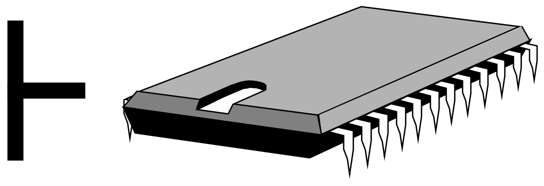
IFIP WG 10.5

Benchmark Circuits for Hardware Verification

v1.2.0



1 Introduction **1**

2 The Benchmark Circuits **2**

2.1 Releases 2

2.2 Verification Problem Presentation 3

2.2.1 VHDL 3

2.2.2 Storage Elements and Multiplexers 3

2.2.3 Base Module Library 4

2.2.4 Graphical Notation 5

2.3 Classification of the Verification tasks 6

2.4 Circuit Classification 6

3 How to get the benchmark circuits **8**

3.1 Physical Organization 8

3.2 Using the benchmarks 9

3.3 Please Contact us if … 9

3.4 Email list 9

4 Present and Future Activity **9**

5 Acknowledgments **10**

6 Literature **10**

7 Single Pulser **11**

7.1 Introduction 11

7.2 Specification 11

7.3 Implementation 11

7.4 Status and Acknowledgments 12

7.5 Literature 12

8 Traffic Light Controller **13**

8.1 Specification 13

8.2 Implementation 13

8.3 Status and Acknowledgments 13

8.4 Literature 13

9 N-bit Adder **14**

9.1 Introduction 14

9.2 Specification 14

9.3 Implementation 14

9.4 Status and Acknowledgments 14

9.5 Literature 14

10 Min_Max Circuit **18**

10.1 Introduction 18

10.2 Specification 18

10.3 Implementation 18

10.4 Status and Acknowledgments 20

10.4.1 Comments to the Specification of MIN_MAX 20

10.5 Literature 22

11 Black-Jack Dealer **23**

11.1 Specification 23

11.2 Implementation 23

11.3 Status and Acknowledgments 25

11.4Literature 25
12 Arbiter 29
12.1Introduction 29
12.2Specification 29
12.2.1Formal specification example 29
12.3Implementation 29
12.4Status and Acknowledgments 31
12.5Literature 31
13 Rollback Chip 34
13.1Specification 34
13.2Implementation 34
13.3Status and Acknowledgments 34
13.4Literature 34
14 The TAMARACK Processor 35
14.1Specification 35
14.2Implementation 35
14.3Status and Acknowledgments 35
14.4Literature 35
15 Stop-Watch 36
15.1Specification 36
15.2Implementation 36
15.3Status and Acknowledgments 37
15.4Literature 37
16 GCD - Greatest Common Divisor 41
16.1Introduction 41
16.2Specification 41
16.2.1Low-Level Specification 41
16.2.2High-Level Specification 41
16.3Implementation 42
16.3.1Controller 42
16.3.2Data Path 43
16.3.3Simulations 43
16.4Status and Acknowledgments 45
16.5Literature 45
17 Multiplier 46
17.1Specification 46
17.2Implementation 46
17.2.1Algorithm 46
17.2.2Global Realization Architecture 46
17.2.3Detailed Description 46
17.3Status and Acknowledgments 49
17.4Literature 49
18 Divider 50
18.1Specification 50
18.2Implementation 50
18.2.1Algorithm 50
18.2.2Global Realization Architecture 50
18.3Detailed Implementation Description 52

18.4Status and Acknowledgments 55
18.5Referencee 56

19 FIFO 57
19.1Specification 57
19.2Implementation 57
19.3Status and Acknowledgments 57
19.4Literature 57

20 Assoc (Associative Memory) 60
20.1Specification 60
20.1.1General Specification 60
20.2Implementation 61
20.2.1General architecture 61
20.2.2Implementation of one cell of AssocMemArray 62
20.2.3Implementation of a Storage Cell Array (AssocMemArray) 62
20.2.4Implementation of the whole Associative memory (AssocMemCirc) 62
20.3Status and Acknowledgments 63
20.4Literature 63

21 1Syst (Filter) 67
21.1Introduction 67
21.2Specification 67
21.3Implementation 67
21.3.1General architecture 67
21.3.2Implementation of one stage 68
21.4Status and Acknowledgments 68
21.5Literature 68

22 2Syst (Matrix Multiplication) 71
22.1Specification 71
22.1.1General Specification 71
22.1.2Timing Requirements 72
22.2Implementation 73
22.2.1General architecture 73
22.2.2Implementation of one cell 74
22.2.3Actual Realization and Example 75
22.3Status and Acknowledgments 75
22.4Literature 75

Benchmark-Circuits for Hardware -Verification

v1.2.1

Thomas Kropf
Institut für Rechnerentwurf und Fehlertoleranz
Universität Karlsruhe, Kaiserstr. 12, 76128 Karlsruhe, Germany
email: Thomas.Kropf@informatik.uni-karlsruhe.de
WWW: http://goethe.ira.uka.de/hvg/

This document describes the IFIP WG10.5 hardware-verification benchmark circuits, intended for evaluating different approaches and algorithms for hardware verification. The paper presents the rationale behind the circuits, describes them briefly and indicates how to get access to the verification benchmark set.

1 Introduction

Although having many drawbacks, benchmark circuits allow a more succinct and direct comparison of different approaches for solving a certain problem. This has lead to sets of widely accepted circuits e.g. in the area of testing [BrPH85, BrBK89] or high-level synthesis [VRMK91].

In the area of hardware verification, this has lead e.g. to the suggestion of “interesting” circuits, like Paillet’s set of seven sequential circuits [Pail85]. One of the first efforts to provide circuits for a broader community has been done by Luc Claesen for the 1990 International Workshop on Applied Formal Methods for VLSI Design [FMVD90]. The most prominent circuit evolving from this effort was the “Min_Max-Circuit”.

The lack of additional, generally available verification benchmark circuits got aware in the preparation of the 2nd International Conference on Theorem Provers in Circuit Design (TPCD94). The motivation to provide additional benchmark circuits together with already ongoing standardization efforts of IFIP, coordinated by Jørgen Staunstrup [Stau93], has led to an enhanced set of circuits. Thanks to J. Staunstrup, in the meantime these circuits have become the official “IFIP WG10.2 Hardware Verification Benchmark Circuit Set”. It will be maintained and enhanced on a long term basis to promote a standardized benchmark set in the hardware verification community.

For the circuits a complete and self-contained implementation is provided, done in a commercial design system [Fram92] (leading e.g. to additional timing diagrams for clarification) as well as a clear specification using the standardized hardware description language VHDL (see Section 2.2 “Verification Problem Presentation”). This puts a comparison of different verification approaches on a sound basis, since - if the given implementations are used - identical circuits are verified instead of different designs implemented in a way especially suited for a certain approach. Moreover, people are not forced to tediously design the circuits before they can be verified - the latter being the main interest of people looking into these circuits. Naturally, when dealing with formal synthesis, the implementations provided here are less interesting and the specifications are the main thing to deal with.

A set of verification benchmark circuits has to provide easily usable circuits to evaluate and to compare different approaches to hardware verification. This comprises:

- availability via the World Wide Web and anonymous ftp,
- a high degree of diversity with regard to the underlying verification task (see Section 2.3 “Classification of the Verification tasks”),
- circuit descriptions without ambiguity, which are succinct and self-contained and
- circuits which span a wide range from introductory examples to real verification challenges [Stau93].

2 The Benchmark Circuits

Currently, there is the set of benchmark circuits given in table 2-1. The main sources for these circuits have been the previous IFIP benchmark set [Stau93] and various textbooks on circuit design.

Circuit Name	Short Circuit Description
Single Pulser	cuts input pulses to a fixed length
Traffic Light Controller	simplified controller for a traffic light
N-bit Adder	sum of two bitvectors of length N
Min_Max	the mean value of incoming integers
Black-Jack Dealer	the dealer’s hand of a card game
Arbiter	access to shared resources for N clients
Rollback Chip	coprocessor for distributed simulation
Tamarack Processor	simplified microprocessor
Stop-Watch	digital stopwatch with 3 digits and 2 buttons
GCD	Greatest Common Divisor
Multiplier	N-bit Multiplier
Divider	N-bit Divider
FIFO	asynchronous FIFO queue with N places
Assotiative Memory	simple assotiative $N \times M$ memory
1dim Systolic Array	onedimensional systolic filter array
2dim Systolic Array	two-dimensional systolic array for matrix multiplication

Table 2-1: Current Benchmark Circuits

2.1 Releases

Each release of the circuit is labeled with a version number, release number and patch level in the form v<Version>.<Release>.<Patchlevel>.

A version number is provided to make the inclusion of new circuits more explicit. Unfortunately we did not succeed in providing full implementation descriptions for all circuits in the first release. As these are provided, the release number is incremented.

Although the circuits have been designed with much care, reality-driven pessimism suggests that there will be the necessity for “patches” (i.e. bug fixes) at least in the first stage of the release process.

All changes are documented in a history file, patches are also explained in the documentation of the relevant circuits. You may want to be added to an emailing list dedicated to inform people about the actual verification benchmark status (see Section 3.4 “Email list”).

2.2 Verification Problem Presentation

Unambiguously specifying circuits in a general way without being forced to a certain description philosophy by the underlying notation is a challenge itself. The only way to circumvent a fixed description formalism is to use only informal descriptions (natural language, drawings[1], timing diagrams, etc.). However, then the descriptions are often not as crisp and exact as it is necessary especially for formal verification, where the information given here has to be translated into formal description languages like predicate logics, temporal logics, process algebras, Z and so forth.

Besides all drawbacks, we decided to provide more (and more formal) information for each circuit. This comprises for each circuit of:

- a description of the specification and the implementation in plain English,
- schematic diagrams of the implementation[2],
- a netlist of the implementation based on structural VHDL and
- a specification of the circuit in VHDL.

2.2.1 VHDL

The decision to use VDHL as a formalization means especially for specifications is probably the most disputable one for obvious reasons: the lack of a clean VDHL semantics, the danger of imposing a certain specification style, impossibility for expressing nondeterminism and so forth. However, in our opinion the advantages of providing a standardized (and simulatable) specification outweighs the disadvantages. Moreover, we tried to avoid ambiguous VDHL constructs and VHDL specifications may further encourage VHDL based verification (or at least the discussion about the “right” specification language is stimulated).

We use a restricted set of VHDL which should be sufficiently simple so that no semantic ambiguities occur.

2.2.2 Storage Elements and Multiplexers

A register is treated as a *base module* (see Section 2.2.3 “Base Module Library”). It is used as follows: The inputs **S** is connected to input lines named **`Store(Name)`**. A signal is stored, if **`Store(Name)`** =1 and there is a $0 \to 1$ transition at the clock input.

Lines named **`SelectXY`** are controlling a multiplexer or a demultiplexer: **`SelectXY`** = 0 connects input 0 of a multiplexer to the output (output 0 of a demultiplexer) and **`SelectXY`** = 1 connects input 1 to the output (output 1 of a demultiplexer).

Busses are provided with their names and number of signal lines: **`DataIn<31:0>`** denotes a 32 bit bus for reading data.

1. For some circuits, original schematics of a realization in the semi-custom design system CADENCE have been added. However, to fit these drawings on a single page they had to be shrunk in many cases so that labels etc. may not be well readable anymore. Nevertheless, the remaining circuit documentation without these figures is completely sufficient as an implementation description.
2. Schematics are based on an implementation of each circuit using a commercial semi-custom design system.

2.2.3 Base Module Library

Most circuit implementations use a predefined base module library which contains parameterized modules. Parameters are delay time and, for certain elements, the bitwidth. The library contains mainly simple gates and storage elements, listed in table 2-2. The respective VHDL descriptions can be found in the file `GateLib.vhd`.

Module Name	Description	Variable Bitwidth
INV	inverter	no
BUF	buffer	no
nBUF	generic n bit buffer	yes
NAND2	two input nand	no
NAND3	three input nand	no
AND2	two input and	no
AND3	three input and	no
AND4	four input and	no
AND5	five input and	no
NOR2	two input nor	no
NOR3	three input nor	no
OR_2[a]	two input or	no
OR3	three input or	no
OR4	four input or	no
OR5	five input or	no
NXOR2	two input equal	no
XOR2	two input exor	no
MUX	two input, one output, one select multiplexer	no
DMUX	one input, two output, one select demultiplexer	no
DL	D-latch	no
DFF	D-flipflop	no
DFFs	setable D-flipflop	no
DFFsr	setable and resetable D-flipflop	no
RSFFR	resetable RS-flipflop	no
RSFF	RS-flipflop	no
nMUX	generic n-bit 2:1 multiplexer	yes
nDMUX	generic n-bit 2:1 demultiplexer	yes
nREG	generic n-bit register with enable	yes
nREGr	generic n-bit register with enable and reset	yes
HA	half adder	no
FA	full adder	no
nINC	n-bit incremented	yes
CRA	n-bit carry ripple adder	yes
AddSub	n-bit carry ripple adder and subtractor	yes
nSREG	n-bit shift register	yes
nCMPO	n-bit equal zero test	yes

Table 2-2: Elements of the Base Module Library

Module Name	Description	Variable Bitwidth
cCMPN	compare tow n-bit vectors	yes
nLSH	n-bit left shift	yes
nRSH	n-bit right shift	yes
expand	expand bit vector by adding zeros	yes
priority	lower bit has priority others are suppressed	yes

Table 2-2: Elements of the Base Module Library

a. underscore necessary due to name conflicts in SYNOPSIS

2.2.4 Graphical Notation

The schematic diagrams consist of modules which are drawn according to the notation given in table 2-3.

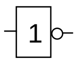
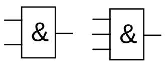
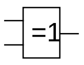
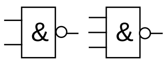
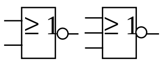
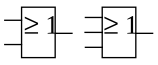
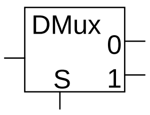
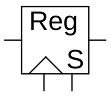
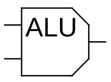
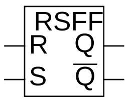
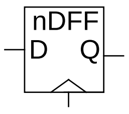
Module type	Graphical Notation
Inverter	
AND gates	
EXOR gate	
NAND gates	
NOR gates	
OR gates	
Demultiplexer 1:2	
Register	
Arithmetic Logic Unit	
RS-Flipflop	
Register with n Flipflops	

Table 2-3: Schematic Drawing Symbols

2.3 Classification of the Verification tasks

There are mainly three verification tasks to be distinguished when talking about hardware verification:

1. verifying that a circuit specification is what it should be,
2. verifying that a given implementation behaves identically to a given specification and
3. verifying important (e.g. safety critical) properties of a given implementation.

According to [Stau93], the first is called *requirements capture*, the second *implementation verification* and the third *design verification*.

The three tasks are often expressed in terms of a specification *S* and an implementation *I*, where a *complete* verification denotes some form of equivalence between *S* and *I* ($S = I$, $S \approx I$, $S \Leftrightarrow I$) and a *partial* verification denotes some form of implication ($I \Rightarrow S$, $I \supseteq S$, $I \mid\!- S$):

The first task is a partial verification (with *S* describing properties of the circuit specification and *I* being the circuit specification), the second is a complete verification (with *S* being the specification and *I* being the implementation) and the third is again a partial verification (with *S* describing the properties of the circuit implementation and *I* being the circuit implementation).

It is to be noted that if e.g. exact computation times are not stated in a specification then we have to cope with a verification problem of the third kind, since in that case an equivalence proof is not possible.

All three verification tasks are covered by the benchmark circuits.

2.4 Circuit Classification

Every classification scheme has its drawbacks, but we found the following circuit properties especially useful for classification purposes.

A circuit may be classified in several dimensions as depicted in table 2-4. Most of the criteria are self-explaining, besides complexity. In the area of testing usually the number of internal lines is directly used as a complexity measure [BrPH85, BrBK89], motivated by the designated application of the circuits: test pattern generation. Using a classical stuck-at fault model, the set of faults to be treated equals the number of internal lines. Hence a circuit s713 denotes a sequential circuit with 713 internal lines [BrBK89].

In the area of hardware verification a similar complexity measure is not as obvious (at least we did not found any meaningful). Hence we use the coarse measure, proposed by J. Staunstrup: an example is either introductory, illustrative or a real challenge [Stau93].

Classification Criterion	Value Set	
Abstraction level	system	(s)
	algorithmic	(a)
	register-transfer	(r)
	gate	(g)
	transistor	(t)
Synchronicity of the implementation	synchronous	(s)
	asynchronous	(a)
	combinational	(c)
Hierarchy of the implementation	hierarchical	(h)
	flat	(f)

Table 2-4: Possible Circuit Classifications

Classification Criterion	Value Set
Determinism	deterministic (d) nondeterministic (n)
Genericity	generic (g) concrete with (optional) bitwidth *n* (c*n*)
Type	controller (c) data path (d) mixed (both) (m)
Complexity	introductory (i) standard illustrative (s) challenge (c)

Table 2-4: Possible Circuit Classifications

The abbreviations given in table 2-4 may be used to characterize each circuit using the following "signature":

`<Name>:<Spec>-<Imp>.<Sync>.<Hier>.<Det>.<Gener>.<Type>.<Compl>`

A circuit `GCD.a-r.s.h.d.g.m.i` denotes the Greatest Common Divisor circuit, which has a specification on algorithmic level and an implementation on register-transfer level. It is a synchronous, hierarchical and deterministic design with arbitrary bit width. Consisting of a controller and a data path it is a small, i.e. an introductory example.

Using this classification scheme, we can characterize all circuits as shown in table 2-5.

