E_o] \\
 & & \mid brs \\
tts & ::= & tt \\
 & & \mid tt \wedge tts \\
tt & ::= & s\ y = tm \\
tm & ::= & c \\
 & & \mid s\ x \\
 & & \mid f\ (tm, \cdots, tm)
\end{array}
$$

Abbreviation:

$ttms$	: term transition model set
ttm	: term transition model
brs	: branch structure model
tts	: term transition set
tt	: term transition
tm	: term
s	: signal
f	: function
c	: constant

Note, (1). in the scheme the function f has covered the predicate and predicate connective. (2). A different point for the term tm of the scheme from one of the behavior specification scheme is that it does not contain variable as which have been eliminated in the construction 6.

8.1.2 Linear Chain

Linear chain is the entity which the construction 7 mainly deals with. In this section let us define what a linear chain is. First we introduce three primary definitions.

Definition 8.1.2.1. [fu : **function unit**]**:**

A function unit fu is a specification body formed as:

$$o\ y \stackrel{\text{a}}{=} f\ (i_1\ x, \cdots, i_n\ x)$$

Here, "a" represents that a device has been allocated to implement the function unit, therefore the ports of the unit, $o,\ i_1, \cdots, i_n$, are the same as the ports of the device. We call o the output (port) of fu and $i_1, \cdots, i_n$ the inputs (port) of fu.

□

Note: here and elsewhere, we often use port instead of signal to emphasize the implementation aspect of a specification.

A corollary from the definition 8.1.2.1 is that if two function unit are implemented by the same device then their corresponding ports are the same.

Definition 8.1.2.2. [cu : **connection unit**]:

A connection unit cu is a specification body formed as:

$s\ y = r\ x$

Here, s and r are different signals and we call s and r output and input (port) of cu respectively. We also take the boolean constant T as a special connection unit.

□

Definition 8.1.2.3. [du : **delay unit**]:

A delay unit du is a specification body formed as:

$s\ y = s\ x$

Here, s is a signal. □

Definition 8.1.2.4. [fl: **function link**]:

A function link fl is a conjunction of one or more function units and zero or more delay units. □

Definition 8.1.2.5. [cl: **connection link**]:

A connection link cl is a conjunction of one or more connection units and zero or more delay units. □

Definition 8.1.2.6. [sl: **special link**]:

A special link sl is a conjunction of one or more delay units. We assume that it can be used as function link or as connection link. □

Definition 8.1.2.7. [tfl: **tolerant function link**]:

A function link is tolerant if in the link there are not two function units to be implemented by the same device. This means that there are not any two ports in the link to be the same. □

Definition 8.1.2.8. [tcl**: tolerant connection link**]**:**

A connection link is tolerant if in the link there are not two connection units having the same output. □

Definition 8.1.2.9. [**linear chain**]**:**

A linear chain is formed as

$$(tcl_1)\ [E_1,\ E_2] \wedge (tfl_2)\ [E_2,\ E_3] \wedge (tcl_3)\ [E_3,\ E_4] \wedge \cdots \wedge$$
$$(tcl_{n-3})\ [E_{n-3},\ E_{n-2}] \wedge (tfl_{n-2})\ [E_{n-2},\ E_{n-1}] \wedge (tcl_{n-1})\ [E_{n-1},\ E_n].$$

Here, n is an odd number. When $n = 1$ we define a linear chain is

$(tcl)\ [E_1,\ E_2].$ □

Intuitively, a linear chain is a sequence of derivation models along an entry predicate sequence $E_1,\ E_2, \cdots, E_n$ and the specification bodies of these models are alternate of tolerant connection links and tolerant function links.

For a linear chain or the part of a linear chain form as

$$\alpha_1\ [E_1,\ E_2] \wedge \alpha_2\ [E_2,\ E_3] \wedge \cdots \wedge \alpha_{n-1}\ [E_{n-1},\ E_n]$$

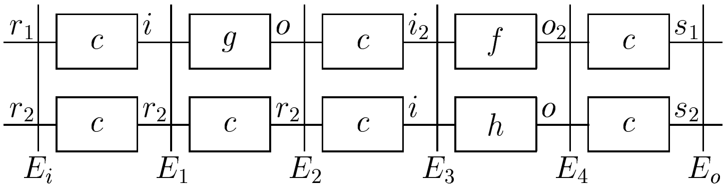
we call E_1 and E_n as its head node and tail node respectively; $\alpha_1\ [E_1,\ E_2]$ and $\alpha_{n-1}\ [E_{n-1},\ E_n]$ as its head link and tail links respectively.

An example of linear chain can be taken from the preceding example: (page 84)

$$(i\ y = r_1\ x \wedge r_2\ y = r_2\ x)\ [E_i,\ E_1] \tag{d1}$$
$$\wedge$$
$$(o\ y = g\ (i\ x) \wedge r_2\ y = r_2\ x)\ [E_1,\ E_2] \tag{d2}$$
$$\wedge$$
$$(i_2\ y = o\ x \wedge i\ y = r_2\ x)\ [E_2,\ E_3] \tag{d3}$$
$$\wedge$$
$$(o_2\ y = f\ (i_2\ x) \wedge o\ y = h\ (i\ x))\ [E_3,\ E_4] \tag{d4}$$

$\wedge$

$$(s_1\ y = o_2\ x \wedge s_2\ y = o\ x)\ [E_4,\ E_o] \tag{d5}$$



Here, (d1), (d3) and (d5) are tolerant connection link models, (d2) and (d4) are tolerant function link models.

8.1.3 Construction 7

Construction 7 deals with each term transition model of the scheme $6'$ to transform it to a linear chain.

Definition 8.1.3.1. [abbreviation for model sequence]:

We write $\Gamma^{\alpha}_{(i,j)}$ as abbreviation for

$\alpha_i\ [E^{\alpha}_i,\ E^{\alpha}_{i+1}] \wedge \alpha_{i+1}\ [E^{\alpha}_{i+1},\ E^{\alpha}_{i+2}] \wedge \cdots \wedge \alpha_j\ [E^{\alpha}_j,\ E^{\alpha}_{j+1}]$

Note: we write E^{α}_i to emphasize that the entry predicate belongs to a model whose specification body is α (with subscript if necessary). □

For example, we write $\Gamma^{\beta}_{(3,5)}$ for $\beta_3\ [E_3, E_4] \wedge \beta_4\ [E_4, E_5] \wedge \beta_5\ [E_5, E_6]$

Definition 8.1.3.2. [abbreviation for fu and cu]:

We write α^{F} for a function unit and α^{C} for a connection unit. □

Definition 8.1.3.3. [merging two function links]:

Two function links are merge-able if putting them together through conjunction we can get a tolerant function link. □

Definition 8.1.3.4. [merging two connection links]:

Two connection links are merge-able if putting them together through conjunction we can get a tolerant connection link. □

Construction 7:

8.1.3.1. $\Psi_7\ ((tts)\ [E_i,\ E_o]) \mathrel{\widehat{=}} \Psi_{7m}\ (\Psi_{7s}\ ((tts)\ [E_i,\ E_o]))$

8.1.3.2. $\Psi_{7s}\ ((tt)\ [E_i,\ E_o]) \mathrel{\widehat{=}} \Psi_{7l}\ ((tt)\ [E_i,\ E_o])$

8.1.3.3. $\Psi_{7s}\ ((tt \wedge tts)\ [E_i,\ E_o]) \mathrel{\widehat{=}} \Psi_{7l}\ ((tt)\ [E_i,\ E_o]),\ \Psi_{7s}\ ((tts)\ [E_i,\ E_o])$

8.1.3.4. $\Psi_{7l}\ ((s\ y = r\ x)\ [E_i,\ E_o]) \mathrel{\widehat{=}} (s\ y = r\ x)\ [E_i,\ E_o]$

Here, r is a signal.

8.1.3.5. $\Psi_{7l}\ ((s\ y = c)\ [E_i,\ E_o])$

$\mathrel{\widehat{=}} (s\ y = o\ x)\ [E_i,\ E_a] \wedge (o\ y \overset{\mathrm{a}}{=} c)\ [E_a,\ E_o] \wedge \mathrm{T}\ [E_o,\ E_o]$

Here and elsewhere, when the "=" of a function unit is replaced by "$\overset{\mathrm{a}}{=}$", this represents that a device has been allocated to implement the unit.

In this case we have an assumption:

Assumption 8.1.3.1. [unifying ports]:

For the function units assigned to the same device to implement their ports (inputs and outputs) are unified to be the same. (However, still to be kept as existent variables). □

8.1.3.6. $\Psi_{7l}\ ((s\ y = f\ (tm_1, \cdots, tm_n)\ [E_i,\ E_o]) \mathrel{\widehat{=}}$

$\Psi_7\ ((i_1\ y = tm_1 \wedge \cdots \wedge i_n\ y = tm_n)\ [E_i,\ E_a]) \wedge$

$(o\ y \overset{\mathrm{a}}{=} f\ (i_1\ x, \cdots, i_n\ x))\ [E_a,\ E_b] \wedge$

$(s\ y = o\ x)\ [E_b,\ E_o]$

Starting from Ψ_{7m} we have an assumption that for each construction all its arguments, except specially stated, are the linear chains or the part of a linear chain, which have the same head node and tail node. For example, in the construction 8.1.3.12 for $\Psi_{7c}\ (\Gamma_1, \Gamma_2)$ we have that Γ_1 and Γ_2 have the same head and tail nodes.

8.1.3.7. $\Psi_{7m}\ (\alpha) \mathrel{\widehat{=}} \alpha$

8.1.3.8. $\Psi_{7m}\ (\alpha_1,\ \alpha_2) \mathrel{\widehat{=}} \Psi_{7f}\ (\alpha_1,\ \alpha_2)$

8.1.3.9. $\Psi_{7m}\ (\alpha_1,\ \alpha_2,\ \alpha_3,\ \cdots, \alpha_n) \mathrel{\widehat{=}} \Psi_{7m}\ (\Psi_{7f}\ (\alpha_1,\ \alpha_2),\ \alpha_3,\ \cdots, \alpha_n)$

For constructions Ψ_{7f}, Ψ_{7c} and Ψ_{7e1} below we assume their two arguments to be commutative for example, we have

$\Psi_{7f}\ (\alpha,\ \beta) = \Psi_{7f}\ (\beta,\ \alpha)$

And for $\Psi_{7c'}$ and Ψ_{7e2} below we assume their first two arguments to be commutative, for example, we have

$\Psi_{7c'}\ (\alpha,\ \beta,\ \gamma) = \Psi_{7c'}\ (\beta,\ \alpha,\ \gamma)$

8.1.3.10. $\Psi_{7f}\ (\Gamma_1,\ \Gamma_2) \mathrel{\widehat{=}} \Psi_{7c}\ (\Gamma_1,\ \Gamma_2)$

For construction equation 8.1.3.10 we suppose that there are not two function units from Γ_1 and Γ_2 respectively being merge-able.

8.1.3.11.
$$\begin{aligned} &\Psi_{7f}\ (\ \Gamma^{\alpha}_{(0,i-1)} \wedge \alpha^{\mathrm{F}}_i\ [E^{\alpha}_i, E^{\alpha}_{i+1}] \wedge \Gamma^{\alpha}_{(i+1,m)}, \\ &\qquad \Gamma^{\beta}_{(0,j-1)} \wedge \beta^{\mathrm{F}}_j\ [E^{\beta}_j, E^{\beta}_{j+1}] \wedge \Gamma^{\beta}_{(j+1,n)}\ ) \\ \mathrel{\widehat{=}}\ &\Psi_{7f}\ (\Gamma^{\alpha}_{(0,i-1)},\ \Gamma^{\beta}_{(0,j-1)}) \wedge \\ &(\alpha^{\mathrm{F}}_i \wedge \beta^{\mathrm{F}}_j)\ [E_x,\ E_y] \wedge \\ &\Psi_{7f}\ (\Gamma^{\alpha}_{(i+1,m)},\ \Gamma^{\beta}_{(j+1,n)}) \end{aligned}$$

For construction equation 8.1.3.11 we suppose that there are at least a pair of function units from Γ^{α} and Γ^{β} respectively being merge-able. And we arbitrarily select a pair:

$\alpha^{\mathrm{F}}_i\ [E^{\alpha}_i, E^{\alpha}_{i+1}]$ and $\beta^{\mathrm{F}}_j\ [E^{\beta}_j, E^{\beta}_{j+1}]$

to be merged as the shown in the construction equation.

Note: here, we introduce E_x and E_y and have $E^{\alpha}_i = E^{\beta}_j = E_x$ and $E^{\alpha}_{i+1} = E^{\beta}_{j+1} = E_y$

8.1.3.12. $\Psi_{7c}\,(\Gamma_1, \Gamma_2) \doteq \Psi_{7e1}\,(\Gamma_1, \Gamma_2)$

For construction equation 8.1.3.12 we suppose that there are not two connection units from Γ_1 and Γ_2 respectively being merge-able.

8.1.3.13.
$$\begin{aligned}&\Psi_{7c}\,(\ \Gamma^{\alpha}_{(0,i-1)} \wedge \alpha^{\mathrm{C}}_i\,[E^{\alpha}_i, E^{\alpha}_{i+1}] \wedge \Gamma^{\alpha}_{(i+1,m)},\\ &\quad \Gamma^{\beta}_{(0,j-1)} \wedge \beta^{\mathrm{C}}_j\,[E^{\beta}_j, E^{\beta}_{j+1}] \wedge \Gamma^{\beta}_{(j+1,n)}\ )\\ \doteq\ &\Psi_{7e2}\,(\Gamma^{\mathrm{L}}_1,\ \Gamma^{\mathrm{L}}_2,\ \gamma) \wedge\\ &(\alpha^{\mathrm{C}}_i \wedge \beta^{\mathrm{C}}_j)\,[E_x,\ E_y] \wedge\\ &\Psi_{7c'}\,(\Gamma^{\mathrm{R}}_1,\ \Gamma^{\mathrm{R}}_2,\ \gamma')\end{aligned}$$

For construction equation 8.1.3.13 we suppose that there are at least a pair of connection units from Γ^{α} and Γ^{β} respectively being merge-able. And we select the pair which first appears in the two chains along direction from left to right:

$\alpha^{\mathrm{C}}_i\,[E^{\alpha}_i, E^{\alpha}_{i+1}]$ and $\beta^{\mathrm{C}}_j\,[E^{\beta}_j, E^{\beta}_{j+1}]$

to be merged as the shown in the construction equation.

In the equation we write $\Psi_{7e2}\,(\Gamma^{\mathrm{L}}_1,\ \Gamma^{\mathrm{L}}_2,\ \gamma)$ to express three possible cases:

(1). $\Psi_{7e2}\,(\Gamma^{\alpha}_{(0,i-1)},\ \Gamma^{\beta}_{(0,j-1)},\ \gamma)$, in this case both $\Gamma^{\alpha}_{(0,i-1)}$ and $\Gamma^{\beta}_{(0,j-1)}$ are not empty and γ is useless.

(2). $\Psi_{7e2}\,(\Gamma^{\alpha}_{(0,i-1)},\ \Gamma^{\beta}_{(0,j-1)},\ \gamma)$, in this case $\Gamma^{\beta}_{(0,j-1)}$ is empty (boolean constant T) and γ is β^{i}_{j}.

(3). $\Psi_{7e2}\,(\Gamma^{\beta}_{(0,j-1)},\ \Gamma^{\alpha}_{(0,i-1)},\ \gamma)$, in this case $\Gamma^{\alpha}_{(0,i-1)}$ is empty and γ is α^{i}_{i}.

Similarly, we write $\Psi_{7c'}$ $(\Gamma_1^{\mathrm{R}},\ \Gamma_2^{\mathrm{R}},\ \gamma')$ to express three possible cases:

(1). $\Psi_{7c'}$ $(\Gamma^{\alpha}_{(i+1,m)},\ \Gamma^{\beta}_{(j+1,n)},\ \gamma')$, in this case both $\Gamma^{\alpha}_{(i+1,m)}$ and $\Gamma^{\beta}_{(j+1,n)}$ are not empty and γ' is useless.

(2). $\Psi_{7c'}$ $(\Gamma^{\alpha}_{(i+1,m)},\ \Gamma^{\beta}_{(j+1,n)},\ \gamma')$, in this case $\Gamma^{\beta}_{(j+1,n)}$ is empty and γ' is β^{o}_{j}.

(3). Ψ_{7e2} $(\Gamma^{\beta}_{(j+1,n)},\ \Gamma^{\alpha}_{(i+1,m)},\ \gamma')$, in this case $\Gamma^{\alpha}_{(i+1,m)}$ is empty and γ' is α^{o}_{i}.

Note: here, we introduce E_x and E_y and have $E^{\alpha}_{i} = E^{\beta}_{j} = E_x$ and $E^{\alpha}_{i+1} = E^{\beta}_{j+1} = E_y$.

8.1.3.14. $\Psi_{7c'}$ $(\Gamma_1,\ \Gamma_2,\ \gamma) \triangleq \Psi_{7e2}$ $(\Gamma_1,\ \Gamma_2,\ \gamma)$

For the construction equation 8.1.3.14 we assume that there are not two connection units from Γ_1 and Γ_2 respectively being mergeable. This includes that case that Γ_2 is empty linear chain, i.e. $\Gamma_2 = \mathrm{T}$ and the case that both Γ_1 and Γ_2 are empty linear chain.

8.1.3.15.
$$\begin{aligned}
&\Psi_{7c'}\ (\ \Gamma^{\alpha}_{(0,i-1)} \wedge \alpha^{\mathrm{C}}_{i}\ [E^{\alpha}_{i}, E^{\alpha}_{i+1}] \wedge \Gamma^{\alpha}_{(i+1,m)},\\
&\qquad \Gamma^{\beta}_{(0,j-1)} \wedge \beta^{\mathrm{C}}_{j}\ [E^{\beta}_{j}, E^{\beta}_{j+1}] \wedge \Gamma^{\beta}_{(j+1,n)},\ \gamma)\\
&\triangleq\ \Psi_{7e1}\ (\Gamma^{\alpha}_{(0,i-1)},\ \Gamma^{\beta}_{(0,j-1)}) \wedge\\
&\quad (\alpha^{\mathrm{C}}_{i} \wedge \beta^{\mathrm{C}}_{j})\ [E_x,\ E_y] \wedge\\
&\quad \Psi_{7c'}\ (\Gamma^{\mathrm{R}}_{1},\ \Gamma^{\mathrm{R}}_{2},\ \gamma')
\end{aligned}$$

For construction equation 8.1.3.15 we suppose that there are at least a pair of connection units from Γ^{α} and Γ^{β} respectively being merge-able. And we select the pair which first appears in the two chains along direction from left to right:

$\alpha_i^{\mathrm{C}}\ [E_i^{\alpha}, E_{i+1}^{\alpha}]$ and $\beta_j^{\mathrm{C}}\ [E_j^{\beta}, E_{j+1}^{\beta}]$

to be merged as the shown in the construction equation.

Similar to the construction equation 8.1.3.13, we write $\Psi_{7c'}\ (\Gamma_1^{\mathrm{R}},\ \Gamma_2^{\mathrm{R}},\ \gamma')$ to express three possible cases:

(1). $\Psi_{7c'}\ (\Gamma^{\alpha}_{(i+1,m)},\ \Gamma^{\beta}_{(j+1,n)},\ \gamma')$, in this case both $\Gamma^{\alpha}_{(i+1,m)}$ and $\Gamma^{\beta}_{(j+1,n)}$ are not empty and γ' is useless.

(2). $\Psi_{7c'}\ (\Gamma^{\alpha}_{(i+1,m)},\ \Gamma^{\beta}_{(j+1,n)},\ \gamma')$, in this case $\Gamma^{\beta}_{(j+1,n)}$ is empty and γ' is β_j^o.

(3). $\Psi_{7e2}\ (\Gamma^{\beta}_{(j+1,n)},\ \Gamma^{\alpha}_{(i+1,m)},\ \gamma')$, in this case $\Gamma^{\alpha}_{(i+1,m)}$ is empty and γ' is α_i^o.

Note: here, we introduce E_x and E_y and have $E_i^{\alpha} = E_j^{\beta} = E_x$ and $E_{i+1}^{\alpha} = E_{j+1}^{\beta} = E_y$.

8.1.3.16. $$\begin{aligned}\Psi_{7e1}\ (\Gamma^{\alpha}_{(a,b)},\ \Gamma^{\beta}_{(c,d)}) \ \widehat{=}\ \ & \Psi_{7d}\ (\Gamma^{\alpha}_{(a,b)},\ \beta_c^i\ [E_w,\ E_x]) \wedge \\ & (\alpha_b^o \wedge \beta_c^i)\ [E_x,\ E_y] \wedge \\ & \Psi_{7d}\ (\Gamma^{\beta}_{(c,d)},\ \alpha_b^o\ [E_y,\ E_z])\end{aligned}$$

In the construction equation 8.1.3.16 we introduce E_w and have $E_w = E_a^{\alpha}$, introduce E_x and use it to replace E_{b+1}^{α}, introduce E_y and use it to replace E_c^{β}, and introduce E_z and have $E_z = E_d^{\beta}$. If there are the same entry predicates between $\Gamma^{\alpha}_{(a,b)}$ and $\Gamma^{\beta}_{(c,d)}$ in two Ψ_{7d} then change them to make them different from each other.

For β_c^i and α_b^o, if you forget its meaning, please check the definition 7.1 (page 58).

8.1.3.17. $\Psi_{7e2}\ (\mathrm{T},\ \mathrm{T},\ \gamma) \triangleq \mathrm{T}$

8.1.3.18. $\Psi_{7e2}\ (\Gamma^{\alpha}_{(a,b)},\ \mathrm{T},\ \beta^{i}_{c}) \triangleq\ \Psi_{7d}\ (\Gamma^{\alpha}_{(a,b)},\ \beta^{i}_{c}\ [E_x,\ E_y])$

8.1.3.19. $\Psi_{7e2}\ (\Gamma^{\alpha}_{(a,b)},\ \Gamma^{\beta}_{(c,d)},\ \gamma)$

$\triangleq\ \Psi_{7d}\ (\Gamma^{\alpha}_{(a,b)},\ \beta^{i}_{c}\ [E_x,\ E_y]) \wedge \Psi_{7d}\ (\Gamma^{\beta}_{(c,d)},\ \alpha^{o}_{b}\ [E_y,\ E_z])$

In the construction Ψ_{7e2} we use boolean constant T to express an empty part chain. In the construction equation 8.1.3.18 we introduce E_x and E_y and have $E_x = E^{\alpha}_{a}$ and $E_y = E^{\alpha}_{b+1}$. In the construction equation 8.1.3.19 we introduce E_x, E_y and E_z and have $E_x = E^{\alpha}_{a}$, $E_y = E^{\alpha}_{b+1} = E^{\beta}_{c}$ and $E_z = E^{\beta}_{d+1}$.

8.1.3.20. $\Psi_{7d}\ (\alpha\ [E_1,\ E_2],\ \beta^{i/o}\ [E_1,\ E_2]) \triangleq (\alpha \wedge \beta^{i/o})\ [E_1,\ E_2]$

8.1.3.21. $\Psi_{7d}\ (\alpha_1\ [E_1,\ E_2] \wedge \Gamma^{\alpha}_{(2,\ n)},\ \beta^{i/o}\ [E_1,\ E_n])$

$\triangleq\ (\alpha_1 \wedge \beta^{i/o})\ [E_1,\ E_2] \wedge \Psi_{7d}\ (\Gamma^{\alpha}_{(2,\ n)},\ \beta^{i/o}\ [E_2,\ E_n])$

In the construction Ψ_{7d} we write $\beta^{i/o}$ to mean is may be β^{i} or β^{o}.

Let us explain the construction 7.

Ψ_7 is a construction which transforms a term transition model formed as $(tts)\ [E_i,\ E_o]$ to a linear chain. The work is done through two construction Ψ_{7s} and Ψ_{7m}.

Ψ_{7s} is a trivial operation. It separates the term transitions tt in a term transition set tts through using comma "," to replace logic conjunctive "$\wedge$" and then passes the tt with the same logic boundary $[E_i,\ E_o]$ of the tts to the construction Ψ_{7l}. The work of Ψ_{7s} is done through the construction equations 8.1.3.2 and 3.

Example 8.1.3.1. [for understanding Ψ_{7s}]:

$\Psi_{7s}\ ((s_1\ y = f\ (g\ (r_1\ x)) \wedge s_2\ y = h\ (r_2\ x))\ [E_i,\ E_o])$

$= \Psi_{7l}\ (s_1\ y = f\ (g\ (r_1\ x))\ [E_i,\ E_o]),\ \Psi_{7l}\ (s_2\ y = h\ (r_2\ x)\ [E_i,\ E_o])$ □

Ψ_{7l} is the main framework of the construction 7. It transforms a term transition model (tt) $[E_i,\ E_o]$ to a linear chain. Construction equation 8.1.3.4 deals with a base case: connection unit model. It does not change any thing. Construction equation 8.1.3.5 deals with another base case: constant tt. It allocates a device to implement the constant term transition and meanwhile adds two connection links among them the T $[E_o,\ E_o]$ is a special one. Construction equation 8.1.3.6 deals with the step case. For a general $s\ y = f\ (tm, \cdots, tm)\ [E_i,\ E_o]$ it first recursively calls Ψ_7 to treat all the argument terms of the function f

$$\Psi_7\ ((i_1\ y = tm_1 \wedge \cdots \wedge i_n\ y = tm_n)\ [E_i,\ E_a])$$

and get a linear chain. Then it produces a function unit and a connection units and makes the three parts form a linear chain.

Example 8.1.3.2. [for understanding Ψ_{7l}]:

$$\begin{aligned}
&\Psi_{7l}\ ((s\ y = f\ (r_1\ x,\ r_2\ x))\ [E_i,\ E_o]) \\
&= \Psi_7\ ((i_1\ y = r_1\ x \wedge i_2\ y = r_2\ x)\ [E_i,\ E_a]) \wedge \\
&\quad (o\ y \stackrel{\text{a}}{=} f\ (i_1\ x,\ i_2\ x))\ [E_a,\ E_b] \wedge \\
&\quad (s\ y = o\ x)\ [E_b,\ E_o] && \text{By construction equation 8.1.3.6} \\
&= (i_1\ y = r_1\ x \wedge i_2\ y = r_2\ x)\ [E_i,\ E_a] \wedge \\
&\quad (o\ y \stackrel{\text{a}}{=} f\ (i_1\ x,\ i_2\ x))\ [E_a,\ E_b] \wedge \\
&\quad (s\ y = o\ x)\ [E_b,\ E_o] && \square
\end{aligned}$$

Note: later in this construction 7 we will see how to get

$$(i_1\ y = r_1\ x \wedge i_2\ y = r_2\ x)\ [E_i,\ E_a]$$

from

$$\Psi_7\ ((i_1\ y = r_1\ x \wedge i_2\ y = r_2\ x)\ [E_i,\ E_a])$$

following a general construction process. But here, it can be intuitively accepted as it contains only two connection units and it is obvious that we can put them into a model to implement.

Ψ_{7m} merge linear chains to get a linear chain. These chains have the same head and tail nodes (see page 87): E_i and E_o. If there is only a chain to be merged then the chain has been the merged one. This is as the construction equation 8.1.3.7. If there are two chains to be merged then, as the construction equation 8.1.3.8 shown, the merging is passed to Ψ_f to do. If there are more than two chains need to be merged then, as the construction equation 8.1.3.9 shown, this is recursively done through calling Ψ_f.

Example 8.1.3.3. [for understanding Ψ_{7m}]:

Ψ_{7m} can be understood through the process from (c1) and (c2) to (d1), (d2), ..., (d5) in the example in page 83 and page 84. □

Ψ_{7f} merge two linear chains to get a linear chain. There are two cases: base one and step one. The base case is that there is not any two function links coming from the two chains respectively being merge-able. In this case the two chains are passed to Ψ_{7c} to treat. This is as the construction equation 8.1.3.10. The step case is that there is at least a pair of function units coming from the two chains respectively being merge-able. In this case, as the construction equation 8.1.3.11 shown, we can arbitrarily select a pair of merge-able function units to merge them to get a model

$(\alpha_i^{\mathrm{F}} \wedge \beta_j^{\mathrm{F}})\ [E_x,\ E_y]$

and leave four sub-chains

$\Gamma^{\alpha}_{(0,i-1)},\ \Gamma^{\beta}_{(0,j-1)},\ \Gamma^{\alpha}_{(i+1,m)}$ and $\Gamma^{\beta}_{(j+1,n)}$

to be recursively dealt with by Ψ_{7f} itself:

$\Psi_{7f}\ (\Gamma^{\alpha}_{(0,i-1)},\ \Gamma^{\beta}_{(0,j-1)})$ and $\Psi_{7f}\ (\Gamma^{\alpha}_{(i+1,m)},\ \Gamma^{\beta}_{(j+1,n)})$

We like to point that the two chains Γ_1 and Γ_2 in the construction equation 8.1.3.10 certainly are non-empty by the structure of linear chain.

Example 8.1.3.4. [for understanding Ψ_{7f}]:

The example is the same as the example 8.1.3.3. Two places should be pointed out to help reader to understand.

1. Two function links

   $(o_2\ y = f\ (i_2\ x))\ [E_3,\ E_4]$

   $(o\ y = h\ (i\ x))\ [E_a,\ E_b]$

   are merged to get a tolerant function link model as (d4)

2. Other links are merged to get two sub-chains:

   (1). (d1) $\wedge$ (d2) $\wedge$ (d3)

   (2). (d5)

Construction Ψ_{7c} deals with two linear chains which are passed from Ψ_{7f}, therefore, have not function links merge-able. If the two chains further have not two connection links merge-able then they are passed Ψ_{7e1} to treat. This is as the construction equation 8.1.3.12 shown. Otherwise, as the construction equation 8.1.3.13 shown, we find the pair of merge-able connection links which is the first one in the two chains along the direction from left to right:

$\alpha_i^{\mathrm{C}}\ [E_i^{\alpha}, E_{i+1}^{\alpha}]$ and $\beta_j^{\mathrm{C}}\ [E_j^{\beta}, E_{j+1}^{\beta}]$

to merge them to get $(\alpha_i^{\mathrm{C}} \wedge \beta_j^{\mathrm{C}})\ [E_x,\ E_y]$

We pass the two left parts of the two linear chains to the construction Ψ_{7e2} and the two right parts of the two linear chains to the construction $\Psi_{7c'}$. However, it is possible that the one left part or two left parts are empty chains, in this case we write boolean constant T to express it.

For the case of a empty chain we use γ for the input signal protection (see 58) of the corresponding merged connection link. And as the Ψ_{7e2} is commutative we can swap its first two arguments to make sure that in the case of there are empty chains the second argument certainly is the empty one (This is for the construction Ψ_{7e2} easier to be treated). Similarly we deal with the two right part.

Example 8.1.3.5. [for understanding Ψ_{7c}]:

$\Psi_{7c}\ ((s_1\ y = r_1\ x)\ [E_1,\ E_2] \wedge \alpha^{\mathrm{F}}\ [E_2,\ E_3] \wedge (s_2\ y = r_2\ x)\ [E_3,\ E_4],$

$(s_3\ y = r_3\ x)\ [E_1,\ E_4])$

$= \Psi_{7e2}\ (\mathrm{T},\ \mathrm{T},\ \gamma) \wedge$

$(s_1\ y = r_1\ x \wedge s_3\ y = r_3\ x)\ [E_1,\ E_2] \wedge$

$\Psi_{7c'}\ (\alpha^{\mathrm{F}}\ [E_2,\ E_3] \wedge (s_2\ y = r_2\ x)\ [E_3,\ E_4],\ \mathrm{T},\ s_3\ y = s_3\ x)$

By the construction equation 8.1.3.13

$= \mathrm{T} \wedge$

$(s_1\ y = r_1\ x \wedge s_3\ y = r_3\ x)\ [E_1,\ E_2] \wedge$

$(\alpha^{\mathrm{F}} \wedge s_3\ y = s_3\ x)\ [E_2,\ E_3] \wedge (s_2\ y = r_2\ x \wedge s_3\ y = s_3\ x)\ [E_3,\ E_4])$

Note: the Ψ_{7e2} and $\Psi_{7c'}$ later will make the construction, however here, we suppose that they are intuitively acceptable by reader.

$= (s_1\ y = r_1\ x \wedge s_3\ y = r_3\ x)\ [E_1,\ E_2] \wedge$

$(\alpha^{\mathrm{F}} \wedge s_3\ y = s_3\ x)\ [E_2,\ E_3] \wedge$

$(s_2\ y = r_2\ x \wedge s_3\ y = s_3\ x)\ [E_3,\ E_4])$

This is a linear chain. □

Construction $\Psi_{7c'}$ is similar to Ψ_{7c}. However, a difference is that the two left parts $\Gamma^{\alpha}_{(0,i-1)}$ and $\Gamma^{\beta}_{(0,j-1)}$ in the construction equation 8.1.3.15 certainly are non-empty. This is because that $\Psi_{7c'}$ is called by Ψ_{7c} and by the structure of linear chain between two connection links there is at least a function link.

Example 8.1.3.6. [for understanding $\Psi_{7c'}$]:

$\Psi_{7c'}\ ((\alpha^{\mathrm{F}} \wedge \beta_1^{\mathrm{F}})\ [E_1,\ E_2]\ \wedge (s_1\ y = r_1\ x)\ [E_2,\ E_3],$

$(\alpha^{\mathrm{F}} \wedge \beta_2^{\mathrm{F}})\ [E_1,\ E_2]\ \wedge (s_2\ y = r_2\ x)\ [E_2,\ E_3],\ \gamma)$

$= \Psi_{7e1}\ ((\alpha^{\mathrm{F}} \wedge \beta_1^{\mathrm{F}})\ [E_1,\ E_2],\ (\alpha^{\mathrm{F}} \wedge \beta_2^{\mathrm{F}})\ [E_1,\ E_2]) \wedge$

$(s_1\ y = r_1\ x \wedge s_2\ y = r_2\ x)\ [E_2,\ E_3] \wedge$

$\Psi_{7c'}\ (\mathrm{T},\ \mathrm{T},\ \gamma')$ By construction equation 8.1.3.15

$= \Psi_{7e1}\ ((\alpha^{\mathrm{F}} \wedge \beta_1^{\mathrm{F}})\ [E_1,\ E_2],\ (\alpha^{\mathrm{F}} \wedge \beta_2^{\mathrm{F}})\ [E_1,\ E_2]) \wedge$

$(s_1\ y = r_1\ x \wedge s_2\ y = r_2\ x)\ [E_2,\ E_3]$

Note: the Ψ_{7e2} later will make that $\Psi_{7c'}$ (T, T, γ') = T , however here, we suppose that they are intuitively acceptable by reader.

$\square$

Construction Ψ_{7e1} deals with two non-empty chains (if it is called by the construction equation 8.1.3.12) or non-empty parts of chains (if it is called by the construction equation 8.1.3.15) whose head and tail link are function link. And the two chain or two parts of chains have all links (function links and connection links) non-merge-able. In the case we have to put the two parts into two different time intervals and to introduce a new link to connect the two parts. The new link is a special one as it only contains delay units (see the definition 8.1.2.6, page 86).

Example 8.1.3.7. [for understanding Ψ_{7e1}]:

$$\begin{aligned}
&\Psi_{7e1}\ (\alpha_1^{\text{F}}\ [E_1,\ E_2] \wedge \alpha_2^{\text{C}}\ [E_2,\ E_3] \wedge \alpha_3^{\text{F}}\ [E_3,\ E_4],\\
&\qquad \beta_1^{\text{F}}\ [E_1,\ E_2] \wedge \beta_2^{\text{C}}\ [E_2,\ E_3] \wedge \beta_3^{\text{F}}\ [E_3,\ E_4])\\
&= \Psi_{7d}\ (\alpha_1^{\text{F}}\ [E_1,\ E_a] \wedge \alpha_2^{\text{C}}\ [E_a,\ E_b] \wedge \alpha_3^{\text{F}}\ [E_b,\ E_c],\ (\beta_1^{\text{F}})^i\ [E_1,\ E_c]) \wedge\\
&\quad ((\alpha_3^{\text{F}})^o \wedge (\beta_1^{\text{F}})^i)\ [E_c,\ E_d] \wedge\\
&\quad \Psi_{7d}\ (\beta_1^{\text{F}}\ [E_d,\ E_e] \wedge \beta_2^{\text{C}}\ [E_e,\ E_f] \wedge \beta_3^{\text{F}}\ [E_f,\ E_4],\ (\alpha_3^{\text{F}})^o\ [E_d,\ E_4])
\end{aligned}$$

$\square$

Construction Ψ_{7e2} deals with two part chains whose head and tail links are connection link and function link respectively (if Ψ_{7e2} is called by the construction equation 8.1.3.13) or function link and connection link respectively (if Ψ_{7e2} is called by the construction equation 8.1.3.14). The two part chains have not any their links merge-able. One or both of the two parts possibly are empty.

To deal with the empty part chain we introduce the third argument in Ψ_{7e2} to record signal protections which is passed from the construction equation 8.1.3.13 or 14. And this is used for the construction equation 8.1.3.18. The construction equation 8.1.3.17 deals with the case of that two part chains are empty. In this case, obviously, the result is also an empty part chain and the third argument is useless. The construction equation 8.1.3.19 deals with the case of that two part chains are non-empty. This case is similar to Ψ_{7e1} but needs not to introduce a special link. And in this case the third argument is too useless.

Example 8.1.3.8. [for understanding Ψ_{7e2}]:

$$\Psi_{7e1}\ (\alpha_1^{\mathrm{C}}\ [E_1,\ E_2] \wedge \alpha_2^{\mathrm{F}}\ [E_2,\ E_3] \wedge \alpha_3^{\mathrm{C}}\ [E_3,\ E_4] \wedge \alpha_4^{\mathrm{F}}\ [E_4,\ E_5],$$
$$\beta_1^{\mathrm{C}}\ [E_a,\ E_b] \wedge \beta_2^{\mathrm{F}}\ [E_b,\ E_c],\ \gamma)$$
$$= \Psi_{7d}\ (\alpha_1^{\mathrm{C}}\ [E_1,\ E_2] \wedge \alpha_2^{\mathrm{F}}\ [E_2,\ E_3] \wedge \alpha_3^{\mathrm{C}}\ [E_3,\ E_4] \wedge \alpha_4^{\mathrm{F}}\ [E_4,\ E_5],\ (\beta_1^{\mathrm{C}})^i\ [E_1,\ E_5])$$
$$\wedge$$
$$\Psi_{7d}\ (\beta_1^{\mathrm{C}}\ [E_5,\ E_b] \wedge \beta_2^{\mathrm{F}}\ [E_b,\ E_c],\ (\alpha_4^{\mathrm{F}})^o\ [E_5,\ E_c]) \qquad \square$$

Construction Ψ_{7d} divide a delay unit and then distributes it to a chain or a part chain. To the chain the construction equation 8.1.3.20 deals with its base case and the construction equation 8.1.3.21 deals with its step case.

Example 8.1.3.9. [for understanding Ψ_{7d}]:

$$\Psi_{7d}\ (\alpha_1^{\mathrm{C}}\ [E_1,\ E_2]\ \wedge \alpha_2^{\mathrm{F}}\ [E_2,\ E_3]\ \wedge \alpha_3^{\mathrm{C}}\ [E_3,\ E_4],\ \beta^i\ [E_1,\ E_4])$$
$$= (\alpha_1^{\mathrm{C}} \wedge \beta^i)\ [E_1,\ E_2]\ \wedge (\alpha_2^{\mathrm{F}} \wedge \beta^i)\ [E_2,\ E_3]\ \wedge (\alpha_3^{\mathrm{C}} \wedge \beta^i)\ [E_3,\ E_4] \qquad \square$$

Finally, let us give a synthetic example. We reconstruction the example (page 82).