Circuit Name	Classification
Single Pulser	`Pulser.g-g.s/a.f.d.c1.c.i`
Traffic Light Controller	`TLC.r-g.s.f.d.c1.c.i`
N-bit Adder	`Adder.a-g.c.h.d.g.d.i`
Min_Max	`Min_Max.a-g.s.h.d.c8/g.d.s`
Black-Jack Dealer	`Dealer.a-g.?.f.n.c.c.s`
Arbiter	`Arbiter.r-g.s.f.n.g.c.s`
Rollback Chip	`Rollback.?-?.?.h.d.?.m.s`
Tamarack Processor	`Tamarack.a-g.s.h.d.?.m.s`
Stop-Watch	`Stopwatch.r-g.s.h.d.c.m.s`
GCD	`GCD.a-g.s.h.d.g.m.i`
Multiplier	`Mult.a-g.c.h.d.g.d.s`
Divider	`Div.a-g.c.h.d.g.d.s`
FIFO	`FIFO.a-g.a.h.d.g.m.s`
Assotiative Memory	`Assoc.r-g.s.h.d.g.d.s`
1dim Systolic Array	`1Syst.a-g.s.h.d.g.d.s`
2dim Systolic Array	`2Syst.a-g.s.h.d.g.d.s`

Table 2-5: Classification of the Benchmark Circuits

3 How to get the benchmark circuits

The most convenient way to a get access to the benchmark suite is via the World Wide Web. Using a WWW browser like Mosaic you can get the latest informations using the URL `http://goethe.ira.uka.de/benchmarks/`.

You can also directly use an anonymous FTP-server. All benchmark circuits as well as PostScript versions of the documents (including this paper) have been made available there in the directory `pub/benchmarks`. The FTP-server is reachable as `goethe.ira.uka.de` (current IP address: `129.13.18.22`). The server will always hold the newest benchmark version (see Section 2.1 “Releases”).

Simple ASCII-files end in `.txt`, Postscript files always end in `.ps`, `.ps.Z` or `ps.gz`. The files with endings `.gz` (`.Z`) has been compressed using the UNIX `gzip` (`compress`) command. They must be transferred using binary ftp mode and must be expanded using the UNIX `gunzip` (`uncompress`) command before they are readable or printable. Tar-Files end in `.tar` (compressed `.tar.gz` or `.tar.Z`) and contain a whole directory in one file. The directory content may be rebuild by executing `tar  -xfv  <name>.tar` after having expanded the respective file.

3.1 Physical Organization

The main directory `/pub/benchmarks` contains:

• `README`	how to use and retrieve the benchmarks
• `It_is_version_x.y.z`	an empty dummy file indicating the current version, release and patchlevel
• `Introduction.ps`	this document as a PostScript file
• `Whole_documentation.ps`	this document plus all (currently available) circuit descriptions as a single PostScript File
• `History.txt`	the version history of the benchmarks
• `GateLib.vhd`	behavioral descriptions of all base modules (see Section 2.2.3 “Base Module Library”)
• `ELEMpack.vhd`	interface and componemt declaration of all base elements
• `<circuit>`	a directory for each benchmark circuit
• `documentation`	a directory containing the whole documentation in its original FrameMaker 4 format (for those who want to get the document “sources”)
• `pending`	a directory containing circuits which will probably be included in future releases of the benchmark circuits.[1]

Each circuit directory `<circuit>` contains at least:

• `<circuit>.ps`	a PostScript file describing the circuit
• `<circuit>.vhd`	different VHDL files covering the implementation (`<circuit>struc.vhd`), a behavioral specification (`<circuit>behave.vhd`), a testbench for simula-

1. For these circuits the documentation may be incomplete, inconsistent or completely missing.

tion (`<circuit>tb.vhd` or `<circuit>test-bench.vhd`) as well as various other files and scripts useful for simulation and synthesis.

3.2 Using the benchmarks

You can use the benchmark circuits in any way which suits your needs. Especially when specifying the problems, you are in no way obliged to use VHDL. The VHDL specifications are mainly provided to clarify the intended proof goals.

However, if for example, you use an implementation completely different from the ones given here (e.g. a simplified version) you should state this clearly whenever you refer to the circuits provided here.

Some of the circuits have been designed hierarchically. If you are flattening the circuits in order to verify them, you should state this also.

3.3 Please Contact us if …

Please contact us if you have any problems, especially

- if you have questions of any kind concerning the benchmarks,
- if you have comments or proposals for changes, additions or even new circuits,
- if you can provide “better” implementations for the circuits,
- if you have problems in printing the PostScript files,
- if you detect errors, inconsistencies or ambiguities, which should be fixed or
- if you have problems in accessing the files.

The easiest way is to send a brief email to Thomas Kropf or to Jørgen Staunstrup (`Thomas.Kropf@informatik.uni-karlsruhe.de, jst@id.dth.dk`).

3.4 Email list

To inform people about the latest release and patches of the benchmark circuits, we do maintain an informal emailing list. If you want to be added (or deleted) from this list, send a short note to `Thomas.Kropf@informatik.uni-karlsruhe.de`.

4 Present and Future Activity

At the moment, we are busy simply with completing all proposed circuits and — probably — by making the current set consistent.

The current set of circuits falls short of asynchronous verification examples. Moreover protocol verification problems are not covered, which may also be viewed at as important subaspects of circuit verification. To cover lower description levels, we also would like to add some switch-level or transistor level verification examples.

There is still a lack of “challenging” verification examples, i.e. circuits which are either of significant size or which reflect “real” commercial designs. As one of these circuits, we will probably provide a large RISC-processor: the DLX of Patterson and Hennesy [HePa90].

5 Acknowledgments

We like to thank the following people, who have helped in creating this benchmark set and documentation:

• Ralf Reetz	for significant ideas and contributions like the parameterized base module library,
• Hans-Peter Eich	for implementing all circuits in a commercial design system and for providing all the figures and timing diagrams,
• Claus-Jürgen Thomas	for updating the circuits and making them synthesizable
• Klaus Schneider	for pointing out interesting verification problems,
• Jørgen Staunstrup,	for supporting this activity via IFIP and suggesting the IFIP circuits and
• the Authors of TPTP	for the excellent documentation of their theorem proving benchmarks [SuSY93], which helped us in structuring this document

6 Literature

[SuSY93] C. Suttner, G. Sutcliff, and T. Yemenis. *The TPTP (Thousands of Problems for Theorem Provers) Problem Library*. TU Muenchen, Germany and James Cook University, Australia, via anonymous ftp flop.informatik.tu-muenchen.de, tptpv1.0.0 edition, 1993.

[BrBK89] F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. In *International Symposium on Circuits and Systems*, May 1989.

[BrPH85] F. Brglez, P. Pownall, and R. Hum. Accelerating ATPG and fault grading via testability analysis. In *International Symposium on Circuits and Systems*, 1985.

[Fram92] CADENCE Design Framework II version 4.2a. Reference Manual, February 1992.

[Pail85] J.-L. Paillet. Un Modele de Fonctions Sequentielles pour la Verification Formelle de Systemes Digitaux. Technical Report 546, IMAG-ARTEMIS, Grenoble, June 1985.

[Stau93] J. Staunstrup. IFIP WG 10.2 collection of circuit verification examples, November 1993.

[HePa90] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A quantitative Approach*. Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, 1990.

[FMVD90] L. Claesen, editor. *International Workshop on Applied Formal Methods for VLSI Design*, Leuven, Belgium, 1990.

[VRMK91] R. Vemuri, J. Roy, P. Mamtora, and N. Kumar. Benchmarks for high-level synthesis. Technical Report ECE-DDE-91-11, Laboratory for Digital Design Environments, ECE Dept., University of Cincinnati, Ohio, USA, November 1991.

7 Single Pulser

Despite its small size, this circuit has shown to be hard to specify in many formalisms like temporal logics and is well-suited as an introductory example.

7.1 Introduction

A Single Pulser is a clocked-sequential device with a one-bit input, *I*, and a one-bit output *O*. The purpose of the circuit is described as follows [WiPr80]:

> We have a debounced push-button, on (true) in the down position, off (false) in the up position. Devise a circuit to sense the depression the button and assert an output signal for one clock pulse. The system should not allow additional assertions of the output until after the operator has released the button.

A design specification of the Single Pulser can be found in the text book “The Art of Digital Design” by D. Winkel and F. Prosser [WiPr80]. A detailed treatment of this example for comparing different specification formalisms can be found in [JoMC94].

7.2 Specification

Assuming that the input is synchronous and debounced, the specification may be stated as:

> For each input pulse on *I*, the Single Pulser issues exactly one pulse of unit duration on *O* regardless of the duration of *I*.

The specification may be also stated by the following three properties [JoMC94]:

1. Whenever there is a rising edge at *I*, *O* becomes true some time later.
2. Whenever *O* is true it becomes false in the next time instance and it remains false at least until the next rising edge on *I*.
3. Whenever there is a rising edge, and assuming that the output pulse doesn’t happen immediately, there are no more rising edges until that pulse happens (There can’t be two rising edges on *I* without a pulse on *O* between them).

In [JoMC94] the specification is given in different formalisms like PVS and CTL [ORSS94, ClEm81].

7.3 Implementation

The implementation is taken from [WiPr80]. The incoming, not yet debounced asynchronous signal **`Pulse_In`** is fed to a D flip_flop and thus becomes the synchronized signal **`Pulse_sync`**, which is then delayed for one clock cycle by using another D flip-flop. Its output is negated, and the AND-connection of the synchronous pulse with its own delay generates the resulting, one clock-cycle lasting signal **`Pulse_Out`** (Fig. 7-1).

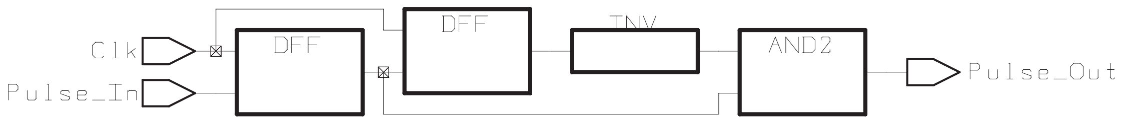


Figure 7-1: Single Pulser

In Fig. 7-2 waveforms are given to illustrate the behavior of the circuit.

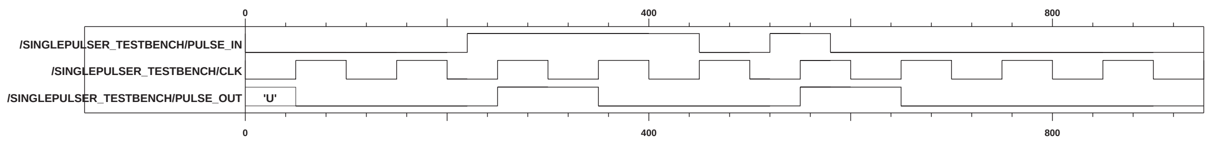


Figure 7-2: Example waveform

7.4 Status and Acknowledgments

The circuit has been originally proposed by J. Staunstrup. Most of the actual text is directly quoted from [Stau93]. Thanks to C.-J. Thomas for creating the VHDL description.

7.5 Literature

[ClEm81] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.

[JoMC94] S.D. Johnson, P.S. Miner, and A. Camilleri. Studies of the single pulser in various reasoning systems. In T. Kropf and R. Kumar, editors, *Proc. 2nd International Conference on Theorem Provers in Circuit Design (TPCD94)*, volume 901 of *Lecture Notes in Computer Science*, pages 126–145, Bad Herrenalb, Germany, September 1994. Springer-Verlag. published 1995.

[ORSS94] S. Owre, J.M. Rushby, N. Shankar, and M.K. Srivas. A tutorial on using PVS for hardware verification. In T. Kropf and R. Kumar, editors, *Proc. 2nd International Conference on Theorem Provers in Circuit Design (TPCD94)*, volume 901 of *Lecture Notes in Computer Science*, pages 258–279, Bad Herrenalb, Germany, September 1994. Springer-Verlag. published 1995.

[Stau93] J. Staunstrup. IFIP WG 10.2 Collection of Circuit Verification Examples, November 1993.

[WiPr80] D. Winkel and F. Prosser. *The Art of Digital Design*. Prentice-Hall Inc., 1980.

8 Traffic Light Controller

The traffic light controller is one of the most famous benchmark circuits also in the area of circuit synthesis. It is a good example for a pure controller circuit for which e.g. safety properties have to be verified.

8.1 Specification

Consider a circuit controlling a simple traffic light placed at the intersection of two roads called NS (North South) and EW (East West). Sensors make it possible to detect whether cars are waiting, and the light is supposed to change in the direction of waiting cars. The sensor detecting waiting cars on NS is called **CarOnNS** and the sensor detecting waiting cars on EW is called **CarOnEW**

There are many ways to design such a traffic light controller; however, it is an indispensable requirement that the light is safe, i.e., that it is *always* red in one of the two directions. So it must be required that

$$(\text{NSLight} = \text{red}) \vee (\text{EWLight} = \text{red}) \qquad \textbf{(8-1)}$$

This is a simplified version of the trafficlight discussed in several introductory textbooks on VLSI design

8.2 Implementation

currently missing

8.3 Status and Acknowledgments

An implementation will be provided after the actual benchmark description guidelines have been revised (probably end of 1994).

The circuit has been originally proposed by J. Staunstrup. Most of the actual text is directly quoted from [Stau93].

8.4 Literature

[Stau93] J. Staunstrup. IFIP WG 10.2 collection of circuit verification examples, November 1993.

9 *N*-bit Adder

This example is “the” verification benchmark circuit. It is useful for demonstrating data abstraction (natural numbers versus bit vectors) and for demonstrating the verification of generic circuits, i.e. circuits with arbitrary bit width.

9.1 Introduction

The circuit computes the sum of two natural number, given as bit vectors of width n. This is probably the most frequently used example in the literature, however, there is a wide variety of restrictions imposed, for example, limitations on n. Is the verification done for arbitrary n? If s is also n-bit what happens at overflow? Furthermore, a variety of realizations are possible ranging from a simple ripple-carry adder to advanced carry lookahead adders.

9.2 Specification

Verify that a realization of an *N*-bit adder computes the sum, *s*, of two n-bit numbers *a* and *b*.

More formally, it is to be shown that

$$\Phi([a_{n-1}, \ldots, a_0]) + \Phi([b_{n-1}, \ldots, b_0]) = \Phi(cout, [s_{n-1}, \ldots, s_0]) \qquad (9\text{-}1)$$

$$\text{with } \Phi([x_{n-1}, \ldots, x_0]) = \sum_{i=0}^{n-1} 2^i \cdot a_i \qquad (9\text{-}2)$$

9.3 Implementation

The N-bit adder was designed scalable with step width four. Its four bit components represent carry lookahead structure as described in [Schm78] (Fig. 9-1). It consists of four one bit adders with carry generate and propagate: G_Out = A_In AND B_In P_Out = A_In OR B_In S_Out = C_In XOR (P_Out AND NOT G_Out).

Signals S_Out(i), P_Out(i), G_Out(i) and carry(0) are fed to the carry lookahead generator (Fig. 9-2), which contains a combinational logic to generate B_Propagate, B_Generate, B-Carry and CarryOut(i). The block carry propagate and generate signals may be used to build a real carry lookahead adder with a cascade structure, which is not that easy to build generic.

By connecting the four bit adding units with ripple carry, we now get the complete adder with its inputs DataIn_A, DataIn_B, sum output DataOut and CarryOut (Fig. 9-3). The figure shows an example with 24 bit.

An example simulation is shown in Fig. 9-4.

9.4 Status and Acknowledgments

The circuit has been originally proposed by J. Staunstrup. Most of the actual text is directly quoted from [Stau93]. Thanks to C.-J. Thomas for generating the VHDL descriptions.

9.5 Literature

[Schm78] V. Schmidt. Digitalschaltungen mit Mikroprozessoren. B.G. Teubner Verlag, Stuttgart, 1979. (in german).

[Stau93] J. Staunstrup. IFIP WG 10.2 collection of circuit verification examples, November 1993.

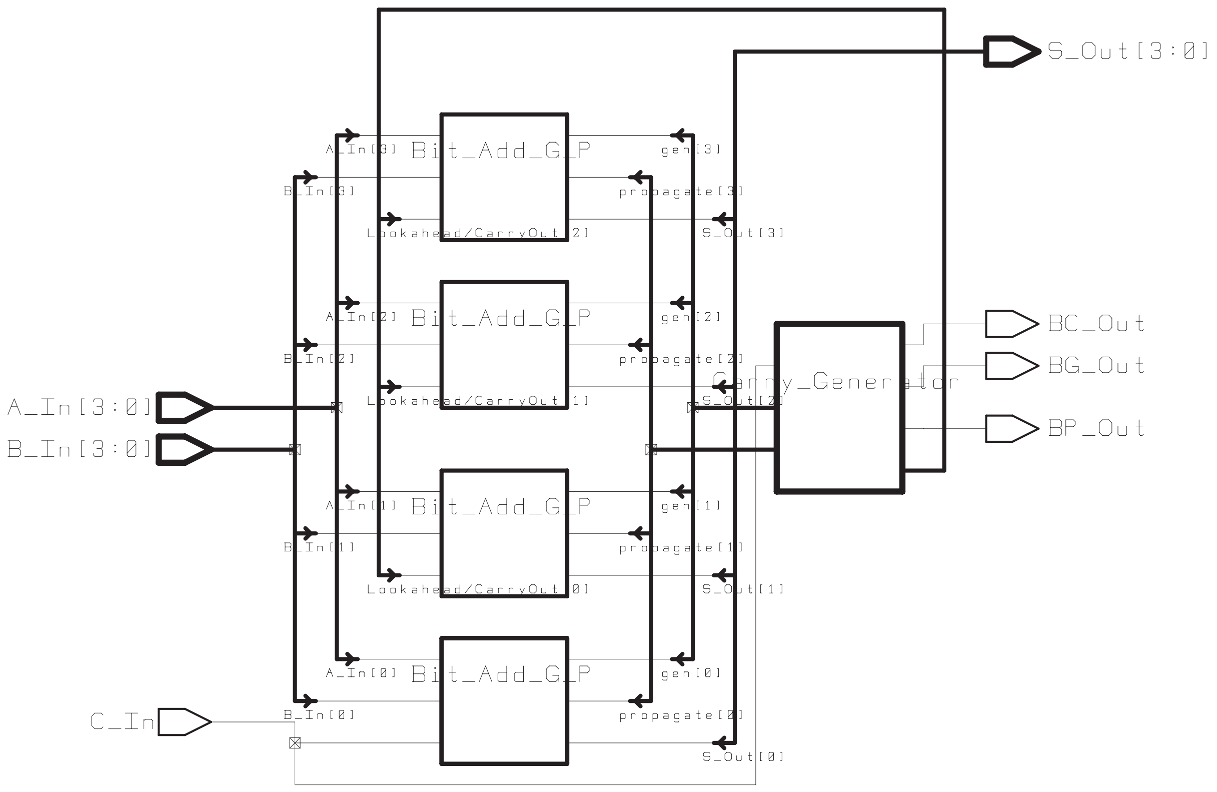


Figure 9-1: 4 bit Base Adder

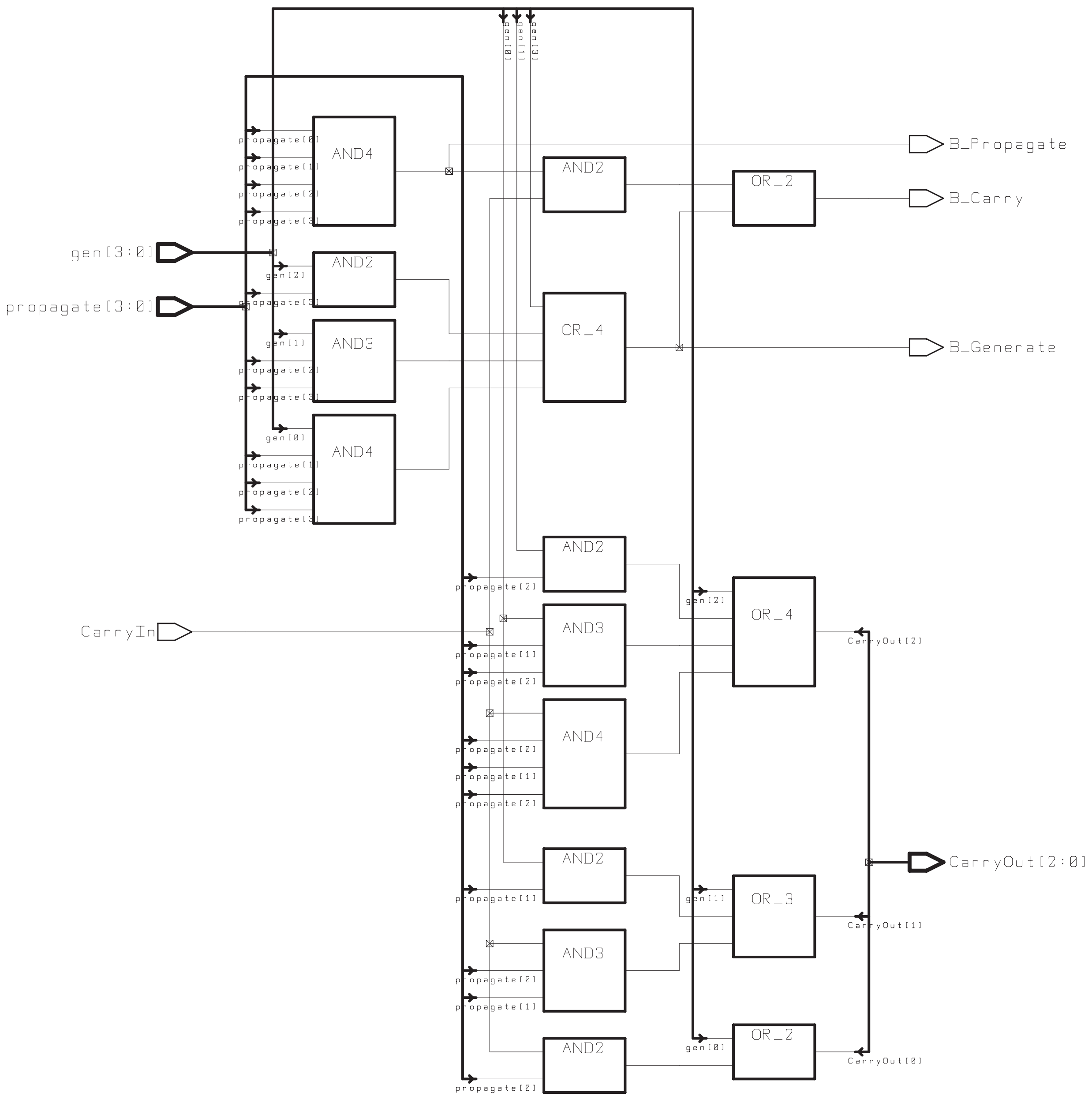


Figure 9-2: Carry Generator

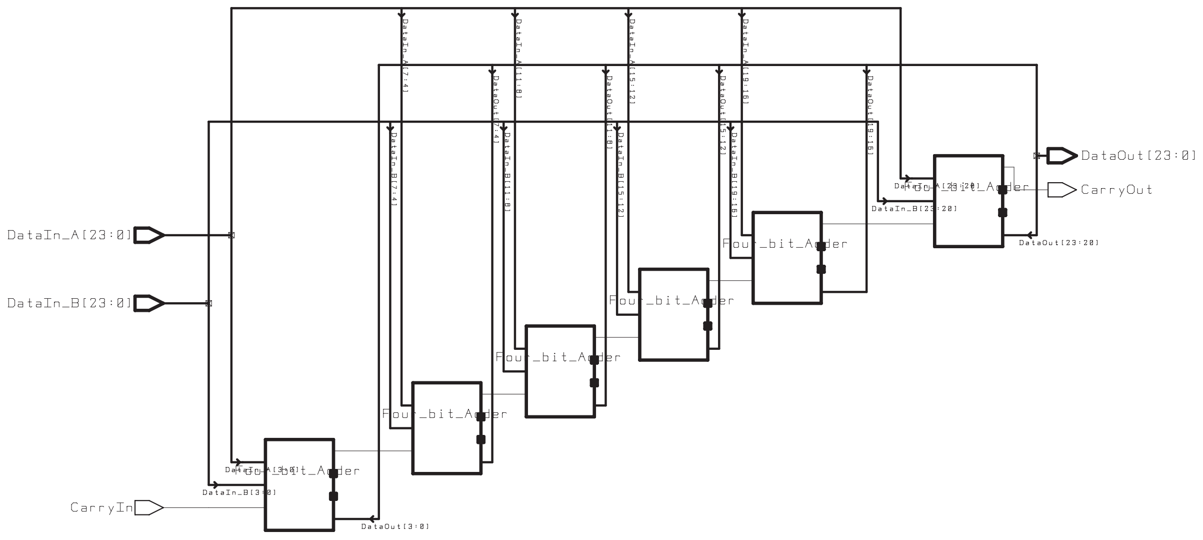


Figure 9-3: 24 Bit Adder

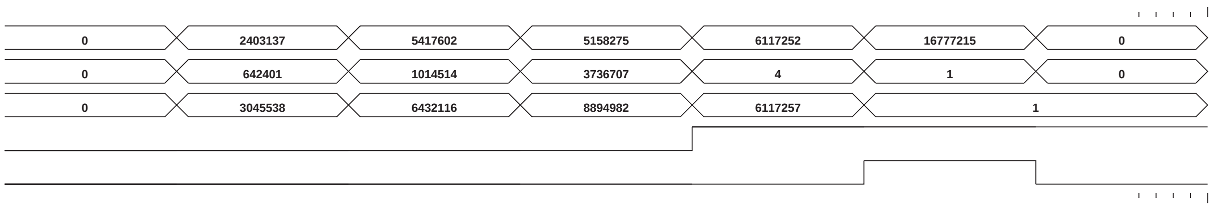


Figure 9-4: Example Simulation

10 Min_Max Circuit

10.1 Introduction

This benchmark circuit has been proposed by Luc Claesen for [FMVD90]. It is the first non-trivial example which has gained some popularity and reveals some problems, arising in the area of digital signal processor verification.

10.2 Specification

The Min_Max unit has an input signal *in* which consists of a sequence of integers in the range of -256 to +255. The Min_Max unit has three boolean control signals *clear, reset*, and *enable*. The unit produces an output sequence *out* at the same rate as *in* in the following way:

- *out* is zero if *clear* is true, independent of the other control signals.
- if *clear* is false and *enable* is false then *out* equals the last value of *in* before *enable* became false.
- if *clear* is false and *enable* is true and *reset* is true then *out* follows *in*.
- if *reset* becomes false, then *out* equals, on each time point *t*, the mean value of the maximum and minimum value of *in* until that time point. So

$$out = \frac{max(in) + min(in)}{2} \tag{10-1}$$

A number of properties of Min_Max have (deliberately) not been specified, e.g. the latency of the system is unspecified. This example was used at the international workshop on “Applied Formal Methods For Correct VLSI Design” and several solutions can be found in the proceedings [FMVD90].

10.3 Implementation

Fig. 10-1 gives an overview of the implementation of the MinMax-Circuit. A storage element Last (Fig. 10-2) may store a new value while Enable is true, otherwise the former value is stored.

The heart of the circuit is the MeanValue device (Fig. 10-3). Here the incoming values are compared with the stored minimum and maximum. The used comparator is a modified version of the nCMPN from GateLib which is able to compare numbers in 2-complemented format. If the new value is greater than the stored maximum or less than the stored minimum, its value is stored in the corresponding branch of the circuit. Otherwise, its value is discarded. The stored minimum and maximum now are added and the result is divided by two, i.e. one shift right. MeanOut contains the mean value of the stored minimum and maximum at each time point, regardless of any control signals. If reset becomes true, the stored maximum and minimum are set to -256 and +255 as soon as Clk becomes true.

To generate the required output signals depending on the control signals, the unit Three_to_four generates 4 condition signals out of the three incoming control signals (Fig. 10-4):

- Condition_1 := Clear : DataOut = ‘0’
- Condition_2 := ~Clear & ~Enable : DataOut = last_out
- Condition_3 := Reset & (Enable & ~Clear) : DataOut = DataIn

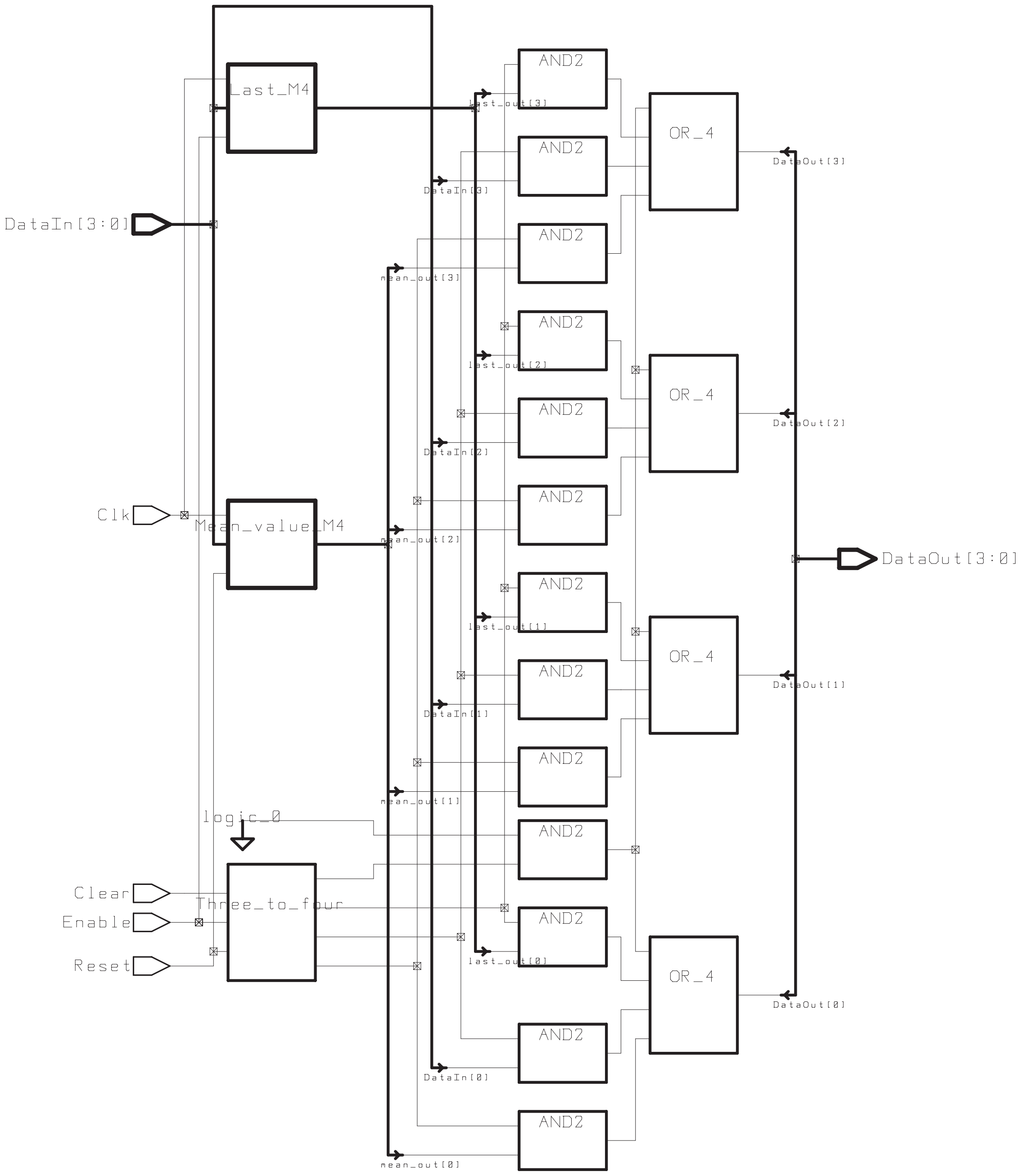


Figure 10-1: MinMax Structure

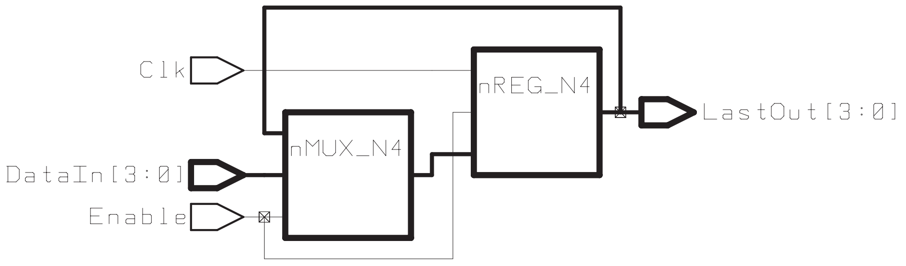


Figure 10-2: LastValue

- Condition_4 := ~Reset & (Enable & ~Clear) : DataOut = mean_out

The condition signals are the and-connected with the corresponding signals to be or-ed for output.

Fig. 10-5 shows different example simulation runs.

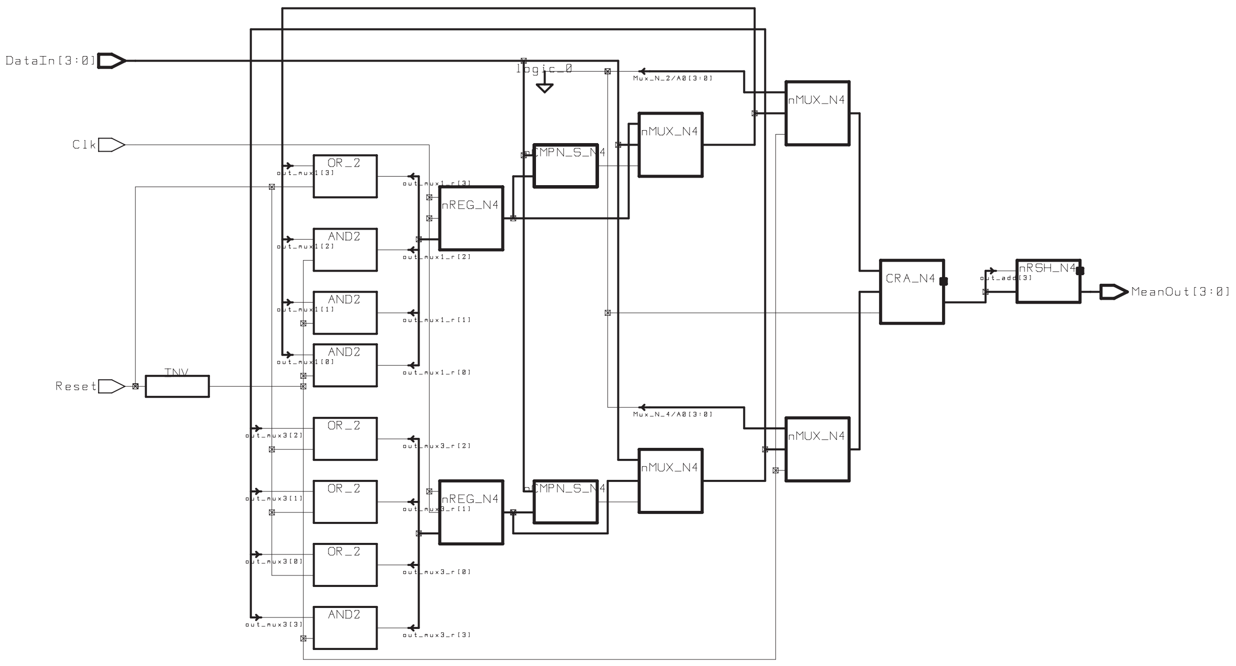


Figure 10-3: MeanValue

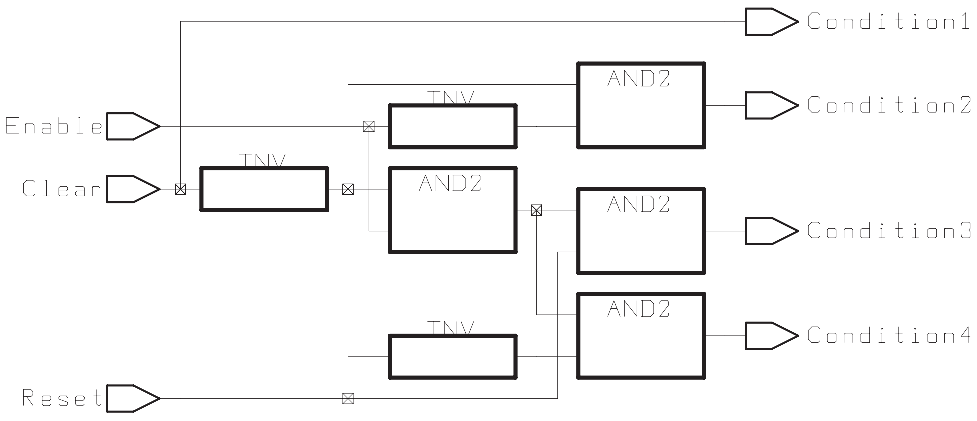


Figure 10-4: 3to4 “glue logic”

10.4 Status and Acknowledgments

Thanks to L. Claesen and J. Staunstrup for providing this example. Thanks to G. Janssen for pointing out the problems of the original specification.