Example 8.1.3.10. [synthetic one]:

$\Psi_7\ ((s_1\ y = f\ (g\ (r_1\ x)) \wedge s_2\ y = h\ (r_2\ x))\ [E_i,\ E_o])$ .... (1)

$= \Psi_{7m}\ (\Psi_{7s}\ ((s_1\ y = f\ (g\ (r_1\ x)) \wedge s_2\ y = h\ (r_2\ x))\ [E_i,\ E_o]))$ ....

.... by the construction equation 8.1.3.1

$= \Psi_{7m}\ (\Psi_{7l}\ ((s_1\ y = f\ (g\ (r_1\ x)))\ [E_i,\ E_o]),\ \Psi_{7l}\ ((s_2\ y = h\ (r_2\ x))\ [E_i,\ E_o]))$ . .

.... by the construction equation 8.1.3.2 and 3

$\Psi_{7l}\ ((s_1\ y = f\ (g\ (r_1\ x)))\ [E_i,\ E_o])$ .... (2)

$= \Psi_7\ ((i_2\ y = g\ (r_1\ x))\ [E_i,\ E_3]) \wedge$

$(o_2\ y = f\ (i_2\ x))\ [E_3,\ E_4] \wedge$

$(s_1\ y = o_2\ x)\ [E_4,\ E_o]$ .... by the construction equation 8.1.3.6

$\Psi_7\ ((i_2\ y = g\ (r_1\ x))\ [E_i,\ E_3])$ .... (3)

$= \Psi_{7m}\ (\Psi_{7s}\ ((i_2\ y = g\ (r_1\ x))\ [E_i,\ E_3]))$ . . by the construction equation 8.1.3.1

$= \Psi_{7m}\ (\Psi_{7l}\ ((i_2\ y = g\ (r_1\ x))\ [E_i,\ E_3]))$ . . by the construction equation 8.1.3.2

$\Psi_{7l}\ ((i_2\ y = g\ (r_1\ x))\ [E_i,\ E_3])$ . . . . (4)
$= \Psi_7\ ((i_1\ y = r_1\ x)\ [E_i,\ E_1]) \wedge$
$\quad (o_1\ y = g\ (i_1\ x))\ [E_1,\ E_2] \wedge$
$\quad (i_2\ y = o_1\ x)\ [E_2,\ E_3]$ . . . . by the construction equation 8.1.3.6

$\Psi_7\ (i_1\ y = r_1\ x)\ [E_i,\ E_1]$ . . . . (5)
$= \Psi_{7m}\ (\Psi_{7s}\ ((i_1\ y = r_1\ x)\ [E_i,\ E_1]))$ . . . . . by the construction equation 8.1.3.1
$= \Psi_{7m}\ (\Psi_{7l}\ ((i_1\ y = r_1\ x)\ [E_i,\ E_1]))$ . . . . . . by the construction equation 8.1.3.2
$= \Psi_{7m}\ ((i_1\ y = r_1\ x)\ [E_i,\ E_1])$ . . . . by the construction equation 8.1.3.4
$= (i_1\ y = r_1\ x)\ [E_i,\ E_1]$ . . . . by the construction equation 8.1.3.7

Going back to (4), we have

$\Psi_{7l}\ ((i_2\ y = g\ (r_1\ x))\ [E_i,\ E_3])$ . . . . $(4')$
$= (i_1\ y = r_1\ x)\ [E_i,\ E_1] \wedge$
$\quad (o_1\ y = g\ (i_1\ x))\ [E_1,\ E_2] \wedge$
$\quad (i_2\ y = o_1\ x)\ [E_2,\ E_3]$

Going back to (3), we get

$\Psi_7\ ((i_2\ y = g\ (r_1\ x))\ [E_i,\ E_3])$ . . . . $(3')$
$= \Psi_{7m}\ (\Psi_{7l}\ ((i_2\ y = g\ (r_1\ x))\ [E_i,\ E_3]))$
$= \Psi_{7m}\ ((i_1\ y = r_1\ x)\ [E_i,\ E_1]\ \wedge (o_1\ y = g\ (i_1\ x))\ [E_1,\ E_2] \wedge$
$\qquad (i_2\ y = o_1\ x)\ [E_2,\ E_3])$
$= (i_1\ y = r_1\ x)\ [E_i,\ E_1] \wedge$
$\quad (o_1\ y = g\ (i_1\ x))\ [E_1,\ E_2] \wedge$
$\quad (i_2\ y = o_1\ x)\ [E_2,\ E_3]$ . . . . by the construction equation 8.1.3.7

Going back to (2) we get

$\Psi_{7l}\ ((s_1\ y = f\ (g\ (r_1\ x)))\ [E_i,\ E_o])$ . . . . $(2')$
$= (i_1\ y = r_1\ x)\ [E_i,\ E_1] \wedge$
$\quad (o_1\ y = g\ (i_1\ x))\ [E_1,\ E_2] \wedge$
$\quad (i_2\ y = o_1\ x)\ [E_2,\ E_3] \wedge$

$$(o_2\ y = f\ (i_2\ x))\ [E_3,\ E_4] \wedge$$
$$(s_1\ y = o_2\ x)\ [E_4,\ E_o]$$

Similarly, we get

$$\Psi_{7l}\ ((s_2\ y = h\ (r_2\ x))\ [E_i,\ E_o]) \dots\dots (6)$$
$$= (i_3\ y = r_3\ x)\ [E_i,\ E_a] \wedge$$
$$(o_3\ y = h\ (i_3\ x))\ [E_a,\ E_b] \wedge$$
$$(s_2\ y = o_3\ x)\ [E_b,\ E_o]$$

Using $(2')$ and (6) and going back to (1) we have

$$\Psi_7\ ((s_1\ y = f\ (g\ (r_1\ x)) \wedge s_2\ y = h\ (r_2\ x))\ [E_i,\ E_o]) \dots\dots (1')$$
$$= \Psi_{7m}\ ((i_1\ y = r_1\ x)\ [E_i,\ E_1] \wedge$$
$$(o_1\ y = g\ (i_1\ x))\ [E_1,\ E_2] \wedge$$
$$(i_2\ y = o_1\ x)\ [E_2,\ E_3] \wedge$$
$$(o_2\ y = f\ (i_2\ x))\ [E_3,\ E_4] \wedge$$
$$(s_1\ y = o_2\ x)\ [E_4,\ E_o],$$
$$(i_3\ y = r_2\ x)\ [E_i,\ E_a] \wedge$$
$$(o_3\ y = h\ (i_3\ x))\ [E_a,\ E_b] \wedge$$
$$(s_2\ y = o_3\ x)\ [E_b,\ E_o])$$

As we select a device to implement both function g and h we use i and o to replace $i_1,\ i_3$ and $o_1,\ o_3$ respectively.

$$\Psi_7\ ((s_1\ y = f\ (g\ (r_1\ x)) \wedge s_2\ y = h\ (r_2\ x))\ [E_i,\ E_o]) \dots\dots (1'')$$
$$= \Psi_{7m}\ ((i\ y = r_1\ x)\ [E_i,\ E_1] \wedge$$
$$(o\ y = g\ (i\ x))\ [E_1,\ E_2] \wedge$$
$$(i_2\ y = o\ x)\ [E_2,\ E_3] \wedge$$
$$(o_2\ y = f\ (i_2\ x))\ [E_3,\ E_4] \wedge$$
$$(s_1\ y = o_2\ x)\ [E_4,\ E_o],$$
$$(i\ y = r_2\ x)\ [E_i,\ E_a] \wedge$$
$$(o\ y = h\ (i\ x))\ [E_a,\ E_b] \wedge$$
$$(s_2\ y = o\ x)\ [E_b,\ E_o])$$
$$= \Psi_{7f}\ ((i\ y = r_1\ x)\ [E_i,\ E_1] \wedge$$

$$(o\ y = g\ (i\ x))\ [E_1,\ E_2] \land$$
$$(i_2\ y = o\ x)\ [E_2,\ E_3] \land$$
$$(o_2\ y = f\ (i_2\ x))\ [E_3,\ E_4] \land$$
$$(s_1\ y = o_2\ x)\ [E_4,\ E_o],$$
$$(i\ y = r_2\ x)\ [E_i,\ E_a] \land$$
$$(o\ y = h\ (i\ x))\ [E_a,\ E_b] \land$$
$$(s_2\ y = o\ x)\ [E_b,\ E_o]) \text{ .... by the construction equation 8.1.3.8}$$
$$= \Psi_{7f}\ ((i\ y = r_1\ x)\ [E_i,\ E_1] \land (o\ y = g\ (i\ x))\ [E_1,\ E_2] \land (i_2\ y = o\ x)\ [E_2,\ E_3],$$
$$(i\ y = r_2\ x)\ [E_i,\ E_3]) \land$$
$$(o_2\ y = f\ (i_2\ x) \land o\ y = h\ (i\ x))\ [E_3,\ E_4] \land$$
$$\Psi_{7f}\ ((s_1\ y = o_2\ x)\ [E_4,\ E_o],\ (s_2\ y = o\ x)\ [E_4,\ E_o]) \text{ ....}$$
$$\text{..................................... by the construction equation 8.1.3.11}$$

$$\Psi_{7f}\ ((i\ y = r_1\ x)\ [E_i,\ E_1] \land (o\ y = g\ (i\ x))\ [E_1,\ E_2] \land (i_2\ y = o\ x)\ [E_2,\ E_3],$$
$$(i\ y = r_2\ x)\ [E_i,\ E_3]) \text{ .... (7)}$$
$$= \Psi_{7c}\ ((i\ y = r_1\ x)\ [E_i,\ E_1] \land (o\ y = g\ (i\ x))\ [E_1,\ E_2] \land (i_2\ y = o_1\ x)\ [E_2,\ E_3],$$
$$(i\ y = r_2\ x)\ [E_i,\ E_3]) \text{ .... by the construction equation 8.1.3.10}$$
$$= \Psi_{7e2}\ ((i\ y = r_1\ x)\ [E_i,\ E_1] \land (o\ y = g\ (i\ x))\ [E_1,\ E_2],\ \mathrm{T},\ r_2\ y = r_2\ x) \land$$
$$(i_2\ y = o\ x \land i\ y = r_2\ x)\ [E_2,\ E_3] \land$$
$$\Psi_{7c'}\ (\mathrm{T},\ \mathrm{T},\ \gamma) \text{ .... by the construction equation 8.1.3.13}$$

$$\Psi_{7e2}\ ((i\ y = r_1\ x)\ [E_i,\ E_1] \land (o\ y = g\ (i\ x))\ [E_1,\ E_2],\ \mathrm{T},\ r_2\ y = r_2\ x) \text{ ...... (8)}$$
$$= \Psi_{7d}\ ((i\ y = r_1\ x)\ [E_i,\ E_1] \land (o\ y = g\ (i\ x))\ [E_1,\ E_2],$$
$$(r_2\ y = r_2\ x)\ [E_i,\ E_2]) \text{ .... by the construction equation 8.1.3.18}$$
$$= (i\ y = r_1\ x \land r_2\ y = r_2\ x)\ [E_i,\ E_1] \land$$
$$(o\ y = g\ (i\ x) \land r_2\ y = r_2\ x)\ [E_1,\ E_2] \text{ ....}$$
$$\text{............................ by the construction equation 8.1.3.21 and 20}$$

$$\Psi_{7c'}\ (\mathrm{T},\ \mathrm{T},\ \gamma) \text{ .... (9)}$$
$$= \Psi_{7e2}\ (\mathrm{T},\ \mathrm{T},\ \gamma) \text{ .... by the construction equation 8.1.3.14}$$
$$= \mathrm{T} \text{ .... by the construction equation 8.1.3.17}$$

Using (8) and (9) and going back to (7) we get

$$\Psi_{7f}\ ((i\ y = r_1\ x)\ [E_i,\ E_1] \wedge (o\ y = g\ (i\ x))\ [E_1,\ E_2] \wedge (i_2\ y = o\ x)\ [E_2,\ E_3],$$
$$(i\ y = r_2\ x)\ [E_i,\ E_3]) \ldots\ldots (7')$$
$$= (i\ y = r_1\ x \wedge r_2\ y = r_2\ x)\ [E_i,\ E_1] \wedge$$
$$(o\ y = g\ (i\ x) \wedge r_2\ y = r_2\ x)\ [E_1,\ E_2] \wedge$$
$$(i_2\ y = o\ x \wedge i\ y = r_2\ x)\ [E_2,\ E_3]$$

$$\Psi_{7f}\ ((s_1\ y = o_2\ x)\ [E_4,\ E_o],\ (s_2\ y = o\ x)\ [E_4,\ E_o]) \ldots\ldots (10)$$
$$= \Psi_{7c}\ ((s_1\ y = o_2\ x)\ [E_4,\ E_o],\ (s_2\ y = o\ x)\ [E_4,\ E_o]) \ldots\ldots$$
$\ldots\ldots$ by the construction equation 8.1.3.10
$$= \Psi_{7e2}\ (\mathrm{T},\ \mathrm{T},\ \gamma) \wedge$$
$$(s_1\ y = o_2\ x \wedge s_2\ y = o\ x)\ [E_4,\ E_o] \wedge$$
$$\Psi_{7c'}\ (\mathrm{T},\ \mathrm{T},\ \gamma') \ldots\ldots$$ by the construction equation 8.1.3.13
$$= (s_1\ y = o_2\ x \wedge s_2\ y = o\ x)\ [E_4,\ E_o] \ldots\ldots$$
$\ldots\ldots$ by the construction equation 8.1.3.17 and 14

Using (7′) and (10) and going back to (1″) we get

$$\Psi_7\ ((s_1\ y = f\ (g\ (r_1\ x)) \wedge s_2\ y = h\ (r_2\ x))\ [E_i,\ E_o]) \ldots\ldots (1''')$$
$$= (i\ y = r_1\ x \wedge r_2\ y = r_2\ x)\ [E_i,\ E_1] \wedge$$
$$(o\ y = g\ (i\ x) \wedge r_2\ y = r_2\ x)\ [E_1,\ E_2] \wedge$$
$$(i_2\ y = o\ x \wedge i\ y = r_2\ x)\ [E_2,\ E_3] \wedge$$
$$(o_2\ y = f\ (i_2\ x) \wedge o\ y = h\ (i\ x))\ [E_3,\ E_4] \wedge$$
$$(s_1\ y = o_2\ x \wedge s_2\ y = o\ x)\ [E_4,\ E_o] \qquad \square$$

8.1.4 Specification Scheme 7

The scheme 7 obtained through the construction 7 has three type structures: function link, connection link and branch structure.

Specification Scheme 7:

$E\ 0 \wedge fcms$

Function unit and connection model set $fcms$:

$$
\begin{array}{lcl}
fcms & ::= & fcm \\
 & & \mid fcm \wedge fcms \\
fcm & ::= & (fus)\ [E_i,\ E_o] \\
 & & \mid (cus)\ [E_i,\ E_o] \\
 & & \mid brs \\
fus & ::= & fu \\
 & & \mid fu \wedge fus \\
cus & ::= & cu \\
 & & \mid cu \wedge cus \\
fu & ::= & o\ y \overset{\text{a}}{=} f\ (i_1\ x, \cdots .i_n\ x) \\
 & & \mid o\ y \overset{\text{a}}{=} c \\
 & & \mid s\ y = s\ x \\
cu & ::= & s\ y = r\ x \\
 & & \mid s\ y = s\ x
\end{array}
$$

Abbreviation:

$fcms$	: function unit and connection model set
fcm	: function unit and connection model
brs	: branch structure
fus	: function unit set
cus	: connection unit set
fu	: function unit
	: constant as function
	: delay unit
cu	: connection unit
	: delay unit

Lemma 8.1.4.1. [different logic boundary and time intervals]:

In the scheme 7 there are not two linear chains which have the same head node or the same time interval.

Proof:

By the theorem 7.1 (page 73), the corollary 7.1.1 (page 73) there is not two term transitions which have the same entry predicates or the same time interval. And by the construction 7 a linear chain keeps its head and tail nodes being the same as the entry and leaving predicates of the term transition model ttm from which the chain is constructed. □

Theorem 8.1.4.1. [uniqueness of entry time point and time interval of model at link level]:

Proof:

By the scheme 7 a model at link level is one among function link, connection link or branch structure. By the lemma 8.1.4.1 we have had that any two linear chains have their head nodes different therefore their entry time points different and their time intervals different. By the construction 7 we further have that any two links have different entry time point and different time intervals. □

8.2 Totalness and Correctness of Construction 7

Theorem 8.2.1. [totalness of construction 7]:

$\vdash \forall s : \mathcal{S}_{6'}.\ \Psi_7\ (s) \in \mathcal{S}_7$

Proof:

The constructions of the construction 7 can be mainly grouped into two classes: Ψ_{7l} for decomposition and Ψ_{7m}, Ψ_{7s}, Ψ_{7f}, Ψ_{7c}, $\Psi_{7c'}$, Ψ_{7e1}, Ψ_{7e2} and Ψ_{7d} for merging two models to get a model. Through Ψ_{7l} function unit and connection unit models are introduced and through other constructions we introduce delay unit model and merge the three type models to get function links and connection links. Branch structures are left unchanged by the construction 7. Therefore for every term transition model of the scheme $6'$ by the construction 7 we certainly get a linear chain which belongs to the scheme 7. □

Lemma 8.2.1. [correctness of Ψ_{7m}]:

If for each term transition tt we have a linear chain $\Psi_{7l}\ ((tt)\ [E_i,\ E_o]$ and

$\vdash \Psi_{7l}\ ((tt)\ [E_i,\ E_o]) \Rightarrow (tt)\ [E_i,\ E_o]$

then we have

$$\vdash \Psi_{7m}\ (\Psi_{7l}\ ((tt_1)\ [E_i,\ E_o]), \cdots, \Psi_{7l}\ ((tt_n)\ [E_i,\ E_o]))$$
$$\Rightarrow (tt_1 \wedge \cdots \wedge tt_n)\ [E_i,\ E_o].$$

Proof:

The lemma assumes that each $\Psi_{7l}\ ((tt_k)\ [E_i,\ E_o])$ is a linear chain. For Ψ_{7m} if there is one linear chain argument then, by the construction equation 8.1.3.7, conclusion is obvious. For the case of two or more chains Ψ_{7m} recursively reduces each two of them to Ψ_{7f} to treat. Therefore the construction is reduced how to merge two linear chains to get a new linear chain.

In the construction 7 there is a construction sequence from Ψ_{7f} to Ψ_{7c}, $\Psi_{7c'}$, Ψ_{te1}, Ψ_{7e2} and Ψ_{7d}. In the construction sequence each function or connection unit in the two given linear chains is rearranged to form or into a function link or a connection link. In these constructions some entry predicates may be re-named or some new entry predicates can be introduced and some new delay units possibly are introduced. All these constructions finally lead to a new linear chain merged from the two given linear chains.

What are changed are only entry predicate. What are newly introduced are only delay units. All function units and connection units of the two given linear chains are kept unchanged. Therefore, we can get the conclusion of this lemma.

□

Theorem 8.2.2. [correctness of construction 7]:

$\Psi_7\ ((tts)\ [E_i,\ E_o])$ is a linear chain and

$$\vdash \forall tts : \mathcal{S}_{6'}.\ \Psi_7\ ((tts)\ [E_i,\ E_o]) \Rightarrow (tts)\ [E_i,\ E_o]$$

Proof:

For an arbitrarily given term transition model $(tts)\ [E_i,\ E_o]$ let us assume that its term transition set tts contains n $(n \geq 1)$ term transitions tt. By the construction Ψ_{7s} and Ψ_{7l} we have

$$\Psi_7\ ((tt_1 \wedge \cdots \wedge tt_n)\ [E_i,\ E_o])$$
$$= \Psi_{7m}\ (\Psi_{7l}\ ((tt_1\ [E_i,\ E_o]), \cdots, \Psi_{7l}\ ((tt_n\ [E_i,\ E_o])$$

For each $\Psi_{7l}\ ((tt_k\ [E_i,\ E_o])$ $(1 \leq k \leq n)$ we can prove it is a linear chain and have

$$\vdash \Psi_{7l}\ ((tt_k)\ [E_i,\ E_o]) \Rightarrow (tt_k)\ [E_i,\ E_o]$$

By the structure of tt_k there are three cases:

(1). $tt_k = (s\ y = r\ x)$: in this case by the construction equation 8.1.3.4 we have

$$\Psi_{7l}\ ((s\ y = r\ x)\ [E_i,\ E_o]) = (s\ y = r\ x)\ [E_i,\ E_o]$$

The $(s\ y = r\ x)\ [E_i,\ E_o]$ is a linear chain and obviously we have

$$\vdash \Psi_{7l}\ ((tt_k)\ [E_i,\ E_o]) \Rightarrow (tt_k)\ [E_i,\ E_o]$$

(2). $tt_k = (s\ y = c)$: in this case by the construction equation 8.1.3.5 we have

$$\Psi_{7l}\ ((s\ y = c)\ [E_i,\ E_o])$$
$$= (s\ y = o\ x)\ [E_i,\ E_a] \wedge (o\ y = c)\ [E_a,\ E_o] \wedge \mathrm{T}\ [E_o,\ E_o]$$

"$(s\ y = o\ x)\ [E_i,\ E_a] \wedge (o\ y = c)\ [E_a,\ E_o] \wedge \mathrm{T}\ [E_o,\ E_o]$" is a linear chain and obviously we have

$$\vdash \Psi_{7l}\ ((tt_k)\ [E_i,\ E_o]) \Rightarrow (tt_k)\ [E_i,\ E_o]$$

(3). $tt_k = (s\ y = f\ (tm_1, \cdots, tm_m))$: in this case by the construction equation 8.1.3.6 we have

$$\Psi_{7l}\ ((s\ y = f\ (tm_1, \cdots, tm_m)\ [E_i,\ E_o])$$
$$= \Psi_7\ ((i_1\ y = tm_1 \wedge \cdots \wedge i_m\ y = tm_m)\ [E_i,\ E_a]) \wedge$$
$$(o\ y \stackrel{\text{a}}{=} f\ (i_1\ x, \cdots, i_m\ x))\ [E_a,\ E_b] \wedge$$
$$(s\ y = o\ x)\ [E_b,\ E_o]$$

By the induction hypothesis we have that

$$\Psi_7\ ((i_1\ y = tm_1 \wedge \cdots \wedge i_m\ y = tm_m)\ [E_i,\ E_a])$$

is a linear chain. Hence,

$$\Psi_{7l}\ ((s\ y = f\ (tm_1, \cdots, tm_m)\ [E_i,\ E_o])$$

also is a linear chain. By the induction hypothesis we also have

$$\vdash \Psi_7\ ((i_1\ y = tm_1 \wedge \cdots \wedge i_m\ y = tm_m)\ [E_i,\ E_a])$$
$$\Rightarrow (i_1\ y = tm_1 \wedge \cdots \wedge i_m\ y = tm_m)\ [E_i,\ E_a]$$

Hence, we have

$$\vdash \Psi_{7l}\ ((s\ y = f\ (tm_1, \cdots, tm_m)\ [E_i,\ E_o])$$
$$\Rightarrow$$
$$(i_1\ y = tm_1 \wedge \cdots \wedge i_m\ y = tm_m)\ [E_i,\ E_a] \wedge$$
$$(o\ y \stackrel{\text{a}}{=} f\ (i_1\ x, \cdots, i_m\ x))\ [E_a,\ E_b] \wedge$$

$$(s\ y = o\ x)\ [E_b,\ E_o]$$
$$\Rightarrow$$
$$(s\ y = f\ (tm_1, \cdots, tm_m)\ [E_i,\ E_o]$$

Therefore, in this case we still have

$$\vdash \Psi_{7l}\ ((tt_k)\ [E_i,\ E_o]) \Rightarrow (tt_k)\ [E_i,\ E_o]$$

By the lemma 8.2.1 we get

$$\vdash \Psi_{7m}\ (\Psi_{7l}\ ((tt_1)\ [E_i,\ E_o]), \cdots, \Psi_{7l}\ ((tt_n)\ [E_i,\ E_o]))$$
$$\Rightarrow (tt_1 \wedge \cdots \wedge tt_n)\ [E_i,\ E_o].$$

Finally, we get

$$\vdash \Psi_7\ ((tt_1 \wedge \cdots \wedge tt_n)\ [E_i,\ E_o]) \Rightarrow (tt_1 \wedge \cdots \wedge tt_n)\ [E_i,\ E_o] \qquad \square$$

Chapter 9

Introducing Register, Bus and Microinstruction

In this chapter we deal with the linear chain of the scheme 7. We discuss why and how to introduce bus and register. And when and where we need to introduce clock and microprogram. As usually we start with an example to help reader to catch the basic ideal. Let us continue the construction of the example of the last chapter (page 82). In the example we have got (page 84):

$$(i\ y = r_1\ x \wedge r_2\ y = r_2\ x)\ [E_i,\ E_1] \tag{d1}$$

$\wedge$

$$(o\ y = g\ (i\ x) \wedge r_2\ y = r_2\ x)\ [E_1,\ E_2] \tag{d2}$$

$\wedge$

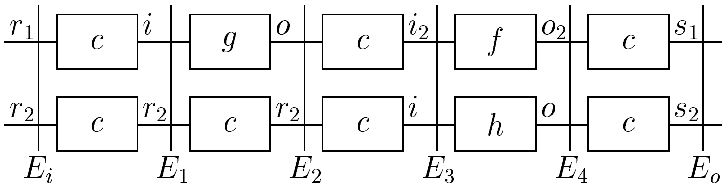
$$(i_2\ y = o\ x \wedge i\ y = r_2\ x)\ [E_2,\ E_3] \tag{d3}$$

$\wedge$

$$(o_2\ y = f\ (i_2\ x) \wedge o\ y = h\ (i\ x))\ [E_3,\ E_4] \tag{d4}$$

$\wedge$

$$(s_1\ y = o_2\ x \wedge s_2\ y = o\ x)\ [E_4,\ E_o] \tag{d5}$$



We can see that in the linear chain there are two connections:

$i\ y = r_1\ x$ of (d1)

$i\ y = r_2\ x$ of (d3)

that have the same output i. For this we need to introduce a bus, therefore, we need to change them to

$i\ y = r_1\ y\ \wedge r_1\ y = r_1\ x$

$i\ y = r_2\ y\ \wedge r_2\ y = r_2\ x$

and introduce a constraint $y > x$ to add to (d2) to guarantee that the two connections to be separated. We get

$$(\underline{i\ y = r_1\ y} \wedge r_1\ y = r_1\ x \wedge r_2\ y = r_2\ x)\ [E_i,\ E_1] \tag{e1}$$

$$\wedge$$

$$(o\ y = g\ (i\ x) \wedge r_2\ y = r_2\ x \wedge\ \underline{y > x})\ [E_1,\ E_2] \tag{e2}$$

$$\wedge$$

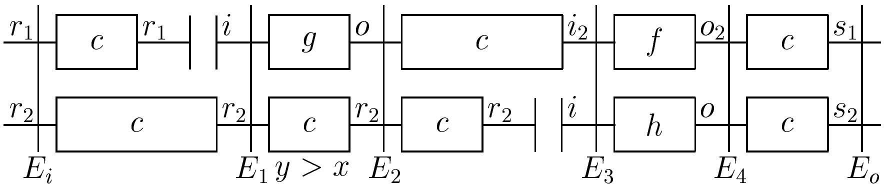
$$(i_2\ y = o\ x \wedge\ \underline{i\ y = r_2\ y} \wedge r_2\ y = r_2\ x)\ [E_2,\ E_3] \tag{e3}$$

$$\wedge$$

$$(o_2\ y = f\ (i_2\ x) \wedge o\ y = h\ (i\ x))\ [E_3,\ E_4] \tag{e4}$$

$$\wedge$$

$$(s_1\ y = o_2\ x \wedge s_2\ y = o\ x)\ [E_4,\ E_o] \tag{e5}$$



In the next step we eliminate unnecessary connections. For (e1) use x to replace y and E_i to replace E_1 to get (f1); for (e2) we use $s\ y = r_2\ x$ to replace $r_2\ y = r_2\ x$, s_2 to replace o and E_i to replace E_1 to get (f2); for (e3) use x to replace y, s to replace r_2, s_2 to replace o and i_2, and E_2 to replace E_3 to get (f3); for (e4) use s_1 to replace o_2, s_2 to replace o and i_2, and finally, E_2 and E_o to replace E_3 and E_4 respectively to get (f4); for (e5) use x to replace y, s_1 to replace o_2, s_2 to replace

o and E_o to replace E_4 to get (f5).

$$(\underline{i\ x = r_1\ x})\ [E_i,\ E_i] \tag{f1}$$

$\wedge$

$$(s_2\ y = g\ (i\ x) \wedge s\ y = r_2\ x \wedge\ \underline{y > x})\ [E_i,\ E_2] \tag{f2}$$

$\wedge$

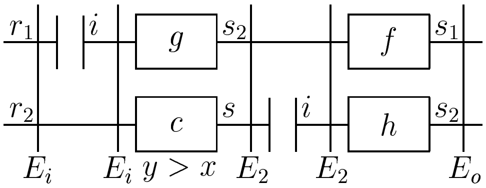
$$(\ \underline{i\ x = s\ x})\ \ [E_2,\ E_2] \tag{f3}$$

$\wedge$

$$(s_1\ y = f\ (s_2\ x) \wedge s_2\ y = h\ (i\ x))\ [E_2,\ E_o] \tag{f4}$$

$\wedge$

$$(s_1\ x = s_1\ x \wedge s_2\ x = s_2\ x)\ [E_o,\ E_o] \tag{f5}$$



To suit the realistic device group of gate-bus for (f1) and (f3) we use

$$i\ x = r_1'\ x \wedge r_1'\ x = r_1\ x$$

$$i\ x = s'\ x \wedge s'\ x = s\ x$$

to replace

$$i\ x = r_1\ x$$

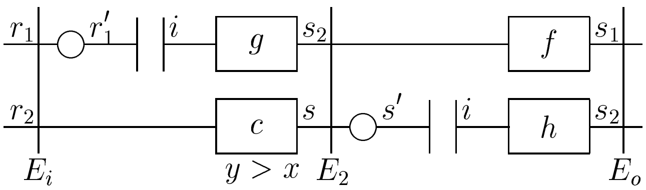
$$i\ x = s\ x$$

Then merge (f1) into (f2), (f3) into (f4). Finally, eliminate (f5) we get:

$$(s_2\ y = g\ (i\ x) \wedge i\ x = r_1'\ x \wedge r_1'\ x = r_1\ x\ \wedge\ s\ y = r_2\ x \wedge y > x)\ [E_i,\ E_2] \tag{g1}$$

$\wedge$

$$(s_1\ y = f\ (s_2\ x) \wedge s_2\ y = h\ (i\ x)) \wedge i\ x = s'\ x \wedge s'\ x = s\ x)\ [E_2,\ E_o] \tag{g2}$$



To get an implementation we need to introduce devices:

$$\mathrm{D}_{gh}\ (\mathrm{c}_g,\ \mathrm{c}_h,\ \mathrm{i},\ \mathrm{s}_2) \equiv \forall t.\ \mathrm{s}_2\ (t+1) = (\mathrm{c}_g\ t \Rightarrow g\ (\mathrm{i}\ t) \downarrow (\mathrm{c}_h\ t \Rightarrow h\ (\mathrm{i}\ t) \downarrow \mathrm{i}\ t))$$

$$\mathrm{D}_f\ (\mathrm{c}_f,\ \mathrm{s}_2,\ \mathrm{s}_1) \equiv \forall t.\ \mathrm{s}_1\ (t+1) = (\mathrm{c}_f\ t \Rightarrow f\ (\mathrm{s}_2\ t) \downarrow \mathrm{s}_2\ t))$$

$$\mathrm{Reg}\ (\mathrm{c}_r,\ \mathrm{r}_2,\ \mathrm{s}) \equiv \forall t.\ \mathrm{s}\ (t+1) = (\mathrm{c}_r\ t \Rightarrow \mathrm{r}_2\ t \downarrow \mathrm{s}\ t)$$

$$\mathrm{Gate}\ (\mathrm{c}_{b1},\ \mathrm{r}_1,\ \mathrm{r}_1') \equiv \forall t.\ \mathrm{r}_1'\ t = (\mathrm{c}_{b1}\ t \Rightarrow \mathrm{Mk\_Tri}\ (\mathrm{r}_1\ t) \downarrow \mathrm{Float})$$

$$\mathrm{Gate}\ (\mathrm{c}_{b2},\ \mathrm{s},\ \mathrm{s}') \equiv \forall t.\ \mathrm{s}'\ t = (\mathrm{c}_{b2}\ t \Rightarrow \mathrm{Mk\_Tri}\ (\mathrm{s}\ t) \downarrow \mathrm{Float})$$

$$\mathrm{Bus}\ (\mathrm{r}_1',\ \mathrm{s}',\ \mathrm{i}) \equiv \forall t.\ \mathrm{i}\ t = \mathrm{Dest\_Tri}\ (\mathrm{r}_1'\ t \cup \mathrm{s}'\ t)$$

We can get the piece of an abstract microprogram, its two microinstruction:

$$(\forall x.\ E_i\ x \Rightarrow \mathrm{c}_{b1}\ x \wedge \mathrm{c}_g\ x \wedge \mathrm{c}_r\ x \wedge E_2\ (x+1))$$

$$\wedge$$

$$(\forall x.\ E_2\ x \Rightarrow \mathrm{c}_{b2}\ x \wedge \mathrm{c}_h\ x \wedge \mathrm{c}_f\ x \wedge E_o\ (x+1))$$

and a datapath:

$$\mathrm{Reg}\ (\mathrm{c}_r,\ \mathrm{r}_2,\ \mathrm{s}) \wedge \mathrm{Gate}\ (\mathrm{c}_{b1},\ \mathrm{r}_1,\ \mathrm{r}_1') \wedge \mathrm{Gate}\ (\mathrm{c}_{b2},\ \mathrm{s},\ \mathrm{s}') \wedge \mathrm{Bus}\ (\mathrm{r}_1',\ \mathrm{s}',\ \mathrm{i}) \wedge$$
$$\mathrm{D}_{gh}\ (\mathrm{c}_g,\ \mathrm{c}_h,\ \mathrm{i},\ \mathrm{s}_2) \wedge \mathrm{D}_f\ (\mathrm{c}_f,\ \mathrm{s}_2,\ \mathrm{s}_1)$$

We can further instantiate the piece of abstract microprogram to a concrete microprogram, for example:

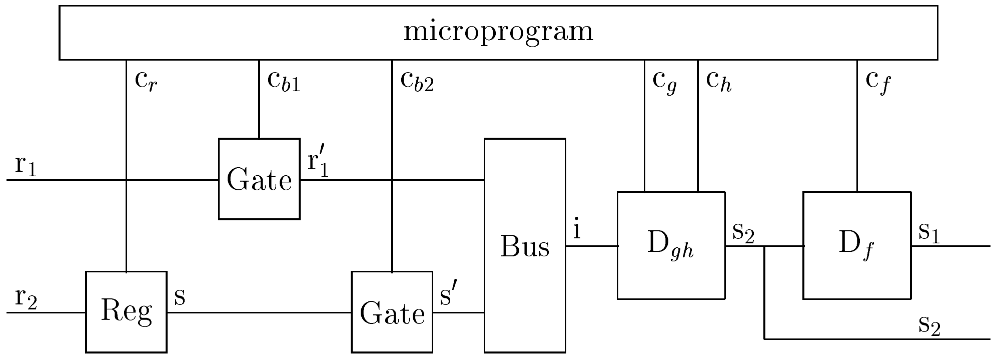
$$(\forall x.\ \mathrm{address}\ x = 11 \Rightarrow \mathrm{c}_{b1}\ x \wedge \mathrm{c}_g\ x \wedge \mathrm{c}_r\ x \wedge \mathrm{address}\ (x+1) = 12)$$

$$\wedge$$

$$(\forall x.\ \mathrm{address}\ x = 12 \Rightarrow \mathrm{c}_{b2}\ x \wedge \mathrm{c}_h\ x \wedge \mathrm{c}_f\ x \wedge \mathrm{address}\ (x+1) = 13)$$

Finally, we get an implementation:

Implementation 1:



In summary, starting from the specification 1 (page 82), we follow a nearly mechanical process to derive an implementation as above. In the process two important selections are allocation and scheduling. By allocation we mean that we select a device D_{gh} to implement two function units g and h and another device for f and by scheduling we mean that we put f and h into the same time interval to implement. Mainly, the selection influence and lead to the implementation: its microprogram and datapath.

Let us introduce other two different selections which lead to quite different implementations.

If we use a device D_{fgh} to implement the three functions f, g and h then we will derive a piece of microprogram and a datapath as below. (Note: in microprogram we always omit the negative control signal, for example, the second microinstruction includes a $\neg(c_{r1}\ x)$.)

$(\forall x.\ \text{address}\ x = 5 \Rightarrow c_{b1}\ x \wedge c_g\ x \wedge c_{r1}\ x \wedge \text{address}\ (x+1) = 6)$

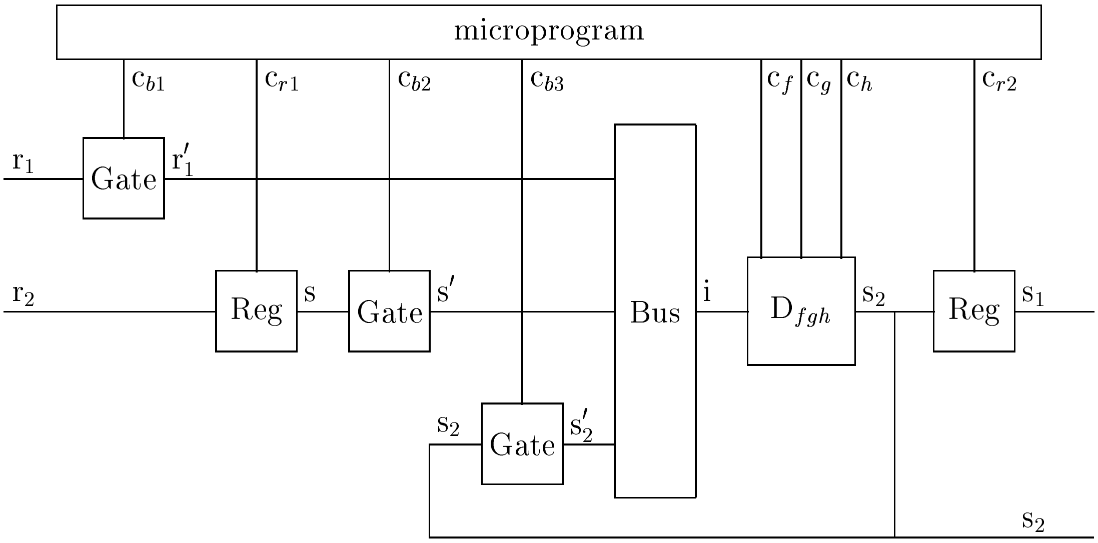
$\wedge$

$(\forall x.\ \text{address}\ x = 6 \Rightarrow c_{b3}\ x \wedge c_f\ x \wedge \text{address}\ (x+1) = 7)$

$\wedge$

$(\forall x.\ \text{address}\ x = 7 \Rightarrow c_{r2}\ x \wedge c_{b2}\ x \wedge c_h\ x \wedge \text{address}\ (x+1) = 8)$

Implementation 2:



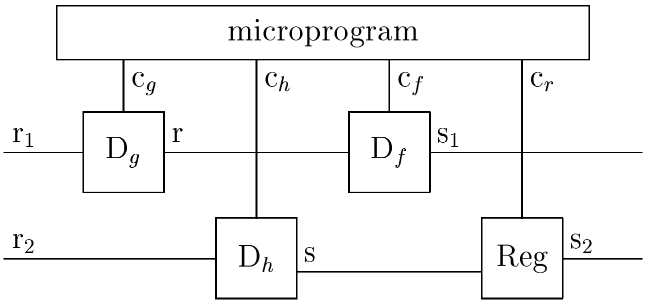
If we use three devices for three function unit, namely a device is used for a function unit, then no bus is needed and we have a clearer microprogram and datapath as below.

$(\forall x.\ \text{address}\ x = 3 \Rightarrow \mathrm{c}_g\ x \wedge \mathrm{c}_h\ x \wedge \text{address}\ (x+1) = 4)$

$\wedge$

$(\forall x.\ \text{address}\ x = 4 \Rightarrow \mathrm{c}_f\ x \wedge \mathrm{c}_r\ x \wedge \text{address}\ (x+1) = 5)$

Implementation 3:



9.1 Introducing Bus, Register and Connection

For a specification at the scheme 7 level we present a general and nearly mechanical process to introduce bus, register and connection.

9.1.1 Introducing Bus

A bus is needed if in the linear chains of the scheme 7 there are two or more connection units which have the same output. In the case we need a device which has the same output as its output and all the inputs of these connection units as its inputs. The device is called "bus".

For any two connection units of a bus if they are in the same linear chain (we need not consider the case that they are in the different linear chains by the theorem 8.1.4.1) then we need to introduce a constraint: that in the all links between the two connection links, which contain the two connection units respectively, there is at least a link which is added with a time constraint:

$x < y$

A skill is that we often select to put the time constraint to the function link as, usually, a function unit needs a time interval to be executed, therefore, the constraint is automatically satisfied.

We need to introduce the constraint otherwise a danger inconsistency can be raised. For example, two connection units in two different connection links have the same output:

$s\ y = r_1\ x$

$s\ y = r_2\ x$

but, if without the time constraint then it is quite possible that they are finally merged into a link whose time interval is zero (y is replaced by x (see the definition

8.1.2.2 (page 86)) and whose specification body contains an inconsistency:

$s\ x = r_1\ x \wedge s\ x = r_2\ x$

If in the scheme 7 there are not two connection units having the same output then the device bus will be unnecessary. However, this means that we have enough devices such that each function unit is assigned to a unique device to implement it (Note: even the same function unit but if they are at different position in the scheme 7 then they are implemented by difference devices).

Here, we need to distinguish two different concepts: device kind and device which should be more exactly called as device instance. Let us use an example. When we write

$\mathrm{Reg}\ (c,\ i,\ o) \equiv \forall t.\ o\ (t+1) = c\ t \Rightarrow i\ t \downarrow o\ t$

we mean a device kind, the register kind. Its ports c, i and o are free-variables. When we use the concrete constants to instantiate the free-variable ports of a device kind we get a device (device instance). For example in the implementation 1 (page 113) we introduce a register instance

$\mathrm{Reg}\ (\mathrm{c}_r,\ \mathrm{r}_2,\ \mathrm{s})$

where, the free-variables c, i and o are replaced by c_r, r_2, s. So, two register instances if they have different ports then are viewed as different registers.

Let us introduce definitions to exactly describe bus.

Definition 9.1.1.1. [exclusive connection units]:

Two connection units are exclusive from each other if they are in the different linear chains or in the same chain but having had the time constraint that:

In the all links between the two connection links, which contain the two connection units respectively, there is at least a link which is added with a time constraint: $x < y$ □

Definition 9.1.1.2. [minimal bus]:

In the scheme 7 if two or more connection units having the same output and different inputs and any two of them are exclusive from each other then we put these connection units together and use a device to implement them. We call the device the minimal bus which has the same output as its output and all the inputs of these connection units as its inputs. □

We read "minimal bus" as that those connection units must be collected together (through the exclusion constraint (definition 9.1.1.1)) to form a device and we cannot separate them (this will possibly lead to an inconsistency). However, we can artificially further put two or more minimal bus and other connection units together to form a bus. In such case we call it the compound bus.

Definition 9.1.1.3. [compound bus]:

In the scheme 7 for two or more minimal buses or connection units which do not belong to minimal bus we can put all their connection units together to form a compound bus if we meanwhile constrain that any two connection units to be exclusive from each other. □

However, in the case of introducing a compound bus a trouble is that we need unify the outputs of all the connection units of the compound bus to get one output as the output of the compound bus. And we need to introduce new connection unit to connect the output of the compound bus to the original port which is unified to the output of the compound bus. For this we must go back to the beginning of the construction 7, in fact, in such a case we need to make the selection for a compound bus at the beginning of the construction 7.

In the real world, for example following the way [50] (page 10 and 11), to implement a bus we need to introduce two kind devices:

$$\text{Gate}\ (c,\ i,\ o) \equiv \forall t.\ o\,t = (c\,t \Rightarrow \text{Mk\_Tri}\ (i\ t) \downarrow \text{Float}) \quad \text{(gs)}$$

$$\text{Bus}\ (i_1,\ i_2, \cdots, i_n,\ o) \equiv \forall t.\ o\,t = \text{Dest\_Tri}\ (i_1\ t \cup i_2\ t \cup \cdots \cup i_n\ t) \quad \text{(bs)}$$

and the two axioms that

$$\vdash \forall s.\ \text{Dest\_Tri}\ (\text{Mk\_Tri}\ s) = s \qquad (s \text{ is a word}) \quad \text{(dmt)}$$

$$\vdash \forall s.\ (\text{Float} \cup s = s) \wedge (s \cup \text{Float} = s) \qquad (s \text{ is a tri\_state}) \quad \text{(bo)}$$

(see page 9 [50] for details)

For this we reconstruct each connection unit in a bus:

$$s\ y = r\ x$$

to three parts:

$$s\ y = r'\ y \quad \text{(c1)}$$

$$\wedge$$

$$r'\ y = r\ y \quad \text{(c2)}$$

$$\wedge$$

$$r\ y = r\ x \quad \text{(c3)}$$

Here, (c1) and (c2) are executed by a Bus and a Gate as above respectively.

Generally, for a case of bus formed as

$$\cdots \wedge\ (\Gamma_{1l} \wedge s\ y = r_1\ x \wedge \Gamma_{1r})\ [E_a,\ E_b] \wedge$$
$$\cdots \wedge\ (\Gamma_{2l} \wedge s\ y = r_2\ x \wedge \Gamma_{2r})\ [E_c,\ E_d] \wedge$$
$$\cdots \wedge\ (\Gamma_{nl} \wedge s\ y = r_n\ x \wedge \Gamma_{nr})\ [E_e,\ E_f]\ \wedge \cdots$$

Here, Γ_{il} and Γ_{ir} $(1 \leq i \leq n)$ represent the conjunction of zero or more specification entities.

We first transform it to

$$\cdots \wedge\ (\Gamma_{1l} \wedge s\ y = r_1'\ y \wedge r_1'\ y = r_1\ y \wedge r_1\ y = r_1\ x \wedge \Gamma_{1r})\ [E_a,\ E_b] \wedge$$
$$\cdots \wedge\ (\Gamma_{2l} \wedge s\ y = r_2'\ y \wedge r_2'\ y = r_2\ y \wedge r_2\ y = r_2\ x \wedge \Gamma_{2r})\ [E_c,\ E_d] \wedge$$
$$\cdots \wedge\ (\Gamma_{nl} \wedge s\ y = r_n'\ y \wedge r_n'\ y = r_n\ y \wedge r_n\ y = r_n\ x \wedge \Gamma_{nr})\ [E_e,\ E_f]\ \wedge \cdots$$

Then we have an implementation formed as

$$\cdots \wedge\ (\Gamma_{1l} \wedge c_{g1}\ y \wedge r_1\ y = r_1\ x \wedge \Gamma_{1r})\ [E_a,\ E_b] \wedge$$
$$\cdots \wedge\ (\Gamma_{2l} \wedge c_{g2}\ y \wedge r_2\ y = r_2\ x \wedge \Gamma_{2r})\ [E_c,\ E_d] \wedge$$
$$\cdots \wedge\ (\Gamma_{nl} \wedge c_{gn}\ y \wedge r_n\ y = r_n\ x \wedge \Gamma_{nr})\ [E_e,\ E_f] \wedge$$
$$\cdots \wedge\ \mathrm{Gate}_1\ (c_{g1}\ r_1,\ r_1') \wedge \mathrm{Gate}_2\ (c_{g2}\ r_2,\ r_2') \wedge \cdots \wedge \mathrm{Gate}_n\ (c_{gn}\ r_n,\ r_n') \wedge$$
$$\cdots \wedge\ \mathrm{Bus}\ (r_1',\ r_2', \cdots, r_n',\ s)\ \wedge \cdots$$

9.1.2 Introducing Register

Similar to the adding the time constraint for introducing bus we also need to add time constraint into the linear chain to avoid those possible inconsistencies incurred by using the same device to implement two or more function units. (In the case it is possible that these function units have the same output).

Definition 9.1.2.1. [exclusive function units]:

Two function units are exclusive from each other if they are in the different linear chains or in the same chain but having had the time constraint that:

In the all links between the two function links, which contain the two function units respectively, there is at least a link which is added with a time constraint:

$x < y$ □

With the definition we need to put the time constraint into a linear chain if it has two or more exclusive function units.

The third case where we need to introduce a time constraint $x < y$ is to satisfy the requirement from the convenience 5.3.1.1 (page 33).

A possible case is that a function unit, for example,

$o\ (x + 1) = f\ (i_1\ x,\ i_2\ x)$

automatically imply a time constraint

$x < x + 1.$

Check-for-Register 9.1.2.1:

In the case that in a connection link there are a connection unit with the time constraint:

$s\ y = r\ x \wedge x < y$

The connection unit is selected to be implemented by a register. □

We introduce a register Reg $(c,\ r,\ s)$ whose kind form is:

Reg $(c,\ i,\ o) \equiv \forall t.\ o\ (t + 1) = (c\ t \Rightarrow i\ t \downarrow o\ t)$

Then a connection link formed as

$(\Gamma_l \wedge s\ y = r\ x \wedge x < y \wedge \Gamma_r)\ [E_i,\ E_o]$

Here, we use Γ_l and Γ_r to represent the conjunction of zero or more specification entities.

will lead to an implementation formed as:

$(\Gamma_l \wedge c\ x \wedge x < y \wedge \Gamma_r)\ [E_i,\ E_o] \wedge \text{Reg}\ (c,\ r,\ s)$

However, a special case is that if in a model formed as

$(s\ y = r\ x)\ [E_i,\ E_o]$

s has been established as the output of a register then we decompose the model to two models:

$(i\ y = r\ x)\ [E_i,\ E_m]$

$(s\ y = i\ x)\ [E_m,\ E_o]$

Here, i is the input of the register. **Check-and-Change-the-Chain-Containing-Delay-Sequence 9.1.2.2:**

In the case that there is a chain or part chain which has form:

$(\Gamma_{1l} \wedge r\ y = r\ x \wedge x < y \wedge \Gamma_{1r})\ [E_1,\ E_2]$ .... (cc1)

$\wedge$

$(\Gamma_{2l} \wedge r\ y = r\ x \wedge \Gamma_{2r})\ [E_2,\ E_3] \wedge \cdots \wedge$

$(\Gamma_{nl} \wedge r\ y = r\ x \wedge \Gamma_{nr})\ [E_n,\ E_{n+1}]$

$(n \geq 1)$

Here, Γ_{il} and Γ_{ir} $(1 \leq i \leq n)$ are the conjunction of zero or more specification entities. (cc1) is the first connection link in the chain, which contains a time constraint $x < y$.

we change it to

$(\Gamma_{1l} \wedge s\ y = r\ x \wedge x < y \wedge \Gamma_{1r})\ [E_1,\ E_2] \wedge$

$(\Gamma_{2l} \wedge s\ y = s\ x \wedge \Gamma_{2r})\ [E_2,\ E_3] \wedge \cdots \wedge$

$(\Gamma_{nl} \wedge s\ y = s\ x \wedge \Gamma_{nr})\ [E_n,\ E_{n+1}]$

and use the substitution $[s/r]$ to the link which has the r. □

We need the change as we have to use register to implement these delays. In the case it can be implemented by

$(\Gamma_{1l} \wedge c\ x \wedge \Gamma_{1r})\ [E_1,\ E_2] \wedge$
$(\Gamma_{2l} \wedge s\ y = s\ x \wedge \Gamma_{2r})\ [E_2,\ E_3] \wedge \cdots \wedge$
$(\Gamma_{nl} \wedge s\ y = s\ x \wedge \Gamma_{nr})\ [E_n,\ E_{n+1}] \wedge$
$\mathrm{Reg}\ (c,\ r,\ s)$

A special case if the r has been established as the output of a register then we need to do nothing.

9.1.3 Introducing Connection

For a connection unit in a connection link formed as

$(\Gamma_l \wedge s\ y = r\ x \wedge \Gamma_r)\ [E_i,\ E_o],$

here, Γ_l and Γ_r represent the conjunction of zero or more connection units and zero or more delay units.

if there is not a time constraint $x < y$ we can use the time point x to replace the time point y, select E_i (E_o) to replace E_o (E_i) and select r (s) to replace s (r). Therefore we can get, for example,

$(\Gamma_l \wedge r\ x = r\ x \wedge \Gamma_r)\ [E_i,\ E_i]$

It is reduced to

$(\Gamma_l \wedge \Gamma_r)\ [E_i,\ E_i]$

In this case we need to pass those replacements, which involve the entry predicates and ports used in other specification entities or links, to the relevant parts.

For example for the example above, we use the substitutions below to all other part in the specification at the scheme 7 level.

$[E_i/E_o,\ r/s]$

Similarly, for a delay unit in a connection link formed as

$(\Gamma_l \wedge s\ y = s\ x \wedge \Gamma_r)\ [E_i,\ E_o]$,

here, Γ_l and Γ_r represent the conjunction of zero or more connection units and zero or more delay units.

if there is not a time constraint $x < y$ we can use the time point x to replace the time point y and select E_i (E_o) to replace E_o (E_i). Therefore we can get, for example,

$(\Gamma_l \wedge s\ x = s\ x \wedge \Gamma_r)\ [E_i,\ E_i]$

It is reduced to

$(\Gamma_l \wedge \Gamma_r)\ [E_i,\ E_i]$

In this case we need to pass the replacement about the entry predicates used in other links, to the relevant parts.

For example for the example above, we use the substitutions below to all other part in the specification at the scheme 7 level.

$[E_i/E_o]$

For a connection link the two processes above can be repeated until finally we get a model, for example,

(T) $[E_i,\ E_i]$

It is further reduced to the boolean constant true T which is called the weakest specification.

In some special case, for example, in a function link all the function units can complete their execution in a zero time interval, for example, such formed function unit:

$o\ x = f\ (i_1\ x,\ i_2\ x)$

This case is equal to the case of without time constraint $x < y$. So, we can deal with all the delay units in the function link as the the delay in the connection link above.

9.1.4 Abstract Algorithm 8, Construction 8 and Specification Scheme 8

Let us use abstract algorithm 8 and construction 8 to summarize the processes of the preceding subsections.

The abstract algorithm 8 is used to mark our selections to add the time constraint and the check for bus, register and connection.

Abstract Algorithm 8.

[$\mathcal{A}_8$: **selecting and marking for bus, register and connection**]:

For a specification at the scheme 7 level

1. Following the definitions 9.1.1.1 and 9.1.2.1 to add time constraint $x < y$ to suitable link.

2. To mark a connection unit $s\ y = r\ x$ which belongs to a bus by

   $s\ y \stackrel{\text{b}}{=} r\ x.$

   And to add subscript to the "b" if it is necessary to distinguish different buses.

3. To mark a connection unit if it is checked being suitable for register by the Check-for-Register 9.1.2.1. For $s\ y = r\ x$ to mark it with

   $s\ y \stackrel{\text{r}}{=} r\ x.$

   And to add subscript to the "r" if it is necessary to distinguish different registers.

4. Following the Check-and-Change-the-Chain-Containing-Delay-Sequence 9.1.2.2 to further mark the connection and delay units like as 3 for the register implementation.

5. Following the subsection 9.1.3 to check and then mark the connection units and delay units which are suitable for "device" Connection. For $s\ y = r\ x$ and $s\ y = s\ x$ to mark them by
   $s\ y \overset{\text{c}}{=} r\ x$
   $s\ y \overset{\text{c}}{=} s\ x$
   And then to pass substitutions for entry predicate and port to the whole specification.

6. Above process may need to be repeated if later step leads to a new situation which needs to be treated with earlier steps. For example, a dealing with connection can lead to introduce a model which needs to be implemented by a Register. □

The construction 8 is applied to the specification at the scheme 7 which has been treated by the abstract algorithm 8 to produce a specification at the scheme 8 level. The construction 8 only changes the connection unit which has been marked for bus to two or three parts for the implementations by Bus, Gate and Register.

Construction 8:

9.1.4.1. $\Psi_8\ (s\ y \overset{\text{b}}{=} r\ x) \mathrel{\hat{=}} s\ y \overset{\text{b}'}{=} r'\ y \wedge r'\ y \overset{\text{g}}{=} r\ y$
if the connection link which the $s\ y \overset{\text{b}}{=} r\ x$ comes from has not a constraint $x < y$

9.1.4.2. $\Psi_8\ (s\ y \overset{\text{b}}{=} r\ x) \mathrel{\hat{=}} s\ y \overset{\text{b}'}{=} r''\ y \wedge r''\ y \overset{\text{g}}{=} r'\ y \wedge r'\ y \overset{\text{r}}{=} r\ x$
if the connection link which the $s\ y \overset{\text{b}}{=} r\ x$ comes from has a constraint $x < y$

In the construction 8 we introduce the new marks "b′" and "g". We write

$s\ y \overset{\text{b}'}{=} r\ y$

to mark that it will be implemented by a Bus in a Bus-Gate device group (see page 118). And we write

$$s\ y \stackrel{\mathrm{g}}{=} r\ y$$

to mark that it will be implemented by a Gate in a Bus-Gate device group.

Example 9.1.4.1. [for understanding $\mathcal{A}_8$ and Ψ_8]:

$$(i\ y = r_1\ x \wedge r_2\ y = r_2\ x)\ [E_i,\ E_1] \tag{d1}$$

$$\wedge$$

$$(o\ y \stackrel{\mathrm{a}}{=} g\ (i\ x) \wedge r_2\ y = r_2\ x)\ [E_1,\ E_2] \tag{d2}$$

$$\wedge$$

$$(i_2\ y = o\ x \wedge i\ y = r_2\ x)\ [E_2,\ E_3] \tag{d3}$$

$$\wedge$$

$$(o_2\ y \stackrel{\mathrm{a}}{=} f\ (i_2\ x) \wedge o\ y \stackrel{\mathrm{a}}{=} h\ (i\ x))\ [E_3,\ E_4] \tag{d4}$$

$$\wedge$$

$$(s_1\ y = o_2\ x \wedge s_2\ y = o\ x)\ [E_4,\ E_o] \tag{d5}$$

$$= (i\ x \stackrel{\mathrm{b}}{=} r_1\ x \wedge r_2\ x \stackrel{\mathrm{c}}{=} r_2\ x)\ [E_i,\ E_i] \tag{e$'$1}$$

$$\wedge$$

$$(o\ y \stackrel{\mathrm{a}}{=} g\ (i\ x) \wedge s\ y \stackrel{\mathrm{r}}{=} r_2\ x \wedge x < y)\ [E_i,\ E_2] \tag{e$'$2}$$

$$\wedge$$

$$(s_2\ x \stackrel{\mathrm{c}}{=} s_2\ x \wedge i\ x \stackrel{\mathrm{b}}{=} s\ x)\ [E_2,\ E_2] \tag{e$'$3}$$

$$\wedge$$

$$(s_1\ y \stackrel{\mathrm{a}}{=} f\ (s_2\ x) \wedge s_2\ y \stackrel{\mathrm{a}}{=} h\ (i\ x))\ [E_2,\ E_o] \tag{e$'$4}$$

$$\wedge$$

$$(s_1\ x \stackrel{\mathrm{c}}{=} s_1\ x \wedge s_2\ x \stackrel{\mathrm{c}}{=} s_2\ x)\ [E_o,\ E_o] \tag{e$'$5}$$

.... by the abstract algorithm 8

$$= (i\ x \stackrel{\mathrm{b}'}{=} r_1'\ x \wedge r_1'\ x \stackrel{\mathrm{g}}{=} r_1\ x \wedge r_2\ x \stackrel{\mathrm{c}}{=} r_2\ x)\ [E_i,\ E_i] \tag{f$'$1}$$

$$\wedge$$

$$(s_2\ y \stackrel{\mathrm{a}}{=} g\ (i\ x) \wedge s\ y \stackrel{\mathrm{r}}{=} r_2\ x \wedge x < y)\ [E_i,\ E_2] \tag{f$'$2}$$

$$\wedge$$

$$(s_2\ x \stackrel{\mathrm{c}}{=} s_2\ x \wedge i\ x \stackrel{\mathrm{b}'}{=} s'\ x \wedge s'\ x \stackrel{\mathrm{g}}{=} s\ x)\ [E_2,\ E_2] \tag{f$'$3}$$

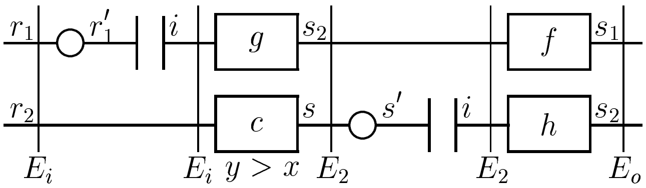
$\wedge$

$$(s_1\ y \overset{\mathrm{a}}{=} f\ (s_2\ x) \wedge s_2\ y \overset{\mathrm{a}}{=} h\ (i\ x))\ [E_2,\ E_o] \qquad \text{(f}'\text{4)}$$

$\wedge$

$$(s_1\ x \overset{\mathrm{c}}{=} s_1\ x \wedge s_2\ x \overset{\mathrm{c}}{=} s_2\ x)\ [E_o,\ E_o] \qquad \text{(f}'\text{5)}$$

. . . . by the construction 8



□

A specification at the scheme 7 level passed through $\mathcal{A}_8$ and Ψ_8 will become a specification at the scheme 8 level as below.

Specification Scheme 8:

$E\ 0 \wedge frms$

Function unit and register model set $frms$**:**

			Abbreviation:
$frms$	$::=$	frm	$frms$: set of frm
		$\mid frm \wedge frms$	frm : function and register model
frm	$::=$	$(fus)\ [E_i,\ E_o]$	fus : set of function units
		$\mid (rus)\ [E_i,\ E_o]$	rus : set of register units
		$\mid brs$	brs : branch structure
fus	$::=$	fu	fus : set of function units
		$\mid fu \wedge fus$	fu : function unit
rus	$::=$	ru	rus : set of register units
		$\mid ru \wedge rus$	ru : register units
fu	$::=$	$o\ y \overset{\mathrm{a}}{=} f\ (i_1\ x, \cdots, i_n\ x)$	: function unit
		$\mid o\ y \overset{\mathrm{a}}{=} c$	: constant function unit
		$\mid s\ y \overset{\mathrm{r}}{=} s\ x$	: delay for Register

$$
\begin{array}{lll}
 & \mid s\ y \overset{\mathrm{c}}{=} s\ x & \text{: delay for Connection} \\
ru ::= & s\ y \overset{\mathrm{b}'}{=} r\ y & \text{: connection for Bus} \\
 & \mid s\ y \overset{\mathrm{g}}{=} r\ y & \text{: connection for Gate} \\
 & \mid s\ y \overset{\mathrm{r}}{=} r\ x & \text{: connection for Register} \\
 & \mid s\ y \overset{\mathrm{r}}{=} s\ x & \text{: delay for Register} \\
 & \mid s\ y \overset{\mathrm{c}}{=} r\ y & \text{: connection for Connection} \\
 & \mid s\ y \overset{\mathrm{c}}{=} s\ x & \text{: delay for Connection}
\end{array}
$$

Note: in the scheme we write "Connection" for implementation and "connection" for specification.

The character of the specification at the scheme 8 is that it has every piece specification to be directly executable by a device at the register transfer level: function unit, bus, gate and register.

It is obvious that the abstract algorithm 8 and the construction 8 is total for the scheme 7 and correct for scheme 8 and 7.

9.2 Introducing Datapath and Microprogram

In the specification scheme 8 each specification piece has been marked for a corresponding device to implement, for example,

$s\ y \overset{\mathrm{r}}{=} r\ x$

is marked to be implemented by a register. In this section we further introduce datapath and microprogram.

By the means of the specification scheme we get an abstract *parameterized* register transfer level. Function unit is kept throughout the whole derivation process of computer as a parameter which we do not further consider its inside structure. What we gain through the specification scheme and formal construction are the relations among these function units. These relations are expressed by connection link which including two aspects: data relation through specification entities: bus, register and connection and control relation through logic boundary: entry predicate which is further constructed to get microprogram.

To gain a realistic implementation for computer we have selected certain devices at register transfer level from the real world, as we have done. However, we still need to introduce some further constraints.

Constraint 9.2.1. [variable time unit computer]:

All the function units in a function link should be implemented in the same time interval. Or at least they can be improved to do so and these improvements are not further discussed. □

Under the constraint all the function units in a function link are guaranteed to be completed in the same time interval. However, the different function links can take the different time intervals. If we take the time interval of a function link as a time unit of computer which also has the time intervals of the other function and connection links as its time units then in this case we can get a *variable* time unit computer.

To get a *constant* time unit computer we need to introduce further constraint as below.

Constraint 9.2.2. [constant time unit computer]:

A computer is of constant time unit if for it the time intervals of all the function and connection links are the same, namely, the execution of its functions and registers can be completed in the same time interval. □

Under the constraint all function units and registers at the register transfer level are assumed to be implemented in the same time unit which is traditionally called as the time unit at microinstruction level.

We can construct the implementation of computer under the looser constraint 9.2.1 if there are not Register and Device for function unit which are used in the same function link. Otherwise, we will use the tighter constraint 9.2.2. Of course, we can consider to introduce some finer constraints. However, this is not essential and is omitted.

9.2.1 Introducing Datapath and Microprogram

The introductions of datapath and microprogram are done simultaneously. When introducing a device we also introduce the corresponding controls if the implementation of the device requires. These devices will constitute the datapath of computer and these controls as codes with the logic boundary (entry and leaving predicates) will constitute the microprogram of computer. For the specification scheme 8 we take the construction steps as below:

(1). To replace

$$o\ y \stackrel{\mathrm{a}}{=} f\ (i_1\ x, \cdots, i_n\ x)$$

by

$$c_f\ x \wedge \mathrm{D}_f\ (\vec{c},\ \vec{i},\ \vec{o})$$

Where, D_f is a device which has the function of f (and possibly has other functions also). $\vec{c}$ is the control vector of the device D_f and c_f is a member of $\vec{c}$. Each i_j $(1 \leq j \leq n)$ is a member of $\vec{i}$ which is the vector of inputs of the device D_f. The o is a member of $\vec{o}$ which is a vector of outputs of the device D_f. Further we assume that

$$\vdash c_f\ x \wedge \mathrm{D}_f\ (\vec{c},\ \vec{i},\ \vec{o}) \Rightarrow o\ (x+\delta) \stackrel{\mathrm{a}}{=} f\ (i_1\ x, \cdots, i_n\ x)$$

Here, y has been replaced by $x+\delta$ and δ is a constant e.g. 1.

For example, for

$$o\ y \stackrel{\mathrm{a}}{=} \mathrm{ADD}\ (i_1\ x,\ i_2\ x)$$

we introduce:

$$add\ x \wedge \mathrm{ALU}\ (add,\ sub,\ inc,\ i_1,\ i_2,\ o)$$

Here,

$$\mathrm{ALU}\ (add,\ sub,\ inc,\ i_1,\ i_2,\ o) \equiv \forall t.\ o\ (t+1) = (add\ t \Rightarrow \mathrm{ADD}\ (i_1\ t,\ i_2\ t) \downarrow$$

$$(sub\, t \Rightarrow \text{SUB}\ (i_1\ t,\ i_2\ t) \downarrow$$
$$(inc\, t \Rightarrow \text{INC}\ (i_2\ t) \downarrow i_1\ t)))$$

Similarly, we can do for the constant function unit.

(2). To replace

$$s\ y \stackrel{\text{b}'}{=} r_i\ y \quad \text{or} \quad s\ x \stackrel{\text{b}'}{=} r_i\ x$$

By

$$\text{Bus}\ (\vec{r},\ s)$$

Where, r_i is a member of $\vec{r}$ which is the input vector of the Bus.

(3). To replace

$$s\ x \stackrel{\text{g}}{=} r\ x$$

By

$$c_g\ x \wedge \text{Gate}\ (c_g,\ r,\ s)$$

Where, c_g is the control signal of the Gate.

(4). To replace

$$s\ y \stackrel{\text{r}}{=} r\ x$$

By

$$c_r\ x \wedge \text{Reg}\ (c_r,\ r,\ s)$$

Where, c_r is the control signal of the Register.

(5). To replace

$$s\ y \stackrel{\text{r}}{=} s\ x$$

By

$$\neg(c_r\ x) \wedge \text{Reg}\ (c_r,\ r,\ s).$$

In the specification of microinstruction the $\neg(c_r\ x)$ is omitted traditionally.

(6). To replace

$$s\ y \stackrel{\text{c}}{=} r\ x \quad \text{and} \quad s\ y \stackrel{\text{c}}{=} s\ x$$

By the boolean constant T.

The replacements above can lead to the redundancy in specification: a device are written two or more copies and boolean constant T. We can use

$\vdash \alpha \wedge \alpha \Leftrightarrow \alpha$

to eliminate those redundant copies and directly eliminate T.

After the replacements above all the specification entities of derivation models in a specification are replaced by control signals. For example, the specification entity $s\ y \stackrel{\mathrm{r}}{=} r\ x$ is replaced by $c_r\ x$. And the end time point y which is an existential variable is replaced by an instance time point formed as $x + \delta$ (Here, δ is an natural number) or by $x + 1$ if under the constraint 9.2.2.

To get a microprogram we instantiate each entry predicate which is an existential variable by an instance predicate about the entry address of a microinstruction. For example, to replace

$E_i\ x$

by

address $x = \mathrm{a}_i$

Here, a_i is a natural number. The meaning of the instance predicate is that at the time point x the address of microprogram is equal to a_i.

So, after these replacements a derivation model formed as, for example,

$(c_f\ x \wedge c_r\ x)\ [E_i,\ E_o]$

is replaced by

$\forall x.$ address $x = 5 \Rightarrow c_f\ x \wedge c_r\ x \wedge$ address $(x + 1) = 6$

This is a microinstruction whose entry address is 5, it sends the control signal c_f and c_r and turns to another microinstruction whose entry address is 6.

9.2.2 About Clock

In the above process it is implied indirectly that we have, in fact, introduced the concept of clock. To gain a variable time unit computer we measure a time point through entry predicate. More exactly, for a computer at the scheme 8 level we collect all the entry predicates of it as the clock of the computer. Suppose that all the entry predicates of a computer are $E_j \quad (1 \leq j \leq n)$ then we define the clock of the computer as

$\forall t.\ E_1\ t \vee E_2\ t \vee \cdots \vee E_n\ t$ .... (c1)

By making the clock be true we get a time point t. By the definition we only need to make one entry predicate in (c1) to be true and we do not require that any two time intervals must be the same.

Under the constraint 9.2.2 we can further introduce one predicate which can be used to make each entry predicate in the clock true. The predicate is

$\forall x.$ address $(x + 1) =$ next-address x .... (n1)

We suppose there is a device at the register transfer level for (n1):

Address (next-address, address)

Note: for the Address the next-address is its input.

However, in this case we need, instead of the preceding replacement, to replace the leaving predicate of a derivation model, for example, "$E_o\ y$" by the instance predicate "next-address $x = \mathrm{a}_o$" and meanwhile to guarantee that when the "$E_o\ x$" is taken as the entry predicate it is replaced by "address $x = \mathrm{a}_o$". Here, the emphasis is that they have the same a_o. For example, to instantiate

$(c_f\ x \wedge c_r\ x)\ [E_i,\ E_o]$

By

$\forall x.$ address $x = 5 \Rightarrow c_f\ x \wedge c_r\ x \wedge$ next-address $x = 6$ .... (i1)

We must guarantee to instantiate, for example,

$(c_g\ x \wedge c_h\ x)\ [E_o,\ E_l]$

By

$\forall x.$ address $x = 6 \Rightarrow c_g\ x \wedge c_h\ x \wedge$ next-address $x = 7$ .... (i2)

Here, E_o is an entry predicate. We call (i1) and (i2) microinstructions.

As we have had E 0, which can be instantiated, for example, by address $0 = 0$, and through using (n1) we can make each entry predicate in (c1) to be true step by step.

To implement all the microinstructions in a computer we need to introduce a device "MicroProgram":

MicroProgram (address, next-address, $\vec{c}$)
Here, "address" is input and $\vec{c}$ is a vector of control signals
used to control the relevant devices in a computer.

It is emphasized that in our construction process for a class of computers we need not directly introduce the concept of clock. Clock is implied by our derivation process. This is just as the shown by the whole preceding construction process. So, we can call these computers to be of *non-clock*. When under constraint 9.2.1 we may, but not certainly necessary, introduce the clock (c1) we call it the *variable-clock* computer. And under constraint 9.2.2 we need to introduce (n1) and through it to define the time point, therefore, we call it the *constant-clock* computer.

For the branch structure models brs of the scheme 8 we replace its entry predicate and all leaving predicates as we do for other models. For example, to replace

$$[\, E_i,\ \mathrm{b}_1\ (E_{o1},\ \mathrm{b}_2\ (E_{o2},\ E_{o3}))\,]$$

by

$$\begin{aligned}\forall x.\ \text{address } x = \mathrm{a}_i \Rightarrow (&\mathrm{b}_1\ x \Rightarrow \text{next-address } x = \mathrm{a}_{o1} \downarrow \\ &(\mathrm{b}_2\ x \Rightarrow \text{next-address } x = \mathrm{a}_{o2} \downarrow \text{next-address } x = \mathrm{a}_{o3}))\end{aligned}$$

And further we suppose that there is a corresponding device

Branch$_1$ (b_1, b_2, address, next-address)

at the register transfer level. More generally, we suppose there are device kind for branch structure model as below:

Branch ($\vec{b}$, address, next-address)

Here, "$\vec{b}$" and "address" are inputs.

9.2.3 Implied Zero Time Point

In [50] J. J. Joyce writes that "One of the most interesting discoveries of this exercise was that the formal verification of the design missed a design error: there is no reset button to initialize the microcode program counter". For our derivation the "reset" is to get $E\ 0$. However, it is just one of the advantages of out approach that derivation which is based on our derivation model cannot miss the "reset". It is very natural to construct the $E\ 0$ which is presented as a goal to be constructed even from specification scheme 1. In the specifications of a computer from the scheme 1 to the scheme 8 an initiation $E\ 0$ is always necessary because the other part of specification scheme, we read it specification body and write it sb, is a loop on the execution of machine instructions and the loop needs a start time point. Although we can use a special device to do the job of $E\ 0$, we will prove in this section that the body sb itself can do this.

In physics when we switch on power to make a computer start its work the start time point is completely stochastic such that it can make any address of a microinstruction become the initial point to start the running of microprogram of the computer. For derivation this is arbitrarily to make an entry predicate of the computer become true. We do not know which entry predicate will be at all but we can always affirm that there is one and only one such predicate.

Postulate 9.2.3.1. [Stochastic starting]:

At the level of specification scheme 8 that a computer powers up at a time point is expressed by the possibility that there is and only is an arbitrary entry predicate which can be true at the time point. □

Here, we assume that a machine at the scheme 8 level. But this is not essential, in fact, we can do at the higher or lower scheme levels.

Let us prove the the initiation $E\ 0$ is implied by specification body sb.

Theorem 9.2.3.1. [implied 0 time point]:

For the specification body sb and initiation $E\ 0$ of the scheme 8 we have $\vdash sb \Rightarrow E\ 0$

Proof:

By the postulate 9.2.3.1 when switching on power we get a time point t_s and an entry predicate E_s such $E_s\ t_s$ is true. Then there are two cases.

1. If the E_s is just at the E then we can take the time point t_s as zero time point. Therefore we get $E\ 0$.

2. If the E_s is not E then we can prove through a finite time interval a t_e will be reached such that $E\ t_e$ true, and we can take the t_e as zero time point of this computer.

   - Commencing from the scheme 2 which has the form as

     $\alpha[E, E]$

     all later constructions continually introduce new model which introduce new entry predicates to decompose and only finite entry predicates are introduced in these construction steps. Therefore, starting from the $E_s\ t_s$ through finite links and branch structures a time point t_e can always be reached such that $E\ t_e$ is true. Note: Branch structures are constituted by conditional structures formed as $P \Rightarrow \alpha \downarrow \beta$. When passing the condition structure the running of computer is directed to either α or β. $\square$

9.2.4 Abstract Algorithm 9 and Specification Scheme 9

We use abstract algorithm 9 to collect that we have done in this section up to this.

Abstract Algorithm 9. [constructing database and microprogram]:

For a specification at the scheme 8 level following the four steps below to construct a specification at the scheme 9 level:

(1). For the specification entities formed as $\alpha \stackrel{\text{a}}{=} \beta$, $\alpha \stackrel{\text{r}}{=} \beta$, $\alpha \stackrel{\text{b}'}{=} \beta$, $\alpha \stackrel{\text{g}}{=} \beta$ and $\alpha \stackrel{\text{c}}{=} \beta$, to replace them, following the described steps in the subsection 9.2.1, by the corresponding control signals which come from the devices for function unit, Register, Bus, Gate and boolean constant T respectively (but for Bus and T there are control signals necessary) and these devices are also introduced separately .

(2). For the logic boundary, entry and leaving predicates, of the derivation models and the branch structure models following the subsections 9.2.1 and 9.2.2 to replace them and then further replace these models by a device "MicroProgram" and a group of device "Branch".

(3). Introduce the device "Address" as mentioned in the subsection 9.2.2.

(4). Eliminate $E\ 0$. □

We will discuss the specification scheme 9 in the section 10.1 as it is taken as the end point of our formal derivation of a class of computers.

For the introductions of Devices for function unit, Register and Connection their corrections are obvious. For Bus-Gate structure we introduce a theorem below to prove its correctness. In the subsection 9.1.1 we have three steps (bg1), (bg2) and (bg3) to construction a Bus-Gate structure as below:

$\cdots \wedge\ (\Gamma_{1l} \wedge s\ y = r_1\ x \wedge \Gamma_{1r})\ [E_a,\ E_b] \wedge$

$\cdots \wedge\ (\Gamma_{2l} \wedge s\ y = r_2\ x \wedge \Gamma_{2r})\ [E_c,\ E_d] \wedge$

$\cdots \wedge\ (\Gamma_{nl} \wedge s\ y = r_n\ x \wedge \Gamma_{nr})\ [E_e,\ E_f]\ \wedge \cdots$ .... (bg1)

Here, Γ_{il} and Γ_{ir} $(1 \leq i \leq n)$ represent the conjunction of zero or more specification entities.