10.4.1 Comments to the Specification of MIN_MAX

The specification given in section 10.2 is not as unambiguous as is seems at the first glace. To point this out, some comments from G. Janssen from Eindhoven University are added below.

- If *clear* is false and *enable* is false then *out* equals the last value of *in* before ENABLE became false.

1. Is this independent of RESET?

2. What if *clear* is false and *enable* is false from the start?

Solution:

1. Yes, we don’t care about *reset*.

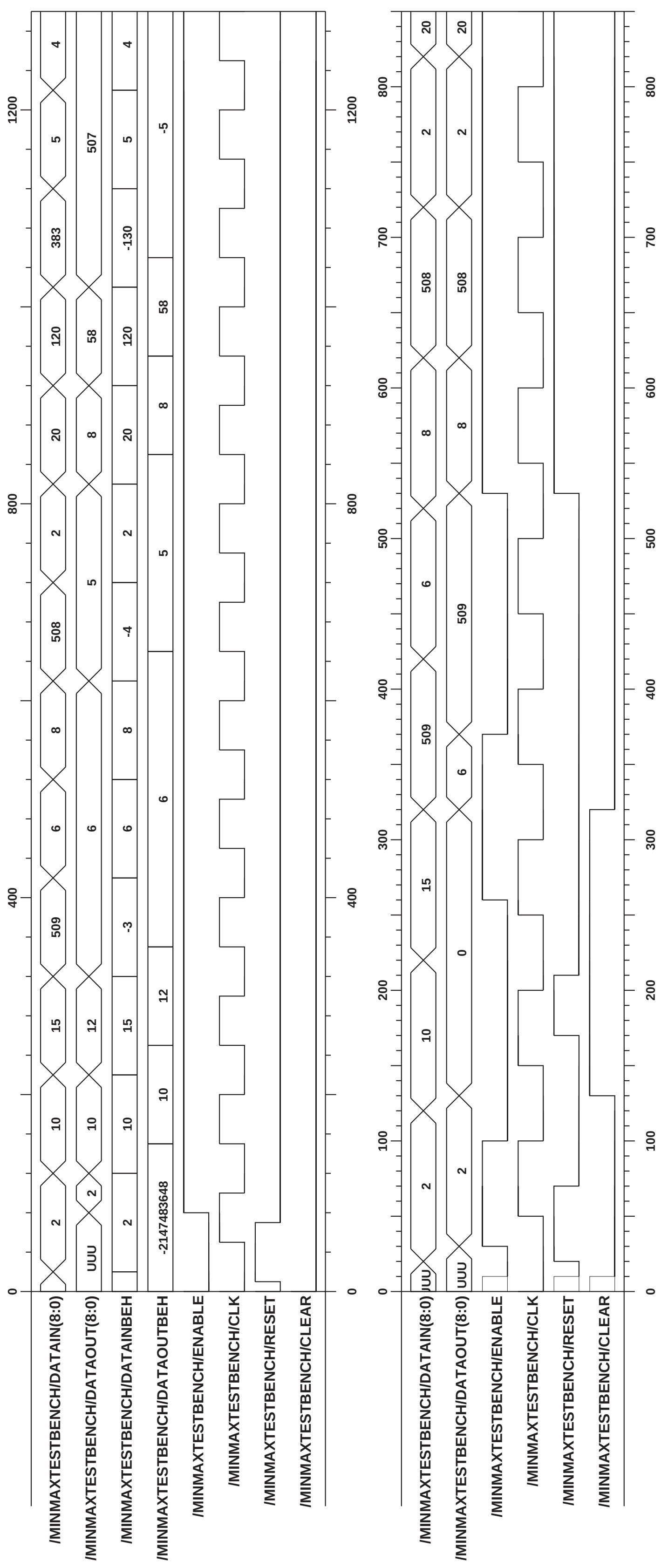


Figure 10-5: Example waveforms

2. Then *out* takes some value in the range -255 to 1 and keeps that value till some other control combination takes over.

- If *clear* is false and *enable* is true and *reset* is true then *out* follows *in*.

1. What does “follow” precisely mean?

Solution:

1. “follow” means equals.

- If *reset* becomes false, then *out* equals, on each time point *t*, the mean value of the maximum and minimum value of *in* until that time point.

1. What must be the values of *clear* and *enable*?
2. Does “until” include or exclude the current time point t?
3. What if *reset* is false from the start?
4. When are the minimum and maximum values (re)initialised and to what value? Where does the averaging starts?
5. What if *clear* and *enable* change during *reset* is false?

Solution:

1. *clear* must be false and *enable* must be true.
2. “until” is interpreted to mean inclusive.
3. Then *min* and *max* are set to *in*.
4. Only when *reset* makes a transition from true to false are *min* and *max* reinitialised; At the time *reset* is false, *min* and *max* are set to *in*; averaging starts then afresh.

10.5 Literature

[FMVD90] L. Claesen, editor. *International Workshop on Applied Formal methods for VLSI Design*, Leuven, Belgium, 1990.

[Stau93] J. Staunstrup. IFIP WG 10.2 collection of circuit verification examples, November 1993.

11 Black-Jack Dealer

11.1 Specification

A Black-Jack dealer is a device which plays the dealer's hand of a card game. Its inputs are *go* (true/false) and *card* (Ace of Spades,...,2 of clubs). Its outputs are *hitme*, *stand*, and *broke* (all truth-valued).

The *go*/*hitme* signals are used for a 4-cycle handshake with the operator. Cards are valued from 2 to 10, and aces may be valued as either 1 or 11 by choice of the player. The Black-Jack dealer is repeatedly presented with cards. It must assert *stand* when its accumulated score reaches 16; and it must assert *broke* when its score exceeds 21. In either case the next card starts a new game.

A design specification of the Black-Jack dealer can be found in the text book "The Art of Digital Design" by D. Winkel and F. Prosser [WiPr80].

11.2 Implementation

The implementation of the black jack dealer has been taken from [WiPr80]. From the abstract description of an FSM controlling the datapath, the following realization has been derived (Fig. 11-1). The FSM has been realized by encoding the states with two flipflops A and B and generating the next state by choosing the right signal with two 4:1 multiplexers (table 11-1).

State	Next State	B	A	Condition
0 Get	Get	0	0	nGet_2
	Add	0	1	Get_2
1 Add	Use	1	0	Acecard & nAce11flag
	Test	1	1	not(Acecard & nAce11flag)
2 Use	Test	1	1	true
3 Test	Get	0	0	nTest_3
	Test	1	1	Test_3

Table 11-1: State transition table

The flipflop output signals are fed to a combinational logic to generate the controller output signals as described in the equations from table 11-2 and table 11-3.:

Get_1	=	S_Get & nCard_r_s
Get_2	=	S_Get & Card_r_s & nCard_r_d
Get_3	=	Get_2 & (Stand v Broke)
Test_1	=	S_Test & ScoreGT16 & nScoreGT21
Test_2	=	S_Test & ScoreGT16 & ScoreGT21 & nAce11flag = S_Test & ScoreGT21 & nAce11flag
Test_3	=	S_Test & ScoreGT16 & ScoreGT21 & Ace11flag = S_Test & ScoreGT21 & Ace11flag

Table 11-2: Internal signals

Hit	=	Get_1
Set_Stand	=	Test_1
Clr_Stand	=	Get_2
Set_Broke	=	Test_2
Clr_Broke	=	Get_2

Table 11-3: Output signals

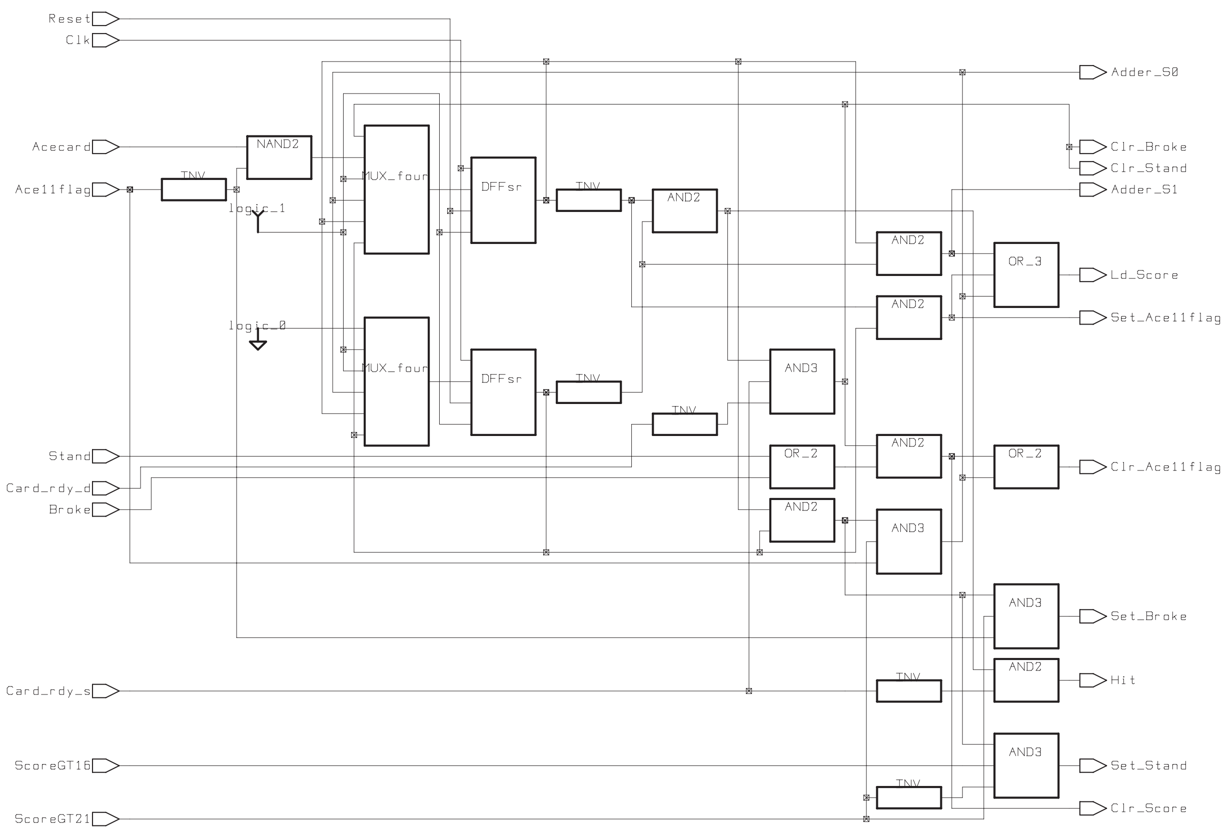


Figure 11-1: Controller of the Black-Jack Dealer

Set_Ace11flag	=	S_Use
Clr_Ace11flag	=	Get_3 v Test_3
Ld_Score	=	S_Add v S_Use v Test_3
Clr_Score	=	Get_3
AdderS0	=	S_Add
AdderS1	=	Test_3

Table 11-3: Output signals

The second part of the dealer is BlackJack_DataPath (Fig. 11-2), including several flipflops to hold the status information for the outside world, hit, stand, broke, to debounce the input signal from a card ready button and to hold the ace11flag. The circuit uses the card_value input signal to discriminate the value of an ace card from others, indicating this by setting the signal ace-card = true. card_value is then expanded by one bit and fed to a 4:1 multiplexer, which has to choose between D'+10', D'-10' and the actual card value according to table 11-4.

Mux input	AdderS1	AdderS0	action
0	0	0	+10
1	0	1	-10
2	1	0	value
3	1	1	-

Table 11-4: Data path function

The multiplexer is followed by a 5-bit carry ripple adder, adding the actual score internal_score and the output of the multiplexer. The sum internal_sum is then fed to a 5-bit register with load and clear, which is controlled by the signals Ld_Score and Clr_Score, to produce the score.

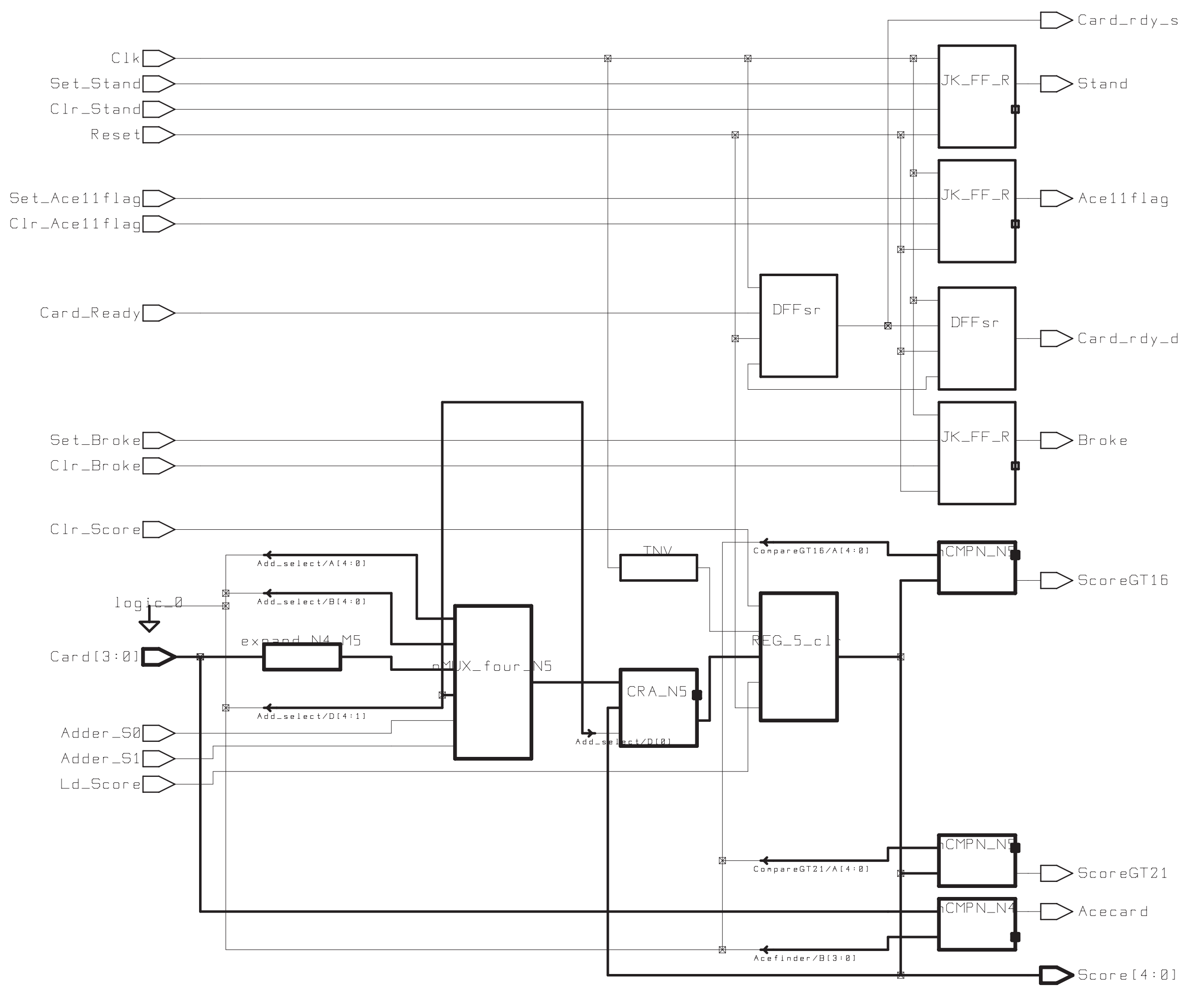


Figure 11-2: Datapath

The score is then compared with D’16’ and D’21’ to generate the signals ScoreGT16 and ScoreGT21. Both units exchange control signals as viewed in Fig. 11-3.

The waveform diagram in Fig. 11-4 shows the exchange of control signals between the units. The signals test(0) to test(3) contain the state signals S_Get, S_Use, S_Add, S_Test. The signal test_dp(4:0) contains the internal sum before it is taken over by the register.

The waveform diagram in Fig. 11-5 shows the behavior of the black jack dealer as a black box. Moreover, the behavior of a algorithmical VHDL description has been added.

11.3 Status and Acknowledgments

Thanks for C.-J. Thomas for providing an implementation description.

The circuit has been originally proposed by J. Staunstrup. Most of the specification has been directly quoted from [Stau93].

11.4 Literature

[WiPr80] D. Winkel and F. Prosser. *The Art of Digital Design*. Prentice-Hall Inc., 1980.