We first transform it to

$\cdots \wedge\ (\Gamma_{1l} \wedge s\ y = r_1'\ y \wedge r_1'\ y = r_1\ y \wedge r_1\ y = r_1\ x \wedge \Gamma_{1r})\ [E_a,\ E_b] \wedge$

$\cdots \wedge\ (\Gamma_{2l} \wedge s\ y = r_2'\ y \wedge r_2'\ y = r_2\ y \wedge r_2\ y = r_2\ x \wedge \Gamma_{2r})\ [E_c,\ E_d] \wedge$

$$\cdots \wedge\ (\Gamma_{nl} \wedge s\ y = r'_n\ y \wedge r'_n\ y = r_n\ y \wedge r_n\ y = r_n\ x \wedge \Gamma_{nr})\ [E_e,\ E_f]\ \wedge \cdots$$
$$\cdots\cdots \text{(bg2)}$$

Then we have an implementation formed as

$$\cdots \wedge\ (\Gamma_{1l} \wedge c_{g1}\ y \wedge r_1\ y = r_1\ x \wedge \Gamma_{1r})\ [E_a,\ E_b]\ \wedge$$
$$\cdots \wedge\ (\Gamma_{2l} \wedge c_{g2}\ y \wedge r_2\ y = r_2\ x \wedge \Gamma_{2r})\ [E_c,\ E_d]\ \wedge$$
$$\cdots \wedge\ (\Gamma_{nl} \wedge c_{gn}\ y \wedge r_n\ y = r_n\ x \wedge \Gamma_{nr})\ [E_e,\ E_f]\ \wedge$$
$$\cdots \wedge\ \text{Gate}_1\ (c_{g1}\ r_1,\ r'_1) \wedge \text{Gate}_2\ (c_{g2}\ r_2,\ r'_2) \wedge \cdots \wedge \text{Gate}_n\ (c_{gn}\ r_n,\ r'_n)\ \wedge$$
$$\cdots \wedge\ \text{Bus}\ (r'_1,\ r'_2, \cdots, r'_n,\ s)\ \wedge \cdots\cdots \text{(bg3)}$$

We can see that (bg1) can be viewed as specification and (bg3) as the implementation. However, we should specially point that the (bg2) is only taken as a middle step and description means but not used to correctness prove.

Theorem 9.2.4.1. [correctness of bus-gate structure]:

$\vdash$ (bg3) $\Rightarrow$ (bg1)

Proof:

By the theorem 8.1.4.1 (page 105) at any time point y there is one and only one model in (bg3), let us suppose it is formed as

$$(\Gamma_{il} \wedge c_{gi}\ y \wedge r_i\ y = r_i\ x \wedge \Gamma_{ir})\ [E_\alpha,\ E_\beta]\quad (1 \le i \le n),$$

its entry predicate E_α gets boolean value True. And all other entry predicates get boolean value False, therefore the corresponding model become True. Further, only the gate-control signal c_{gi} of the model gets value True and all others gate-control signal get value False. By the specification of Gate (gs) (page 118) at the time point y we only have one $\text{Gate}_i\ (c_{gi},\ r_i,\ r'_i)$

$$r'_i\ y = r_i\ y$$

and all other gates:

$$r'_j\ y = \text{Float}\quad (1 \le j \le n \wedge j \ne i)$$

By the specification of Bus (bs) (page 118) we have

$$s\ y = \text{Dest\_Tri}\ (\text{Float} \cup \cdots \cup \text{Float} \cup \text{Mk\_Tri}\ (r_i\ y) \cup \text{Float} \cup \cdots \cup \text{Float})$$

$= \text{Dest\_Tri}\ (\text{Mk\_Tri}\ (r\ y))$ .... by the axiom (bo) (see page 118)

$= r_i\ y$ .... by the axiom (dmt) (see page 118)

Hence, we get

$$(\Gamma_{il} \wedge s\ y = r_i\ x \wedge \Gamma_{ir})\ [E_\alpha,\ E_\beta].$$

As all other model in (bg1) have become True therefore we can get (bg1). □

Based on the theorem 9.2.4.1 and refer to the specification scheme 9 in the next chapter it is obvious about totalness and correctness of the abstract algorithm 9.

Chapter 10

Structural Specification Scheme

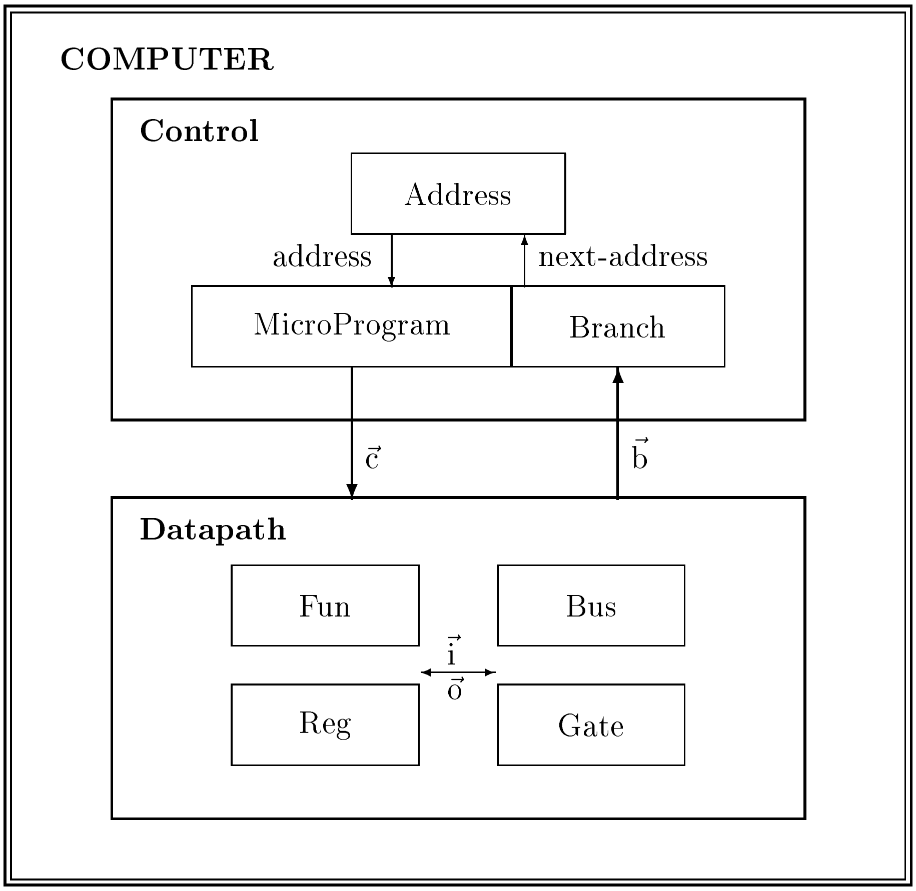
10.1 Structural Specification Scheme

We call the specification scheme 9 the structural specification scheme as it is considered to have enough detail for implementation at register transfer level, therefore, can be taken as the end point of our formal derivation for a class of computers.

Specification Scheme 9 as Structural Specification Scheme

Specification Scheme: SSS			**Abbreviation:**
SSS	::=	$Control$	SSS : Structural Spec. Scheme
		$\wedge$ $Datapath$	
$Control$	::=	Address (next-address, address)	
		$\wedge$ MicroProgram (address, next-address, $\vec{c}$)	
		$\wedge$ Brs	Brs : branch structure set
$Datapath$	::=	$Device$	
		$\mid$ $Device \wedge Datapath$	
Brs	::=	Branch ($\vec{b}$, address, next-address)	
		$\mid$ Branch ($\vec{b}$, address, next-address) $\wedge$ Brs	
$Device$	::=	Fun ($\vec{c}$, $\vec{i}$, $\vec{o}$)	Fun : Device for function unit
		$\mid$ Reg (c_r, i, o)	c_r : control signal of register
		$\mid$ Bus ($\vec{i}$, o)	i/o : in/output signal of gate

$\mid$ Gate (c_g, i, o)	c_g	: control signal
	$\vec{b}$	: boolean signal vector
	$\vec{c}$	: control signal vector
	$\vec{i}/\vec{o}$	: in/output signal vector



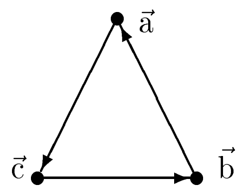
A computer at the structural specification scheme level consists of two parts: control and datapath. The control is constituted by a device Address, a device MicroProgram and a kind of devices Branch. The datapath is constituted by the four kinds of devices: device for function unit, bus, register and gate. In the inside of the control there are internal signals: address and next-address and in the inside of the datapath there are internal signals: input and output signals $\vec{i}$

and $\vec{\text{o}}$. The communications between control and datapath are through control (boolean valued) signals $\vec{\text{c}}$ from control to datapath for controlling the data flow $\vec{\text{i}}$ and $\vec{\text{o}}$ and control signals $\vec{\text{b}}$ from datapath to control for controlling the address flow of microprogram.

An interesting observation to the formally derived computer class is that there is a control circle:

$\{\ \vec{\text{a}}, \vec{\text{c}}, \vec{b}\ \}$.

Here, $\vec{\text{a}} =<$ address, next-address $>$. However, $\vec{\text{a}}$ is the center of the circle.



Part III

Case Study and Further Discussion

Chapter 11

Deriving Gordon's Computer

In this chapter we will apply the stepwise constructions to Gordon's computer following the schemes of specification, construction, and proof as laid down in Part 2. we will formally construct three correct implementations at register transfer level from a formal specification at machine instruction level. Finally we will compare these derived implementations and the implementation verified by [50].

One point should be emphasized. In the formal derivation we deal with pure structure and try to keep the specification as general as possible. We leave any further detail uninterpreted if it is irrelevant to the pure structure and construction.

11.1 Introduction to Gordon's Computer

Gordon's computer is a simple general-purpose computer invented as a nontrivial example for the formal specification and verification of hardware [50]. We take this computer as a case study for formal derivation.

A top-level specification of the computer is at machine instruction level. This is typically what an assembler language programmer would need to write a program.

At the top level Gordon's computer has a memory and two registers: the program counter PC and the accumulator ACC. The top-level specification includes two parts: the front panel operation and the machine instructions.

The front panel of the computer has a button, a four-position knob, and switches. When the computer is running pushing the button interrupts the execution of the program and the computer idles. The effect of pushing the button when the computer is idling is determined by the position of the knob. If the knob is in position 0 the effect is that the word determined by the switches is loaded into the PC. Pushing the button when the knob is in position 1 loads the word determined by the switches into the ACC. When the knob is in position 2, the contents of the ACC is stored in the memory at the location held in the PC. Finally, knob position 3 is used to start the execution of the program stored in memory beginning at the location in the PC.

Knob = 0	Load PC
Knob = 1	Load ACC
Knob = 2	Store ACC at PC
Knob = 3	Start execution at PC

The computer has eight machine instructions HALT, JMP, JZR, ADD, SUB, LOAD, STORE and SKIP, with opcode 0-7, respectively.

HALT	Stop execution
JMP L	Jump to address L
JZR L	Jump to address L if ACC = 0
ADD L	Add contents of address L to ACC
SUB L	Subtract contents of address L from ACC
LD L	Load contents of address L into ACC
ST L	Store contents of ACC in address L
SKIP	Skip to next instruction

Above is the informal description for Gordon's computer that will formally be dealt with in the next section. We will follow [50] exactly and, as mentioned, we only deal formally with pure structure and leave out all irrelevant details.

11.2 Formal Derivation

In this section we show 11 construction steps staring from the behavior specification and ending in the structural specification for constructing an implementation for Gordon's computer.

11.2.1 Behavior Specification

Specification 0:

$$
\begin{array}{l}
\text{Computer}\ (\ \mathit{button},\ \mathit{knob},\ \mathit{switches},\ \mathit{memory},\ \mathit{pc},\ \mathit{acc},\ \mathit{idle}) \equiv \\
\forall u.\ \ (\mathit{memory}\ (u+1),\ \mathit{pc}\ (u+1),\ \mathit{acc}\ (u+1),\ \mathit{idle}\ (u+1)) = \\
\qquad (\mathit{idle}\ u \Rightarrow \mathit{Idle} \downarrow \mathit{Busy}) \\
\mathit{Idle} \mathrel{\hat{=}}\ \ \mathit{button}\ u \Rightarrow \\
\qquad\qquad (\ (\text{Val}_2\ (\mathit{knob}\ u) = 0) \Rightarrow \\
\qquad\qquad\qquad (\mathit{memory}\ u,\ \text{Cut}_{16\_13}\ (\mathit{switches}\ u),\ \mathit{acc}\ u,\ \text{T}) \downarrow \\
\qquad\qquad\quad (\text{Val}_2\ (\mathit{knob}\ u) = 1) \Rightarrow \\
\qquad\qquad\qquad (\mathit{memory}\ u,\ \mathit{pc}\ u,\ \mathit{switches}\ u,\ \text{T}) \downarrow \\
\qquad\qquad\quad (\text{Val}_2\ (\mathit{knob}\ u) = 2) \Rightarrow \\
\qquad\qquad\qquad (\text{Store}_{13}\ (\mathit{pc}\ u)\ (\mathit{acc}\ u)\ (\mathit{memory}\ u),\ \mathit{pc}\ u,\ \mathit{acc}\ u,\ \text{T}) \downarrow \\
\qquad\qquad\qquad (\mathit{memory}\ u,\ \mathit{pc}\ u,\ \mathit{acc}\ u,\ \text{F})) \downarrow \\
\qquad\qquad (\mathit{memory}\ u,\ \mathit{pc}\ u,\ \mathit{acc}\ u,\ \text{T})
\end{array}
$$

$$
\begin{array}{l}
\mathit{Busy} \mathrel{\hat{=}}\ \ \mathit{button}\ u \Rightarrow \\
\qquad\qquad (\mathit{memory}\ u,\ \mathit{pc}\ u,\ \mathit{acc}\ u,\ \text{T}) \downarrow \\
\qquad\qquad \text{let}\ \mathit{op} = \text{Val}_3\ (\text{Opcode}\ (\text{Fetch}_{13}\ (\mathit{memory}\ u)\ (\mathit{pc}\ u)))\ \ \text{in}
\end{array}
$$

$$
\begin{array}{l}
\text{let } addr = \mathrm{Cut}_{16\_13}\ (\mathrm{Fetch}_{13}\ (memory\ u)\ (pc\ u))\ \text{ in} \\
(\ (op = 0) \Rightarrow (memory\ u,\ pc\ u,\ acc\ u,\ \mathrm{T}) \downarrow \\
\quad (op = 1) \Rightarrow (memory\ u,\ addr,\ acc\ u,\ \mathrm{F}) \downarrow \\
\quad (op = 2) \Rightarrow \\
\qquad ((\mathrm{Val}_{16}\ (acc\ u) = 0) \Rightarrow \\
\qquad\quad (memory\ u,\ addr,\ acc\ u,\ \mathrm{F}) \downarrow \\
\qquad\quad (memory\ u,\ \mathrm{Inc}_{13}\ (pc\ u),\ acc\ u,\ \mathrm{F})) \downarrow \\
\quad (op = 3) \Rightarrow (memory\ u, \\
\qquad\qquad\quad \mathrm{Inc}_{13}\ (pc\ u), \\
\qquad\qquad\quad \mathrm{Add}_{16}\ (acc\ u)\ \ (\mathrm{Fetch}_{13}\ (\mathrm{memory}\ u)\ addr), \\
\qquad\qquad\quad \mathrm{F}) \downarrow \\
\quad (op = 4) \Rightarrow (memory\ u, \\
\qquad\qquad\quad \mathrm{Inc}_{13}\ (pc\ u), \\
\qquad\qquad\quad \mathrm{Sub}_{16}\ (acc\ u)\ \ (\mathrm{Fetch}_{13}\ (memory\ u)\ addr), \\
\qquad\qquad\quad \mathrm{F}) \downarrow \\
\quad (op = 5) \Rightarrow (memory\ u, \\
\qquad\qquad\quad \mathrm{Inc}_{13}\ (pc\ u), \\
\qquad\qquad\quad \mathrm{Fetch}_{13}\ (memory\ u)\ addr, \\
\qquad\qquad\quad \mathrm{F}) \downarrow \\
\quad (op = 6) \Rightarrow (\mathrm{Store}_{13}\ addr\ (acc\ u)\ (memory\ u), \\
\qquad\qquad\quad \mathrm{Inc}_{13}\ (pc\ u), \\
\qquad\qquad\quad acc\ u, \\
\qquad\qquad\quad \mathrm{F}) \downarrow \\
\qquad\qquad (memory\ u,\ \mathrm{Inc}_{13}\ (pc\ u),\ acc\ u,\ \ \mathrm{F}))
\end{array}
$$

The meanings of the functions Val_2, Cut_{16_13}, Store_{13}, Opcode and Fetch_{13} are exactly as in [50]. T and F are our boolean constants for true and false.

11.2.2 Refining Time

The specification 0 is at the machine instruction time level. In this subsection following the construction 1 and 2 we change the time level to the microinstruction level, namely, from time unit $< u,\ u+1 >$ to the time interval $< t_1,\ t_2 >$. In the construction we introduce a global entry predicate E and two constraints:

$\forall t.\ t_1 < t < t_2 \Rightarrow \neg(E\ t)$ and $t_1 < t_2$.

Specification 1:

$$
\begin{array}{l}
E\ 0\ \wedge \\
\forall t_1.\ \exists t_2.\ E\ t_1 \Rightarrow (\mathit{memory}\ (t_2),\ \mathit{pc}\ (t_2),\ \mathit{acc}\ (t_2),\ \mathit{idle}\ (t_2)) = \\
\qquad\qquad\qquad (\mathit{idle}\ t_1 \Rightarrow \mathit{Idle} \downarrow \mathit{Busy})\ \wedge \\
\qquad\qquad (\forall t.\ t_1 < t < t_2 \Rightarrow \neg(E\ t))\ \wedge \\
\qquad\qquad t_1 < t_2\ \wedge \\
\qquad\qquad E\ t_2
\end{array}
$$

$$
\begin{array}{l}
\mathit{Idle} \mathrel{\widehat{=}}\ \mathit{button}\ t_1 \Rightarrow \\
\qquad (\ (\mathrm{Val}_2\ (\mathit{knob}\ t_1) = 0) \Rightarrow \\
\qquad\qquad (\mathit{memory}\ t_1,\ \mathrm{Cut}_{16\_13}\ (\mathit{switches}\ t_1),\ \mathit{acc}\ t_1,\ \mathrm{T}) \downarrow \\
\qquad\quad (\mathrm{Val}_2\ (\mathit{knob}\ t_1) = 1) \Rightarrow \\
\qquad\qquad (\mathit{memory}\ t_1,\ \mathit{pc}\ t_1,\ \mathit{switches}\ t_1,\ \mathrm{T}) \downarrow \\
\qquad\quad (\mathrm{Val}_2\ (\mathit{knob}\ t_1) = 2) \Rightarrow \\
\qquad\qquad (\mathrm{Store}_{13}\ (\mathit{pc}\ t_1)\ (\mathit{acc}\ t_1)\ (\mathit{memory}\ t_1),\ \mathit{pc}\ t_1,\ \mathit{acc}\ t_1,\ \mathrm{T}) \downarrow \\
\qquad\qquad (\mathit{memory}\ t_1,\ \mathit{pc}\ t_1,\ \mathit{acc}\ t_1,\ \mathrm{F})) \downarrow \\
\qquad (\mathit{memory}\ t_1,\ \mathit{pc}\ t_1,\ \mathit{acc}\ t_1,\ \mathrm{T})
\end{array}
$$

$$
\begin{array}{l}
\mathit{Busy} \mathrel{\widehat{=}}\ \mathit{button}\ t_1 \Rightarrow \\
\qquad (\mathit{memory}\ t_1,\ \mathit{pc}\ t_1,\ \mathit{acc}\ t_1,\ \mathrm{T}) \downarrow \\
\qquad \mathrm{let}\ \mathit{op} = \mathrm{Val}_3\ (\mathrm{Opcode}\ (\mathrm{Fetch}_{13}\ (\mathit{memory}\ t_1)\ (\mathit{pc}\ t_1)))\ \ \mathrm{in} \\
\qquad \mathrm{let}\ \mathit{addr} = \mathrm{Cut}_{16\_13}\ (\mathrm{Fetch}_{13}\ (\mathit{memory}\ t_1)\ (\mathit{pc}\ t_1))\ \ \mathrm{in} \\
\qquad (\ (\mathit{op} = 0) \Rightarrow (\mathit{memory}\ t_1,\ \mathit{pc}\ t_1,\ \mathit{acc}\ t_1,\ \mathrm{T}) \downarrow
\end{array}
$$

$$
\begin{array}{l}
(op = 1) \Rightarrow (\mathit{memory}\ t_1,\ \mathit{addr},\ \mathit{acc}\ t_1,\ \mathrm{F}) \downarrow \\
(op = 2) \Rightarrow \\
\quad ((\mathrm{Val}_{16}\ (\mathit{acc}\ t_1) = 0) \Rightarrow \\
\qquad (\mathit{memory}\ t_1,\ \mathit{addr},\ \mathit{acc}\ t_1,\ \mathrm{F}) \downarrow \\
\qquad (\mathit{memory}\ t_1,\ \mathrm{Inc}_{13}\ (\mathit{pc}\ t_1),\ \mathit{acc}\ t_1,\ \mathrm{F})) \downarrow \\
(op = 3) \Rightarrow (\mathit{memory}\ t_1, \\
\qquad\qquad\quad \mathrm{Inc}_{13}\ (\mathit{pc}\ t_1), \\
\qquad\qquad\quad \mathrm{Add}_{16}\ (\mathit{acc}\ t_1)\ \ (\mathrm{Fetch}_{13}\ (\mathrm{memory}\ t_1)\ \mathit{addr}), \\
\qquad\qquad\quad \mathrm{F}) \downarrow \\
(op = 4) \Rightarrow (\mathit{memory}\ t_1, \\
\qquad\qquad\quad \mathrm{Inc}_{13}\ (\mathit{pc}\ t_1), \\
\qquad\qquad\quad \mathrm{Sub}_{16}\ (\mathit{acc}\ t_1)\ \ (\mathrm{Fetch}_{13}\ (\mathit{memory}\ t_1)\ \mathit{addr}), \\
\qquad\qquad\quad \mathrm{F}) \downarrow \\
(op = 5) \Rightarrow (\mathit{memory}\ t_1, \\
\qquad\qquad\quad \mathrm{Inc}_{13}\ (\mathit{pc}\ t_1), \\
\qquad\qquad\quad \mathrm{Fetch}_{13}\ (\mathit{memory}\ t_1)\ \mathit{addr}, \\
\qquad\qquad\quad \mathrm{F}) \downarrow \\
(op = 6) \Rightarrow (\mathrm{Store}_{13}\ \mathit{addr}\ (\mathit{acc}\ t_1)\ (\mathit{memory}\ t_1), \\
\qquad\qquad\quad \mathrm{Inc}_{13}\ (\mathit{pc}\ t_1), \\
\qquad\qquad\quad \mathit{acc}\ t_1, \\
\qquad\qquad\quad \mathrm{F}) \downarrow \\
\qquad\qquad\quad (\mathit{memory}\ t_1,\ \mathrm{Inc}_{13}\ (\mathit{pc}\ t_1),\ \mathit{acc}\ t_1,\ \ \mathrm{F}))
\end{array}
$$

11.2.3 Setting State Transition

Following the construction 3 we set a corresponding state transition for each machine instruction. There are eleven state_transitions which correspond to all possible front panel operations and machine instructions of Gordon's computer.

Specification 2:

$$
\begin{array}{l}
E\ 0 \wedge \\
\forall t_1.\ \exists t_2.\ \ E\ t_1 \Rightarrow (\mathit{idle}\ t_1 \Rightarrow \mathit{Idle}_{st} \downarrow \mathit{Busy}_{st}) \wedge \\
\qquad\qquad\qquad (\forall t.\ t_1 < t < t_2 \Rightarrow \neg E\ t) \wedge t_1 \leq t_2 \wedge E\ t_2
\end{array}
$$

$$
\begin{array}{ll}
\mathit{Idle}_{st} \mathrel{\widehat{=}} & \mathit{button}\ t_1 \Rightarrow \\
& (\ (\mathrm{Val}_2\ (\mathit{knob}\ t_1) = 0) \Rightarrow \mathit{state\_transition}_0 \downarrow \\
& \ \ (\mathrm{Val}_2\ (\mathit{knob}\ t_1) = 1) \Rightarrow \mathit{state\_transition}_1 \downarrow \\
& \ \ (\mathrm{Val}_2\ (\mathit{knob}\ t_1) = 2) \Rightarrow \mathit{state\_transition}_2 \downarrow \mathit{state\_transition}_3) \downarrow \\
& \mathit{state\_transition}_4
\end{array}
$$

$$
\begin{array}{ll}
\mathit{Busy}_{st} \mathrel{\widehat{=}} & \mathit{button}\ t_1 \Rightarrow \mathit{state\_transition}_4 \downarrow \\
& \mathrm{let}\ \mathit{op} = \mathrm{Val}_3\ (\mathrm{Opcode}\ (\mathrm{Fetch}_{13}\ (\mathit{memory}\ t_1)\ (\mathit{pc}\ t_1)))\ \ \mathrm{in} \\
& \mathrm{let}\ \mathit{addr} = \mathrm{Cut}_{16\_13}\ (\mathrm{Fetch}_{13}\ (\mathit{memory}\ t_1)\ (\mathit{pc}\ t_1))\ \ \mathrm{in} \\
& (\ (\mathit{op} = 0) \Rightarrow \mathit{state\_transition}_4 \downarrow \\
& \ \ (\mathit{op} = 1) \Rightarrow \mathit{state\_transition}_5 \downarrow \\
& \ \ (\mathit{op} = 2) \Rightarrow \\
& \qquad ((\mathrm{Val}_{16}\ (\mathit{acc}\ t_1) = 0) \Rightarrow \mathit{state\_transition}_5 \downarrow \mathit{state\_transition}_6) \downarrow \\
& \ \ (\mathit{op} = 3) \Rightarrow \mathit{state\_transition}_7 \downarrow \\
& \ \ (\mathit{op} = 4) \Rightarrow \mathit{state\_transition}_8 \downarrow \\
& \ \ (\mathit{op} = 5) \Rightarrow \mathit{state\_transition}_9 \downarrow \\
& \ \ (\mathit{op} = 6) \Rightarrow \mathit{state\_transition}_{10} \downarrow \mathit{state\_transition}_6)
\end{array}
$$

$$
\begin{array}{ll}
\mathit{state\_transition}_0 \mathrel{\widehat{=}} & (\ \mathit{memory}\ t_2 = \mathit{memory}\ t_1 \wedge \\
& \ \ \mathit{pc}\ t_2 = \mathrm{Cut}_{16\_13}\ (\mathit{switches}\ t_1) \wedge \\
& \ \ \mathit{acc}\ t_2 = \mathit{acc}\ t_1 \wedge \\
& \ \ \mathit{idle}\ t_2 = \mathrm{T}) \\
\mathit{state\_transition}_1 \mathrel{\widehat{=}} & (\ \mathit{memory}\ t_2 = \mathit{memory}\ t_1 \wedge \\
& \ \ \mathit{pc}\ t_2 = \mathit{pc}\ t_1 \wedge \\
& \ \ \mathit{acc}\ t_2 = \mathit{switches}\ t_1 \wedge \\
& \ \ \mathit{idle}\ t_2 = \mathrm{T})
\end{array}
$$

$$
\begin{aligned}
\mathit{state\_transition}_2 \mathrel{\widehat{=}} \ (\ & \mathit{memory}\ t_2 = \mathrm{Store}_{13}\ (\mathit{pc}\ t_1)\ (\mathit{acc}\ t_1)\ (\mathit{memory}\ t_1) \wedge \\
& \mathit{pc}\ t_2 = \mathit{pc}\ t_1 \\
& \mathit{acc}\ t_2 = \mathit{acc}\ t_1 \wedge \\
& \mathit{idle}\ t_2 = \mathrm{T})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{state\_transition}_3 \mathrel{\widehat{=}} \ (\ & \mathit{memory}\ t_2 = \mathit{memory}\ t_1 \wedge \\
& \mathit{pc}\ t_2 = \mathit{pc}\ t_1 \wedge \\
& \mathit{acc}\ t_2 = \mathit{acc}\ t_1 \wedge \\
& \mathit{idle}\ t_2 = \mathrm{F})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{state\_transition}_4 \mathrel{\widehat{=}} \ (\ & \mathit{memory}\ t_2 = \mathit{memory}\ t_1 \wedge \\
& \mathit{pc}\ t_2 = \mathit{pc}\ t_1 \wedge \\
& \mathit{acc}\ t_2 = \mathit{acc}\ t_1 \wedge \\
& \mathit{idle}\ t_2 = \mathrm{T})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{state\_transition}_5 \mathrel{\widehat{=}} \ (\ & \mathit{memory}\ t_2 = \mathit{memory}\ t_1 \wedge \\
& \mathit{pc}\ t_2 = \mathit{addr} \wedge \\
& \mathit{acc}\ t_2 = \mathit{acc}\ t_1 \wedge \\
& \mathit{idle}\ t_2 = \mathrm{F})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{state\_transition}_6 \mathrel{\widehat{=}} \ (\ & \mathit{memory}\ t_2 = \mathit{memory}\ t_1 \wedge \\
& \mathit{pc}\ t_2 = \mathrm{Inc}_{13}\ (\mathit{pc}\ t_1) \wedge \\
& \mathit{acc}\ t_2 = \mathit{acc}\ t_1 \wedge \\
& \mathit{idle}\ t_2 = \mathrm{F})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{state\_transition}_7 \mathrel{\widehat{=}} \ (\ & \mathit{memory}\ t_2 = \mathit{memory}\ t_1 \wedge \\
& \mathit{pc}\ t_2 = \mathrm{Inc}_{13}\ (\mathit{pc}\ t_1) \wedge \\
& \mathit{acc}\ t_2 = \mathrm{Add}_{16}\ (\mathit{acc}\ t_1)\ \ (\mathrm{Fetch}_{13}\ (\mathit{memory}\ t_1)\ \mathit{addr}) \wedge \\
& \mathit{idle}\ t_2 = \mathrm{F})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{state\_transition}_8 \mathrel{\widehat{=}} \ (\ & \mathit{memory}\ t_2 = \mathit{memory}\ t_1 \wedge \\
& \mathit{pc}\ t_2 = \mathrm{Inc}_{13}\ (\mathit{pc}\ t_1) \wedge \\
& \mathit{acc}\ t_2 = \mathrm{Sub}_{16}\ (\mathit{acc}\ t_1)\ \ (\mathrm{Fetch}_{13}\ (\mathit{memory}\ t_1)\ \mathit{addr}) \wedge \\
& \mathit{idle}\ t_2 = \mathrm{F})
\end{aligned}
$$

$$
\mathit{state\_transition}_9 \mathrel{\widehat{=}} \ (\ \mathit{memory}\ t_2 = \mathit{memory}\ t_1 \wedge
$$

$$
\begin{aligned}
& pc\ t_2 = \mathrm{Inc}_{13}\ (pc\ t_1) \wedge \\
& acc\ t_2 = \mathrm{Fetch}_{13}\ (memory\ t_1)\ addr \wedge \\
& idle\ t_2 = \mathrm{F})
\end{aligned}
$$

$$
\begin{aligned}
state\_transition_{10} \mathrel{\widehat{=}} (\ & memory\ t_2 = \mathrm{Store}_{13}\ addr\ (acc\ t_1)\ (memory\ t_1) \wedge \\
& pc\ t_2 = \mathrm{Inc}_{13}\ (pc\ t_1) \wedge \\
& acc\ t_2 = acc\ t_1 \wedge \\
& idle\ t_2 = \mathrm{F})
\end{aligned}
$$

11.2.4 Resetting Global Structure

From the specification 2 we rearrange its let_in structure and then follow the construction 4 and 5 to select common blocks (sub-state transition set), and use the bracket to mark our selection for the parallel/sequential execution for the assignments of let_in sentences and predicate parts of condition sentences.

- **Re-arranging let_in structure:**

  We introduce

  let $v = \mathrm{Fetch}_{13}\ (memory\ t_1)\ (pc\ t_1)$ in

  to replace the two let_in sentences of the specification 2

  let $op = \mathrm{Val}_3\ (\mathrm{Opcode}\ (\mathrm{Fetch}_{13}\ (memory\ t_1)\ (pc\ t_1)))$ in

  let $addr = \mathrm{Cut}_{16\_13}\ (\mathrm{Fetch}_{13}\ (memory\ t_1)\ (pc\ t_1))$ in

  and use Val_3 (Opcode v) to replace op and use $\mathrm{Cut}_{16\_13}\ (v)$ to replace $addr$ for the specification 2, because we suppose that Val_3, Opcode, Fetch_{13}, and Cut_{13} can be implemented directly by primitive components at register transfer level.

- **Selecting common block:**

  We select

  $sub\_state\_transition_5 \mathrel{\widehat{=}} (pc\ t_2 = \mathrm{Inc}_{13}\ (pc\ t_1) \wedge idle\ t_2 = \mathrm{F})$

because it is shared as common sub_state transition set by state_transition 6, 7, 8, 9, and 10.

- **Selecting for parallel/sequential execution:**

  We select

  $\mathrm{Val}_2\ (knob\ t_1) = i \ \ (0 \leq i \leq 3)$ and

  $\mathrm{Val}_3\ (\mathrm{Opcode}\ v) = i \ \ (0 \leq i \leq 7)$

  as parallel condition predicates, because they are easy to be executed in parallel by a branch structure such as the one which is a part of DECODE in [50].

Specification 3:

$$
\begin{array}{l}
E\ 0 \wedge \\
\forall t_1.\ \exists t_2.\ \ E\ t_1 \Rightarrow (idle\ t_1 \Rightarrow \{Idle_{st}\} \downarrow \{Busy_{rs}\}) \wedge \\
\qquad\qquad\qquad\quad (\forall t.\ t_1 < t < t_2 \Rightarrow \neg E\ t) \ \wedge t_1 \leq t_2 \wedge E\ t_2
\end{array}
$$

$$
\begin{array}{l}
Idle_{st} \mathrel{\hat{=}}\ button\ t_1 \Rightarrow \\
\qquad\qquad \{\ (\mathrm{Val}_2\ (knob\ t_1) = 0) \Rightarrow \{state\_transition_0\} \downarrow \\
\qquad\qquad\quad (\mathrm{Val}_2\ (knob\ t_1) = 1) \Rightarrow \{state\_transition_1\} \downarrow \\
\qquad\qquad\quad (\mathrm{Val}_2\ (knob\ t_1) = 2) \Rightarrow \{state\_transition_2\} \downarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{state\_transition_3\}\} \downarrow \\
\qquad\qquad \{state\_transition_4\}
\end{array}
$$

$$
\begin{array}{l}
Busy_{rs} \mathrel{\hat{=}}\ button\ t_1 \Rightarrow \{state\_transition_4\} \downarrow \\
\qquad\qquad \{\mathrm{let}\ v = \mathrm{Fetch}_{13}\ (memory\ t_1)\ (pc\ t_1)\ \ \mathrm{in} \\
\qquad\qquad \{\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ v) = 0) \Rightarrow \{state\_transition_4\} \downarrow \\
\qquad\qquad\quad (\mathrm{Val}_3\ (\mathrm{Opcode}\ v) = 1) \Rightarrow \{state\_transition_5\} \downarrow \\
\qquad\qquad\quad (\mathrm{Val}_3\ (\mathrm{Opcode}\ v) = 2) \Rightarrow \\
\qquad\qquad\qquad \{(\mathrm{Val}_{16}\ (acc\ t_1) = 0) \Rightarrow \{state\_transition_5\} \downarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \{state\_transition_6\}\} \downarrow \\
\qquad\qquad\quad (\mathrm{Val}_3\ (\mathrm{Opcode}\ v) = 3) \Rightarrow \{state\_transition_7\} \downarrow
\end{array}
$$

$$(\mathrm{Val}_3\ (\mathrm{Opcode}\ v) = 4) \Rightarrow \{state\_transition_8\} \downarrow$$
$$(\mathrm{Val}_3\ (\mathrm{Opcode}\ v) = 5) \Rightarrow \{state\_transition_9\} \downarrow$$
$$(\mathrm{Val}_3\ (\mathrm{Opcode}\ v) = 6) \Rightarrow \{state\_transition_{10}\} \downarrow$$
$$\{state\_transition_6\}\}\}$$

$$\begin{aligned}
state_transition_0 \ \widehat{=}\ & (\ memory\ t_2 = memory\ t_1 \wedge \\
& pc\ t_2 = \mathrm{Cut}_{16_13}\ (switches\ t_1) \wedge \\
& acc\ t_2 = acc\ t_1 \wedge \\
& idle\ t_2 = \mathrm{T}) \\
state_transition_1 \ \widehat{=}\ & (\ memory\ t_2 = memory\ t_1 \wedge \\
& pc\ t_2 = pc\ t_1 \wedge \\
& acc\ t_2 = switches\ t_1 \wedge \\
& idle\ t_2 = \mathrm{T}) \\
state_transition_2 \ \widehat{=}\ & (\ memory\ t_2 = \mathrm{Store}_{13}\ (pc\ t_1)\ (acc\ t_1)\ (memory\ t_1) \wedge \\
& pc\ t_2 = pc\ t_1 \\
& acc\ t_2 = acc\ t_1 \wedge \\
& idle\ t_2 = \mathrm{T}) \\
state_transition_3 \ \widehat{=}\ & (\ memory\ t_2 = memory\ t_1 \wedge \\
& pc\ t_2 = pc\ t_1 \wedge \\
& acc\ t_2 = acc\ t_1 \wedge \\
& idle\ t_2 = \mathrm{F}) \\
state_transition_4 \ \widehat{=}\ & (\ memory\ t_2 = memory\ t_1 \wedge \\
& pc\ t_2 = pc\ t_1 \wedge \\
& acc\ t_2 = acc\ t_1 \wedge \\
& idle\ t_2 = \mathrm{T}) \\
state_transition_5 \ \widehat{=}\ & (\ memory\ t_2 = memory\ t_1 \wedge \\
& pc\ t_2 = \mathrm{Cut}_{16_13}\ v \wedge \\
& acc\ t_2 = acc\ t_1 \wedge \\
& idle\ t_2 = \mathrm{F}) \\
state_transition_6 \ \widehat{=}\ & sub_state_transition_0 \wedge [sub_state_transition_5]
\end{aligned}$$

$$state\_transition_7 \mathrel{\hat{=}} sub\_state\_transition_1 \wedge [sub\_state\_transition_5]$$

$$state\_transition_8 \mathrel{\hat{=}} sub\_state\_transition_2 \wedge [sub\_state\_transition_5]$$

$$state\_transition_9 \mathrel{\hat{=}} sub\_state\_transition_3 \wedge [sub\_state\_transition_5]$$

$$state\_transition_{10} \mathrel{\hat{=}} sub\_state\_transition_4 \wedge [sub\_state\_transition_5]$$

$$\begin{aligned} sub\_state\_transition_0 \mathrel{\hat{=}} (\ & memory\ t_2 = memory\ t_1 \wedge \\ & acc\ t_2 = acc\ t_1) \end{aligned}$$

$$\begin{aligned} & sub\_state\_transition_1 \\ & \mathrel{\hat{=}} (\ memory\ t_2 = memory\ t_1 \wedge \\ & \quad acc\ t_2 = \mathrm{Add}_{16}\ (acc\ t_1)\ \ (\mathrm{Fetch}_{13}\ (memory\ t_1)\ (\mathrm{Cut}_{16\_13}\ v))) \end{aligned}$$

$$\begin{aligned} & sub\_state\_transition_2 \\ & \mathrel{\hat{=}} (\ memory\ t_2 = memory\ t_1 \wedge \\ & \quad acc\ t_2 = \mathrm{Sub}_{16}\ (acc\ t_1)\ \ (\mathrm{Fetch}_{13}\ (memory\ t_1)\ (\mathrm{Cut}_{16\_13}\ v))) \end{aligned}$$

$$\begin{aligned} & sub\_state\_transition_3 \\ & \mathrel{\hat{=}} (\ memory\ t_2 = memory\ t_1 \wedge \\ & \quad acc\ t_2 = \mathrm{Fetch}_{13}\ (memory\ t_1)\ (\mathrm{Cut}_{16\_13}\ v)) \end{aligned}$$

$$\begin{aligned} & sub\_state\_transition_4 \\ & \mathrel{\hat{=}} (\ memory\ t_2 = \mathrm{Store}_{13}\ (\mathrm{Cut}_{16\_13}\ v)\ (acc\ t_1)\ (memory\ t_1) \wedge \\ & \quad acc\ t_2 = acc\ t_1) \end{aligned}$$

$$\begin{aligned} sub\_state\_transition_5 \mathrel{\hat{=}} (\ & pc\ t_2 = \mathrm{Inc}_{13}\ (pc\ t_1) \wedge \\ & idle\ t_2 = \mathrm{F}) \end{aligned}$$

11.2.5 Decomposing to Term Transition

This step is essential for the construction of a class of computers. We follow the construction 6 to decompose the specification 3 to a specification 4 below which is at term transition level.

As the mentioned by the convenience 5.3.1.1 in the further construction process the two constraints introduced in the specification 1

$\forall t.\ t_1 < t < t_2 \Rightarrow \neg(E\ t)$ and $t_1 < t_2$.

will be omitted but they will be certainly satisfied by the process.

From this subsection we will use the forms of the derivation models. We suggest reader to review them if you have not been familiar with them. To suit the convention we use x and y to replace t_1 and t_2.

Specification 4:

$$
\begin{array}{l}
E\ 0\ \wedge \\
(idle\ y = idle\ x \wedge kss_o)\ [E,\ E] \wedge \\
[E,\ (idle\ x)\ (E_1,\ E_2)] \wedge \\
(button\ y = button\ x \wedge kss_1)\ [E_1,\ E_1] \wedge \\
[E_1,\ (button\ x)\ (E_3,\ E_4)] \wedge \\
(button\ y = button\ x \wedge kss_2)\ [E_2,\ E_2] \wedge \\
[E_2,\ (button\ x)\ (E_4,\ E_5)] \wedge \\
(knob\ y = knob\ x \wedge kss_3)\ [E_3,\ E_3] \wedge \\
[E_3,\ (\mathrm{Val}_2\ (knob\ x) = 0)\ (E_6, \\
\qquad\qquad\qquad (\mathrm{Val}_2\ (knob\ x) = 1)\ (E_7, \\
\qquad\qquad\qquad\qquad\qquad (\mathrm{Val}_2\ (knob\ x) = 2)\ (E_8, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad E_9)))] \wedge \\
(v\ y = \mathrm{Fetch}_{13}\ (memory\ x)\ (pc\ x) \wedge kss_4)\ [E_5,\ E_{10}] \wedge \\
(v\ y = v\ x \wedge kss_5)\ [E_{10},\ E_{10}] \\
[E_{10},\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (v\ x)) = 0) \\
\quad (E_4,\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (v\ x)) = 1) \\
\qquad (E_{11},\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (v\ x)) = 2) \\
\qquad\quad (E_{12},\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (v\ x)) = 3) \\
\qquad\qquad (E_{13},\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (v\ x)) = 4) \\
\qquad\qquad\quad (E_{14},\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (v\ x)) = 5) \\
\qquad\qquad\qquad (E_{15},\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (v\ x)) = 6) \\
\qquad\qquad\qquad\quad (E_{16},\ E_{17})))))))]
\end{array}
$$

$$(acc\ y = acc\ x \wedge kss_6)\ [E_{12},\ E_{12}] \wedge$$
$$[E_{12}, (\mathrm{Val}_{16}\ (acc\ x) = 0)\ (E_{11},\ E_{18})] \wedge$$
$$st_0\ [E_6,\ E] \wedge st_1\ [E_7,\ E] \wedge st_2\ [E_8,\ E] \wedge\ \wedge$$
$$st_3\ [E_9,\ E] \wedge st_4\ [E_4,\ E] \wedge st_5\ [E_{11},\ E] \wedge$$
$$(sst_1 \wedge kss_7)\ [E_{13},\ E_{19}] \wedge (sst_2 \wedge kss_7)\ [E_{14},\ E_{19}] \wedge$$
$$(sst_3 \wedge kss_7)\ [E_{15},\ E_{19}] \wedge (sst_4 \wedge kss_7)\ [E_{16},\ E_{19}] \wedge$$
$$(sst_0 \wedge kss_7)\ [E_{17},\ E_{19}] \wedge (sst_0 \wedge kss_7)\ [E_{18},\ E_{19}] \wedge$$
$$(sst_5 \wedge kss_8)\ [E_{19},\ E]$$

Here,

$$\begin{aligned}
st_0 \ \widehat{=}\ \ (\ & memory\ y = memory\ x \wedge \\
& pc\ y = \mathrm{Cut}_{16_13}\ (switches\ x) \wedge \\
& acc\ y = acc\ x \wedge \\
& idle\ y = \mathrm{T})
\end{aligned}$$

$$\begin{aligned}
st_1 \ \widehat{=}\ \ (\ & memory\ y = memory\ x \wedge \\
& pc\ y = pc\ x \wedge \\
& acc\ y = switches\ x \wedge \\
& idle\ y = \mathrm{T})
\end{aligned}$$

$$\begin{aligned}
st_2 \ \widehat{=}\ \ (\ & memory\ y = \mathrm{Store}_{13}\ (pc\ x)\ (acc\ x)\ (memory\ x) \wedge \\
& pc\ y = pc\ x \\
& acc\ y = acc\ x \wedge \\
& idle\ y = \mathrm{T})
\end{aligned}$$

$$\begin{aligned}
st_3 \ \widehat{=}\ \ (\ & memory\ y = memory\ x \wedge \\
& pc\ y = pc\ x \wedge \\
& acc\ y = acc\ x \wedge \\
& idle\ y = \mathrm{F})
\end{aligned}$$

$$\begin{aligned}
st_4 \ \widehat{=}\ \ (\ & memory\ y = memory\ x \wedge \\
& pc\ y = pc\ x \wedge \\
& acc\ y = acc\ x \wedge \\
& idle\ y = \mathrm{T})
\end{aligned}$$

$$\begin{aligned}
st_5 \mathrel{\widehat{=}} (\ & \mathit{memory}\ y = \mathit{memory}\ x\ \wedge \\
& \mathit{pc}\ y = \mathrm{Cut}_{16_13}\ (v\ x)\ \wedge \\
& \mathit{acc}\ y = \mathit{acc}\ x\ \wedge \\
& \mathit{idle}\ y = \mathrm{F})
\end{aligned}$$

$$\begin{aligned}
sst_0 \mathrel{\widehat{=}} (\ & \mathit{memory}\ y = \mathit{memory}\ x\ \wedge \\
& \mathit{acc}\ y = \mathit{acc}\ x)
\end{aligned}$$

$$\begin{aligned}
sst_1 \mathrel{\widehat{=}} (\ & \mathit{memory}\ y = \mathit{memory}\ x\ \wedge \\
& \mathit{acc}\ y = \mathrm{Add}_{16}\ (\mathit{acc}\ x)\ \ (\mathrm{Fetch}_{13}\ (\mathit{memory}\ x)\ (\mathrm{Cut}_{16_13}\ v)))
\end{aligned}$$

$$\begin{aligned}
sst_2 \mathrel{\widehat{=}} (\ & \mathit{memory}\ y = \mathit{memory}\ x\ \wedge \\
& \mathit{acc}\ y = \mathrm{Sub}_{16}\ (\mathit{acc}\ x)\ \ (\mathrm{Fetch}_{13}\ (\mathit{memory}\ x)\ (\mathrm{Cut}_{16_13}\ v)))
\end{aligned}$$

$$\begin{aligned}
sst_3 \mathrel{\widehat{=}} (\ & \mathit{memory}\ y = \mathit{memory}\ x\ \wedge \\
& \mathit{acc}\ y = \mathrm{Fetch}_{13}\ (\mathit{memory}\ x)\ (\mathrm{Cut}_{16_13}\ v))
\end{aligned}$$

$$\begin{aligned}
sst_4 \mathrel{\widehat{=}} (\ & \mathit{memory}\ y = \mathrm{Store}_{13}\ (\mathrm{Cut}_{16_13}\ v)\ (\mathit{acc}\ x)\ (\mathit{memory}\ x)\ \wedge \\
& \mathit{acc}\ y = \mathit{acc}\ x)
\end{aligned}$$

$$\begin{aligned}
sst_5 \mathrel{\widehat{=}} (\ & \mathit{pc}\ y = \mathrm{Inc}_{13}\ (\mathit{pc}\ x)\ \wedge \\
& \mathit{idle}\ y = \mathrm{F})
\end{aligned}$$

$$kss_0 \mathrel{\widehat{=}} \mathit{button}\ y = \mathit{button}\ x \wedge kss_1$$

$$kss_1 \mathrel{\widehat{=}} \mathit{knob}\ y = \mathit{knob}\ x \wedge kss_3$$

$$kss_2 \mathrel{\widehat{=}} kss_6$$

$$kss_3 \mathrel{\widehat{=}} \mathit{switches}\ y = \mathit{switches}\ x \wedge kss_4$$

$$kss_4 \mathrel{\widehat{=}} kss_6$$

$$kss_5 \mathrel{\widehat{=}} v\ y = v\ x \wedge kss_6$$

$$kss_6 \mathrel{\widehat{=}} kss_7 \wedge kss_8$$

$$kss_7 \mathrel{\widehat{=}} \mathit{pc}\ y = \mathit{pc}\ x$$

$$kss_8 \mathrel{\widehat{=}} \mathit{memory}\ y = \mathit{memory}\ x \wedge \mathit{acc}\ y = \mathit{acc}\ x$$

It should be aware that although two models:

$$(sst_0 \wedge kss_7)\ [E_{17},\ E_{19}] \quad \text{and} \quad (sst_0 \wedge kss_7)\ [E_{18},\ E_{19}]$$

have the same specification entity but which has not been selected as common block. This is because its implementation does not expense device resource.

From the specification 4 we can get an abstract microprogram structure which describes the logic connection relation between the term transitions of the specification 4. This is a generally existent high level architecture in the design process of computer. It is significant that this high level architecture is derived by out derivation model very naturally.

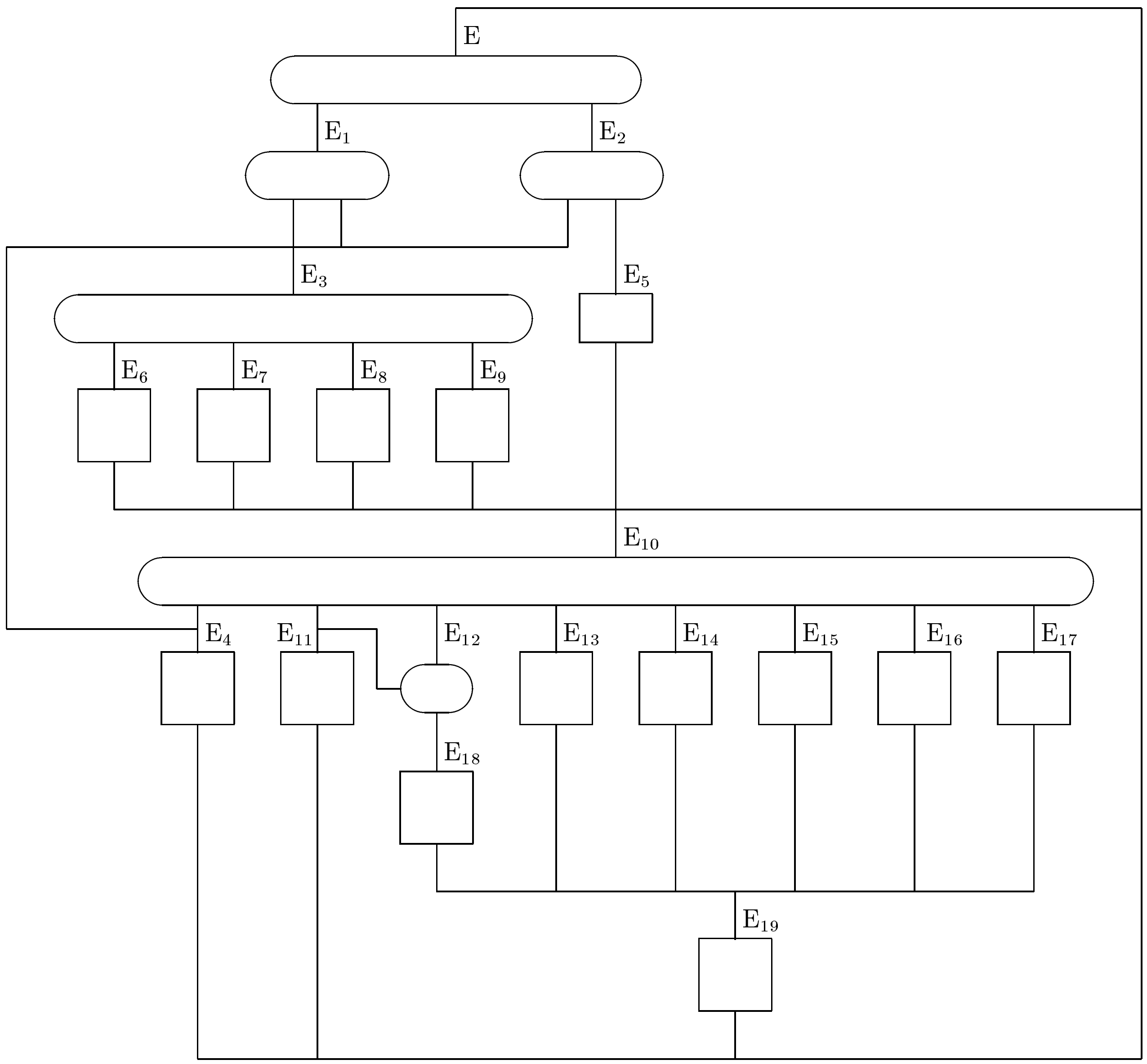


Figure 11.2.5.1 Flowchart of the specification of Gordon's computer at term transition level.

11.2.6 Decomposing to Function Unit and Connection

We follow the construction 7 to decompose the specification 4 to a specification which is at function unit and connection level and has its function units to be assigned to devices. We need to give more details of this construction process as, we will see, some Register and Gate from realistic world are assumed to have function of, instead of only a function, composition of two or more functions. For example, instead of ideal register:

$$\text{Reg}\ (c,\ i,\ o) \equiv \forall t.\ o\ (t+1) = (c\ t \Rightarrow i\ t \downarrow o\ t)$$

we use one from [50]:

$$\text{Reg}_c\ (c,\ i,\ o) \equiv \forall t.\ o\ (t+1) = (c\ t \Rightarrow \text{Cut}_{16\_13}\ (i\ t) \downarrow o\ t)$$

The register has an additional function Cut_{16_13}.

Convenience 11.2.6.1:

From this subsection we have a convenience that when we write a specification we will only write its new part and will omit the unchanged part. □

Separating out Single Complex Term Transition

We introduce the specification 5 as the first step of construction from the specification 4. In the specification 5 the single complex term transitions $term\_transition_i$ $(0 \leq i \leq 9)$ are pick out from state transition st_i $(0 \leq i \leq 5)$ and sub-state transition sst_i $(0 \leq i \leq 5)$.

Specification 5:

$st_0 = keep\_memory \wedge \underline{term\_transition_0} \wedge keep\_acc \wedge idle\_is\_true,$

$st_1 = keep\_memory \wedge keep\_pc \wedge \underline{term\_transition_1} \wedge idle\_is\_true,$

$st_2 = \underline{term\_transition_2} \wedge keep\_pc \wedge keep\_acc \wedge idle\_is\_true,$

$st_3 = keep\_memory \wedge keep\_pc \wedge keep\_acc \wedge idle\_is\_false,$

$st_4 = keep\_memory \wedge keep\_pc \wedge keep\_acc \wedge idle\_is\_true,$

$st_5 = keep\_memory \wedge \underline{term\_transition_3} \wedge keep\_acc \wedge idle\_is\_false,$

$sst_0 = keep\_memory \wedge keep\_acc,$

$sst_1 = keep\_memory \wedge \underline{term\_transition_4},$

$sst_2 = keep\_memory \wedge \underline{term\_transition_5},$

$sst_3 = keep\_memory \wedge \underline{term\_transition_6},$

$sst_4 = \underline{term\_transition_7} \wedge keep\_acc,$

$sst_5 = \underline{term\_transition_8} \wedge idle\_is\_false$

Here,

$keep\_memory \mathrel{\widehat{=}} memory\ y = memory\ x$

$keep\_pc \mathrel{\widehat{=}} pc\ y = pc\ x$

$keep\_acc \mathrel{\widehat{=}} acc\ y = acc\ x$

$idle\_is\_true \mathrel{\widehat{=}} idle\ y = \mathrm{T}$

$idle\_is\_false \mathrel{\widehat{=}} idle\ y = \mathrm{F}$

$term\_transition_0 \mathrel{\widehat{=}} pc\ y = \mathrm{Cut}_{16\_13}\ (switches\ x)$

$term\_transition_1 \mathrel{\widehat{=}} acc\ y = switches\ x$

$term\_transition_2 \mathrel{\widehat{=}} memory\ y = \mathrm{Store}_{13}\ (pc\ x)\ (acc\ x)\ (memory\ x)$

$term\_transition_3 \mathrel{\widehat{=}} pc\ y = \mathrm{Cut}_{16\_13}\ (v\ x)$

$term\_transition_4 \mathrel{\widehat{=}} acc\ y = \mathrm{Add}_{16}\ (acc\ x)\ \ (\mathrm{Fetch}_{13}\ (memory\ x)\ (\mathrm{Cut}_{16\_13}\ (v\ x)))$

$term\_transition_5 \mathrel{\widehat{=}} acc\ y = \mathrm{Sub}_{16}\ (acc\ x)\ \ (\mathrm{Fetch}_{13}\ (memory\ x)\ (\mathrm{Cut}_{16\_13}\ (v\ x)))$

$term\_transition_6 \mathrel{\widehat{=}} acc\ y = \mathrm{Fetch}_{13}\ (memory\ x)\ (\mathrm{Cut}_{16\_13}\ (v\ x))$

$term\_transition_7 \mathrel{\widehat{=}} memory\ y = \mathrm{Store}_{13}\ (\mathrm{Cut}_{16\_13}\ (v\ x))\ (acc\ x)\ (memory\ x)$

$term\_transition_8 \mathrel{\widehat{=}} pc\ y = \mathrm{Inc}_{13}\ (pc\ x)$

And we assume

$\underline{term\_transition_9} \mathrel{\widehat{=}} v\ y = \mathrm{Fetch}_{13}\ (memory\ x)\ (pc\ x)$

Decomposing to Function Unit and Connection Level

In this subsubsection we construction a specification 6 from the specification 5 by decomposing those complex term transitions to function units and connections.

Before the decomposition we need a data refinement: for Inc_{13} we invoke the extra axiom from [50] as below:

Axiom 11.2.6.1. [for $\mathbf{Inc}_{13}$]:

$\vdash \forall x.\ \mathrm{Inc}_{13}\ x = \mathrm{Cut}_{13}\ (\mathrm{Inc}_{16}\ (\mathrm{Pad}_{16}\ x))$

Therefore for the specification 5 its $\mathrm{Inc}_{13}\ x$ will be replaced by

$\mathrm{Cut}_{13}\ (\mathrm{Inc}_{16}\ (\mathrm{Pad}_{16}\ x))$.

In this construction step we introduce new entry predicates to break up each term transition $term\_transition_i$ $(0 \leq i \leq 9)$ and introduce new ports (as output and input ports). For example, for $term\_transtion_0$ we introduce the entry predicates E_{0a} and E_{0b} and the new ports i_{0a} and o_{0a}.

Specification 6:

$(term\_transition_0)\ [E_6,\ E] =$

$\quad (i_{0a}\ y = switches\ x)\ [E_6,\ E_{0a}] \wedge$

$\quad (o_{0a}\ y = \mathrm{Cut}_{13}\ (i_{0a}\ x))\ [E_{0a},\ E_{0b}] \wedge$

$\quad (pc\ y = o_{0a}\ x)\ [E_{0b},\ E]$

$(term\_transition_1)\ [E_7,\ E] =$

$\quad acc\ y = switches\ x[E_7,\ E]$

$(term\_transition_2)\ [E_8,\ E] =$

$\quad (i_{2a}\ y = pc\ x)\ [E_8,\ E_{2a}] \wedge$

$\quad (i_{2b}\ y = acc\ x)\ [E_8,\ E_{2a}] \wedge$

$\quad (i_{2c}\ y = memory\ x)\ [E_8,\ E_{2a}] \wedge$

$\quad (o_{2a}\ y = \mathrm{Store}_{13}\ (i_{2a}\ x)\ (i_{2b}\ x)\ (i_{2c}\ x))\ [E_{2a},\ E_{2b}] \wedge$

$\quad (memory\ y = o_{2a}\ x)\ [E_{2b},\ E]$

$(term\_transition_3)\ [E_{11},\ E] =$

$(i_{3a}\ y = v\ x)\ [E_{11},\ E_{3a}] \wedge$

$(o_{3a}\ y = \mathrm{Cut}_{16\_13}\ (i_{3a}\ x))\ [E_{3a},\ E_{3b}] \wedge$

$(pc\ y = o_{3a}\ x)\ [E_{3b},\ E]$

$(term\_transition_4)\ [E_{13},\ E_{19}] =$

$(i_{4a}\ y = acc\ x)\ [E_{13},\ E_{4a}] \wedge$

$(i_{4b}\ y = memory\ x)\ [E_{13},\ E_{4b}] \wedge$

$(i_{4c}\ y = v\ x)\ [E_{13},\ E_{4c}] \wedge$

$(o_{4a}\ y = \mathrm{Cut}_{16\_13}\ (i_{4c}\ x))\ [E_{4c},\ E_{4d}] \wedge$

$(i_{4d}\ y = o_{4a}\ x)\ [E_{4d},\ E_{4b}] \wedge$

$(o_{4b}\ y = \mathrm{Fetch}_{13}\ (i_{4b}\ x)\ (i_{4d}\ x))\ [E_{4b},\ E_{4e}] \wedge$

$(i_{4e}\ y = o_{4b}\ x)\ [E_{4e},\ E_{4a}] \wedge$

$(o_{4c}\ y = \mathrm{Add}_{16}\ (i_{4a}\ x)\ (i_{4e}\ x))\ [E_{4a},\ E_{4f}] \wedge$

$(acc\ y = o_{4c}\ x)\ [E_{4f},\ E_{19}]$

$(term\_transition_5)\ [E_{14},\ E_{19}] =$

$(i_{5a}\ y = acc\ x)\ [E_{14},\ E_{5a}] \wedge$

$(i_{5b}\ y = memory\ x)\ [E_{14},\ E_{5b}] \wedge$

$(i_{5c}\ y = v\ x)\ [E_{14},\ E_{5c}] \wedge$

$(o_{5a}\ y = \mathrm{Cut}_{16\_13}\ (i_{5c}\ x))\ [E_{5c},\ E_{5d}] \wedge$

$(i_{5d}\ y = o_{5a}\ x)\ [E_{5d},\ E_{5b}] \wedge$

$(o_{5b}\ y = \mathrm{Fetch}_{13}\ (i_{5b}\ x)\ (i_{5d}\ x))\ [E_{5b},\ E_{5e}] \wedge$

$(i_{5e}\ y = o_{5b}\ x)\ [E_{5e},\ E_{5a}] \wedge$

$(o_{5c}\ y = \mathrm{Sub}_{16}\ (i_{5a}\ x)\ (i_{5e}\ x))\ [E_{5a},\ E_{5f}] \wedge$

$(acc\ y = o_{5c}\ x)\ [E_{5f},\ E_{19}]$

$(term\_transition_6)\ [E_{15},\ E_{19}] =$

$(i_{6a}\ y = memory\ x)\ [E_{15},\ E_{6a}] \wedge$

$(i_{6b}\ y = v\ x)\ [E_{15},\ E_{6b}]$

$(o_{6a}\ y = \mathrm{Cut}_{16\_13}\ (i_{6b}\ x))\ [E_{6b},\ E_{6c}] \wedge$

$(i_{6c}\ y = o_{6a}\ x)\ [E_{6c},\ E_{6a}] \wedge$

$(o_{6b}\ y = \mathrm{Fetch}_{13}\ (i_{6a}\ x)\ (i_{6c}\ x))\ [E_{6a},\ E_{6d}] \wedge$

$(acc\ y = o_{6b}\ x)\ [E_{6d},\ E_{19}]$

$(term\_transition_7)\ [E_{16},\ E_{19}] =$

$(i_{7a}\ y = v\ x)\ [E_{16},\ E_{7a}] \wedge$

$(o_{7a}\ y = \mathrm{Cut}_{13}\ (i_{7a}\ x))\ [E_{7a},\ E_{7b}] \wedge$

$(i_{7b}\ y = o_{7a}\ x)\ [E_{7b},\ E_{7c}] \wedge$

$(i_{7c}\ y = acc\ x)\ [E_{16},\ E_{7c}] \wedge$

$(i_{7d}\ y = memory\ x)\ [E_{16},\ E_{7c}] \wedge$

$(o_{7b}\ y = \mathrm{Store}_{13}\ (i_{7b}\ x)\ (i_{7c}\ x)\ (i_{7d}\ x))\ [E_{7c},\ E_{7d}] \wedge$

$(memory\ y = o_{7b}\ x)\ [E_{7d},\ E_{19}]$

$(term\_transition_8)\ [E_{19},\ E] =$

$(i_{8a}\ y = pc\ x)\ [E_{19},\ E_{8a}] \wedge$

$(o_{8a}\ y = \mathrm{Pad}_{16}\ (i_{8a}\ x))\ [E_{8a},\ E_{8b}] \wedge$

$(i_{8b}\ y = o_{8a}\ x)\ [E_{8b},\ E_{8c}] \wedge$

$(o_{8b}\ y = \mathrm{Inc}_{16}\ (i_{8b}\ x))\ [E_{8c},\ E_{8d}] \wedge$

$(i_{8c}\ y = o_{8b}\ x)\ [E_{8d},\ E_{8e}] \wedge$

$(o_{8c}\ y = \mathrm{Cut}_{13}\ (i_{8c}\ x))\ [E_{8e},\ E_{8f}] \wedge$

$(pc\ y = o_{8c}\ x)\ [E_{8f},\ E]$

$(term\_transition_9)\ [E_5,\ E_{10}] =$

$(i_{9a}\ y = memory\ x)\ [E_5,\ E_{9a}] \wedge$

$(i_{9b}\ y = pc\ x)\ [E_5,\ E_{9a}] \wedge$

$(o_{9a}\ y = \mathrm{Fetch}_{13}\ (i_{9a}\ x)\ (i_{9b}\ x))\ [E_{9a},\ E_{9b}] \wedge$

$(v\ y = o_{9a}\ x)\ [E_{9b},\ E_{10}]$

Allocation

In the specification 6 we have not allocate devices to function units. In this subsubsection we do for this. Meanwhile we will treat register like as function unit as they have some non-register functions. We introduce two device kinds:

Mem $(read, write, addrs, date, mem, memout)$

$$\begin{aligned}&\equiv \forall t.\ (memout\ t = (read\ t \Rightarrow \text{Fetch}_{13}\ (mem\ t)\ (addrs\ t) \downarrow \text{Float}_{16}))\\ &\qquad \wedge\\ &\qquad (mem\ (t+1) = (write\ t \Rightarrow \text{Store}_{13}\ (addrs\ t)\ (data\ t)\ (mem\ t) \downarrow mem\ t))\end{aligned}$$

$$\begin{aligned}&\text{Alu}\ (add, sub, inc, i_1, i_2, o)\\ &\equiv \forall t.\ \ o\ t = (\ add\ t \Rightarrow \text{Add}_{16}\ (i_1\ t)\ (i_2\ t) \downarrow\\ &\qquad\qquad sub\ t \Rightarrow \text{Sub}_{16}\ (i_1\ t)\ (i_2\ t) \downarrow\\ &\qquad\qquad inc\ t \Rightarrow \text{Inc}_{16}\ (i_2\ t) \downarrow \text{Float}_{16})\end{aligned}$$

Correspondingly, in the specification 7 we introduce device instances:

Mem $(rmem,\ wmem,\ addrs,\ data,\ mem,\ mout)$ .... (m)

Alu $(add,\ sub,\ inc,\ ain1,\ ain2,\ aout)$ .... (a)

And four registers:

Reg $(wacc,\ i_{acc},\ acc)$ .... called Acc .... (r_a)

Reg_c $(wpc,\ i_{pc},\ o_{pc})$ .... called Pc .... (r_p)

Reg $(wir,\ i_{ir},\ v)$ .... called Ir .... (r_i)

Reg_c $(wmar,\ i_{mar},\ o_{mar})$ .... called Mar .... (r_m)

Reg and Reg_c are two different registers:

$\text{Reg}\ (c,\ i,\ o) \equiv \forall t.\ o\ (t+1) = (c\ t \Rightarrow i\ t \downarrow o\ t)$

$\text{Reg}_c\ (c,\ i,\ o) \equiv \forall t.\ o\ (t+1) = (c\ t \Rightarrow \text{Cut}_{16\_13}\ (i\ t) \downarrow o\ t)$

We still keep the arguments of these devices as variables until the last step to instantiating them to get an implementation.

In the construction we decompose the models formed as

$(acc\ y = \alpha)\ [E_i,\ E_o]$

$(v\ y = \beta)\ [E_i,\ E_o]$

to

$(i_{acc}\ y = \alpha)\ [E_i,\ E_m]\ \wedge (acc\ y = i_{acc}\ x)\ [E_m,\ E_o]$

$(i_{ir}\ y = \beta)\ [E_i,\ E_m]\ \wedge (v\ y = i_{ir}\ x)\ [E_m,\ E_o]$

This is because in the specification 4 and 5 the state signal *acc* and the signal v

appear in delay units with time constraint $x < y$ (the constraint must be taken as, at least, the realistic devices require a time unit delay) and they must be selected as the output of registers.

Specification 7:

$(term\_transition_0)\ [E_6,\ E] =$

$\quad (i_{i_{pc}}\ y = switches\ x)\ [E_6,\ E_{0a}] \wedge$

$\quad (o_{pc}\ y \overset{\mathrm{r}_p}{=} \mathrm{Cut}_{13}\ (i_{pc}\ x))\ [E_{0a},\ E_{0b}] \wedge$

$\quad (pc\ y = o_{pc}\ x)\ [E_{0b},\ E]$

$(term\_transition_1)\ [E_7,\ E] =$

$\quad i_{acc}\ y = switches\ x[E_7,\ E_{7a}] \wedge$

$\quad acc\ y \overset{\mathrm{r}_a}{=} i_{acc}\ x[E_{7a},\ E]$

$(term\_transition_2)\ [E_8,\ E] =$

$\quad (addrs\ y = pc\ x)\ [E_8,\ E_{2a}] \wedge$

$\quad (data\ y = acc\ x)\ [E_8,\ E_{2a}] \wedge$

$\quad (mem\ y = memory\ x)\ [E_8,\ E_{2a}] \wedge$

$\quad (mem\ y \overset{\mathrm{m}}{=} \mathrm{Store}_{13}\ (addrs\ x)\ (data\ x)\ (mem\ x))\ [E_{2a},\ E_{2b}] \wedge$

$\quad (memory\ y = mem\ x)\ [E_{2b},\ E]$

$(term\_transition_3)\ [E_{11},\ E] =$

$\quad (i_{pc}\ y = v\ x)\ [E_{11},\ E_{3a}] \wedge$

$\quad (o_{pc}\ y \overset{\mathrm{r}_p}{=} \mathrm{Cut}_{16\_13}\ (i_{pc}\ x))\ [E_{3a},\ E_{3b}] \wedge$

$\quad (pc\ y = o_{pc}\ x)\ [E_{3b},\ E]$

$(term\_transition_4)\ [E_{13},\ E_{19}] =$

$\quad (ain1\ y = acc\ x)\ [E_{13},\ E_{4a}] \wedge$

$\quad (mem\ y = memory\ x)\ [E_{13},\ E_{4b}] \wedge$

$\quad (i_{mar}\ y = v\ x)\ [E_{13},\ E_{4c}] \wedge$

$\quad (o_{mar}\ y \overset{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}\ (i_{mar}\ x))\ [E_{4c},\ E_{4d}] \wedge$

$\quad (addrs\ y = o_{mar}\ x)\ [E_{4d},\ E_{4b}] \wedge$

$\quad (mout\ y \overset{\mathrm{m}}{=} \mathrm{Fetch}_{13}\ (mem\ x)\ (addrs\ x))\ [E_{4b},\ E_{4e}] \wedge$

$\quad (ain2\ y = mout\ x)\ [E_{4e},\ E_{4a}] \wedge$

$$(aout\ y \stackrel{\mathrm{a}}{=} \mathrm{Add}_{16}\ (ain1\ x)\ (ain2\ x))\ [E_{4a},\ E_{4f}] \wedge$$
$$(i_{acc}\ y = aout\ x)\ [E_{4f},\ E_{4g}] \wedge$$
$$(acc\ y \stackrel{\mathrm{r}_a}{=} i_{acc}\ x)\ [E_{4g},\ E_{19}]$$
$$(term\_transition_5)\ [E_{14},\ E_{19}] =$$
$$(ain1\ y = acc\ x)\ [E_{14},\ E_{5a}] \wedge$$
$$(mem\ y = memory\ x)\ [E_{14},\ E_{5b}] \wedge$$
$$(i_{mar}\ y = v\ x)\ [E_{14},\ E_{5c}] \wedge$$
$$(o_{mar}\ y \stackrel{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}\ (i_{mar}\ x))\ [E_{5c},\ E_{5d}] \wedge$$
$$(addrs\ y = o_{mar}\ x)\ [E_{5d},\ E_{5b}] \wedge$$
$$(mout\ y \stackrel{\mathrm{m}}{=} \mathrm{Fetch}_{13}\ (mem\ x)\ (addrs\ x))\ [E_{5b},\ E_{5e}] \wedge$$
$$(ain2\ y = mout\ x)\ [E_{5e},\ E_{5a}] \wedge$$
$$(aout\ y \stackrel{\mathrm{a}}{=} \mathrm{Sub}_{16}\ (ain1\ x)\ (ain2\ x))\ [E_{5a},\ E_{5f}] \wedge$$
$$(i_{acc}\ y = aout\ x)\ [E_{5f},\ E_{5g}] \wedge$$
$$(acc\ y \stackrel{\mathrm{r}_a}{=} i_{acc}\ x)\ [E_{5g},\ E_{19}]$$
$$(term\_transition_6)\ [E_{15},\ E_{19}] =$$
$$(mem\ y = memory\ x)\ [E_{15},\ E_{6a}] \wedge$$
$$(i_{mar}\ y = v\ x)\ [E_{15},\ E_{6b}] \wedge$$
$$(o_{mar}\ y \stackrel{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}\ (i_{mar}\ x))\ [E_{6b},\ E_{6c}] \wedge$$
$$(addrs\ y = o_{mar}\ x)\ [E_{6c},\ E_{6a}] \wedge$$
$$(mout\ y \stackrel{\mathrm{m}}{=} \mathrm{Fetch}_{13}\ (mem\ x)\ (addrs\ x))\ [E_{6a},\ E_{6d}] \wedge$$
$$(i_{acc}\ y = mout\ x)\ [E_{6d},\ E_{6e}] \wedge$$
$$(acc\ y \stackrel{\mathrm{r}_a}{=} i_{acc}\ x)\ [E_{6e},\ E_{19}]$$
$$(term\_transition_7)\ [E_{16},\ E_{19}] =$$
$$(i_{mar}\ y = v\ x)\ [E_{16},\ E_{7a}] \wedge$$
$$(o_{mar}\ y \stackrel{\mathrm{r}_m}{=} \mathrm{Cut}_{13}\ (i_{mar}\ x))\ [E_{7a},\ E_{7b}] \wedge$$
$$(addrs\ y = o_{mar}\ x)\ [E_{7b},\ E_{7c}] \wedge$$
$$(data\ y = acc\ x)\ [E_{16},\ E_{7c}] \wedge$$
$$(mem\ y = memory\ x)\ [E_{16},\ E_{7c}] \wedge$$
$$(mem\ y \stackrel{\mathrm{m}}{=} \mathrm{Store}_{13}\ (addrs\ x)\ (data\ x)\ (mem\ x))\ [E_{7c},\ E_{7d}] \wedge$$

$$(memory\ y = mem\ x)\ [E_{7d},\ E_{19}]$$

$$(term\_transition_8)\ [E_{19},\ E] =$$

$$(i_{8a}\ y = pc\ x)\ [E_{19},\ E_{8a}] \wedge$$

$$(o_{8a}\ y = \mathrm{Pad}_{16}\ (i_{8a}\ x))\ [E_{8a},\ E_{8b}] \wedge$$

$$(ain2\ y = o_{8a}\ x)\ [E_{8b},\ E_{8c}] \wedge$$

$$(aout\ y \overset{\mathrm{a}}{=} \mathrm{Inc}_{16}\ (ain2\ x))\ [E_{8c},\ E_{8d}] \wedge$$

$$(i_{pc}\ y = aout\ x)\ [E_{8d},\ E_{8e}] \wedge$$

$$(o_{pc}\ y \overset{\mathrm{r}_p}{=} \mathrm{Cut}_{13}\ (i_{pc}\ x))\ [E_{8e},\ E_{8f}] \wedge$$

$$(pc\ y = o_{pc}\ x)\ [E_{8f},\ E]$$

$$(term\_transition_9)\ [E_5,\ E_{10}] =$$

$$(mem\ y = memory\ x)\ [E_5,\ E_{9a}] \wedge$$

$$(addrs\ y = pc\ x)\ [E_5,\ E_{9a}] \wedge$$

$$(mout\ y \overset{\mathrm{m}}{=} \mathrm{Fetch}_{13}\ (mem\ x)\ (addrs\ x))\ [E_{9a},\ E_{9b}] \wedge$$

$$(i_{ir}\ y = mout\ x)\ [E_{9b},\ E_{9c}] \wedge$$

$$(v\ y \overset{\mathrm{r}_i}{=} i_{ir}\ x)\ [E_{9c},\ E_{10}]$$

Scheduling

In specification 8 we give the result of scheduling for $term\_transition_i$ $(0 \leq i \leq 9)$. The scheduling is somewhat simple as the main work is to introduce delays for state signal *memory* and *acc*.