[Stau93] J. Staunstrup. IFIP WG 10.2 collection of circuit verification examples, November 1993.

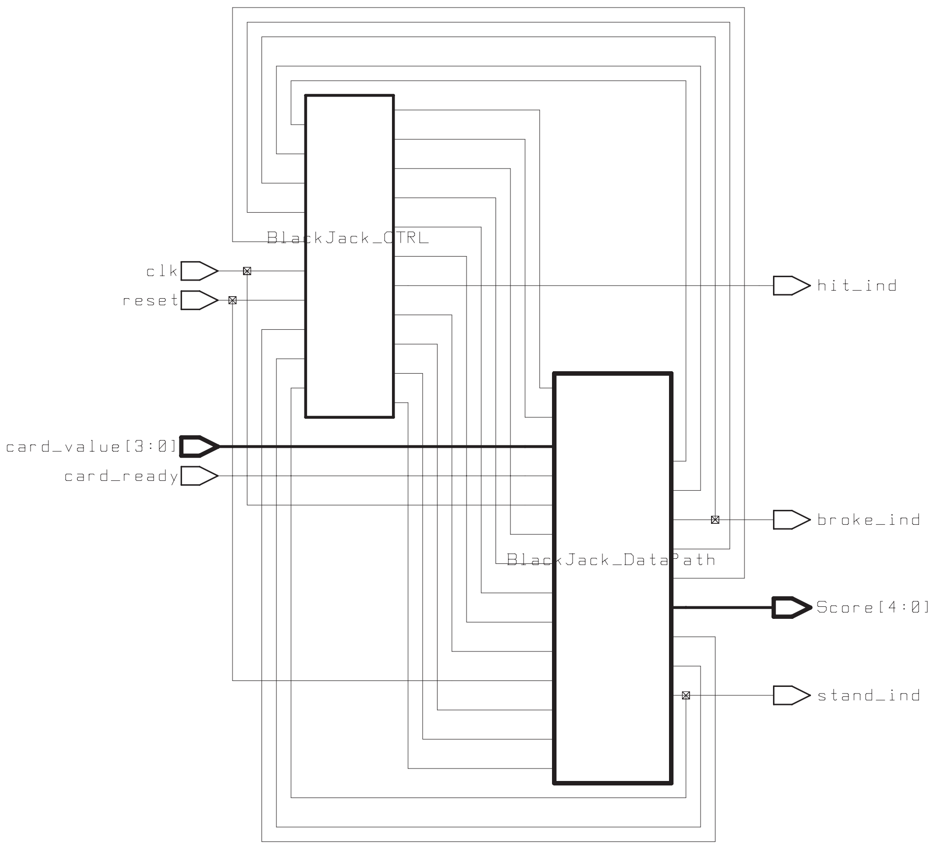


Figure 11-3: Control - data path communication

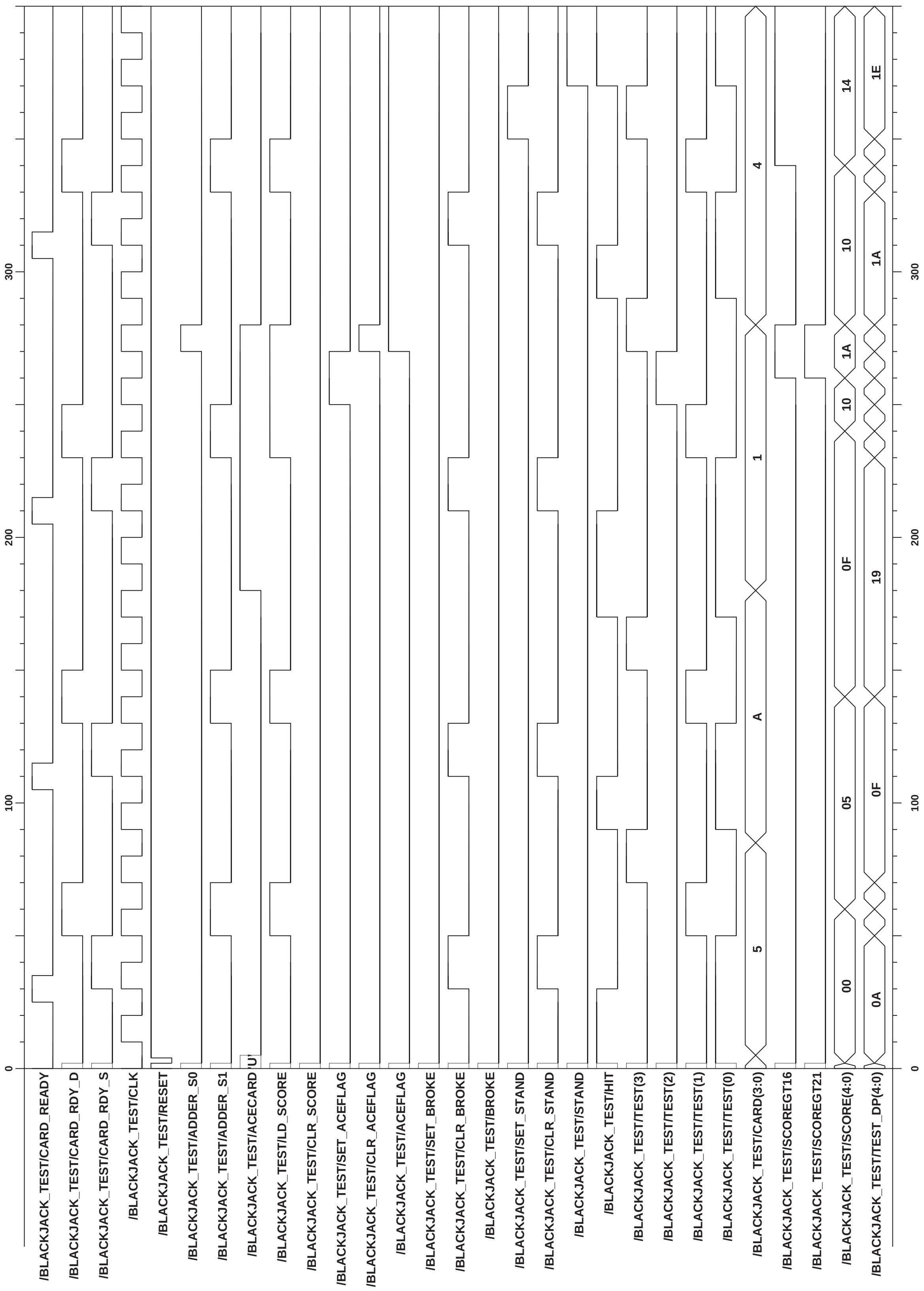


Figure 11-4: Waveform diagram of internal signals

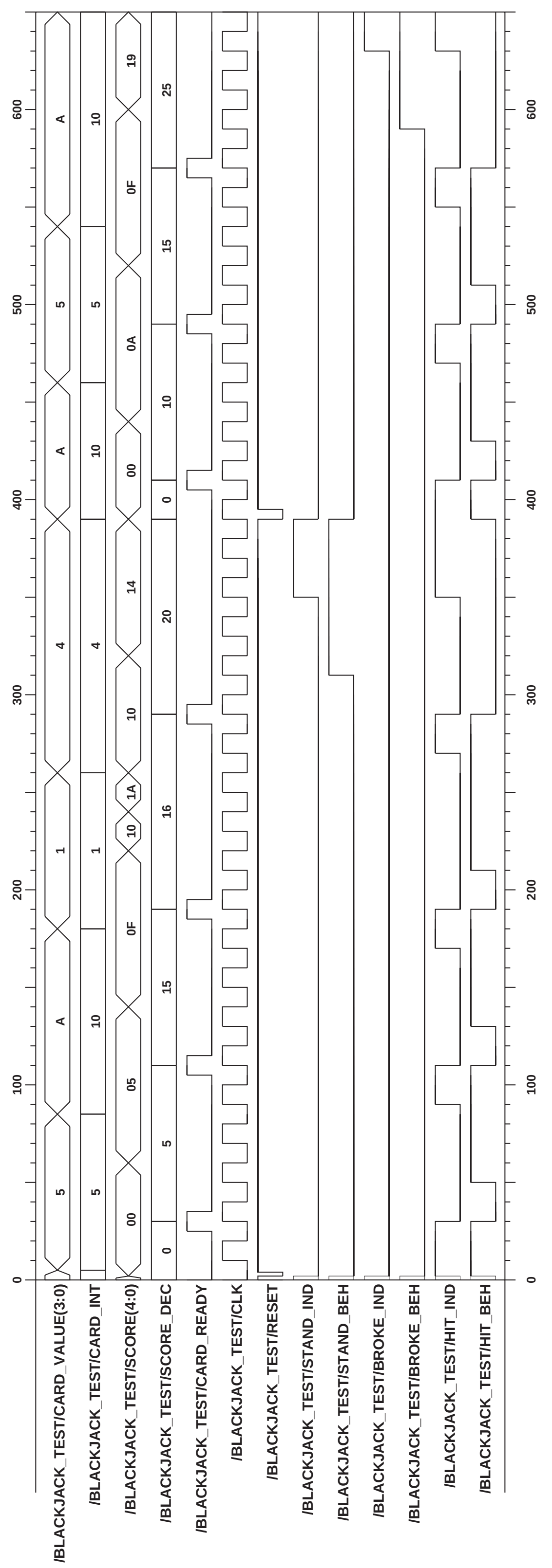


Figure 11-5: Blackbox simulation of the circuit

12 Arbiter

This arbiter is a good example for a synchronous scalable state machine.

12.1 Introduction

The purpose of the bus arbiter is to grant access on each clock cycle to a single client among a number of clients contending for the use of a bus (or another resource). The inputs to the circuit are a set of request signals $req_0, \ldots, req_{k-1}$ and the outputs are a set of acknowledge signals $ack_0, \ldots, ack_{k-1}$ (Fig. 12-1). Normally the arbiter asserts the acknowledge signal of the requesting client with the lowest index. However, as requests become more frequent, the arbiter is designed to fall back on a round robin scheme, so that every requester is eventually acknowledged. This is done by circulating a token in a ring of arbiter cells, with one cell per client. The token moves once every clock cycle. If a given client's request persists for the time it takes for the token to make a complete circuit, that client is granted immediate access to the bus.

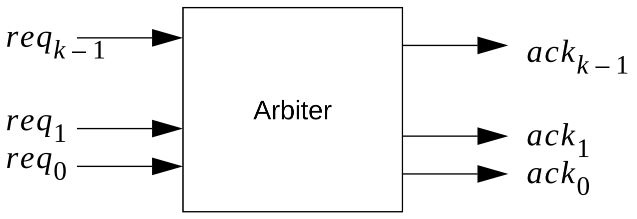


Figure 12-1: Black box view of the arbiter

12.2 Specification

The desired properties to be verified are:

1. No two acknowledge outputs are asserted simultaneously.
2. Every persistent request is eventually acknowledged.
3. Acknowledge is not asserted without request.

By restricting *N* to be two (or some other small constant), the problem becomes simple yet illustrative. Ultimately, it should be possible to verify a design where *N* is a parameter, i.e. do the verification for an arbitrary *N*.

12.2.1 Formal specification example

Data path free control circuits may be described easily using propositional temporal logics like CTL [ClEm81]. The above properties result in the following CTL expressions:

1. $\bigwedge_{i \neq j} \mathrm{AG}\neg(ack_i \wedge ack_j)$
2. $\bigwedge_i \mathrm{AGAF}(req_i \rightarrow ack_i)$
3. $\bigwedge_i \mathrm{AG}(ack_i \rightarrow req_i)$

12.3 Implementation

The basic cell of the arbiter is shown in Fig. 12-2. This cell is repeated k times, as shown in Fig. 12-3 for $k = 4$. Each cell has a request input and an acknowledge output. The grant of cell i is passed to cell $i + 1$, and indicates that no client of index less than or equal to i are requesting. Hence a cell may assert its acknowledge output if its grant input is asserted. Each cell has a register T which stores a one when the token is present. The T registers form a circular shift register which shifts up one place each clock cycle. Each cell also has a register W (for “waiting”) which is set to one when the request input is asserted and the token is present. The register remains set while the request persists, until the token returns. At this time, the cell’s override and acknowledge outputs are asserted. The override signal propagates through the cells below, negating the grant input of cell 0, and thus preventing any other cells from acknowledging at the same time.

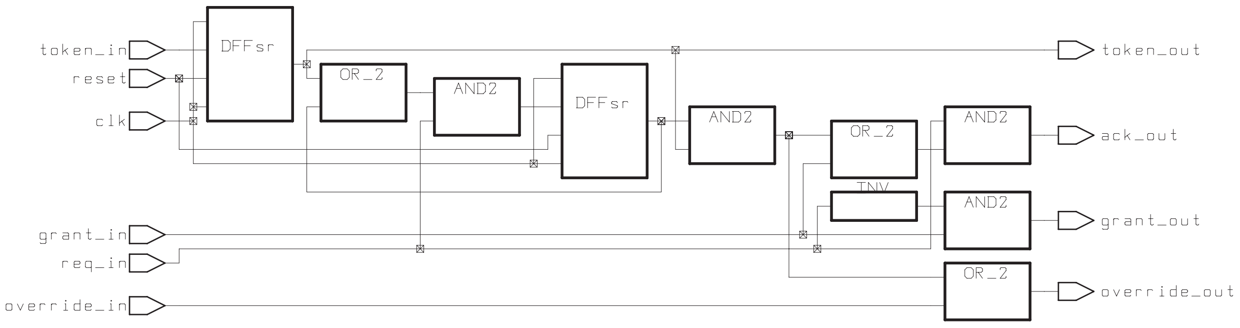


Figure 12-2: Cell_2_plus

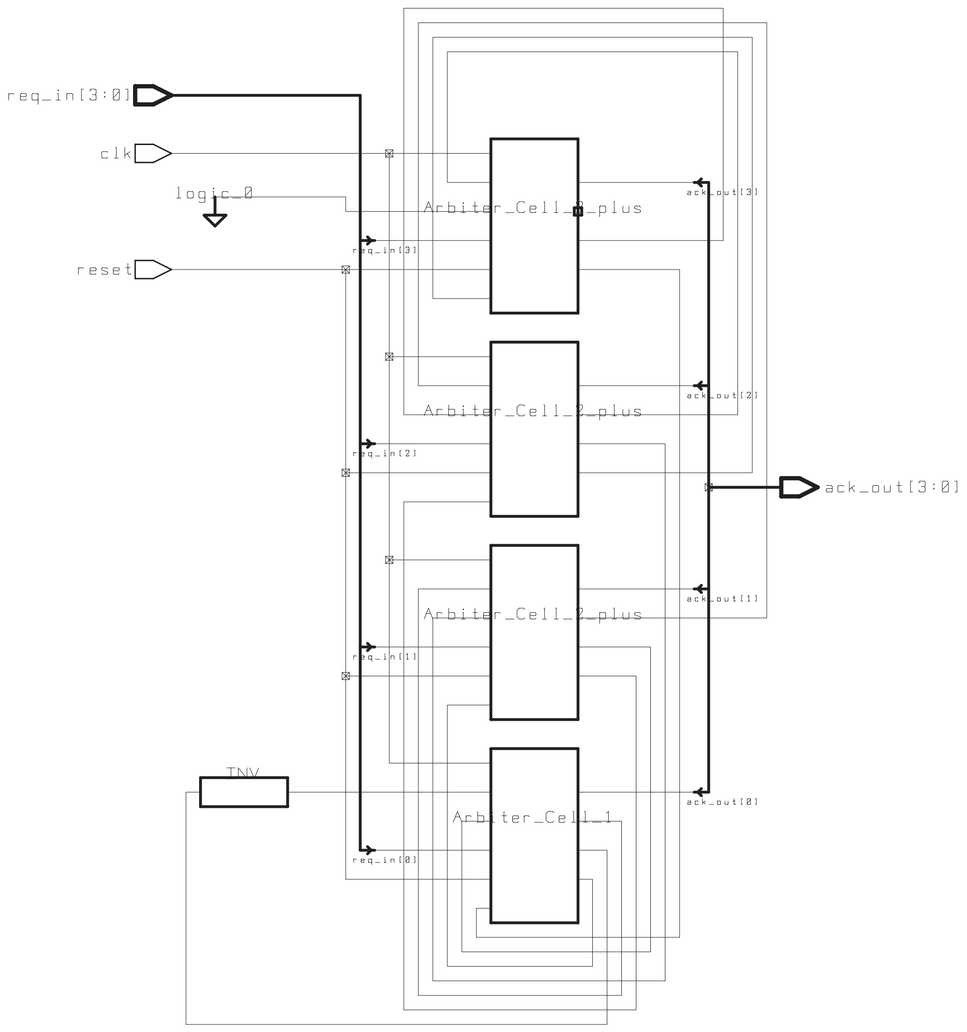


Figure 12-3: Arbiter with four inputs/outputs

The circuit is initialized so that all of the W registers are reset and exactly one T register is set. To achieve this one cell has a different implementation as shown in Fig. 12-4.

Fig. 12-5 and Fig. 12-6 show different simulation runs with and without request collisions.

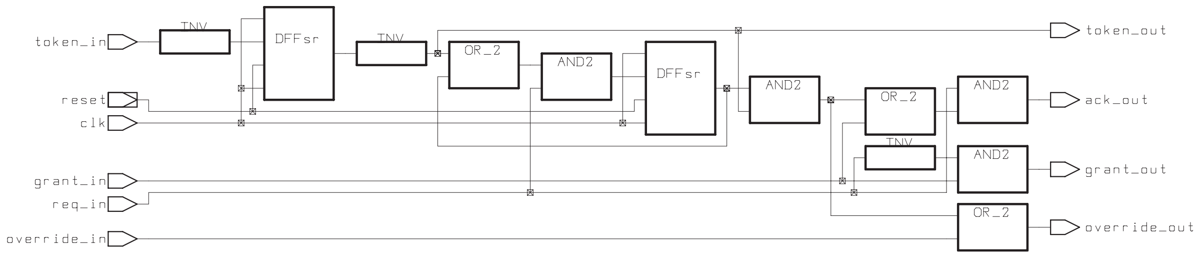


Figure 12-4: Cell_1

12.4 Status and Acknowledgments

The circuit has been originally proposed by J. Staunstrup [Stau93]. The implementation and formal specification has been taken from [McMi93a, p. 40ff.]. Parts of the description are directly quoted from there. Thanks to C.J. Thomas for providing the VHDL descriptions.