Specification 8:

$$(term\_transition_0)\ [E_6,\ E] =$$

$$(i_{i_{pc}}\ y = switches\ x)\ [E_6,\ E_{0a}] \wedge$$

$$(o_{pc}\ y \overset{\mathrm{r}_p}{=} \mathrm{Cut}_{13}\ (i_{pc}\ x))\ [E_{0a},\ E_{0b}] \wedge$$

$$(pc\ y = o_{pc}\ x)\ [E_{0b},\ E]$$

$$(term\_transition_1)\ [E_7,\ E] =$$

$$i_{acc}\ y = switches\ x[E_7,\ E_{7a}] \wedge$$

$$acc\ y \overset{\mathrm{r}_a}{=} i_{acc}\ x[E_{7a},\ E]$$

$(term\_transition_2)\ [E_8,\ E] =$

$(addrs\ y = pc\ x \wedge data\ y = acc\ x \wedge mem\ y = memory\ x)\ [E_8,\ E_{2a}] \wedge$

$(mem\ y \stackrel{\mathrm{m}}{=} \mathrm{Store}_{13}\ (addrs\ x)\ (data\ x)\ (mem\ x))\ [E_{2a},\ E_{2b}] \wedge$

$(memory\ y = mem\ x)\ [E_{2b},\ E]$

$(term\_transition_3)\ [E_{11},\ E] =$

$(i_{pc}\ y = v\ x)\ [E_{11},\ E_{3a}] \wedge$

$(o_{pc}\ y \stackrel{\mathrm{r}_p}{=} \mathrm{Cut}_{16\_13}\ (i_{pc}\ x))\ [E_{3a},\ E_{3b}] \wedge$

$(pc\ y = o_{pc}\ x)\ [E_{3b},\ E]$

$(term\_transition_4)\ [E_{13},\ E_{19}] =$

$(i_{mar}\ y = v\ x \wedge memory\ y = memory\ x \wedge acc\ y = acc\ x)\ [E_{13},\ E_{4c}] \wedge$

$(o_{mar}\ y \stackrel{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}\ (i_{mar}\ x) \wedge$

$memory\ y = memory\ x \wedge acc\ y = acc\ x)\ [E_{4c},\ E_{4d}] \wedge$

$(addrs\ y = o_{mar}\ x \wedge mem\ y = memory\ x \wedge acc\ y = acc\ x)\ [E_{4d},\ E_{4b}] \wedge$

$(mout\ y \stackrel{\mathrm{m}}{=} \mathrm{Fetch}_{13}\ (mem\ x)\ (addrs\ x) \wedge acc\ y = acc\ x)\ [E_{4b},\ E_{4e}] \wedge$

$(ain2\ y = mout\ x \wedge ain1\ y = acc\ x)\ [E_{4e},\ E_{4a}] \wedge$

$(aout\ y \stackrel{\mathrm{a}}{=} \mathrm{Add}_{16}\ (ain1\ x)\ (ain2\ x))\ [E_{4a},\ E_{4f}] \wedge$

$(i_{acc}\ y = aout\ x)\ [E_{4f},\ E_{4g}] \wedge$

$(acc\ y \stackrel{\mathrm{r}_a}{=} i_{acc}\ x)\ [E_{4g},\ E_{19}]$

$(term\_transition_5)\ [E_{14},\ E_{19}] =$

$(i_{mar}\ y = v\ x \wedge memory\ y = memory\ x \wedge acc\ y = acc\ x)\ [E_{14},\ E_{5c}] \wedge$

$(o_{mar}\ y \stackrel{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}\ (i_{mar}\ x) \wedge$

$memory\ y = memory\ x \wedge acc\ y = acc\ x)\ [E_{5c},\ E_{5d}] \wedge$

$(addrs\ y = o_{mar}\ x \wedge mem\ y = memory\ x \wedge acc\ y = acc\ x)\ [E_{5d},\ E_{5b}] \wedge$

$(mout\ y \stackrel{\mathrm{m}}{=} \mathrm{Fetch}_{13}\ (mem\ x)\ (addrs\ x) \wedge acc\ y = acc\ x)\ [E_{5b},\ E_{5e}] \wedge$

$(ain2\ y = mout\ x \wedge ain1\ y = acc\ x)\ [E_{5e},\ E_{5a}] \wedge$

$(aout\ y \stackrel{\mathrm{a}}{=} \mathrm{Sub}_{16}\ (ain1\ x)\ (ain2\ x))\ [E_{5a},\ E_{5f}] \wedge$

$(i_{acc}\ y = aout\ x)\ [E_{5f},\ E_{5g}] \wedge$

$(acc\ y \stackrel{\mathrm{r}_a}{=} i_{acc}\ x)\ [E_{5g},\ E_{19}]$

$(term\_transition_6)\ [E_{15},\ E_{19}] =$

$(i_{mar}\ y = v\ x \wedge memory\ y = memory\ x)\ [E_{15},\ E_{6b}] \wedge$

$(o_{mar}\ y \stackrel{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}\ (i_{mar}\ x) \wedge memory\ y = memory\ x)\ [E_{6b},\ E_{6c}] \wedge$

$(addrs\ y = o_{mar}\ x \wedge mem\ y = memory\ x)\ [E_{6c},\ E_{6a}] \wedge$

$(mout\ y \stackrel{\mathrm{m}}{=} \mathrm{Fetch}_{13}\ (mem\ x)\ (addrs\ x))\ [E_{6a},\ E_{6d}] \wedge$

$(i_{acc}\ y = mout\ x)\ [E_{6d},\ E_{6e}] \wedge$

$(acc\ y \stackrel{\mathrm{r}_a}{=} i_{acc}\ x)\ [E_{6e},\ E_{19}]$

$(term\_transition_7)\ [E_{16},\ E_{19}] =$

$(i_{mar}\ y = v\ x \wedge memory\ y = memory\ x \wedge acc\ y = acc\ x)\ [E_{16},\ E_{7a}] \wedge$

$(o_{mar}\ y \stackrel{\mathrm{r}_m}{=} \mathrm{Cut}_{13}\ (i_{mar}\ x) \wedge$

$memory\ y = memory\ x \wedge acc\ y = acc\ x)\ [E_{7a},\ E_{7b}] \wedge$

$(addrs\ y = o_{mar}\ x \wedge data\ y = acc\ x \wedge mem\ y = memory\ x)\ [E_{7b},\ E_{7c}] \wedge$

$(mem\ y \stackrel{\mathrm{m}}{=} \mathrm{Store}_{13}\ (addrs\ x)\ (data\ x)\ (mem\ x))\ [E_{7c},\ E_{7d}] \wedge$

$(memory\ y = mem\ x)\ [E_{7d},\ E_{19}]$

$(term\_transition_8)\ [E_{19},\ E] =$

$(i_{8a}\ y = pc\ x)\ [E_{19},\ E_{8a}] \wedge$

$(o_{8a}\ y = \mathrm{Pad}_{16}\ (i_{8a}\ x))\ [E_{8a},\ E_{8b}] \wedge$

$(ain2\ y = o_{8a}\ x)\ [E_{8b},\ E_{8c}] \wedge$

$(aout\ y \stackrel{\mathrm{a}}{=} \mathrm{Inc}_{16}\ (ain2\ x))\ [E_{8c},\ E_{8d}] \wedge$

$(i_{pc}\ y = aout\ x)\ [E_{8d},\ E_{8e}] \wedge$

$(o_{pc}\ y \stackrel{\mathrm{r}_p}{=} \mathrm{Cut}_{13}\ (i_{pc}\ x))\ [E_{8e},\ E_{8f}] \wedge$

$(pc\ y = o_{pc}\ x)\ [E_{8f},\ E]$

$(term\_transition_9)\ [E_5,\ E_{10}] =$

$(mem\ y = memory\ x \wedge addrs\ y = pc\ x)\ [E_5,\ E_{9a}] \wedge$

$(mout\ y \stackrel{\mathrm{m}}{=} \mathrm{Fetch}_{13}\ (mem\ x)\ (addrs\ x))\ [E_{9a},\ E_{9b}] \wedge$

$(i_{ir}\ y = mout\ x)\ [E_{9b},\ E_{9c}]$

$(v\ y \stackrel{\mathrm{r}_i}{=} i_{ir}\ x)\ [E_{9c},\ E_{10}]$

The specification 8 has established the main structure of the implementation of Gordon's computer because from the specification 4 and 5 we can see all other parts only are about delay unit and branch structures.

11.2.7 Introducing Bus, Connection and Microinstruction

Introducing Bus, Gate and Connection

Following the specification 8 we introduce bus, gate and connection.

We introduce a group of Buses and Gates (see page 118):

Bus $(o_{11},\ o_{12},\ o_{13},\ i_{acc})$ .... called Bus1 .... (r_1)

Bus $(o_{11},\ o_{12},\ o_{23},\ i_{pc})$ .... called Bus2 .... (r_2)

Bus $(o_{31},\ o_{32},\ i_{a2})$ .... called Bus3 .... (r_3)

Bus $(o_{32},\ v,\ i_a)$ .... called Bus4 .... (r_4)

Gate $(rsw,\ switches,\ o_{11})$ .... (Gate1)

Gate $(ralu,\ o_a,\ o_{12})$ .... (Gate2)

Gate $(rm_1,\ o_m,\ o_{13})$ .... (Gate3)

Gate $(rir,\ v,\ o_{23})$ .... (Gate4)

$\text{Gate}_p\ (rm_3,\ o_m,\ o_{31})$ .... (Gate5)

$\text{Gate}_p\ (rpc,\ pc, o_{32})$ .... (Gate6)

Note: above we write o_a and o_m for *aout* and *mout* respectively.

Here, we introduce a new type gate:

$\text{Gate}_p\ (c,\ i,\ o) \equiv \forall t.\ o\ t = (c\ t \Rightarrow \text{Mk\_Tri}\ (\text{Pad}_{16\_13}\ (i\ t)) \downarrow \text{Float}_{16})$ ...... (gs_p)

To introduce connection we use pc, $memory$, o_{mar}, acc and $mout$ to replace o_{pc}, mem, $addrs$, $ain1$ and i_{ir} respectively. Meanwhile when introduce a connection we make time interval become zero (instantiating y by x)

As the function Fetch_{13} and the device Alu takes zero time interval to complete their functions we can make the logic boundary of their corresponding models become to have the same entry and leaving predicates and the same begin and end time points (instantiating y by x).

For the connection link which has been selected for Bus its logic boundary is also constructed with the same entry and leaving predicate and zero time interval.

For example

$$(i_{pc}\ y \overset{\mathrm{b}_2}{=} switches\ x)\ [E_6,\ E_{0a}]$$

is constructed to

$$(i_{pc}\ x \overset{\mathrm{b}_2}{=} switches\ x)\ [E_6,\ E_6].$$

Here, E_{0a} is replaced by E_6 and y is replaced by x.

A special case is that we replace

$$(i_{8a}\ y = pc\ x)\ [E_{19},\ E_{8a}] \wedge$$
$$(o_{8a}\ y = \mathrm{Pad}_{16}\ (i_{8a}\ x))\ [E_{8a},\ E_{8b}] \wedge$$
$$(ain2\ y = o_{8a}\ x)\ [E_{8b},\ E_{8c}]$$

by

$$(ain2\ x \overset{\mathrm{b}_3}{=} \mathrm{Pad}_{16}\ (pc\ x))\ [E_{19},\ E_{19}]$$

because we need to introduce a special Gate_p for Bus [50].

As $addrs$ is replaced by o_{mar} which is the output of register Mar in $term\_transirion_2$ and $term\_transirion_9$ we replace

$$(\alpha_1 \wedge addrs\ y = \mathrm{Pad}_{16\_13}\ (pc\ x) \wedge \alpha_2)\ [E_\beta,\ E_\gamma]$$

by

$$(\alpha_1 \wedge i_{mar}\ x \overset{\mathrm{b}_4}{=} pc\ x \wedge \alpha_2)\ [E_\beta,\ E_\beta]\ \text{(Again, Gate}_p\text{ for Bus)}$$
$$\wedge$$
$$(\alpha_1 \wedge o_{mar}\ x \overset{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}\ (i_{mar}\ x) \wedge \alpha_2)\ [E_\beta,\ E_\gamma]$$

Specification 9:

$$(term\_transition_0)\ [E_6,\ E] =$$
$$(i_{i_{pc}}\ x \overset{\mathrm{b}_2}{=} switches\ x)\ [E_6,\ E_6] \wedge$$
$$(pc\ y \overset{\mathrm{r}_p}{=} \mathrm{Cut}_{13}\ (i_{pc}\ x))\ [E_6,\ E] \wedge$$
$$(pc\ x \overset{\mathrm{c}}{=} pc\ x)\ [E,\ E]$$
$$(term\_transition_1)\ [E_7,\ E] =$$
$$(i_{acc}\ x \overset{\mathrm{b}_1}{=} switches\ x)\ [E_7,\ E_7] \wedge$$

$$(acc\ y \stackrel{\mathrm{r}_a}{=} i_{acc}\ x)\ [E_7,\ E]$$

$$(term\_transition_2)\ [E_8,\ E] =$$

$$(i_{mar}\ x \stackrel{\mathrm{b}_4}{=} \mathrm{Pad}_{16\_13}\ (pc\ x) \wedge acc\ x = acc\ x \wedge memory\ x = memory\ x)\ [E_8,\ E_8] \wedge$$

$$(o_{mar}\ y \stackrel{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}\ (i_{mar}\ x) \wedge$$

$$acc\ y = acc\ x \wedge memory\ y = memory\ x)\ [E_8,\ E_{2a}] \wedge$$

$$(memory\ y \stackrel{\mathrm{m}}{=} \mathrm{Store}_{13}\ (o_{mar}\ x)\ (acc\ x)\ (memory\ x))\ [E_{2a},\ E] \wedge$$

$$(memory\ x \stackrel{\mathrm{c}}{=} memory\ x)\ [E,\ E]$$

$$(term\_transition_3)\ [E_{11},\ E] =$$

$$(i_{pc}\ x \stackrel{\mathrm{b}_2}{=} v\ x)\ [E_{11},\ E_{11}] \wedge$$

$$(pc\ y \stackrel{\mathrm{r}_p}{=} \mathrm{Cut}_{16\_13}\ (i_{pc}\ x))\ [E_{11},\ E] \wedge$$

$$(pc\ x \stackrel{\mathrm{c}}{=} pc\ x)\ [E,\ E]$$

$$(term\_transition_4)\ [E_{13},\ E_{19}] =$$

$$(i_{mar}\ x \stackrel{\mathrm{b}_4}{=} v\ x \wedge memory\ x = memory\ x \wedge acc\ x = acc\ x)\ [E_{13},\ E_{13}] \wedge$$

$$(o_{mar}\ y \stackrel{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}\ (i_{mar}\ x) \wedge$$

$$memory\ y = memory\ x \wedge acc\ y = acc\ x)\ [E_{13},\ E_{4d}] \wedge$$

$$(o_{mar}\ x \stackrel{\mathrm{c}}{=} o_{mar}\ x \wedge memory\ x = memory\ x \wedge acc\ x = acc\ x)\ [E_{4d},\ E_{4d}] \wedge$$

$$(mout\ x \stackrel{\mathrm{m}}{=} \mathrm{Fetch}_{13}\ (memory\ x)\ (o_{mar}\ x) \wedge acc\ x = acc\ x)\ [E_{4d},\ E_{4d}] \wedge$$

$$(ain2\ x \stackrel{\mathrm{b}_3}{=} mout\ x \wedge acc\ x \stackrel{\mathrm{c}}{=} acc\ x)\ [E_{4d},\ E_{4d}] \wedge$$

$$(aout\ x \stackrel{\mathrm{a}}{=} \mathrm{Add}_{16}\ (acc\ x)\ (ain2\ x))\ [E_{4d},\ E_{4d}] \wedge$$

$$(i_{acc}\ x \stackrel{\mathrm{b}_1}{=} aout\ x)\ [E_{4d},\ E_{4d}] \wedge$$

$$(acc\ y \stackrel{\mathrm{r}_a}{=} i_{acc}\ x)\ [E_{4d},\ E_{19}]$$

$$(term\_transition_5)\ [E_{14},\ E_{19}] =$$

$$(i_{mar}\ x \stackrel{\mathrm{b}_4}{=} v\ x \wedge memory\ x = memory\ x \wedge acc\ x = acc\ x)\ [E_{14},\ E_{14}] \wedge$$

$$(o_{mar}\ y \stackrel{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}\ (i_{mar}\ x) \wedge$$

$$memory\ y = memory\ x \wedge acc\ y = acc\ x)\ [E_{14},\ E_{5d}] \wedge$$

$$(o_{mar}\ x \stackrel{\mathrm{c}}{=} o_{mar}\ x \wedge memory\ x = memory\ x \wedge acc\ x = acc\ x)\ [E_{5d},\ E_{5d}] \wedge$$

$$(mout\ x \stackrel{\mathrm{m}}{=} \mathrm{Fetch}_{13}\ (memory\ x)\ (o_{mar}\ x) \wedge acc\ x \stackrel{\mathrm{c}}{=} acc\ x)\ [E_{5d},\ E_{5d}] \wedge$$

$$(ain2\ x \stackrel{\mathrm{b}_3}{=} mout\ x \wedge acc\ x = acc\ x)\ [E_{5d},\ E_{5d}] \wedge$$

$$(aout\ x \stackrel{\mathrm{a}}{=} \mathrm{Sub}_{16}\ (acc\ x)\ (ain2\ x))\ [E_{5d},\ E_{5d}] \wedge$$

$$(i_{acc}\; x \stackrel{\mathrm{b_1}}{=} aout\; x)\; [E_{5d},\; E_{5d}] \wedge$$
$$(acc\; y \stackrel{\mathrm{r}_a}{=} i_{acc}\; x)\; [E_{5d},\; E_{19}]$$
$$(term\_transition_6)\; [E_{15},\; E_{19}] =$$
$$(i_{mar}\; x \stackrel{\mathrm{b_4}}{=} v\; x \wedge memory\; x = memory\; x)\; [E_{15},\; E_{15}] \wedge$$
$$(o_{mar}\; y \stackrel{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}\; (i_{mar}\; x) \wedge memory\; y = memory\; x)\; [E_{15},\; E_{6c}] \wedge$$
$$(o_{mar}\; x \stackrel{\mathrm{c}}{=} o_{mar}\; x \wedge memory\; x = memory\; x)\; [E_{6c},\; E_{6c}] \wedge$$
$$(mout\; x \stackrel{\mathrm{m}}{=} \mathrm{Fetch}_{13}\; (memory\; x)\; (o_{mar}\; x))\; [E_{6c},\; E_{6c}] \wedge$$
$$(i_{acc}\; x \stackrel{\mathrm{b_1}}{=} mout\; x)\; [E_{6c},\; E_{6c}] \wedge$$
$$(acc\; y \stackrel{\mathrm{r}_a}{=} i_{acc}\; x)\; [E_{6c},\; E_{19}]$$
$$(term\_transition_7)\; [E_{16},\; E_{19}] =$$
$$(i_{mar}\; x \stackrel{\mathrm{b_4}}{=} v\; x \wedge memory\; x = memory\; x \wedge acc\; x = acc\; x)\; [E_{16},\; E_{16}] \wedge$$
$$(o_{mar}\; y \stackrel{\mathrm{r}_m}{=} \mathrm{Cut}_{13}\; (i_{mar}\; x) \wedge$$
$$memory\; y = memory\; x \wedge acc\; y = acc\; x)\; [E_{16},\; E_{7b}] \wedge$$
$$(o_{mar}\; x \stackrel{\mathrm{c}}{=} o_{mar}\; x \wedge acc\; x \stackrel{\mathrm{c}}{=} acc\; x \wedge memory\; x \stackrel{\mathrm{c}}{=} memory\; x)\; [E_{7b},\; E_{7b}] \wedge$$
$$(memory\; y \stackrel{\mathrm{m}}{=} \mathrm{Store}_{13}\; (o_{mar}\; x)\; (acc\; x)\; (memory\; x))\; [E_{7b},\; E_{19}] \wedge$$
$$(memory\; y \stackrel{\mathrm{c}}{=} memory\; x)\; [E_{19},\; E_{19}]$$
$$(term\_transition_8)\; [E_{19},\; E] =$$
$$(ain2\; x \stackrel{\mathrm{b_3}}{=} \mathrm{Pad}_{16}\; (pc\; x))\; [E_{19},\; E_{19}] \wedge$$
$$(aout\; x \stackrel{\mathrm{a}}{=} \mathrm{Inc}_{16}\; (ain2\; x))\; [E_{19},\; E_{19}] \wedge$$
$$(i_{pc}\; x \stackrel{\mathrm{b_2}}{=} aout\; x)\; [E_{19},\; E_{19}] \wedge$$
$$(pc\; y \stackrel{\mathrm{r}_p}{=} \mathrm{Cut}_{13}\; (i_{pc}\; x))\; [E_{19},\; E] \wedge$$
$$(pc\; x \stackrel{\mathrm{c}}{=} pc\; x)\; [E,\; E]$$
$$(term\_transition_9)\; [E_5,\; E_{10}] =$$
$$(memory\; x \stackrel{\mathrm{c}}{=} memory\; x \wedge i_{mar}\; x \stackrel{\mathrm{b_4}}{=} \mathrm{Pad}_{16\_13}\; (pc\; x))\; [E_5,\; E_5] \wedge$$
$$(memory\; y = memory\; x \wedge o_{mar}\; y \stackrel{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}(i_{mar}\; x))\; [E_5,\; E_{9a}] \wedge$$
$$(mout\; y \stackrel{\mathrm{m}}{=} \mathrm{Fetch}_{13}\; (memory\; x)\; (o_{mar}\; x))\; [E_{9a},\; E_{9a}] \wedge$$
$$(mout\; y \stackrel{\mathrm{c}}{=} mout\; x)\; [E_{9a},\; E_{9a}] \wedge$$
$$(v\; y \stackrel{\mathrm{r}_i}{=} mout\; x)\; [E_{9a},\; E_{10}]$$

A more technical detail is about the structure of group of Bus and Gatas which are connected each other as a fixed module. In principle a Gate is only used for a Bus. But for saving device recourse it is possible to use a Gate for two or more Bus.

For example, we must use two Gates: Gate3: Gate $(rm_1,\ o_m,\ o_{13})$ for the Bus1 connecting *mout* to i_{acc} in $term\_transition_6$ and Gate5: Gate $(rm_3,\ o_m,\ o_{31})$ for the Bus3 connecting *mout* to *ain*2 in the $term\_transition_4$ and $term\_transition_5$ respectively. Otherwise, namely we use one Gate, let suppose its control signal is rm to replace rm_1 and rm_3, for the two Buses, it can lead to an inconsistency. By $term\_transition_5$ we get one i_{acc}, however, by $term\_transition_6$ we can also get another i_{acc} *simultaneously.*

However, we can use one Gate for two Buses, for example Gate1 for Bus1 and Bus2 as this does not lead to inconsistency.

Introducing Abstract Microinstruction

In this subsubsection we clean up the specification 9 by combine models such that a model can be just corresponding to an abstract microinstruction.

Specification 10:

$(term\_transition_0)\ [E_6,\ E] =$

$(i_{i_{pc}}\ x \stackrel{\text{b}_2}{=} switches\ x) \wedge$

$pc\ y \stackrel{\text{r}_p}{=} \text{Cut}_{13}\ (i_{pc}\ x)$

$)\ [E_6,\ E]$

$(term\_transition_1)\ [E_7,\ E] =$

$(i_{acc}\ x \stackrel{\text{b}_1}{=} switches\ x) \wedge$

$acc\ y \stackrel{\text{r}_a}{=} i_{acc}\ x$

$)\ [E_7,\ E]$

$(term\_transition_2)\ [E_8,\ E] =$

$(i_{mar}\ x \overset{\mathrm{b_4}}{=} \mathrm{Pad}_{16\_13}\ (pc\ x) \wedge acc\ x = acc\ x \wedge memory\ x = memory\ x) \wedge$
$o_{mar}\ y \overset{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}\ (i_{mar}\ x) \wedge$
$acc\ y = acc\ x \wedge memory\ y = memory\ x$
$)\ [E_8,\ E_{2a}] \wedge$
$(memory\ y \overset{\mathrm{m}}{=} \mathrm{Store}_{13}\ (o_{mar}\ x)\ (acc\ x)\ (memory\ x)$
$)\ [E_{2a},\ E]$

$(term\_transition_3)\ [E_{11},\ E] =$
$(i_{pc}\ x \overset{\mathrm{b_2}}{=} v\ x \wedge$
$pc\ y \overset{\mathrm{r}_p}{=} \mathrm{Cut}_{16\_13}\ (i_{pc}\ x)$
$)\ [E_{11},\ E]$

$(term\_transition_4)\ [E_{13},\ E_{19}] =$
$(i_{mar}\ x \overset{\mathrm{b_4}}{=} v\ x \wedge memory\ x = memory\ x \wedge acc\ x = acc\ x \wedge$
$o_{mar}\ y \overset{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}\ (i_{mar}\ x) \wedge$
$memory\ y = memory\ x \wedge acc\ y = acc\ x$
$)\ [E_{13},\ E_{4a}] \wedge$
$(mout\ x \overset{\mathrm{m}}{=} \mathrm{Fetch}_{13}\ (memory\ x)\ (o_{mar}\ x) \wedge acc\ x = acc\ x \wedge$
$ain2\ x \overset{\mathrm{b_3}}{=} mout\ x \wedge acc\ x \overset{\mathrm{c}}{=} acc\ x \wedge$
$aout\ x \overset{\mathrm{a}}{=} \mathrm{Add}_{16}\ (acc\ x)\ (ain2\ x) \wedge$
$i_{acc}\ x \overset{\mathrm{b_1}}{=} aout\ x \wedge$
$acc\ y \overset{\mathrm{r}_a}{=} i_{acc}\ x$
$)\ [E_{4a},\ E_{19}]$

$(term\_transition_5)\ [E_{14},\ E_{19}] =$
$(i_{mar}\ x \overset{\mathrm{b_4}}{=} v\ x \wedge memory\ x = memory\ x \wedge acc\ x = acc\ x \wedge$
$o_{mar}\ y \overset{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}\ (i_{mar}\ x) \wedge$
$memory\ y = memory\ x \wedge acc\ y = acc\ x$
$)\ [E_{14},\ E_{5a}] \wedge$
$(mout\ x \overset{\mathrm{m}}{=} \mathrm{Fetch}_{13}\ (memory\ x)\ (o_{mar}\ x) \wedge acc\ x \overset{\mathrm{c}}{=} acc\ x \wedge$
$ain2\ x \overset{\mathrm{b_3}}{=} mout\ x \wedge acc\ x = acc\ x \wedge$
$aout\ x \overset{\mathrm{a}}{=} \mathrm{Sub}_{16}\ (acc\ x)\ (ain2\ x) \wedge$

$i_{acc}\ x \overset{\text{b}_1}{=} aout\ x \wedge$

$acc\ y \overset{\text{r}_a}{=} i_{acc}\ x$

$)\ [E_{5a},\ E_{19}]$

$(term\_transition_6)\ [E_{15},\ E_{19}] =$

$(i_{mar}\ x \overset{\text{b}_4}{=} v\ x \wedge memory\ x = memory\ x \wedge$

$o_{mar}\ y \overset{\text{r}_m}{=} \text{Cut}_{16\_13}\ (i_{mar}\ x) \wedge memory\ y = memory\ x \wedge$

$)\ [E_{15},\ E_{6a}] \wedge$

$(mout\ x \overset{\text{m}}{=} \text{Fetch}_{13}\ (memory\ x)\ (o_{mar}\ x) \wedge$

$i_{acc}\ x \overset{\text{b}_1}{=} mout\ x \wedge$

$acc\ y \overset{\text{r}_a}{=} i_{acc}\ x$

$)\ [E_{6a},\ E_{19}]$

$(term\_transition_7)\ [E_{16},\ E_{19}] =$

$(i_{mar}\ x \overset{\text{b}_4}{=} v\ x \wedge memory\ x = memory\ x \wedge acc\ x = acc\ x \wedge$

$o_{mar}\ y \overset{\text{r}_m}{=} \text{Cut}_{13}\ (i_{mar}\ x) \wedge$

$memory\ y = memory\ x \wedge acc\ y = acc\ x$

$)\ [E_{16},\ E_{7a}] \wedge$

$(memory\ y \overset{\text{m}}{=} \text{Store}_{13}\ (o_{mar}\ x)\ (acc\ x)\ (memory\ x)$

$)\ [E_{7a},\ E_{19}]$

$(term\_transition_8)\ [E_{19},\ E] =$

$(ain2\ x \overset{\text{b}_3}{=} \text{Pad}_{16}\ (pc\ x) \wedge$

$aout\ x \overset{\text{a}}{=} \text{Inc}_{16}\ (ain2\ x) \wedge$

$i_{pc}\ x \overset{\text{b}_2}{=} aout\ x \wedge$

$pc\ y \overset{\text{r}_p}{=} \text{Cut}_{13}\ (i_{pc}\ x)$

$)\ [E_{19},\ E]$

$(term\_transition_9)\ [E_5,\ E_{10}] =$

$(memory\ x \overset{\text{c}}{=} memory\ x \wedge i_{mar}\ x \overset{\text{b}_4}{=} \text{Pad}_{16\_13}\ (pc\ x) \wedge$

$memory\ y = memory\ x \wedge o_{mar}\ y \overset{\text{r}_m}{=} \text{Cut}_{16\_13}(i_{mar}\ x)$

$)\ [E_5,\ E_{9a}] \wedge$

$(mout\ x \overset{\text{m}}{=} \text{Fetch}_{13}\ (memory\ x)\ (o_{mar}\ x) \wedge$

$v\ y \overset{\mathrm{r}_i}{=} mout\ x$

$)\ [E_{9a},\ E_{10}]$

For a shorter writing form before to derive an implementation we have replacements for signals. We use "i_a" to replace "i_{mar}", "a" to replace "o_{mar}", "i_{a2}" to replace "$ain2$", "o_a" to replace "$aout$" and "o_m" to replace "$mout$" to get a specification $10'$.

Specification 10′:

$(term\_transition_0)\ [E_6,\ E] =$

$(i_{i_{pc}}\ x \overset{\mathrm{b}_2}{=} switches\ x) \wedge$

$pc\ y \overset{\mathrm{r}_p}{=} \mathrm{Cut}_{13}\ (i_{pc}\ x)$

$)\ [E_6,\ E]$

$(term\_transition_1)\ [E_7,\ E] =$

$(i_{acc}\ x \overset{\mathrm{b}_1}{=} switches\ x) \wedge$

$acc\ y \overset{\mathrm{r}_a}{=} i_{acc}\ x$

$)\ [E_7,\ E]$

$(term\_transition_2)\ [E_8,\ E] =$

$(i_a\ x \overset{\mathrm{b}_4}{=} \mathrm{Pad}_{16\_13}\ (pc\ x) \wedge acc\ x = acc\ x \wedge memory\ x = memory\ x) \wedge$

$a\ y \overset{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}\ (i_a\ x) \wedge$

$acc\ y = acc\ x \wedge memory\ y = memory\ x$

$)\ [E_8,\ E_{2a}] \wedge$

$(memory\ y \overset{\mathrm{m}}{=} \mathrm{Store}_{13}\ (a\ x)\ (acc\ x)\ (memory\ x)$

$)\ [E_{2a},\ E]$

$(term\_transition_3)\ [E_{11},\ E] =$

$(i_{pc}\ x \overset{\mathrm{b}_2}{=} v\ x \wedge$

$pc\ y \overset{\mathrm{r}_p}{=} \mathrm{Cut}_{16\_13}\ (i_{pc}\ x)$

$)\ [E_{11},\ E]$

$(term\_transition_4)\ [E_{13},\ E_{19}] =$

$(i_a\ x \overset{\mathrm{b}_4}{=} v\ x \wedge memory\ x = memory\ x \wedge acc\ x = acc\ x \wedge$

$$
\begin{aligned}
&\quad a\ y \overset{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}\ (i_a\ x) \wedge \\
&\quad memory\ y = memory\ x \wedge acc\ y = acc\ x \\
&\ )\ [E_{13},\ E_{4a}] \wedge \\
&\ (o_m\ x \overset{\mathrm{m}}{=} \mathrm{Fetch}_{13}\ (memory\ x)\ (a\ x) \wedge acc\ x = acc\ x \wedge \\
&\quad i_{a2}\ x \overset{\mathrm{b}_3}{=} o_m\ x \wedge acc\ x \overset{\mathrm{c}}{=} acc\ x \wedge \\
&\quad o_a\ x \overset{\mathrm{a}}{=} \mathrm{Add}_{16}\ (acc\ x)\ (i_{a2}\ x)\ \wedge \\
&\quad i_{acc}\ x \overset{\mathrm{b}_1}{=} o_a\ x \wedge \\
&\quad acc\ y \overset{\mathrm{r}_a}{=} i_{acc}\ x \\
&\ )\ [E_{4a},\ E_{19}] \\
(&term\_transition_5)\ [E_{14},\ E_{19}] = \\
&\ (i_a\ x \overset{\mathrm{b}_4}{=} v\ x \wedge memory\ x = memory\ x \wedge acc\ x = acc\ x \wedge \\
&\quad a\ y \overset{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}\ (i_a\ x) \wedge \\
&\quad memory\ y = memory\ x \wedge acc\ y = acc\ x \\
&\ )\ [E_{14},\ E_{5a}] \wedge \\
&\ (o_m\ x \overset{\mathrm{m}}{=} \mathrm{Fetch}_{13}\ (memory\ x)\ (a\ x) \wedge acc\ x \overset{\mathrm{c}}{=} acc\ x \wedge \\
&\quad i_{a2}\ x \overset{\mathrm{b}_3}{=} o_m\ x \wedge acc\ x = acc\ x \wedge \\
&\quad o_a\ x \overset{\mathrm{a}}{=} \mathrm{Sub}_{16}\ (acc\ x)\ (i_{a2}\ x) \wedge \\
&\quad i_{acc}\ x \overset{\mathrm{b}_1}{=} o_a\ x \wedge \\
&\quad acc\ y \overset{\mathrm{r}_a}{=} i_{acc}\ x \\
&\ )\ [E_{5a},\ E_{19}] \\
(&term\_transition_6)\ [E_{15},\ E_{19}] = \\
&\ (i_a\ x \overset{\mathrm{b}_4}{=} v\ x \wedge memory\ x = memory\ x \wedge \\
&\quad a\ y \overset{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}\ (i_a\ x) \wedge memory\ y = memory\ x \wedge \\
&\ )\ [E_{15},\ E_{6a}] \wedge \\
&\ (o_m\ x \overset{\mathrm{m}}{=} \mathrm{Fetch}_{13}\ (memory\ x)\ (a\ x) \wedge \\
&\quad i_{acc}\ x \overset{\mathrm{b}_1}{=} o_m\ x \wedge \\
&\quad acc\ y \overset{\mathrm{r}_a}{=} i_{acc}\ x \\
&\ )\ [E_{6a},\ E_{19}] \\
(&term\_transition_7)\ [E_{16},\ E_{19}] =
\end{aligned}
$$

$(i_a\ x \overset{\mathrm{b}_4}{=} v\ x \land memory\ x = memory\ x \land acc\ x = acc\ x \land$

$a\ y \overset{\mathrm{r}_m}{=} \mathrm{Cut}_{13}\ (i_a\ x) \land$

$memory\ y = memory\ x \land acc\ y = acc\ x$

$)\ [E_{16},\ E_{7a}] \land$

$(memory\ y \overset{\mathrm{m}}{=} \mathrm{Store}_{13}\ (a\ x)\ (acc\ x)\ (memory\ x)$

$)\ [E_{7a},\ E_{19}]$

$(term\_transition_8)\ [E_{19},\ E] =$

$(i_{a2}\ x \overset{\mathrm{b}_3}{=} \mathrm{Pad}_{16}\ (pc\ x) \land$

$o_a\ x \overset{\mathrm{a}}{=} \mathrm{Inc}_{16}\ (i_{a2}\ x) \land$

$i_{pc}\ x \overset{\mathrm{b}_2}{=} o_a\ x \land$

$pc\ y \overset{\mathrm{r}_p}{=} \mathrm{Cut}_{13}\ (i_{pc}\ x)$

$)\ [E_{19},\ E]$

$(term\_transition_9)\ [E_5,\ E_{10}] =$

$(memory\ x \overset{\mathrm{c}}{=} memory\ x \land i_a\ x \overset{\mathrm{b}_4}{=} \mathrm{Pad}_{16\_13}\ (pc\ x) \land$

$memory\ y = memory\ x \land a\ y \overset{\mathrm{r}_m}{=} \mathrm{Cut}_{16\_13}(i_a\ x)$

$)\ [E_5,\ E_{9a}] \land$

$(o_m\ x \overset{\mathrm{m}}{=} \mathrm{Fetch}_{13}\ (memory\ x)\ (a\ x) \land$

$v\ y \overset{\mathrm{r}_i}{=} o_m\ x$

$)\ [E_{9a},\ E_{10}]$

To get a complete specification of Gordon's computer at this level we need to complement six points:

1. Following the subsection 9.2.3 to eliminate $E\ 0$.

2. Omitting all negative control signals. For example in a model formed as

   $(\alpha \land acc\ y = acc\ x \land \beta)\ [E_i,\ E_o]$

   for the $acc\ y = acc\ x$ the control signal of its corresponding Register is $\neg(wacc\ x)$ but when transforming the model to a corresponding microinstruction we omit the negative signal. This is because that we suppose that in a microinstruction a negative signal is existent if corresponding positive signal is not existent.

3. According to 2 above the control signals of st_3 $[E_9,\ E]$, $(sst_0 \wedge kss_7)$ $[E_{17},\ E]$ and $(sst_0 \wedge kss_7)$ $[E_{18},\ E]$ become empty, therefore they become the weakest specification T. For this we use E replace E_9 and E_{19} to replace E_{17} and E_{18}.