12.5 Literature

[ClEm81] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.

[McMi93a] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[Stau93] J. Staunstrup. IFIP WG 10.2 Collection of Circuit Verification Examples, November 1993.

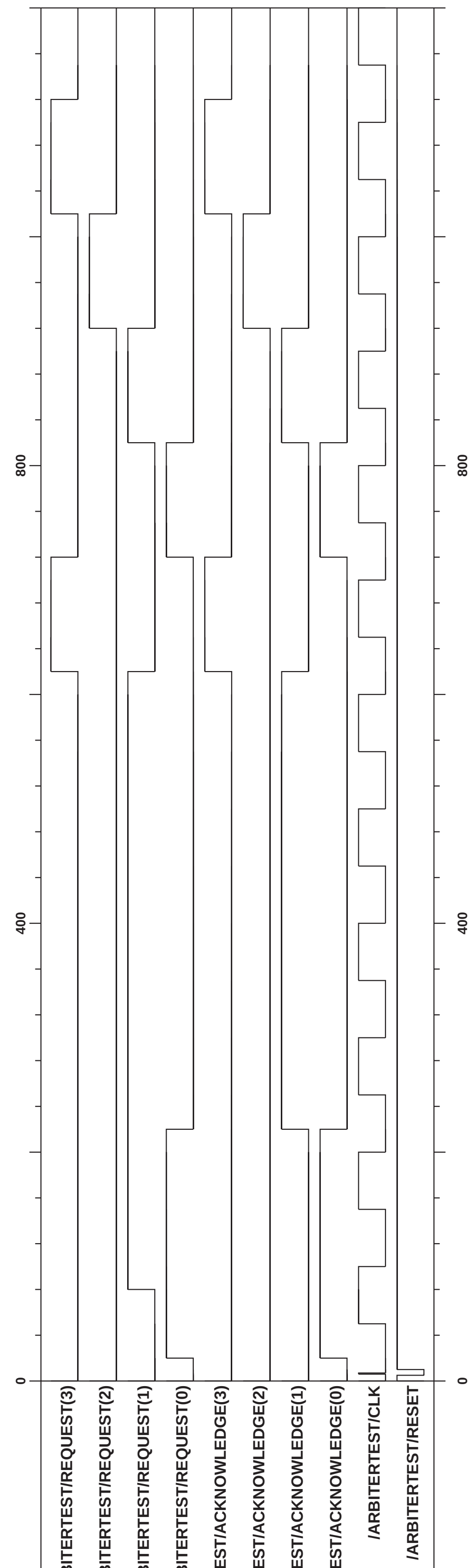


Figure 12-5: Example simulation with collision

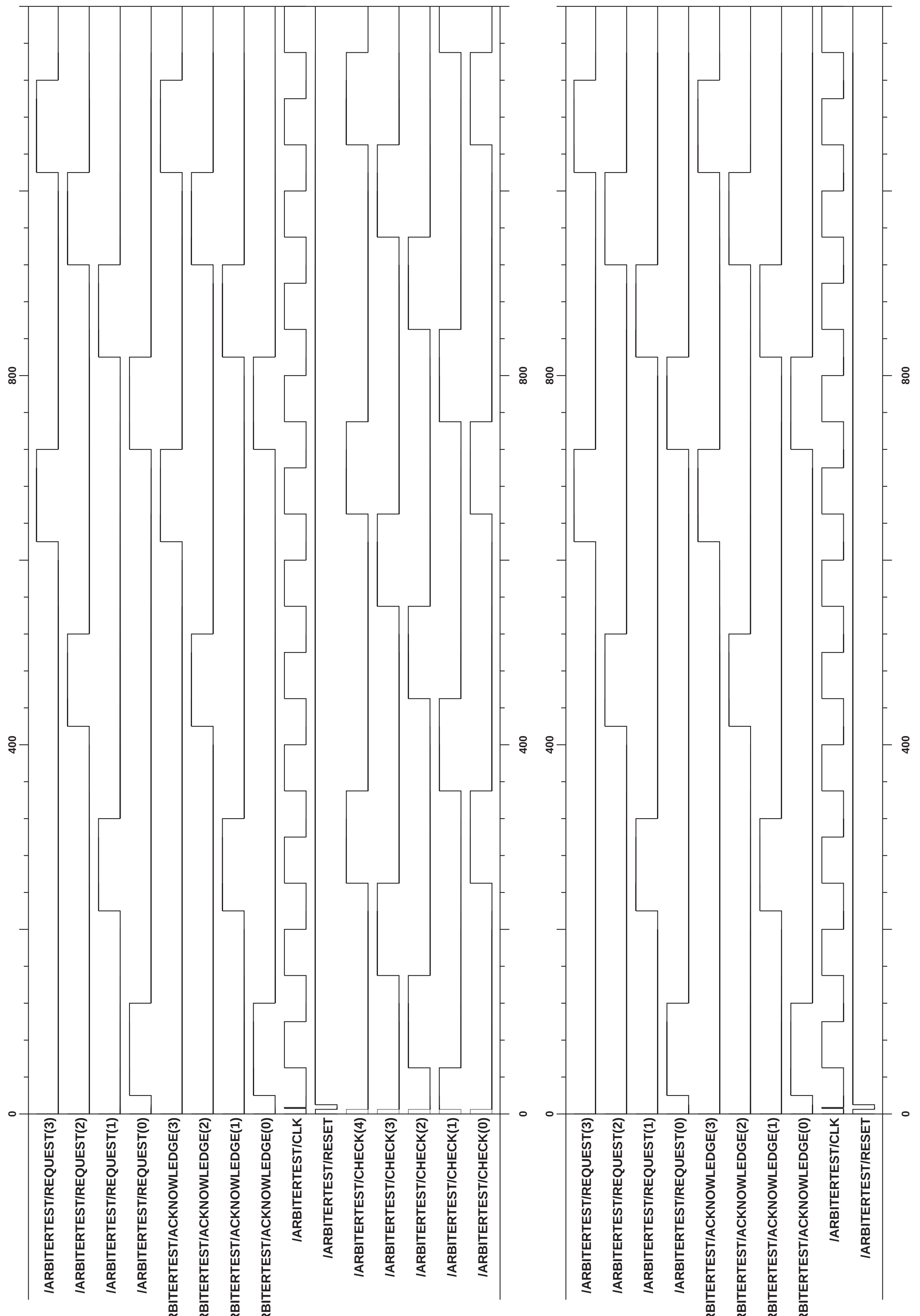


Figure 12-6: Example simulation without collision

13 Rollback Chip

This circuit is a special processor of non-trivial size.

13.1 Specification

The Rollback chip is a co-processor for speeding up a distributed simulation, the functionality is described in the paper [GoFu93]. This paper also presents several refinements and a formal verification of these.

13.2 Implementation

currently missing

13.3 Status and Acknowledgments

An implementation will be provided after the actual benchmark description guidelines have been revised (probably end of 1994).

The circuit has been originally proposed by J. Staunstrup. Most of the actual text is directly quoted from [Stau93].

13.4 Literature

[GoFu93] G. Gopalakrishnan and R. Fujimoto. Design and verification of the rollback chip using HOP: A case study of formal methods applied to hardware design. *ACM Transactions on Computer Systems*, 11(2):109–145, May 1993.

[Stau93] J. Staunstrup. IFIP WG 10.2 collection of circuit verification examples, November 1993.

14 The TAMARACK Processor

This circuit has been one of the first microprocessors, which have been formally verified. It may be used to demonstrate the problems, arising in the area of microprocessor verification.

14.1 Specification

Tamarack is a simplified microprocessor which has been specified and verified formally, this example is described in [Joyc88].

14.2 Implementation

currently missing

14.3 Status and Acknowledgments

An implementation will be provided after the actual benchmark description guidelines have been revised (probably end of 1994).

The circuit has been originally proposed by J. Staunstrup. Most of the actual text is directly quoted from [Stau93].

14.4 Literature

[Joyc88] J.J. Joyce. Formal verification and implementation of a microprocessor. In G. Birtwistle and P.A Subramanyam, editors, *VLDI Specification and Synthesis*, pages 129–157. Kluwer Academic Publishers, 1988.

[Stau93] J. Staunstrup. IFIP WG 10.2 collection of circuit verification examples, November 1993.

15 Stop-Watch

It has been pointed out by Mike Fourman that this example illustrates the need for temporal abstraction which might be treated differently than the data abstraction found in the arithmetic examples like the *N*-bit adder (section 9) [Four90].

15.1 Specification

Consider designing a digital stopwatch with a three digit, seven-segment display (to read tens of seconds, seconds and tenths of seconds), and two control buttons, “*reset*” and “*start/stop*”. When the *reset* button is pressed the display is cleared. The *start/stop* button is used to start and stop the clock. The design is driven by an 1MHz external clock signal. The stop watch is constructed as a synchronous design with one iteration in each clock cycle.

15.2 Implementation

The stopwatch basically consists of 3 synchronous, self starting counters (Fig. 15-1) which generate output data tenths_out, seconds_out, tens_out. For counting the tens of seconds, o modulo 6 counter is used, for counting seconds and tenths of seconds, modulo 10 counters are required. (Fig. 15-2, Fig. 15-3).

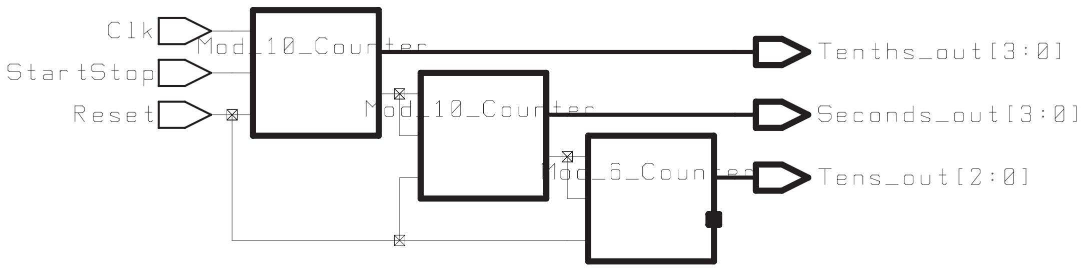


Figure 15-1: StopWatchCount

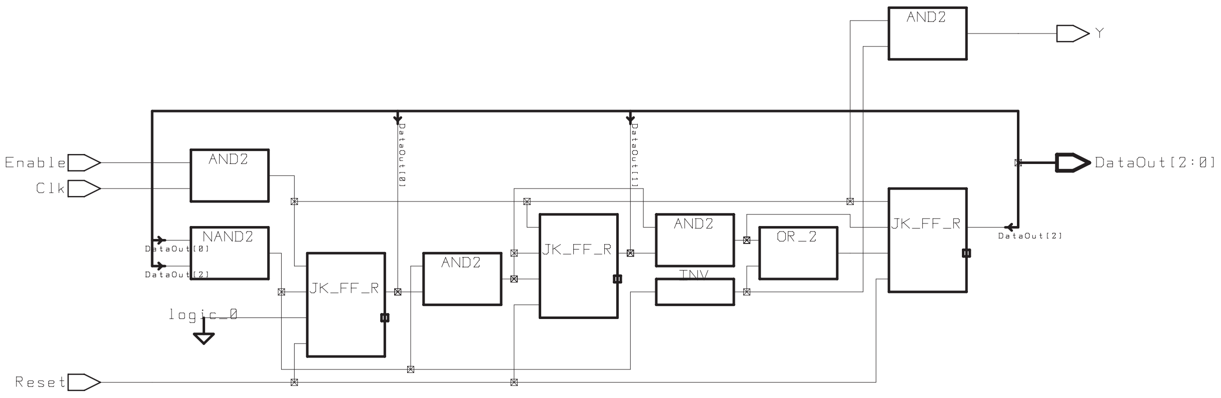


Figure 15-2: Mod_6_Counter

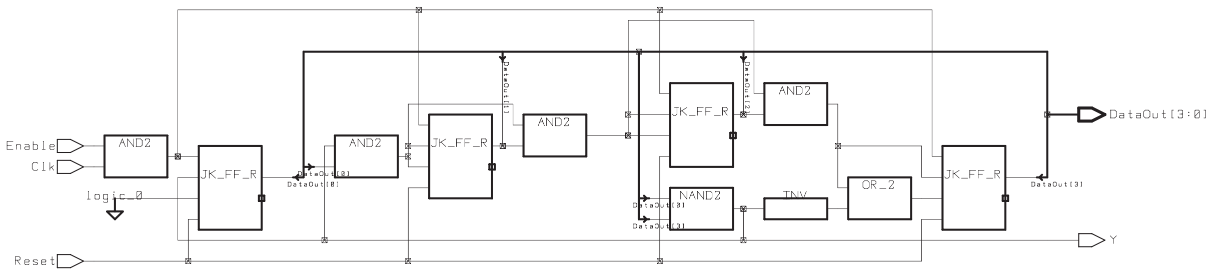


Figure 15-3: Mod_10_Counter

Because of the 1 Mhz clock input, there must be a clock reducing device, build of modulo 10 counters, to generate an internal clock signal (Fig. 15-4).

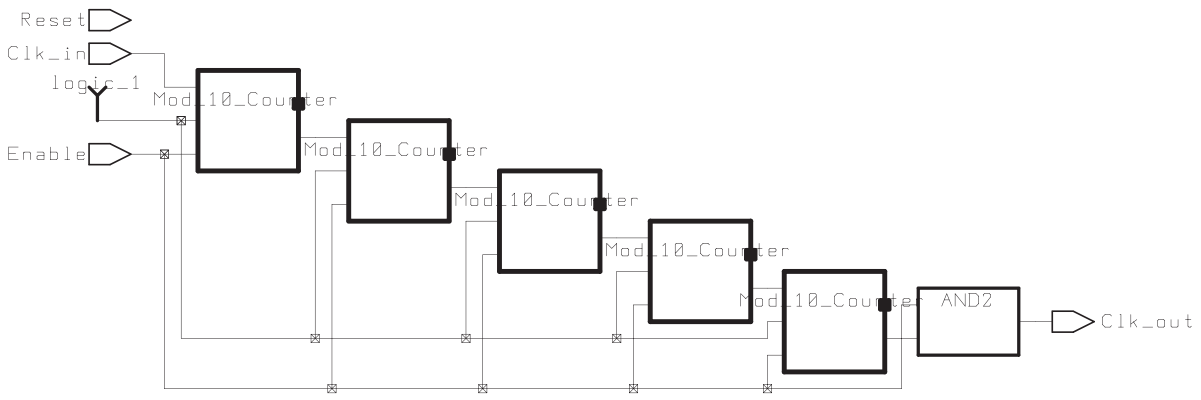


Figure 15-4: StopWatchReduce

In accordance to the specification, a combinational logic is provided to generate the output signals for a 7 segment display out of the data vectors (Fig. 15-5). The overall structure is shown in Fig. 15-6. Fig. 15-7 shows the waveform of the counting device in comparison with a behavioral description of the stopwatch.

15.3 Status and Acknowledgments

The circuit has been originally proposed by J. Staunstrup. Most of the actual text is directly quoted from [Stau93]. Thanks to C.-J. Thomas for generating the VHDL descriptions.

15.4 Literature

[Stau93] J. Staunstrup. IFIP WG 10.2 collection of circuit verification examples, November 1993.

[Four90] M. Fourman. Formal system design. In J. Staunstrup, editor, *Formal Methods for VLSI Design*, pages 191–236. North-Holland/Elsevier, 1990.

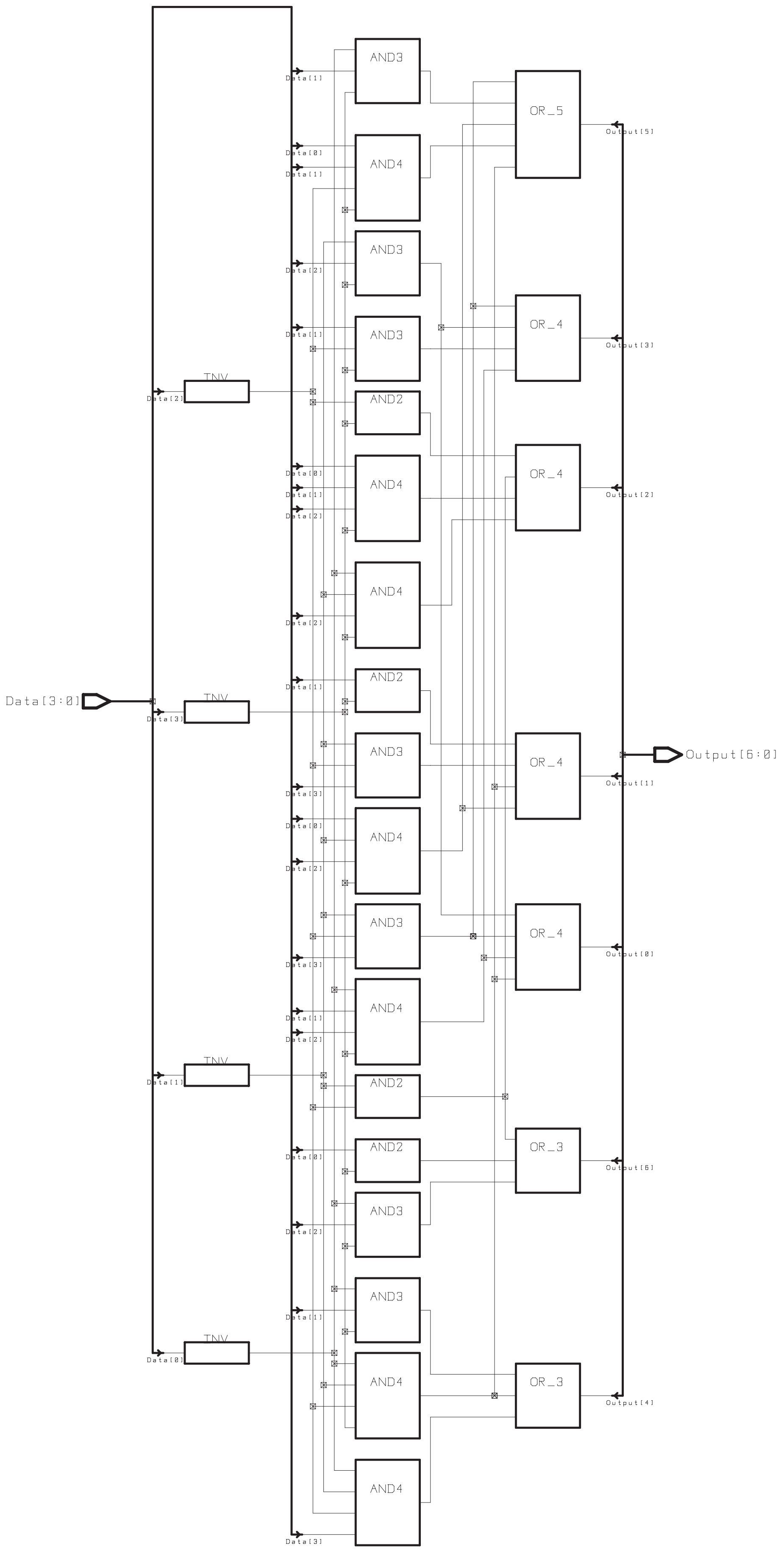


Figure 15-5: Unit for 7 Segment Display

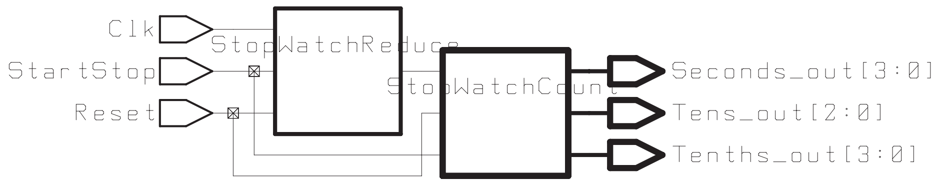
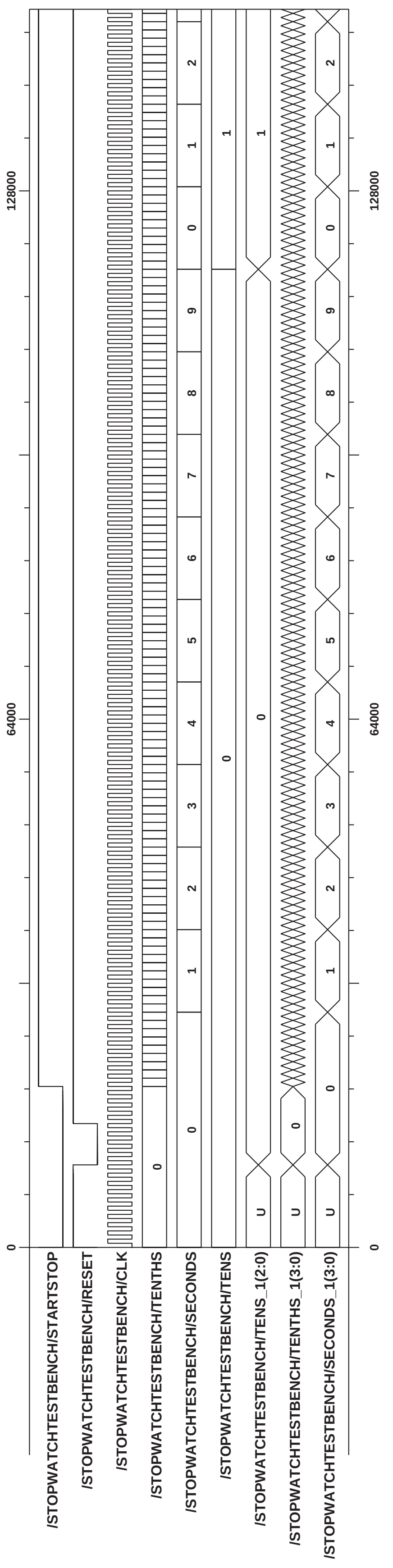


Figure 15-6: Overall Structure of Stop Watch



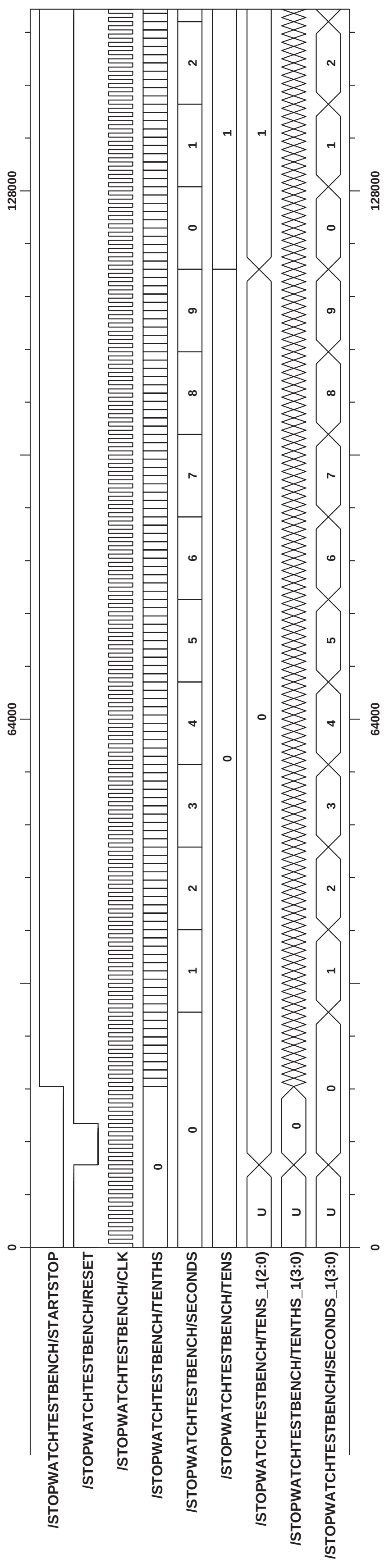


Figure 15-7: Example Waveforms

16 GCD - Greatest Common Divisor

Although being a small and worn out example, this circuit reveals some interesting verification problems: data dependent loops and data abstraction from bits and bit vectors to natural numbers. Moreover, if a more abstract specification is used (see Section 16.2.2 “High-Level Specification”) then even simple *program* verification tasks arise.

16.1 Introduction

The circuit simply computes the greatest common divider (GCD)of two natural numbers, given as bit vectors of fixed bit width.

16.2 Specification

In the following, two different specifications are given. The first “low level” version only requires an implementation verification, namely that the given algorithm to compute the GCD is correctly implemented. The second specification is based on the mathematical definition of the GCD and hence requires some sort of “program verification”, i.e. is has to be shown that the implemented algorithm really computes the GCD.

The circuit is based on a simple handshake protocol: It starts running after the signal **`start`** has been set to true, signalling that new input values are available at the inputs **`A`** and **`B`**. The latter carry binary encoded natural numbers. The **`stop`** signal indicates the termination of a computation.

16.2.1 Low-Level Specification

Given two natural numbers *A* and *B* (of fixed bit width). Compute the natural number *GCD* according to the algorithm of table 16-1.

1	*X1*:= Max(*A*,*B*);
2	*X2*:= Min(*A*,*B*);
3	**repeat**
4	*M*:= *X1* MOD *X2*;
5	**if** $(M \neq 0)$ **then**
6	**begin**
7	*X1*:= *X2*;
8	*X2*:= *M*;
9	**end**
10	**until** (*M*=0)
11	*GCD*:= *X2*

Table 16-1: Computation of the GCD of two numbers A and B

16.2.2 High-Level Specification

Given two natural numbers *A* and *B* (of fixed bit width). Compute the natural number *GCD* with the following property:

$$\mathrm{Max}(\{g|((g \in N)\text{and } g \text{ divides } A \text{ without rest and } G \text{ divides } B \text{ without rest})\})$$

16.3 Implementation

The implementation follows the specified algorithm (table 16-1). It is divided into control and data path. The data path computes (without controller interaction) the maximum of two n-bit vectors, to be interpreted as unsigned integers. Afterwards, the iteration of the algorithm is performed.

There are 4 signals between controller and data path. **StoreComp** activates the storage of the minimum/maximum sorted values in the data path. The sorting is performed by a comparator (Fig. 16-1).

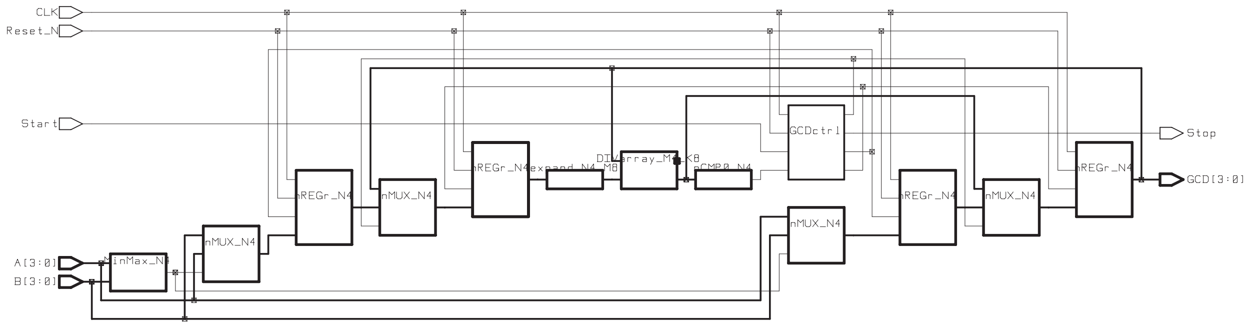


Figure 16-1: Controller and data path

SelectLoopInit determines, if there will be an additional loop computation or if new values are used for the loop. **StoreLoop** activates a register assignment, if the computation has not finished. The end of a computation is determined by **eq0**. All lines are active high.

16.3.1 Controller

The controller is specified by table 16-2. **Start** resets the controller, **Stop** signals the end of a computation. The FSM uses 3 states: 00 is the starting state after **Start** = 1. In state 01 the values of X1 and X2 are stored. Dependent on **eq0**, the FSM remains in 01 or jumps to the final state 10. States are stored in 2 D-flip-flops (q_0, q_1). State 11 is unreachable.

Start	eq0	state $(q_1q_0)^t$	next state $(q_1q_0)^{t+1}$	StoreComp	StoreLoop	SelectLoopInit	Stop
0	-[a]	00	01	0	1	0	0
0	0	01	01	-	1	0	0
0	1	01	10	-	0	0	1
0	-	10	10	-	0	0	1
1	-	--	00	1	1	1	0
0	-	11	10	-	0	0	1

Table 16-2: Controller transition table

a. "-" denotes don't cares

Using table 16-2, the transition functions of table 16-3 result.

$q_1^{t+1} = d_1^t =$	$(\mathbf{Start}\ \mathrm{q1}) \vee (\mathbf{Start}\ \mathbf{eq0}\ q_0)$
$q_0^{t+1} = d_0^t =$	$(\mathbf{Start}\ \overline{q_1}\overline{q_0}) \vee (\mathbf{Start}\ \mathbf{eq0}\ \overline{q_1})$
StoreComp =	**Start**

Table 16-3: FSM transition functions

StoreLoop =	**Start** $\vee d_0^t$
SelectLoopInit =	**Start**
Stop =	d_1^t

Table 16-3: FSM transition functions

An implementation of the controller is given in Fig. 16-2.

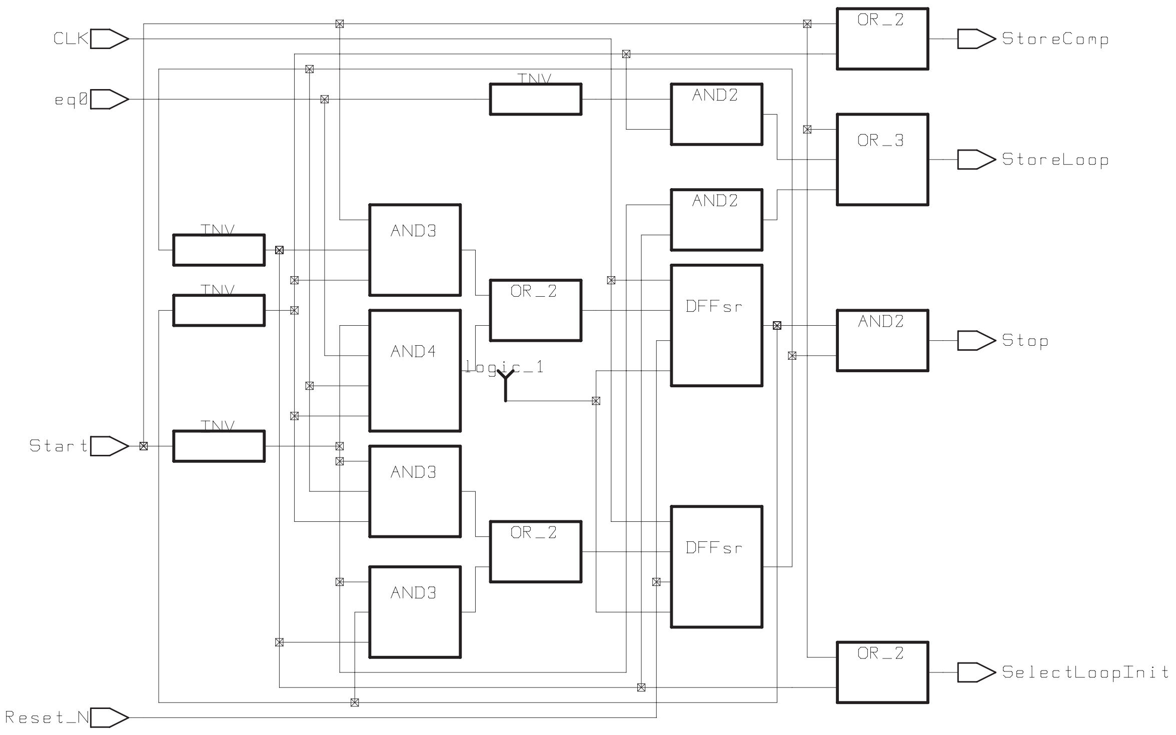


Figure 16-2: Controller realization

16.3.2 Data Path

The realization of the data path need the following modules: 4 n-bit multiplexer, 4 n-bit registers, 1 n-bit comparator, 1 n-bit divider and 1 n-bit “zero”-tester. These are connected as shown in (Fig. 16-1).

16.3.3 Simulations

In Fig. 16-3 two example computations are given. The GCD of 15_{10} (F_{16}) and 2 (result 1) and the GCD of 14_{10} (E_{16}) and 10_{10} (A_{16}) with result 2 are computed. Busses named GCD<3:0> and Div<7:0> carry divisor and dividend. Loop iterations occur on value transitions on these busses. The first computation needs 4 and the second computation needs 3 iterations.

To initialize the circuits, the Start and input signals have to be applied for a minimum duration of 2 clock cycles.

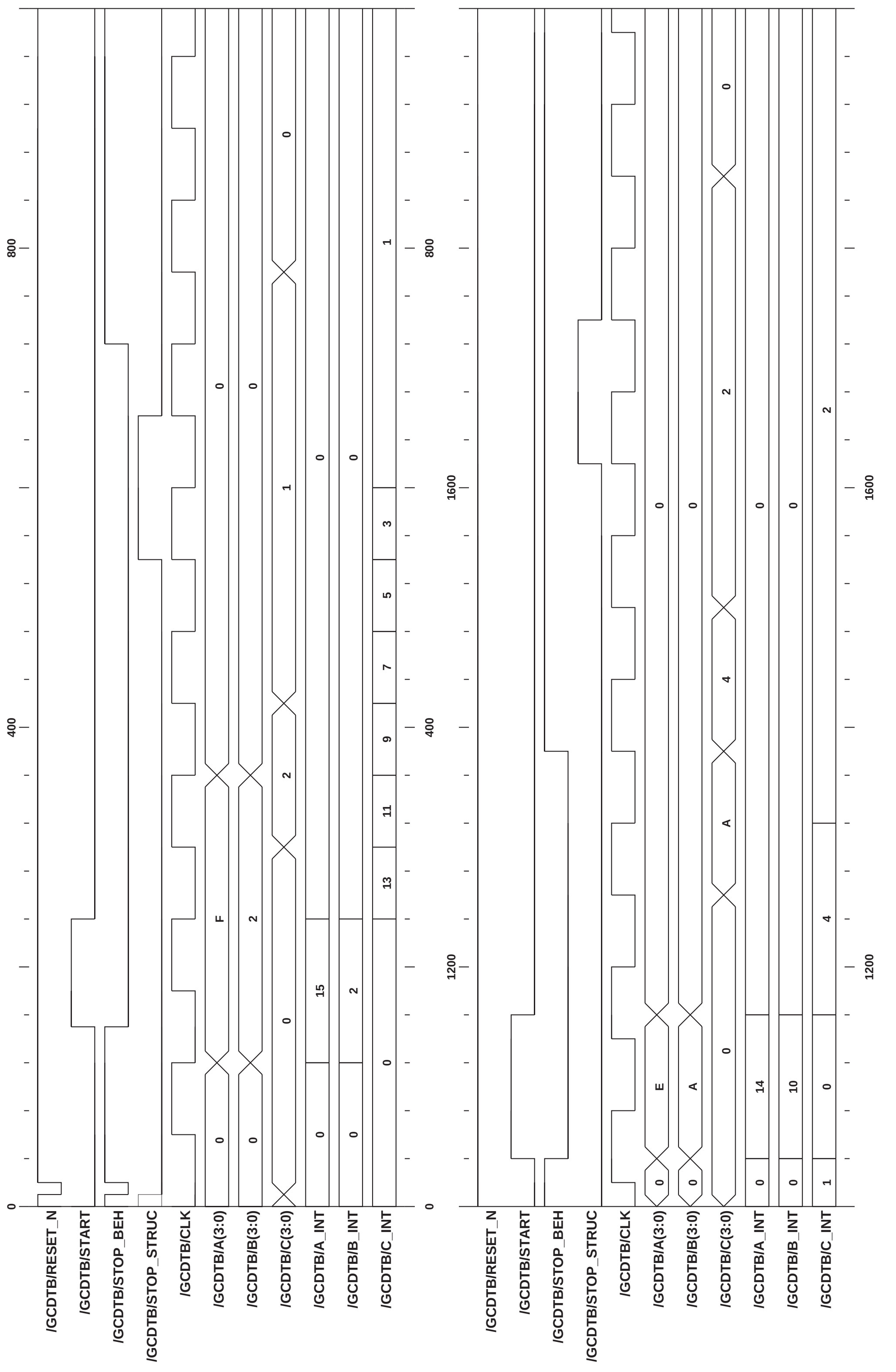


Figure 16-3: Waveforms for two GCD computations

16.4 Status and Acknowledgments

The GCD circuit is a variant of a high-level synthesis benchmark example [VRMK91]. The circuit presented here uses a ‘modulo” operation instead of repeated subtractions in order to put more emphasis on arithmetic. However, for uniformity reasons in a future release the algorithm may be adapted to the one used in [VRMK91].