4. We set a signal "idle$_i$" in microinstruction for state signal idle. Therefore to implement *idle_is_true* can be setting the idle$_t$ to be true and by the convenience 2 above we omit to write the negative case *idle_is_false* into the microinstructions. And also we introduce a device Idle

   $\text{Idle}\ (\text{idle}_t,\ \text{idle}) \equiv \forall t.\ \text{idle}\ t = \text{idle}_t\ t.$

5. For hand handing signal *hhs* such as *switches*, *botton* and *knob* we introduce extra axioms

   $(hhs\ y = hhs\ x)\ [E_i,\ E_o]$

6. Finally, to instantiate all variables except the entry predicates.

Then we get

Specification 11 = Control $\wedge$ Datapath

Control $\equiv$

$[E,\ (idle\ x)\ (E_1,\ E_2)]$

$\wedge$

$[E_1,\ (button\ x)\ (E_3,\ E_4)]$

$\wedge$

$[E_3,\ (\text{Val}_2\ (knob\ x) = 0)\ (E_6,$

$(\text{Val}_2\ (knob\ x) = 1)\ (E_7,$

$(\text{Val}_2\ (knob\ x) = 2)\ (E_8,$

$E)))]$

$\wedge$

$(\mathrm{rsw}\ x \wedge \mathrm{wpc}\ x \wedge \mathrm{idle}_t)\ [E_6,\ E]$

$\wedge$

$(\mathrm{rsw}\ x \wedge \mathrm{wacc}\ x \wedge \mathrm{idle}_t)\ [E_7,\ E]$

$\wedge$

$(\mathrm{rpc}\ x \wedge \mathrm{wmar}\ x)\ [E_8,\ E_{2a}] \wedge$

$(\mathrm{wmem}\ x \wedge \mathrm{idle}_t)\ [E_{2a},\ E]$

$\wedge$

$[E_2,\ (\mathit{button}\ x)\ (E_4,\ E_5)]$

$\wedge$

$(\mathrm{idle}_t\ x)\ [E_4,\ E]$

$\wedge$

$(\mathrm{rpc}\ x \wedge \mathrm{wmar}\ x)\ [E_5,\ E_{9a}] \wedge$

$(\mathrm{rmem}\ x \wedge \mathrm{wir}\ x)\ [E_{9a},\ E_{10}]$

$\wedge$

$$
\begin{array}{l}
[E_{10},\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (v\ x)) = 0) \\
\quad (E_4,\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (v\ x)) = 1) \\
\quad\quad (E_{11},\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (v\ x)) = 2) \\
\quad\quad\quad (E_{12},\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (v\ x)) = 3) \\
\quad\quad\quad\quad (E_{13},\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (v\ x)) = 4) \\
\quad\quad\quad\quad\quad (E_{14},\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (v\ x)) = 5) \\
\quad\quad\quad\quad\quad\quad (E_{15},\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (v\ x)) = 6) \\
\quad\quad\quad\quad\quad\quad\quad (E_{16},\ E_{19})))))))]
\end{array}
$$

$\wedge$

$(\mathrm{rir}\ x \wedge \mathrm{wpc}\ x)\ [E_{11},\ E]$

$\wedge$

$[E_{12}, (\mathrm{Val}_{16}\ (\mathit{acc}\ x) = 0)\ (E_{11},\ E_{19})] \wedge$

$\wedge$

$(\text{rir } x \wedge \text{wmar } x)\ [E_{13},\ E_{4a}] \wedge$

$(\text{rmem } x \wedge \text{rm}_3\ x \wedge \text{add } x \wedge \text{ralu } x \wedge \text{wacc } x)\ [E_{4a},\ E_{19}]$

$\wedge$

$(\text{rir } x \wedge \text{wmar } x)\ [E_{14},\ E_{5a}] \wedge$

$(\text{rmem } x \wedge \text{rm}_3\ x \wedge \text{sub } x \wedge \text{ralu } x \wedge \text{wacc } x)\ [E_{5a},\ E_{19}]$

$\wedge$

$(\text{rir } x \wedge \text{wmar } x)\ [E_{15},\ E_{6a}] \wedge$

$(\text{rmem } x \wedge \text{rm}_1\ x \wedge \text{wacc } x)\ [E_{6a},\ E_{19}]$

$\wedge$

$(\text{rir } x \wedge \text{wmar } x)\ [E_{16},\ E_{7a}] \wedge$

$(\text{wmem } x)\ [E_{7a},\ E_{19}]$

$\wedge$

$(\text{rpc } x \wedge \text{inc } x \wedge \text{ralu } x \wedge \text{wpc } x)\ [E_{19},\ E]$

Datapath $\equiv$

Mem (rmem, wmem, a, acc, memory, o_m) $\wedge$

Alu (add, sub, inc, acc, i_{a2}, o_a) $\wedge$

Reg (wacc, i_{acc}, acc) $\wedge$

Reg_p (wpc, i_{pc}, pc) $\wedge$

Reg (wir, o_m, v) $\wedge$

Reg_p (wmar, i_a, a) $\wedge$

(Gate (rsw, switches, o_{11}) $\wedge$ Gate (ralu, o_a, o_{12}) $\wedge$ Gate (rm_1, o_m, o_{13}) $\wedge$
Bus (o_{11}, o_{12}, o_{13}, i_{acc})) $\wedge$

(Gate (rir, v, o_{23}) $\wedge$ Bus (o_{11}, o_{12}, o_{23}, i_{pc})) $\wedge$

(Gate_p (rm_3, o_m, o_{31}) $\wedge$ Gate_p (rpc, pc, o_{32}) $\wedge$ Bus (o_{31}, o_{32}, i_{a2})) $\wedge$

Bus (o_{32}, v, i_a)

11.2.8 Structural Specification

To construct the microprogram of Gordon's computer we introduce below devices:

For **control** we introduce MicroProgram (note: which has included branch structure) and Address.

MicroProgram (address,
idle, button, knob, v, acc,
next-address,
add, sub, inc, rmem, wmem,
rsw, ralu, rm_1, rir, rm_3, rpc,
wacc, wpc, wir, wmar,
idle_t) $\equiv$

$$
\begin{array}{l}
(\forall x.\ \text{address } x = a_0 \Rightarrow \\
\qquad \text{next-address } x = (\text{idle } x \Rightarrow a_1 \downarrow a_2)) \\
\wedge \\
(\forall x.\ \text{address } x = a_1 \Rightarrow \\
\qquad \text{next-address } x = (\text{button } x \Rightarrow a_3 \downarrow a_4)) \\
\wedge \\
(\forall x.\ \text{address } x = a_3 \Rightarrow \\
\qquad \text{next-address } x = (\text{Val}_2\ (\text{knob } x) = 0) \Rightarrow a_6 \downarrow \\
\qquad\qquad (\text{Val}_2\ (\text{knob } x) = 1) \Rightarrow a_7 \downarrow \\
\qquad\qquad (\text{Val}_2\ (\text{knob } x) = 2) \Rightarrow a_8 \downarrow a_0) \\
\wedge \\
(\forall x.\ \text{address } x = a_6 \Rightarrow (\text{rsw } x \wedge \text{wpc } x)\ \wedge \text{idle}_t\ x \wedge \text{next-address } x = a_0) \\
\wedge \\
(\forall x.\ \text{address } x = a_7 \Rightarrow (\text{rsw } x \wedge \text{wacc } x)\ \wedge \text{idle}_t\ x \wedge \text{next-address } x = a_0) \\
\wedge \\
(\forall x.\ \text{address } x = a_8 \Rightarrow (\text{rpc } x \wedge \text{wmar } x) \wedge \text{next-address } x = a_{2a}) \\
\wedge \\
(\forall x.\ \text{address } x = a_{2a} \Rightarrow \text{wmem } x\ \wedge \text{idle}_t\ x \wedge \text{next-address } x = a_0)
\end{array}
$$

$\wedge$

$(\forall x.\ \text{address}\ x = a_4 \Rightarrow \text{idle}_t\ x \wedge \text{next-address}\ x = a_0)$

$\wedge$

$(\forall x.\ \text{address}\ x = a_2 \Rightarrow$

$\quad \text{next-address}\ x = (\text{button}\ x \Rightarrow a_4 \downarrow a_5))$

$\wedge$

$(\forall x.\ \text{address}\ x = a_5 \Rightarrow (\text{rpc}\ x\ \wedge \text{wmar}\ x) \wedge \text{next-address}\ x = a_{9a})$

$\wedge$

$(\forall x.\ \text{address}\ x = a_{9a} \Rightarrow (\text{rmem}\ x\ \wedge \text{wir}\ x) \wedge \text{next-address}\ x = a_{10})$

$\wedge$

$(\forall x.\ \text{address}\ x = a_{10} \Rightarrow$

$\quad \text{next-address}\ x =$

$\qquad (\ \text{Val}_3\ (\text{Opcode}\ (\text{v}\ x)) = 0 \Rightarrow a_4 \downarrow$

$\qquad\ \ \text{Val}_3\ (\text{Opcode}\ (\text{v}\ x)) = 1 \Rightarrow a_{11} \downarrow$

$\qquad\ \ \text{Val}_3\ (\text{Opcode}\ (\text{v}\ x)) = 2 \Rightarrow a_{12} \downarrow$

$\qquad\ \ \text{Val}_3\ (\text{Opcode}\ (\text{v}\ x)) = 3 \Rightarrow a_{13} \downarrow$

$\qquad\ \ \text{Val}_3\ (\text{Opcode}\ (\text{v}\ x)) = 4 \Rightarrow a_{14} \downarrow$

$\qquad\ \ \text{Val}_3\ (\text{Opcode}\ (\text{v}\ x)) = 5 \Rightarrow a_{15} \downarrow$

$\qquad\ \ \text{Val}_3\ (\text{Opcode}\ (\text{v}\ x)) = 6 \Rightarrow a_{16} \downarrow address_{19}))$

$\wedge$

$(\forall x.\ \text{address}\ x = a_{11} \Rightarrow (\text{rir}\ x\ \wedge \text{wpc}\ x) \wedge \text{next-address}\ x = a_0)$

$\wedge$

$(\forall x.\ \text{address}\ x = a_{12} \Rightarrow$

$\quad \text{next-address}\ x = (\text{acc}\ x \Rightarrow a_{11} \downarrow a_{19}))$

$\wedge$

$(\forall x.\ \text{address}\ x = a_{13} \Rightarrow (\text{rir}\ x\ \wedge \text{wmar}\ x) \wedge \text{next-address}\ x = a_{4a})$

$\wedge$

$(\forall x.\ \text{address}\ x = a_{4a} \Rightarrow$

$\quad (\text{rmem}\ x\ \wedge \text{rm}_3\ x \wedge \text{add}\ x\ \wedge \text{ralu}\ x \wedge \text{wacc}\ x) \wedge$

next-address $x = a_{19}$)

$\wedge$

($\forall x$. address $x = a_{14} \Rightarrow$ (rir x $\wedge$ wmar x) $\wedge$ next-address $x = a_{5a}$)

$\wedge$

($\forall x$. address $x = a_{5a} \Rightarrow$

(rmem x $\wedge$ rm_3 x $\wedge$ sub x $\wedge$ ralu x $\wedge$ wacc x) $\wedge$

next-address $x = a_{19}$)

$\wedge$

($\forall x$. address $x = a_{15} \Rightarrow$ (rir x $\wedge$ wmar x) $\wedge$ next-address $x = a_{6a}$)

$\wedge$

($\forall x$. address $x = a_{6a} \Rightarrow$

(rmem x $\wedge$ rm_1 x $\wedge$ wacc x) $\wedge$ next-address $x = a_{19}$)

$\wedge$

($\forall x$. address $x = a_{16} \Rightarrow$ (rir x $\wedge$ wmar x) $\wedge$ next-address $x = a_{7a}$)

$\wedge$

($\forall x$. address $x = a_{7a} \Rightarrow$ wmem x $\wedge$ next-address $x = a_{19}$)

$\wedge$

($\forall x$. address $x = a_{19} \Rightarrow$

(rpc x $\wedge$ inc x $\wedge$ ralu x $\wedge$ wpc x) $\wedge$ next-address $x = a_0$)

Note: we use the a_i $(0 \leq i \leq 19)$ as variables above which leaves them for further design selection and keeps our microprogram more general.

Address (next-address, address) $\equiv \forall t.$ address $(t+1) =$ next-address t.

For **datapath** in addition the one in the preceding subsection we introduce a device Idle as below:

Idle (idle_i, idle) $\equiv \forall t.$ idle $(t+1) = \mathrm{idle}_i$ t.

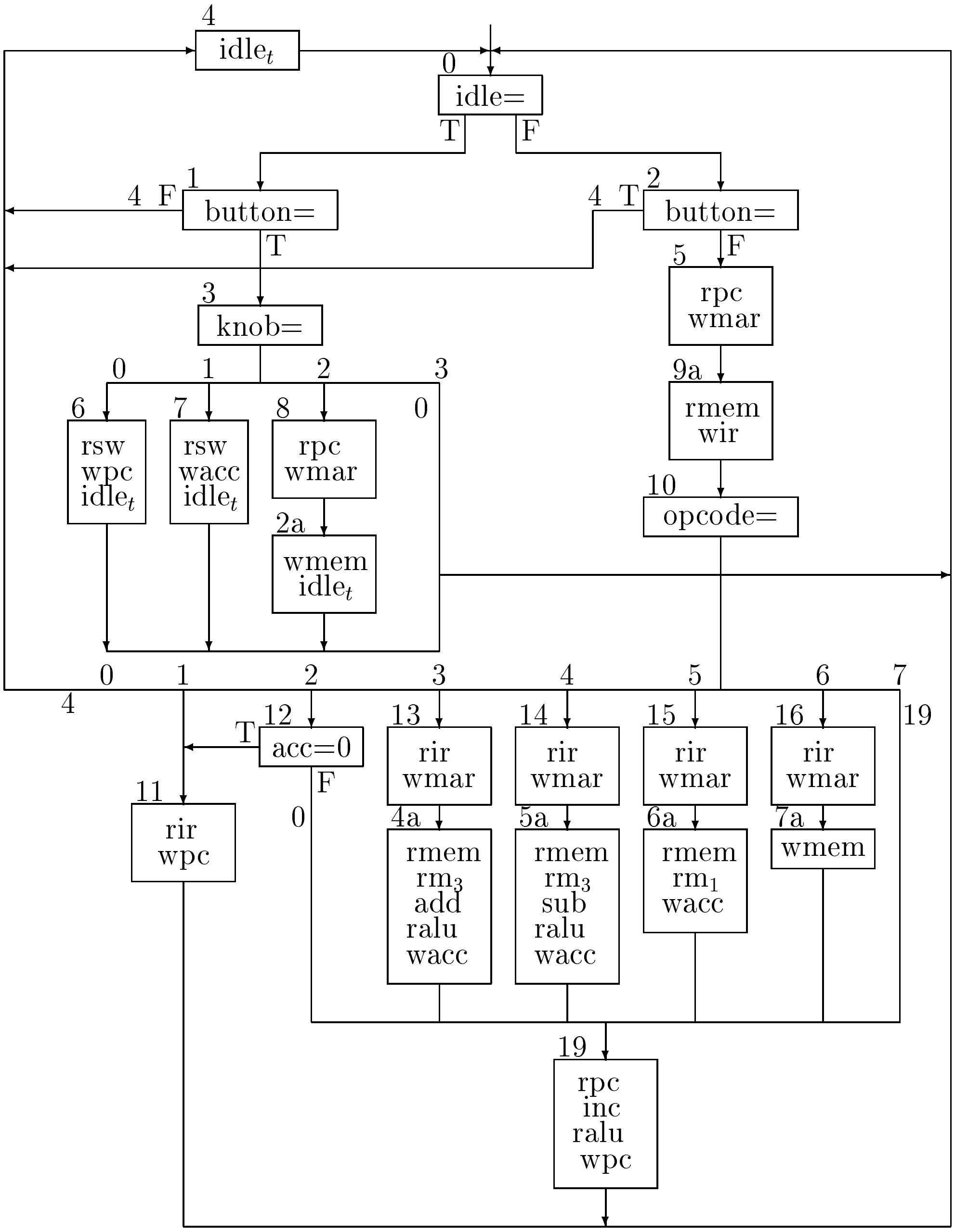


Figure 2: Flowchart of the microprogram of the first implementation

Note: the number on the top left of each frame-box is the subscript of the microinstruction address

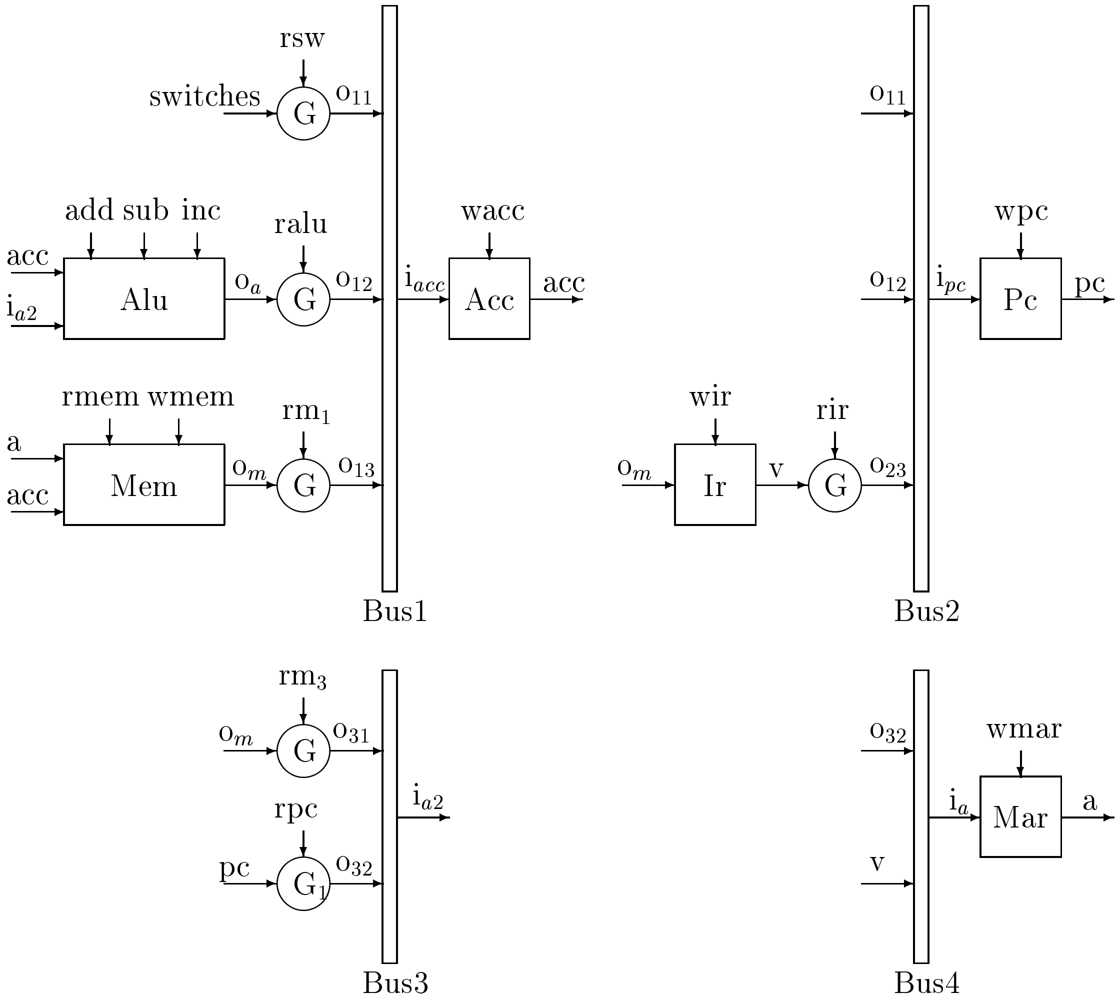


Figure 1: Datapath of the first implementation

11.3 Deriving other Implementations

In the whole derivation process each nondeterministic construction step offers different selections which can lead to different constructions and finally result in different implementations. In our derivation the main selections are done in three stages: the selecting of the global structure which will lead to different global structures of the whole computer, the allocation which will lead to different datapaths and microprograms and the merging of parallel chains which will lead to different linear microprograms. In this section we give two other implementations which are derived by the different selections for buses. We call them the second and third implementation to distinguish them from the preceding one which we refer to as the first implementation.

11.3.1 Two-Bus Implementation

We can merge several minimal buses to get a compound bus. Instead of the three minimal buses Bus1, Bus2, and Bus4 with corresponding outputs i_{acc}, i_{pc} and i_a we may use a compound bus Bus5. We use "i" as the output of the compound bus, i.e. we use "i" to replace the original three outputs i_{acc}, i_{pc} and i_a. This results in the following new datapath, which is graphically depicted in Figure 3.

Datapath =

Mem (rmem, wmem, a, acc, memory, o_m) $\wedge$

Alu (add, sub, inc, acc, i_{a2}, o_a) $\wedge$

Reg_c (wpc, i, pc) $\wedge$

Reg (wacc, i, acc) $\wedge$

Reg (wmar, i, a) $\wedge$

Reg_c (wir, o_m, v) $\wedge$

(Gate (rsw, switches, o_{11}) $\wedge$ Gate (ralu, o_a, o_{12}) $\wedge$ Gate (rm_1, o_m, o_{13}) $\wedge$

Gate (rir, v, o_{23}) $\wedge$ Bus (o_{11}, o_{12}, o_{13}, o_{23}, o_{32}, i)) $\wedge$

(Gate_p (rm_3, o_m, o_{31}) $\wedge$ Gate_p (rpc, pc, o_{32}) $\wedge$ Bus (o_{31}, o_{32}, i_{a2}))

The second implementation has the same microprogram as the first one. The second implementation can keep each microinstruction and for each bus there is only a connection executed. This is same as in the first implementation.

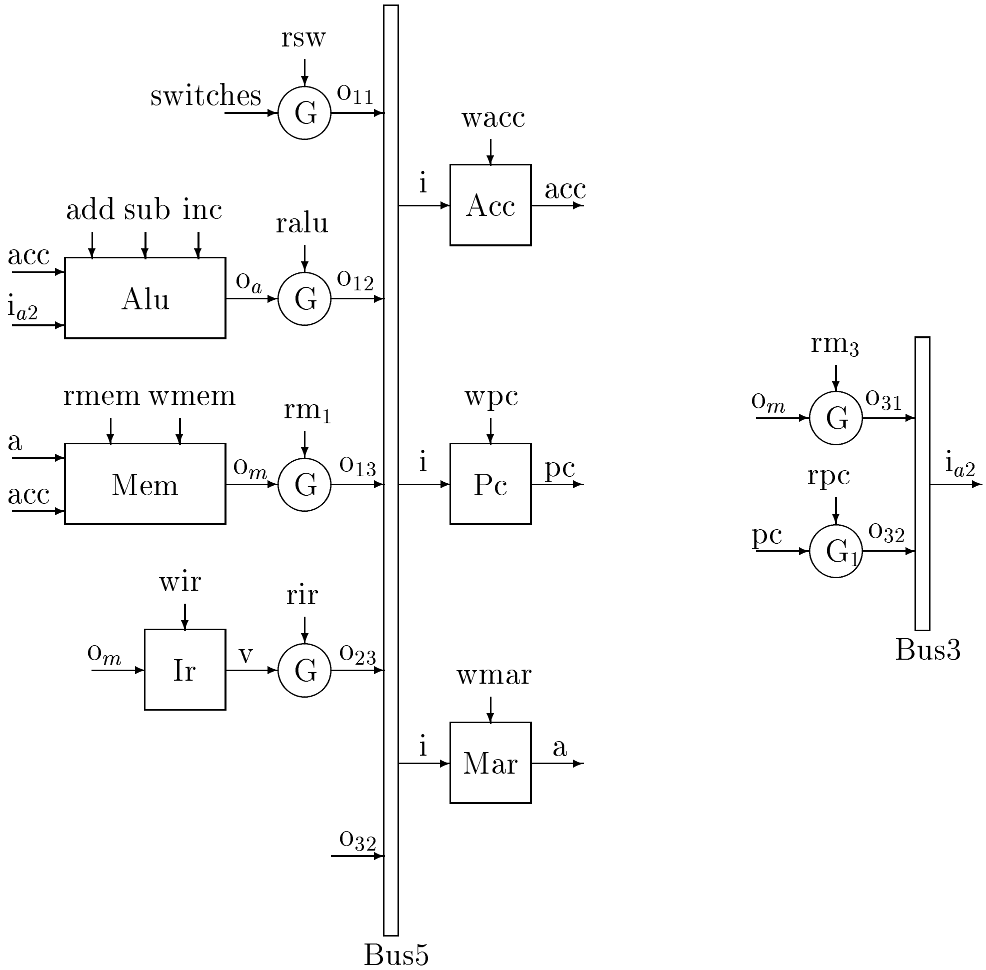


Figure 3: Datapath of the second implementation

11.3.2 One-Bus Implementation

To get the third implementation we further merge the Bus5 and Bus3 of the second implementation into a single Bus6. To this end we need to introduce a Reg for the second input i_{a2} of Alu and merge the two outputs of Bus5 and Bus3 to get one output for Bus6. These two changes need to be taken as a compound bus may have only a pair of input and output signals connected at any time point. The change will further lead to a change in the microprogram which is based on the new datapath. For the third implementation we have the datapath below which is also shown in Figure 4.

Datapath $\equiv$

Mem (rmem, wmem, a, acc, memory, o_m) $\wedge$

Alu (add, sub, inc, acc, i, o_a) $\wedge$

Reg_c (wpc, i, pc) $\wedge$

Reg (wacc, i, acc) $\wedge$

Reg (wir, o_m, v) $\wedge$

Reg_c (wmar, i, a) $\wedge$

Reg (warg, i, i_{a2}) $\wedge$

(Gate (rsw, switches, o_{11}) $\wedge$ Gate (ralu, o_a, o_{12}) $\wedge$ Gate (rm, o_m, o_{13}) $\wedge$

Gate (rir, v, o_{23}) $\wedge$ Gate_p (rpc, pc, o_{32}) $\wedge$

Bus (o_{11}, o_{12}, o_{13}, o_{23}, o_{32}, i))

The construction for the microprogram of the third implementation is the same as for the first one except for the microinstructions 4a, 5a, and 19 of the Figure 2. As we introduced a new Reg (Named Arg in Figure 4) we need to put in an extra microinstruction step since to pass a data. For this we replace the microinstruction 4a (5a, 19) of the second implementation by two microinstructions 4a and 4b (5a and 5b, 19 and 19a). Finally we get a microprogram as shown in Figure 5.

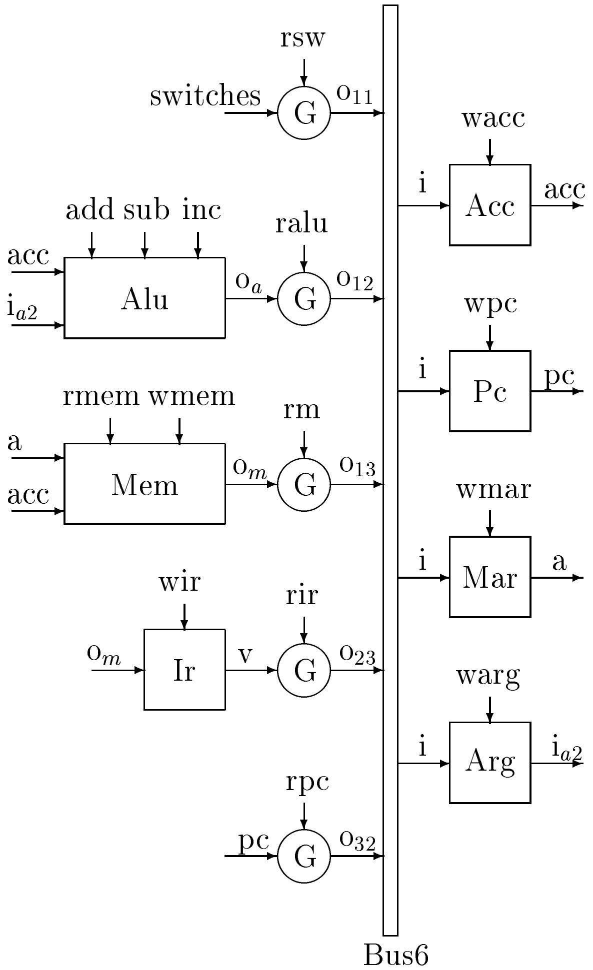


Figure 4: Datapath of the third implementation

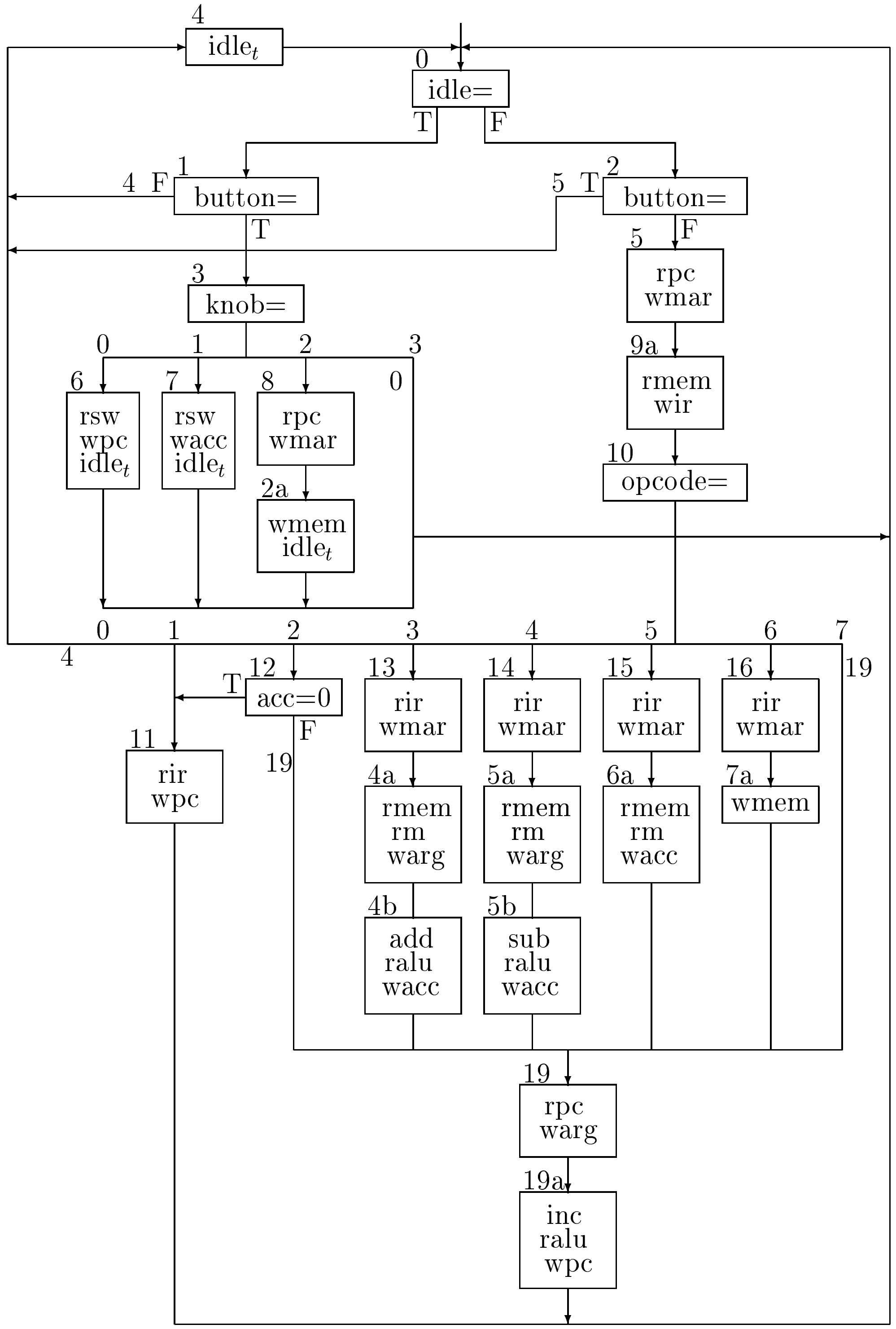


Figure 5: Flowchart of the microprogram of the third implementation

11.4 Discussion

Through the derivation of Gordon's computer we show a concrete construction process and its result. The process and result have, directly or indirectly, told us nearly the whole story about what our formal derivation means and how it differs from other methods. In this section let us summarize some important aspects.

We may first compare our implementation with the one of [50]. In the table below "Impl_i" $(1 \leq i \leq 3)$ refer to our three derived implementations where "Impl_G" means the implementation of Gorden's computer. "Microprogram" means the number of microinstructions, "Microcode" the number of microcodes in a microinstruction, and "Max length" the number of microinstructions of the maximal length execution of a machine instruction.

Implementation	Impl_1	Impl_2	Impl_3	Impl_G
Bus	4	2	1	1
Gate	6	6	5	5
Register	4	4	5	6
Microprogram	23	23	26	26
Microcode	14	14	15	16
Max length	8	8	10	10

Of course, the truly interesting story is the method and treatment process behind these results. In Part 2 we use abstract structure and its construction in a process of formally deriving a class of computers. Therefore, this is is a formal design driven by formal description and construction. A distinct point is that the basic string of derivation is *proof-driven* as opposed to being *artificially given*.

For example, in the time refinement, the early stage of derivation, we identify the global loop structure of the computer. We introduce a global entry predicate E which has four functions:

1. the invariant of the global loop,
2. the clock at machine instruction level,
3. the synchronization for different time layers,
4. the global entry address of the microprogram.

Therefore, E can replace the "ready" of Gordon's computer for time synchronization. For our implementations that "ready" is redundant.

In the datapath aspect some connections through bus and some registers of Gordon's computer are not used by our implementations. For example, the output of register Acc can directly be connected to the corresponding input ports of Memory and Alu, therefore it does not need to pass via the bus. This is similar for the output of register Mar which may be directly connected to the Memory and the output of Memory can be directly connected to the instruction register Ir. Also for our first and second implementations the register Arg and Buf are unnecessary. An essential difference is that with our methods all these above implementation structures are very naturally obtained in a formal derivation process.

The microprogram is the center of a computer and in fact our entire derivation process can be seen as a process of constructing a microprogram. The process includes a top-down hierarchy:

1. Global loop
2. Abstract microprogram structure
3. Linear microprogram
4. Microinstruction
5. Microcode

This hierarchy is unfolded in our formal derivation process and is based on our central concept of the derivation model.

Derivation is structure oriented. Derivation always deals with pure structure and leaves many elements uninterpreted. It is unnecessary to specify details, for example, for Val_2, Opcode, etc except for introducing extra axioms to describe some basic relations among them if this is necessary for the derivation. To deal with pure structure and structure construction is the fundamental point for formal derivation. All irrelevant details are ignored focusing on just that structure which is related to a particular construction step.

Chapter 12

Proof-Based Architecture of Computer

By formal derivation for the formal design of computer we hold two things: a formal construction process and a formal specification sequence. When we interpret them to the design in real world we gain a new view — proof-based architecture of computer. In this chapter let us summarize what we have done by the two aspects: formal construction as design and formal specification as architecture.

We use specification (a) as a start point to describe a hardware device

$\forall t : \text{time}.\ \alpha$ . . . . (a)

because it is a basic requirement that a hardware device should be of work order for any time point. The quantification $\forall t$ is written at the most outside of the (a) because time is a general existential primitive factor for hardware device.

We transform a specification formed as

$\forall t.\ S\ (t,\ t+1)$ . . . . (b)

to another specification formed as

$\forall x.\ \exists y.\ S\ (x,\ y)$ . . . . (c)

Because we need to introduce finer time level to support more detailed implementation. This corresponds to the time refinement in the design of computer from a time unit $< t,\ t+1 >$ at the machine instruction level to a time interval $< x,\ y >$ at the microinstruction level.

We introduce entry predicate E as a higher order variable to express (c) to (d):

$$\forall x.\ \exists y.\ E\ x \Rightarrow \beta \wedge E\ y \quad \text{(d)}$$

By derivation model (d) can be rewritten as

$$(\beta)\ [E,\ E] \quad \text{(e)}$$

Through (e) we can see that at the highest level a computer has a loop structure through a global entry predicate E which finally is interpreted as the global entry address of the whole microprogram of computer.

β of (e) has state transition to describe the semantics of machine instruction of computer – using recursion to express infinite iteration of computer running. The state transition can have a quite complex structure which reflects the complexity of a computer.

To deal with the state transition following the strategy of "divide and conquer", we decompose condition structure, let_in structure and state transition to a specification whose components are term transition formed as

$$(s\ t = tm)\ [E_i,\ E_o] \quad \text{(f)}$$

Here, tm is a term with time point x.

At the level the specification of a computer is reduced to term transitions and their connection relation through entry predicate.

Each term transition is further decomposed to a linear chain with a continue entry predicate sequence, for example, $E_1,\ E_2,\ \cdots,\ E_n$. The link of the chain is formed as

$$(\gamma)\ [E_i,\ E_o] \quad \text{(g)}$$

Here, γ has two kind forms: of function unit or of connection.

This leads to introducing datapath by assigning γ to devices and introducing microprogram by instantiating the entry predicate variables E_i and E_o to instance ones about the entry address of microprogram.

Therefore the basic process of formal construction of computer is to introducing the finer derivation models of three levels:

$(\alpha)\ [E_i,\ E_o]$ .... (m1)

Here, α is about state transition.

$(\beta)\ [E_i,\ E_o]$ .... (m2)

Here, β is about term transition.

$(\gamma)\ [E_i,\ E_o]$ .... (m3)

Here, γ is about function unit and connection.

The process is mainly through microprogramming of the five levels:

1. Global loop
2. Abstract microprogram structure
3. Linear microprogram
4. Microinstruction
5. Microcode

Derivation model is the foundation. In fact it has more general significance. In the next chapter we will discuss that the model can be taken as a foundation to clean up our understanding and moreover present a new understanding for hardware construction for its basic structure about logic, space and time.

Chapter 13

Logic, Space and Time of Hardware Construction

The work of hardware verification is an one-dimensional *signal* world: free-variable as signal argument for predicate representing a hardware component and existential quantified port as connection representing the component composition. The other relationships between circuits are expressed through the dimension in space.

The world of construction is a three-dimensional world in logic (Boolean value), space (signal/port) and time (point). Such a world is for describing a dynamic construction *process* and the structure of a hardware, the relation of its components, is derived in the process.

13.1 Logic, Space and Time Structure of Model

If we describe circuit by static known specification then we only need an one-dimensional signal world where an inverter circuit and the structure of a delay-inverter are described as below respectively:

$\mathrm{Inv}\ (i,\ o) \equiv \forall t.\ o\ (t+1) = \neg(i\ t),$

$\exists z.\ \mathrm{Del}\,(i,z) \wedge \mathrm{Inv}\ (z,o)$

Such a one-dimensional model originates from the obvious static-topological understanding of physical hardware focusing on spatial structural abstraction [57] (p. 269).

However, if we describe hardware circuit for a dynamic construction to derive unknown specification then we need a more complex three-dimensional world about Boolean value, signal (port) and time, which can be more generally called as logic, space and time. In such a world an Inverter is written as

$$\text{Inv}\ (i,\ o)\ [E_i,\ E_o] \equiv \forall x.\ \exists y.\ E_i\ x \Rightarrow o\ y = \neg(i\ x) \wedge E_o\ y$$

In addition to i and o two higher order predicate free-variables E_i and E_o are introduced and the fixed time interval 1 is made flexible by $\exists$ variable y. This is a timed logic relationship: at any time point x under condition $E_i\,x$ Inverter is activated and does its work in the time interval $\langle x, y\rangle$, then at time point y to hold up E_o, y to activate other circuit.

This is a higher order predicate model about a circuit related to its environment — through E_i and E_o in logic, i and o in space and x and y in time. They are all *variables* for constructibility. These variables can be instantiated to obtain different implementations: synchronous, asynchronous etc. and typically the microprogrammed ones.

We present a construction model as the box below:

Construction Model:			Syntactic Class:
cm	$::=$	$mp\,[E,\ lps]$	cm : construction model
lps	$::=$	E	lps : leaving predicate structure
		$\mid P\ (lps,\ lps)$	P : Boolean term
			mp : model predicate (body)
			E : (entry or leaving) predicate

The semantics of the model is defined by the induction translation:

$$(mp\,[E, lps])^* = \forall x.\exists y.\, E\,x \Rightarrow mp \wedge lps^*$$

$$E^* = E\,y$$

$$(P\,(lps_1, lps_2))^* = P \Rightarrow lps_1^* \downarrow lps_2^*$$

To give an example we may write

$$(pc\; y = pc\; x)\,[E_2, b_1\, x\,(E_3, b_2\, x\,(E_4, E_5))]$$

to stand for

$$\forall x.\exists y.\, E_2\, x \Rightarrow pc\; y = pc\; x \wedge (b_1\, x \Rightarrow E_3\, y \downarrow (b_2\, x \Rightarrow E_4\, y \downarrow E_5\, y)).$$

Where $a \Rightarrow b \downarrow c$ abbreviates the term "if a then b else c" [33].

13.2 Logic, Space and Time Structure of Construction

Construction is to divide specification to subspecifications by introducing new entry predicates in logic, new signals (port) in space and new points in time. For example, decomposing a specification delay-inverter (1) to the subspecifications Delay (2) and Inverter (3):

$$(o\, y = \neg(i\, x))\ [E_i,\ E_o] \tag{1}$$

$$(z\, y = i\, x)\ [E_i,\ E_z] \tag{2}$$

$$(o\, y = \neg(z\, x))\ [E_z,\ E_o] \tag{3}$$

By introducing predicate variable E_z, signal variable z and a time point as the argument of E_z. Generally construction is a map:

$$\begin{aligned}(R\,(A_1, \cdots, A_n))\ [E_i,\ E_o] \rightarrow\ & R'\ [E_{r\_i},\ lps_{r\_o}] \wedge \\ & \textstyle\bigwedge_{k=1}^{n} A'_k\ [E_{ak\_i},\ E_{ak\_o}] \wedge \\ & \textstyle\bigwedge_{l=1}^{m} C_l\ [E_{cl\_i},\ E_{cl\_o}]\end{aligned}$$

which maps a specification to the implementation. The specification is about n parts A_k $(1 \leq k \leq n)$ with their relation R. The implementation is constituted by $n + m + 1$ parts: R' constructed from R, A'_k from A_k $(1 \leq k \leq n)$ and C_l $(1 \leq l \leq m)$ is the connections introduced. In the implementation E_i and

E_o are kept and each introduced new entry predicate must appears two times as leaving one and entry one for two different parts. Each introduced connection is about two new signals as input and output signals from two different parts.

Definition

[Input and output signal protection]:

For a syntactic construct S the input protection is

$S^i \mathrel{\hat=} (r_1\ y = r_1\ x) \wedge \cdots \wedge (r_n\ y = r_n\ x)$,

where r_i $(1 \le i \le n)$ are all the input signals in S. Similarly,

$S^o \mathrel{\hat=} (s_1\ y = s_1\ x) \wedge \cdots \wedge (s_m\ y = s_m\ x)$,

where s_i $(1 \le i \le m)$ are all the output signals is the output protection for S. □

Example 13.2.1. [decomposition]:

A conditional structure is formed as

$(P \Rightarrow R \downarrow S)\ [E_i,\ E_o]$

By decomposition we get

$$\begin{array}{c}
(b\ y = P \wedge R^i \wedge S^i)\ [E_i,\ E_a]\ \wedge \\
(c\ y = b\ x \wedge R^i \wedge S^i)\ [E_a,\ E_{a'}]\ \wedge \\
(R^i \wedge S^i)\ [E_{a'},\ c\ x\ (E_b,\ E_c)]\ \wedge \\
R\ [E_b,\ E_o]\ \wedge \\
S\ [E_c,\ E_o]
\end{array}$$

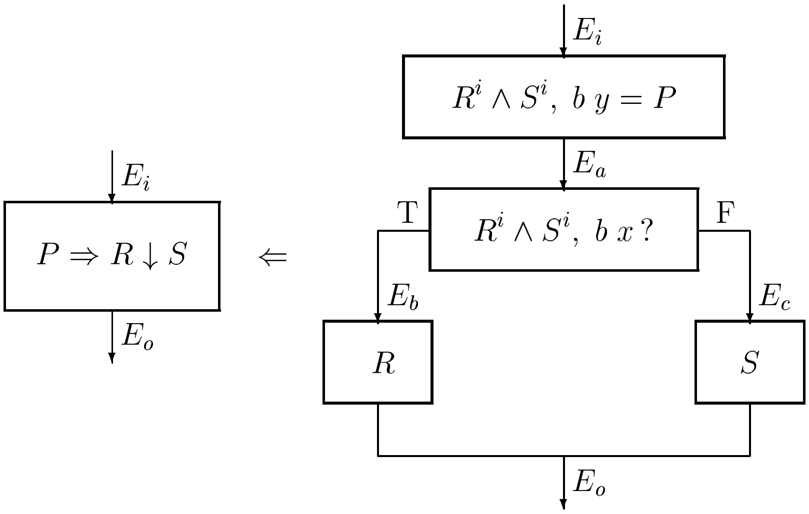
We can reduce

$(c\ y = b\ x \wedge R^i \wedge S^i)\ [E_a,\ E_{a'}]$

to the Boolean constant T by use b, x and E_a to instantiate c, y and E'_a respectively. Therefore we get

$$\begin{array}{c}
(b\ y = P \wedge R^i \wedge S^i)\ [E_i,\ E_a]\ \wedge \\
(R^i \wedge S^i)\ [E_a,\ b\ x\ (E_b,\ E_c)]\ \wedge
\end{array}$$

$$R\,[E_b,\ E_o]\ \wedge$$
$$S\,[E_c,\ E_o]$$



The construction introduces predicate variables E_a, E_b and E_c and corresponding time variables and port variable b.

13.2.1 Logic Structure of Construction

To keep consistency when introducing an entry predicate variable to divide a specification into two subspecifications the variable always to be made to appears in one subspecification as leaving predicate and in the other subspecification as entry predicate. This lead to the logic structure of construction as followings.

1. Composition: Through E_m to compose $\alpha\,[E_i,\ E_m]$ and $\beta\,[E_m,\ E_o]$

2. Condition: $\alpha\,[E_a,\ \beta\,(E_b,\ E_c)]$

Note: by the construction model we can get more general condition combinations.

3. Loop: Through E to get a loop: $\alpha\,[E,\ E_a] \wedge \cdots \wedge \gamma\,[E_z,\ E]$

13.2.2 Space Structure of Construction

To suit further constructing the three possible cases: port-connection, bus and register for a new possible connection of two components, initially, we always introduce a pair of ports: one, say r, as the output of a component and the other one, say s, as the input of the other component. Therefore, in the case we always introduce a new *connection* formed as

$$(s\, y = r\, r)\ [E_i,\ E_o]$$

The connection can further lead to a port-connection, a bus or a register.

1. Port-Connection: s only appears once. In the case to instantiate $s\,(r)$, y and $E_i\,(E_o)$ by $r\,(s)$, x and $E_o\,(E_i)$ to get the weakest specification — Boolean constant T.

2. Bus: s is the same output of two or more connections.

$$(s\, y = r_k)\ [E_{ik},\ E_{ok}] \quad (k \geq 2)$$

In the case a bus is introduced for these connections.

3. Register: if a time constraint is added:

$$(s\, y = r\, r \wedge y > x)\ [E_i,\ E_o]$$

Therefore, through a special model *connection* the space structure of a circuit can be *systematically* constructed.

13.2.3 Time Structure of Construction

Time variables are always introduced together with entry predicates and we use entry predicate to measure time.

Definition 13.2.3.1. [time point]:

Time point t is one making an entry predicate E_x at the t true. □

Definition 13.2.3.2. [time clock & time layer]:

For a circuit its clock is a disjunction formed as

$$\vee_{k=1}^{n} E_k$$

Here each E_k is an entry predicate of the circuit. When one and only one predicate in the clock gets true the clock gets a time point. We call the set of all entry predicates of a circuit as its time layer. Therefore time clock and layer is relative to the specification level. □

Definition 13.2.3.3. [time construction]:

The construction from a high specification level whose time layer is L_i to the adjacent low specification level whose time layer is L_{i+1} is through introducing a set Δ of new middle entry predicates at the low level. Therefore the time construction is

$$L_i \cup \Delta = L_{i+1}$$

As Δ for each level is always non-empty the whole time construction process of a circuit from top specification 1 to bottom implementation n is

$$L_1 \subset L_2 \subset \cdots \subset L_n$$ □

Definition 13.2.3.4. [time unit]:

For a circuit when each construction model formed as

$$S\,(x,\ y)\ [E_i,\ E_o]$$

is instantiated by

$$\forall x.\ E_i\, x \Rightarrow S\,(x, x+\delta) \wedge E_o\,(x+\delta)$$

The δ is called as a time unit of the circuit. □

So, we can have a sequence of clocks for a circuit:

1. Non-Clock: Do not introduce clock.

2. Variable-Clock: Introducing clock and time unit.

3. Constant-Clock: Introducing clock and time unit and moreover introducing a constraint that the δ of every model is the same.

13.3 Case Study

The construction of a computer can form mainly the three levels:

1. State Transition: This is the behavior specification at machine instruction level and each state transition describes the semantics of a instruction.

$S\,[E,\,E]$

2. Term Transition: This is the mixed behavior-structural specification at term transition, which is a substate transition and similar other ones and corresponds to a linear microprogram, level.

$\wedge_{k=1}^{n} T_k\,[E_i,\,E_o]\ (k \geq 1)$

3. Register Transfer: This is structural specification considered as implementation. Each R_l of the model in the specification is either a function which is considered implementable at the register transfer level or a connection for port-connection, bus or register.

$\wedge_{l=1}^{m} R_l\,[E_a,\ E_b]\ (l \geq 1)$

The computer has its logic, space and time structure for each level. Mainly, this is a hierarchy: global loop, abstract microprogram, linear microprogram, microinstruction and microcode.

13.4 Conclusion

For a circuit its three aspects of logic, space and time structures can transform each other by different selections to instantiate variables.

The logic, space and time structures of a circuit are relative to the specification level the circuit currently is at. Construction as a process can unfold a circuit through a sequence of levels. Therefore, we must distinguish these structures for different levels. It is an interesting new understanding to realize, for example, the same entry predicate, the same bus, the same register or the same time point can express quite different things as it can be at different specification levels. For example, an entry predicate is taken as the entry address of a large microprogram and meanwhile as the entry address of only a microinstruction. A bus or a register is at a high level and the other is at the low level. This time point is shared by three levels and that time point is shared by five levels.

For a circuit through its construction process it forms a logic, space and time world of its own and by itself and for the same circuit the different construction processes can lead to quite different worlds. The world can be encapsulated from the environment but only communicate to through the entry predicate, port and time variables of the top specification of the circuit. Therefore a hierarchy and multi-colored structure of circuit in logic, space and time three aspects can be naturally constructed and unfolded by our construction model, construction scheme and methodology.

Chapter 14

Related Work

In this section we describe the work related specially to our main area of interest: computer verification and formal synthesis.

14.1 Computer Verification

Gordon's verification of a simple computer, called Gordon's computer by others, using the LCF_LSM system [31] was an early example of how formal proof and mechanical proof-generation could be used to reason about the design of a microprocessor. This is an elegant example which has a suitable size as a complex case study and holds nearly all main aspects a realistic computer should have. Moreover, its verification process has captured essential aspects of computer verification from an academic research viewpoint. Joyce reverified Gordon's computer using HOL [50]. Gordon's computer [31], [50] is a main source for this thesis and is generalized to a class of computers. Gordon's computer was taken as a case study of formal derivation by the author [73].

Two notable case studies widely seen as the current state-of-the-art in microprocessor verification are Hunt's verification of the FM8501 [44] and Cohn's verification of the VIPER [14], [15]. The FM8501 is a 16-bit microprocessor similar in complexity to a PDP-11. The instruction set of this microprocessor is rich enough to support realistic application. The specification and implementation of this computer are described in pure Lisp and the verification is based on the Boyer-Moore theorem-prover. The work of Hunt is followed by further studies [18] in the FM series and among them an interesting work is [6] on the derivation of a Verified Microprocessor, the DDD-FM9001. The DDD-FM9001 is a 32-bit general purpose microprocessor formally derived directly from Hunt's mechanically verified Nqthm FM9001 microprocessor specification. This example witnesses an inherent trend from verification to derivation, although the derivation of [6] is based on algebra transformation.

VIPER is a commercially available microprocessor which was designed by the British Ministry of Defence for use in life-critical applications [19], [20]. Cohn has completed two verification levels: The first level is from the major state machine to the top-level specification [14] and the second level is from the block level description to the major state machine [15]. As pointed out in [53] a particularly interesting outcome of the VIPER project is that it revealed weaknesses in the links between designer, verifier and manufacturer. The reason was seen in the fact that a formal description of the block level must be derived from a mixture of engineering documents, partly pictorial and partly textual, supplied by the VIPER designers. In this context Cohn pointed out the scope and limitations of using formal proof to verify microprocessor systems [16].

Still more than the above two notable case studies the works of J.J. Joyce [53] and P. J. Windley [78] have relation to our research concept. Joyce verified the microprocessor TAMARACK-3 and moreover a microprocessor-based system including a simple compiler. An important concept in his work is the generic specification that is parameterized with uninterpreted data types and uninterpreted primitives. Using this concept of generic specification the verification of TAMARACK-3 can encompass the wider domain of microprocessors for which the uninterpreted data types and primitives are abstract enough to cover each one among them [51] [52] [53].

J.J. Joyce's concept of generic specification still does not directly describe the specification for a class of computers. His aim is through a concrete microprocessor to introduce somewhat general specification structure and his method is in a concrete microprocessor, TAMARACK-3, to introduce the more generic and abstract notations which are called the uninterpreted data types and primitives. Our aim is to derive a class of computer and therefore we must *directly* describe what the specification of a class of computers is. For this our top specification scheme and all further derived specification schemes are, fundamentally, to describe abstract computers: not only their primitive components but also their high level structure. Therefore what we deal with is a generic structure called specification scheme and the generic structure can be instantiated to get any concrete computer in a set of computers. The set is described by the generic structure. We also call the set as a class of computers as the title of our thesis.

The work of Windley is even more abstract than Joyce's. It concentrates on the abstract structure of general verification of microprocessors [74] [75] [77] [78]. Windley presents the concept of generic interpreters including state, state transition function, selection function and interpreter predicate, and, based on these definitions, the abstract representation and the theory obligations. The generic interpreters are a framework covering the common and basic process of the verification of microprocessors.

Windley's work reveals a generic structure and process of verification of microprocessors. This is a very interesting result. However, two aspects should be pointed out. First, his abstract interpreter is a generic and still somewhat coarser framework. Second, the framework is suitable for verification only. Main limitation is that each level of interpreters of microprocessors are *given by the verifier* and Windley's theory of generic interpreters [78] only describes relations between these given interpreters. For our derivation of a class of computers instead of the given specification our framework must be able to construct specification in each specification scheme. To construct a class of computers what is given is only the top specification scheme at machine instruction level and all lower level specification schemes down to the bottom specification scheme at register transfer level should be constructed step by step from the top specification scheme. Therefore our top specification scheme must be generic and meanwhile accurate enough to make these constructions available. So, for our derivation of a class of computers, we are at a more complex and difficult position that we must present a generic framework not only for the top specification scheme but also for whole construction and verification process from the top scheme down to the bottom scheme.

14.2 Formal Synthesis

Fourman, Hanna, Milne and others pointed out that there are many advantages in combining the processes of design and formal verification [26], [36], [61]. In [26] Fourman presented a formal approach for system level design to deal with complexity and abstraction. It is mainly using formal methodology to express goal-directed design including goal-directed problem solving, proof-search as problem solving, modeling the design state and formalizing design refinement. The formal approach is based on the Lambda logic system [24]. Lambda consists of a prover kernel with rewriting, unification, subgoal and many other functions available in state-of-the-art systems, structured libraries of rules and common logic definitions, and a user-interface which includes a browser and the graphics tool DIALOG. The system is implemented and runs in Poly/ML. The LAMBDA logic is axiomatized as a classical higher-order predicate logic with polymorphic types. The syntax of LAMBDA terms is kept close to ML, enhanced with the common operators of higher-order logic.

[26] describes a general design methodology and a key point is that it can be formally described and executed in LAMBDA system [24]. Some primitive programming tests of our thesis have been done in LAMBDA system. The HOL system [33] and LAMBDA system provide quite powerful support for mechanical proof not only for hardware but also for software and wider areas. The work of our thesis are based on these systems to use proof-based method to deal with certain application area: a class of computer. Although our work concentrates on a limited area comparing with the work of universal design methodology and general design logic below, however, some general significant concepts are presented through our work. This is mainly the concept of derivation model which describe a general logical structure to cover whole construction process of a class of computer and possibly to be extended to whole hardware area.

Hanna clearly introduced the concept of design logic [38], [39]. He used dependent types to achieve a rigorous and yet theoretically simple connection between circuits and their behavioral specifications [40]. He used the VERITAS system in a goal-directed manner to generate interactively a design and a correctness proof. The synthesized design is then translated into the MODEL hardware description language [36]. Dany Suk used Hanna's idea in his hardware type theory [70].

The concept of dependent type is the cornerstone of Hanna's system and it is concise and beautiful. He presented the concept of design logic for hardware. This is bold and creative. However, with respect to realistic design practices this is still at a quite abstract description level. Using the design logic to describe complex realistic problems is still quite difficult. This is just as Hanna himself said in [40] pp 55: "in the authors' opinion, a very open question as to whether these theoretical gains can be translated into practical ones. Thus far, our own computational trials have been limited to very simple circuits and it is not clear how both the computational load and the amount of human guidance required will scale with increasing size of circuit."

We think that we should find a way to bridge from general design methodology and general design logic to realistic and complex applications. For this we need to select a trade-off: instead of universal design methodology and general design logic we limit our goal in certain area, for example, the formal design of a class of computers, which is at a level of nearly realistic hardware complexity and meanwhile has a concise form of general logical structure. However, for the limited area we work forward to a goal which is as generic as possible, for example, for our work we deal with a quite generic top specification scheme but not one concrete computer. In the process we can obtain a trade-off point: it is general enough for the certain area of hardware, for us it is the formal design of computers, but still particular for more general design logic and design methodology. For our work these trade-off points are the concepts of specification scheme and derivation model. Instead of abstract and general design methodology and design logic our derivation model as a general hardware model can cover whole construction process of a class of computers and we expect that it can be generalized to cover even whole hardware area.

14.3 Other Relevant Work

In this subsection we cite some work which do not directly related to our work but are interesting referring to them to understand our work. Mainly, these work deal with the verification of microprocessors or proof-based formal hardware design methods.

D. Borrione et al. [5] presented μSPEED, a framework for specifying and verifying microprocessors. It is a proof methodology associating functional semantics to each abstraction level of specification and verification and is implemented by a menu-driven tool working as a specialized editor. This tool automatically produces the functional expressions associated to these specification levels, by way of a user-friendly interaction with the hardware designer, and calls on a specialized formal verifier based on rewriting techniques.

M. Langevin and E. Cerny [54] used the formalism of a specific kind of logic formulas which can be represented using BDDs in a verification algorithm for processor-like circuits. The verification algorithm consists in extracting the observable behavior of the circuit implementation and in comparing it with the specification.

J. Chazarain and H. Collavizza [11] developed an object oriented programming based method and a computer algorithm system to prove automatically the correctness of the translation into micro-instructions of each assembly language instruction for a given processor.

N. A. Harman and J. V. Tucker, [41] introduced a general algebraic method for modeling microprocessors at different levels of abstraction by means of iterated maps to support equational specification and design.

D.A. Basin and P. B. Jackson reported about their work using Nuprl to extract circuits from constructive proofs [1] [2] [47]. D.A. Basin described a foundation for using constructive type theory as a tool for combining hardware synthesis and verification. He specialized the proofs-as-programs paradigm for the Nuprl constructive type theory to "proofs-as-circuits", whereby representations of circuits can be automatically synthesized from constructive proofs.

H. Busch [7], [8] presented a framework for proof-based transformation of hardware designs.

Chapter 15

Conclusions and Further Work

15.1 Conclusions

Two intentions stimulated our work. One is to investigate the question of how to apply formal methods to complex and realistic hardware designs. The other is to investigate what best formal methods can do for hardware design. These lead to our thesis: the formal derivation of a class of computers.

We formally specified, constructed and verified a class of computers from the machine instruction specification at the behavior level to the register transfer description at the structural level.

The "class" is defined by a behavior specification scheme which is an abstract syntax structure. The structure can cover a set of specifications: any specification which is an instance of the abstract syntax structure belongs to the behavior specification scheme. The behavior level structure is a general semantic framework to describe the state transition semantics of the computer's machine instructions. It is a scheme because it can be instantiated to specify the semantics of a particular instance of a computer. Therefore the behavior specification scheme establishes a wide class of computers as the target of formal derivation.

Formal construction is a main characteristic of derivation. It can describe exactly and accurately how to produce a new specification from a given one. This is a basic and essential step for formal design. Instead of using proof to replace construction we proposed to view construction and proof as two independent parts and in particular make a formal construction which is independent of proof. Both aim at the same goal: to treat the design process formally.

We singled out 10 levels of specification schemes, and correspondingly, presented 9 construction steps. Except for the behavior specification scheme which is given each specification scheme was formally constructed from its parent specification scheme and was formally proved correct with respect to its parent scheme. Because a specification scheme covers a set of specifications for each construction step we proved its totalness as a mapping to guarantee that the construction step covers each specification in the parent specification scheme, while correctness guarantees that each constructed specification satisfies its corresponding parent specification. The specification schemes and construction steps cover the formal derivation of a class of computers from the behavior specification scheme to the structural specification scheme (implementation). Thereby we completed the formal derivation for a class of computers.

Derivation differs from verification. Verification concentrates on consistent proofs relating implementation and specification, and passively accepts the designed implementation as a given fact. Derivation on the other hand has the freedom of design: only the specification is given from which it actively constructs the implementation and at the same time to verifies the construction process. Therefore, derivation must deal with both aspects: formal construction and formal verification, especially, it should make clear what formal construction means and how to do it.

However, we should fairly comment on verification, derivation, and their relationship. Verification must be able to treat a *having-been-designed* implementation by rigorous proof. This makes verification sometimes more difficult than design including derivation as formal design. At root, the great significance of verification is the formal method. By the formal method we can enter a new world: using rigorous mathematics to hold an object. Derivation is developing verification further in that it avoids the difficulty of verification by putting construction and verification together. Derivation should make its distinct point: formal construction.

Derivation differs from synthesis. Generally speaking, derivation is only a part of synthesis. Synthesis is a somewhat loose concept. Usually the term refers to any process leading from a specification to an implementation. In contrast, derivation only concentrates on the basic aspect of formal synthesis: formally to produce a new specification from a given specification. It cannot include other aspects of synthesis like, for example, optimization. Derivation emphasizes formal construction and logical proof-based verification.

Formal construction is a leading string throughout the whole derivation process. The formal specification and verification of a derivation only serves the formal construction.

In this thesis we presented the derivation model which is the center of our derivation method. With the model we have a clear form to express the *relationship* between specifications. We described the construction process in the three-dimensional world of logic, space and time. A key concept in the model is the entry predicate which "glues" specifications together and is a general relation existing in the construction of microprogrammed computers. Based on the concepts of the derivation model and entry predicate the construction process of a computer mainly becomes a process of introducing new entry predicates for the logical dimension, new time points in the temporal dimension, and new ports (signal) in the spatial dimension.

Our work of deriving a class of computers established a concept of proof-based computer architecture. It demonstrated that the hierarchical structure of a whole computer can be organized through a dynamic construction and proof process. To understand such a computer architecture we proposed to put it into a three dimensional world of logic, space and time where the logical aspect is the leading one. Based on the derivation model and entry predicate we showed that the two aspects of unity of opposites, the controlling and the controlled, are visible throughout the whole hierarchical construction process of a computer from behavior to structural specification and, similarly, the hierarchical construction of the microprogram is visible throughout the whole derivation process too. The primitive element of control is the microinstruction address. We established a hierarchy of microprogramming whose three high levels are global loop, abstract microprogram structure, and linear microprogram. We identified a sequence of time layers throughout the whole derivation process in which the two traditional time layers of machine instruction and microinstruction are only two extreme ends (special cases).

We took Gordon's computer as a typical case study to illustrate the presented concepts and methods. We derived three different implementations and compared them with the one of Gordon's computer.

15.2 Further Work

15.2.1 Mechanical Design System

The natural next step to be taken is to implement and mechanize our work. We would aim for a semi-automatic or nearly automatic tool for the design and verification of computers. It would give exact documentations to explain the middle design processes and results, and especially the abstract and high level structure of a computer. What the user would need to do is only to give the behavior specification describing the meaning of machine instruction of the intended computer, and some high level selection including the optimization from the high-level synthesis. The mechanization of our work is feasible because the formal base and basic process have been set up and made clear in the present thesis. In fact the essential part of the mechanization has been reduced to be such exquisite that it can be implemented by about forty clauses of Prolog.

If we develop a computer-aided design system for the design of computers along the lines set out in this thesis we have more freedoms of choice. First, for each of the two aspects of derivation, construction and verification, we can choose to be either formal or mechanical. For example, we can choose a mechanical design which can automatically produce a whole class of implementations and a formal verification which verifies the entire set of constructed specifications to be correct rather than on a one-by-one basis. Second, we can choose only to derive a concrete computer following the general scheme of specification, construction, and verification of a class of computers.

A first step towards an implementation of a mechanical construction and mechanical verification using Prolog and LAMBDA has been undertaken by the author [73] [12] [24]. The design of the whole implementation will be an interesting non-trivial work. We wish to have a chance to complete this work.

15.2.2 Mathematical Foundation of Hardware Design

To represent circuits in logic it is usual to adopt the following paradigm [32], [40]:

- Components are predicates and ports are free variable.
- Port connections amount to existential hiding.
- Component composition is logical conjunction.

In this thesis the paradigm was extended to the derivation model for deriving *microprogrammed computers*. As a further work we will investigate the possibility of extending the derivation model to a wider area. The idea is that we should find a suitable form to express the logical relation of hardware components *directly* instead of dealing with it in the proof process indirectly. Based on the extended model the formal construction of hardware will become *Logic Structure Refinement*.

Finally, it is author's personal viewpoint that I believe that in the area of hardware we can also find a formal model and a logical system which is at such a position as Hoare Logic's fundamental, concise and as a start point for its following formal programming researches. only need about forty clauses

Bibliography

[1] D. A. Basin and P. D. Vecchio, Verification of Combinatorial Logic in Nuprl, In M Leeser and G Brown, ed., Hardware Specification, verification and Synthesis: Mathematical Aspects, Springer Verlag, July 1989.

[2] D. A. Basin, Extracting Circuits from Constructive Proofs, In 1991 International Workshop on Formal Verification in VLSI Design. ACM IFIP WG 10.2, Jan. 1991.

[3] J. Bormann, H Nusser-Mwhlan, and G. Venzl, Formal Design in an Industrial Research Laboratory: Lessons and Perspectives, In J. Staunstrup and R. Sharp, ed., Second Workshop on Designing Correct Circuits, Lynbgy, Denmark, Jan. 1992 WG10.2, WG 10.5.

[4] D. Borrione and J. L. Paillel, An Approach to the Formal Verification of VHDL Descriptions, Techical report RR 683-1, IMAG/ARTEMIS, Nov. 1987.

[5] D. Borrione, H. Collavizza and C. Le Faou, μSPEED: A Framework for Specifying and Verifying Microprocessors, in Int. Workshop on Formal Methods in VLSI Design 9-11 Janv. 1991, Miami, Springer-Verlag.

[6] B. Bose and S. D. Johnson, DDD-FM9001: Derivation of a Verified Microprocessor, G. J. Milne and L. Pierre eds. Correct Hardware Design and Verification Methods, LNCS 683, Springer-Verlag 1983.

[7] H. Busch, Proof-Based Transformation of Hardware Models, in Proceedings of the Workshop on Designing Correct Circuits, Oxford, edited by G. Jones and M. Sheeran, Springer, 1990.

[8] H. Busch, Transformation Design in a Theorem Prover, Theorem Provers in Circuit Design, pp. 175-196, Edited by V. Stavridou, T.F. Melham and R.T. Boute, North-Holland, 1992.

[9] R. M. Burstall, Proving Properties of Programs by Structural Induction, Comput. J., 12(1) Feb. 1969.

[10] P. Camurati, P. Prinetto, Formal Verification of Hardware Correctness: Introduction and Survey of Current Research, IEEE Computer, July 1988.

[11] J. Chazarain and H. Collavizza, Combining Symbolic Evaluation and Object Oriented Approach for Verifying Processor-like Architectures at the RT-level, pp. 109 - 121, G. J. Milne and L. Pierre Eds. Correct Hardware Design and Verification Methods, LNCS 683 Springer-Verlag, May 1993.

[12] W. F. Clocksin and C. S. Mellish, Programming in Prolog, Third, revised and Extended Edition, Springer-Verlag, 1987.

[13] Avra Cohn and M. Gordon, A Mechanised Proof of Correctness of a Simple Counter, Technical report 94, University of Cambridge Laboratory, July 1986.

[14] Avra Cohn, A Proof of Correctness of the Viper Microprocessor: First Level, in G. Birtwistle and P. Subrahmanyam, eds., VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, Boston, 1988, pp. 27-71.

[15] Avra Cohn, Correctness Properties of the Viper Block Model: The Second Level, in G. Birtwistle and P. Subrahmanyam, eds.,Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, 1989, pp.1-91.

[16] Avra Cohn, The Notion of Proof in Hardware Verification, Journal of Automated Reasoning, Vol. 5, May 1989, pp. 127-139.

[17] R. L. Constable and D. J. Howe, Nuprl as a general Logic, In P. Odifreddi, ed., Logic and Computer Science, Vol 31 of APIC Studies in Data Processing, Academic Press, 1990.

[18] S.D. Crocker, E. Cohen, S. Landauer, and H. Orman. Reverification of a microprocessor. In Proceedings of the IEEE Symposium on Security and Privacy, pages 166-176, April 1988.

[19] W.J. Cullyer, Implementing Safety-Critical Systems: The VIPER Microprocessor, in: G. Birtwistle and P. Subrahmanyam. eds., VLSI Specification, verification and Synthesis, Kluwer Academic Publishers, 1988, pp. 1-25.

[20] W.J. Cullyer, High Integrity Computing, in: M. Joseph, ed., Formal Techniques in Real-Time and Fault-Tolerant Systems, Lecture Notes in Computer Science, No. 331, Springer-Verlag, Berlin, 1988. pp. 1-35.

[21] H. Eveking, Axiomatizing Hardware Description Languages, International Journal of Computer Aided VLSI Design, 2:263-280, 1990

[22] S. Finn, M. Fourman, M. Francis and R. Harris, Formally Based System Design - Interactive Synthesis Based on Computer-Assisted Formal Reasoning, in Formal VLSI Specification and Synthesis - VLSI Design Methods I, edited by L. Claesen. (North-Holland, 1990), pp. 139 - 152.

[23] M. P. Fourman, W. J. Palmer, and R. M. Zimmer, Using Higher-Order Functions to Describe Hardware, In Proceedings of the Electronic Design Automation Conference, Wembley, 1987.

[24] M. P. Fourman, R. L. Harris, Lambda-Logic and Mathematics Behind Design Automation, 26th ACD/IEEE Design Automation Conference, 1988.

[25] M. P. Fourman, W. J. Palmer, and R. M. Zimmer, Proof and Synthesis, In Proceedings ICCD '88, Rye Brook, NY, 1988.

[26] M. P. Fourman, Formal System Design, in J. Staunstrup, editor, IFIP WG 10.5, Formal Methods for VLSI Design, pp 191-236, North-Holland, 1990.

[27] M. P. Fourman and R.A. Hexsel, Formal Synthesis, in Graham Birtwistle, editor, *IV Higher Order Workshop, Banff 1990,* pages 245-264, Springer-Verlag.

[28] Kees G. W. Goossens, Embedding Hardware Description Language in Proof Systems, PhD Thesis, CST-100-93, Department of Computer Science, University of Edinburgh, May 1993.

[29] M. J. C. Gordon, R. Milner, and C. Wadsworth, Edinburgh LCF, Vol 78 of LNCS, Springer Verlag, 1979.

[30] M. J. C. Gordon, The Denotational Semantics of Sequential Machines. Information Processing Letters, 10(1):1-3, Feb. 1980.

[31] M. J. C. Gordon, Proving a Computer Correct with the LCF_LSM Hardware Verification System, Technical Report No. 42, Computer Laboratory, University of Cambridge, 1983.

[32] M. J. C. Gordon, Why higher-order logic is a good formalism for specifying and verifying hardware, in: G. Milne and P. Subrahmanyam, eds., Formal Aspects of VLSI Design, Proceedings of the 1985 Edinburgh Conference on VLSI, North-Holland, 1986, pp. 153-177.

[33] M. J. C. Gordon, HOL: A Proof Generating System for Higher-Order Logic, University of Cambridge, Computer Laboratory, Tech Report No. 103, 1987.

[34] M. J. C. Gordon, Programming Language Theory and its Implementation, C.A.R. Hoare Series Editor, Prestice Hall, 1988.

[35] F.K. Hanna and N.Daeche. Specification and Verification of Digital Systems using Higher-Order Predicate Logic, IEE Proceedings, Vol. 133, Part E, No. 5, September 1986, pp. 242-254.

[36] F.K. Hanna, M.Longley, and N.Daeche. Formal synthesis of digital systems. In Luc Claesen, editor, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Volume 2,* pages 532-548, Elsevier Science Publishers, B.V. North-Holland, Amsterdam, 1989.

[37] F.K. Hanna, N.Daeche and M.Longley, Specification and Verification using Dependent Types, pp. 949 - 964, IEEE Trans on SE, Vol. 16, No. 9, Sept 1990.

[38] F.K. Hanna, N.Daeche and G. Howells, Implementation of the Veritas Design Logic, Theorem Provers in Circuit Design, pp. 77-94, Edited by V. Stavridou, T.F. Melham and R.T. Boute, North-Holland, 1992

[39] F.K. Hanna and N.Daeche, The Veritas Design Logic: A User's View, Theorem Provers in Circuit Design, pp. 301-310, Edited by V. Stavridou, T.F. Melham and R.T. Boute, North-Holland, 1992.

[40] F.K. Hanna and N.Daeche. Strongly-Typed Theory of Structure and Behaviors, pp. 39 - 54, G. J. Milne and L. Pierre Eds. Correct Hardware Design and Verification Methods, LNCS 683, Springer-Verlag, May 1993.

[41] N.A. Harman and J. V. Tucker, Algebraic Models and the Correctness of Microprocessors, pp. 92 - 108, G. J. Milne and L. Pierre Eds. Correct Hardware Design and Verification Methods, LNCS 683 Springer-Verlag, May 1993.

[42] C.A.R. Hoare, An Axiomatic basis for Computer Programming, Commun. ACM 12, 10 Oct. 1969, 576-580, 583.

[43] C.A.R. Hoare, A Calculus of Total Correctness for Communicating Processes, Science of Computer Programming, Vol. 1, No. 1, 1981.

[44] Warren A. Hunt, FM8501, A Verified Microprocessor, Ph.D. Thesis, Report No. 47, Institute for Computing Science, University of Texas, Austin, December 1985.

[45] Warren A. Hunt, Microprocessor Design Verification, Journal of Automated Reasoning, 5:429-460, 1989.

[46] C. D. Kloos, Semantics of Digital Circuits, Vol 285 of LNCS, Springer Verlag, 1987.

[47] P.B. Jackson, Nuprl and its Use in Circuit Design, Theorem Provers in Circuit Design, pp. 311-336, Edited by V. Stavridou, T.F. Melham and R.T. Boute, North-Holland, 1992.

[48] S. D. Johnson, Synthesis of Digital Design from Recursion Equations, ACM Distinguished Dissertation, The MIT Press, 1833, Indiana University PhD Thesis, May 1983.

[49] G. Jones and M. Sheeran, Circuit Design in Ruby, In J. Staunstrup, ed., Formal Methods for VLSI Design, North-Holland, 1990.

[50] J. J. Joyce, G. Birtwistle and M. Gordon, Proving a Computer Correct in Higher Order Logic, Report No. 100, Computer Laboratory, Cambridge University, 1986.

[51] J. J. Joyce, General Structure in the Formal Specification and verification of Digital Circuits, in G. Milne, ed., The Fusion of Hardware Design and Veri-

fication, Proceedings of the IFIP WG 10.2 International Working Conference, Glasgow, July 1988, North-Holland, 1988.

[52] J. J. Joyce, Using Higher-Order Logic to Specify Computer Hardware and Architecture, in D. Edwards, ed., Design Methodology for VLSI and Computer Architecture, Proceedings of the IFIP TC10 Working Conference on Design Methodology in VLSI and Computer Architecture, Italy, Sept 1988, North-Holland, 1989.

[53] J. J. Joyce, Multi-Level Verification of Microprocessor-Based System, Report No. 195, Computer Laboratory, Cambridge University, May 1990.

[54] M. Langevin and E. Cerny, Verification of Processor-like Circuits, In Proc. of The Advanced Work on Correct Hardware Design Methodology, Turin, 12-14 June 1991.

[55] Z. Manna, Mathematical Theory of Computation, McGraw-Hill Book Company, 1974.

[56] Michael Mendler, A Modal Logic for Handling Behavioural Constraints in Formal Hardware Verification, PhD Thesis, CST-97-93, Department of Computer Science, University of Edinburgh, March 1993.

[57] Thomas F. Melham, Abstraction mechanism for hardware verification. In G. Birtwhistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis,* pages 267-291. Kluwer Academic Publishers, 1988.

[58] Thomas F. Melham, Automating Recursive Type Definitions in Higher Order Logic, In G. Birtwistle and P. A. Subrahmanyam, ed., Current Trends in Hardware Verification and Automated Theorem Proving, New York, 1988, Springer Verlag.

[59] Thomas F. Melham, Using Recursive Types to Reason about Hardware in Higher Order Logic, Technical Report 135, University of Combridge Computer Laboratory, may 1988.

[60] M. C. McFarland, A. C. Parker, and R. Camposano, The High-Level Synthesis of Digital System, Proceedings of the IEEE, pp. 301-318, Feb. 1990.

[61] G. J. Milne, Design for Verifiability, In Proc. Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects, Cornel University, USA, July 1989.

[62] R. Milner, A Calculus of Communicating System. Lecture Notes in Computer Science No. 92. Springer-Verlag, 1980.

[63] R. Milner, An Interview with Robin Milner, Communication of the ACM, Vol.36, No.1 January 1993, pp. 96.

[64] L. C. Paulson, Logic and Computation: Interactive Proof with Cambridge LCF, Cambridge University press, 1987.

[65] L. Rossen, Formal Ruby, In Summer School on Formal Methods for VLSI Design, IFIP WG 10.5, June 1990.

[66] J. B. Saxe, S. J. Garland, J. V. Guttag, and J. J. Horning, Using Transformations and Verification in Circuit Design. In J. Staunstrup and R. Sharp, ed., Second Workshop on Designing Correct Circuits, Lynbgy, Denmark, Jan. 1992, IFIP WG 10.2, WG 10.5.

[67] D. A. Schmidt, Denotational Semantics A Methodology for Language Development, Allyn and Bacon, Inc. 1986.

[68] V. Stavridon, J. A. Goguen, A. Stevens, S. M. Eker, S. N. Alonefits, and K. M. Hobley, FUNNEL and 2OBJ: Towards and Integrated Hardware Design

Environment, In V. Stavridou, T. F. melham, and R. T. Boute, ed., Theorem Provers in Circuit Design: Theory, Practice and Experience, IFIP TC10/WG 10.2, North Holland, June 1992.

[69] J. E. Stoy, Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. The MIT Press, 1977.

[70] Dany Suk, Hardware Synthesis in Constructive Type Theory, Designing Correct Circuits, Edited by Jones and Sheeran, pp 29-49, Oxford, Springer-Verlag, 1990.

[71] R. R. Vemuri, A Transformational Approach to Register-Transfer-Level Design-Spece Exploration, PhD Thesis, Department of Computer Engineering and Science, Case Western Reserve University, Jan.1989.

[72] Li-Guo Wang, Hardware Synthesis Logic and its Independence, Manuscript, LFCS, Department of Computer Science, University of Edinburgh, Nov. 1991.

[73] Li-Guo Wang, Deriving a Correct Computer, In L. J. M. Claesen and M. J. C. Gordon, Editors, *IFIP Transactions, Higher Order Logic Theorem Proving and its Application,* pages 449-458, Elsevier Science Publisher B.V. North-Holland, Amsterdam 1993.

[74] P. J. Windley, The Formal verification of Generic Interpreters, OhD thesis, University of California, Davis, Division of Computer Science, June 1990.

[75] P. J. Windley, A Hierarchical Methodology for the Verification of Microprogrammed Microprocessors, In Proceedings of the IEEE Symposium on Security and Privacy, may 1990.

[76] P. J. Windley, The Formal Specification of a High-Speed CMOS Correlator, In Proceedings of the Third Annual IEEE/NASAS Symposium on VLSI Design, Oct 1991.

[77] P. J. Windley, Abstract Theories in HOL. In Proceedings of the 1992 International Conference on the HOL Theorem Prover and its Application, Oct 1992.

[78] P. J. Windley, A Theory of Generic Interpreters, pp.122 - 134, G. J. Milne and L. Pierre, eds., Correct Hardware Design and Verification Methods, LNCS 683, Springer-Verlag, May 1993.