Thanks to H.-P. Eich and C.-J. Thomas for designing the circuit in a commercial design system.

16.5 Literature

[VRMK91] R. Vemuri, J. Roy, P. Mamtora, and N. Kumar. Benchmarks for high-level synthesis. Technical Report ECE-DDE-91-11, Laboratory for Digital Design Environments, ECE Dept., University of Cincinnati, Ohio, USA, November 1991.

17 Multiplier

Although multipliers are usually purely combinational circuits, they are known to represent a class of hard to verify circuits. This is due to the high degree of dependability of the Boolean variables from each other (i.e. implementations have a high degree of connectivity between the gates). For this reason representations for Boolean functions like ROBDDs [Brya86] fail to provide efficient representations for multipliers, leading to problems when verifying multipliers of large bit width.

Moreover, the verification must perform a data abstraction, since the specification is given in terms of natural numbers and the implementation is based on bit vectors.

17.1 Specification

Given two natural numbers *A* and *B* (of fixed bit width). Compute the natural number *P* with the following property

$$P = A \cdot B \tag{17-1}$$

17.2 Implementation

17.2.1 Algorithm

The multiplication of two unsigned integers, represented as bit vectors $A = a_0 2^0 + a_1 2^1 + ... + a_{n-1} 2^{n-1}$ and $B = b_0 2^0 + b_1 2^1 + ... + b_{n-1} 2^{n-1}$ may be described by equation 17-2 and equation 17-3 in case of $n = 4$.

$$P = A \cdot B = \sum_{i=0}^{n-1} a_i 2^i B = \sum_{i=0}^{n-1} 2^i \left(\sum_{j=0}^{n-1} a_i b_j 2^j \right) \tag{17-2}$$

$$\begin{aligned} P = A \cdot B = {} & (a_0b_0)+2(a_1b_0+a_0b_1)+4(a_2b_0+a_1b_1+a_0b_2)+8(a_3b_0+a_2b_1+a_1b_2+a_0b_3) \\ & +16(a_3b_1+a_2b_2+a_1b_3)+32(a_3b_2+a_2b_3)+64(a_3b_3) \end{aligned} \tag{17-3}$$

17.2.2 Global Realization Architecture

Looking at equation 17-2, it is apparent, that the n^2 product terms (16 in equation 17-3) are all used only once. This leads to a hierarchical implementation, where first the product terms are computed, which are then fed into n^2 full adders. Finally, the resulting carry values are taken care of by a serial addition (Fig. 17-1).

In Fig. 17-1, 16 (n^2 with $n = 4$) multiplier base modules are used, which compute one product term and then perform an addition with carry. The module is specified in table 17-1 and realized as given in Fig. 17-1.

S =	$(ab) \oplus (c \oplus CIN)$
COUT =	$abc \vee abCIN \vee cCIN$
=	$((ab \wedge c) \wedge (ab \wedge CIN) \wedge \overline{(c \wedge CIN)})$

Table 17-1: Specification of the multiplier base module

17.2.3 Detailed Description

In this section, the implementation is described at a more detailed level.

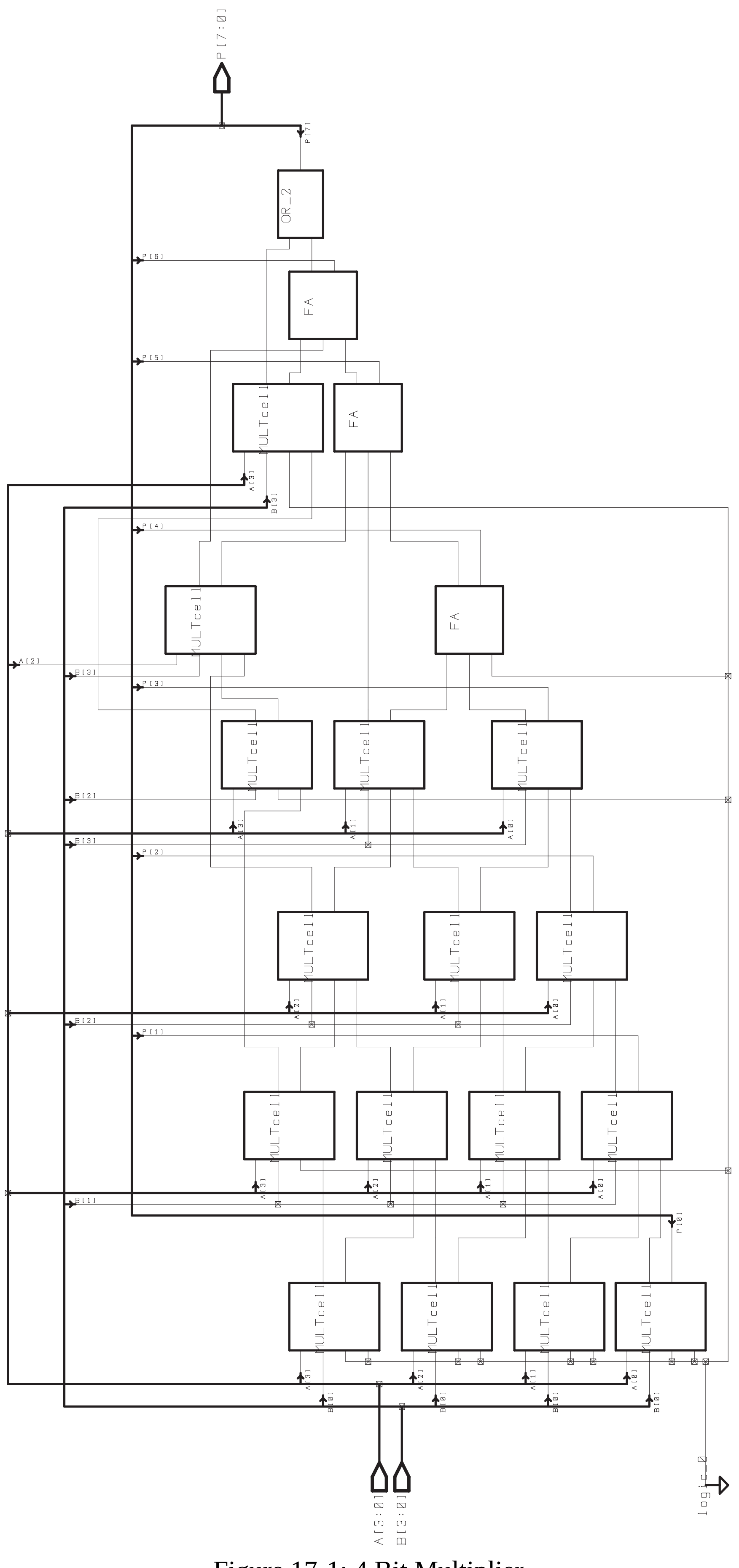


Figure 17-1: 4 Bit Multiplier

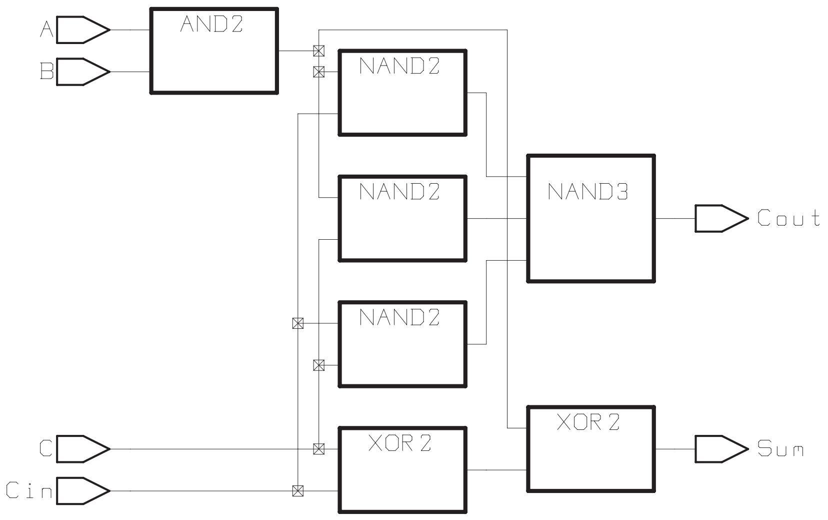


Figure 17-2: Multiplier base module (MULTcell in Fig. 17-1)

First the product terms a_0b_0 to $a_{n-1}b_0$ are computed, using the modules of figure Fig. 17-1. The inputs **c** and **CIN** are set to 0. The result of multiplying the least significant bits a_0b_0 is available at the global output of the circuit after computation.

In a second step, the product terms a_0b_1 to $a_{n-1}b_1$ are computed. The input **c** of a base cell which computes a_ib_1 $(0 \le i \le n-2)$ is set to a_ib_0 (the result of the first computation).The input **c** of the cell, which computes $a_{n-1}b_1$ is set to 0. Input **CIN** of all modules is set to the carry of the computation of $a_{i-1}b_0$. Now the computation of the second least significant bit of *P* is finished and the results, available at the outputs **S**, are needed to proceed further.

In the *n*-th step, the circuit computes the product terms a_0b_{n-1} to $a_{n-1}b_{n-1}$ using *n* multiplier base modules. The input **c** of the module which gets the most significant bits for computation (a_{n-1} and b_{n-1}) is set to 0 again. All other modules get the result of the previous step via **c**. Input **CIN** of the cell computing a_ib_{n-1} gets the carry of the cell computing $a_{i-1}b_{n-2}$. The result of the cell computing the least significant bit (a_0b_{n-1}) is available at the primary outputs of the whole circuit. The outputs **S** of all other cells is connected to inputs **b** of a full adder. Hence *n*-l full adders are needed. The second input **a** of each adder is connected to the carry out of the multipliers base module one position left (with regard to the module providing the signal for the adder input **b**). **CIN** of the rightmost adder is set to 0. All outputs **COUT** are connected to the inputs **CIN** of those full adders computing the next higher significant bit of the product *P*. The lowest row of the combinational circuit consists of a ripple-carry adder. The most significant bit of P results Or-ing **COUT** of the rightmost full adder and **COUT** of the rightmost multiplier cell placed above the adder.

A simulation can be found in Fig. 17-3.

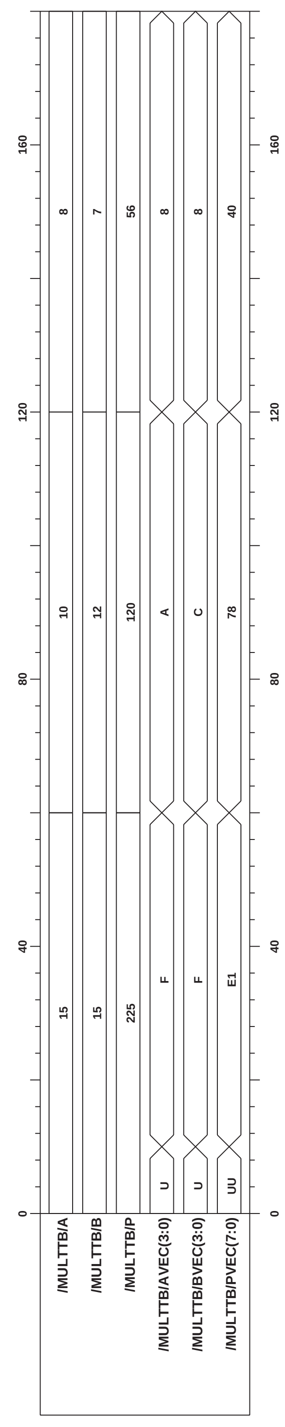


Figure 17-3: Simulation of the Multiplier

17.3 Status and Acknowledgments

Thanks to H.-P. Eich and C.-J. Thomas for designing the circuit in a commercial design system.

17.4 Literature

[Brya86] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

18 Divider

A divider is mainly introduced for having an additional purely combinational example besides a multiplier and because it is used in other verification examples like the GCD.

Moreover, the verification must perform a data abstraction, since the specification is given in terms of natural numbers and the implementation is based on bit vectors.

18.1 Specification

Given two natural numbers, the dividend N and the divisor D, calculate the quotient Q and the rest R such that Q is the largest natural number and R is a number with $R < D$ such that

$$N = D \cdot Q + R \qquad (18\text{-}1)$$

18.2 Implementation

18.2.1 Algorithm

The implementation closely follows the “restoring cellular array divider”, presented in [Hwan79] (pp. 264ff., see also [Haye88]). As the multiplier, presented in section 17, it is based on a regular, cell based design.

In contrast to other implementations, this design performs two operations at each computation cycle. The first subtracts the divisor for the dividend, staring with the most significant bit. If the subtraction results in a negative difference, a bit 0 will be produced for the quotient. The second operation is a restoring addition, if a negative difference occurred. The latter operation produces parts of the rest *R*. Each computation cycle is finished by a left-shift. If a positive result was achieved by the substraction, then a 1 bit is generated for the quotient and no addition is performed.

18.2.2 Global Realization Architecture

The circuit is built by a iterative cellular array as depicted in Fig. 18-1. Each cell DIVcell uses a controlled subtraction with 4 inputs and outputs.

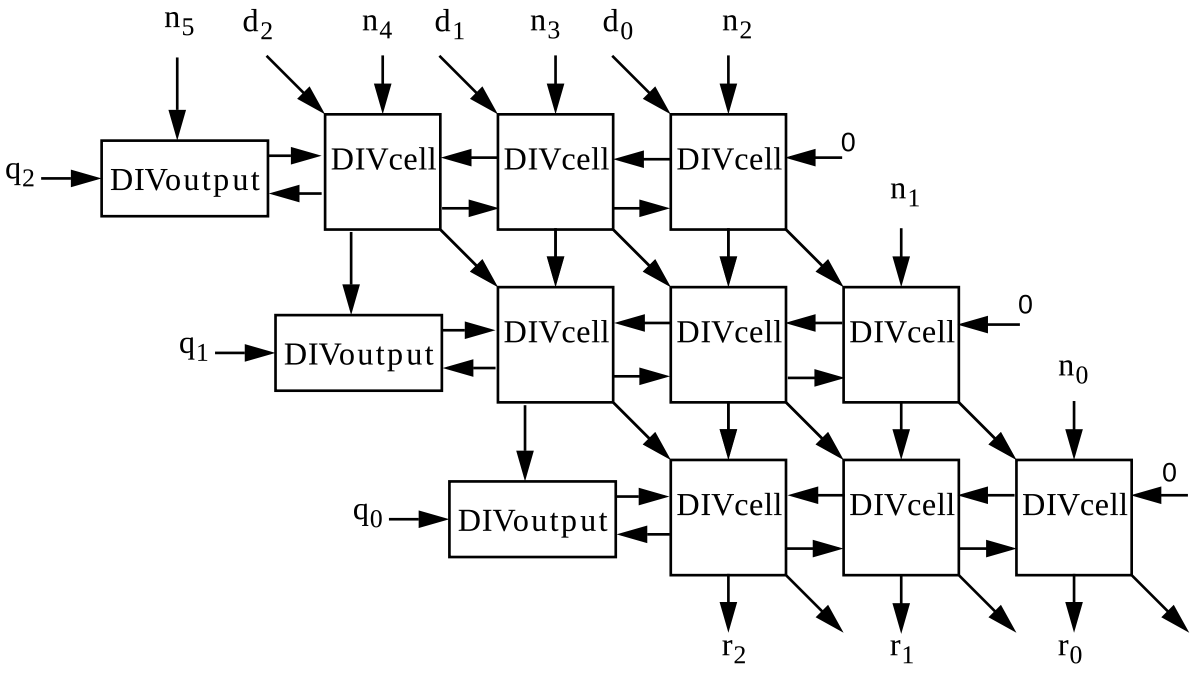


Figure 18-1: Divider

The signals of one DIVcell module are given in Fig. 18-2 and are specified in table 18-1. The signals **DivOut** and **SubOut** are omitted in the table, since they are identical to **DivIn** and **SubIn**, respectively.

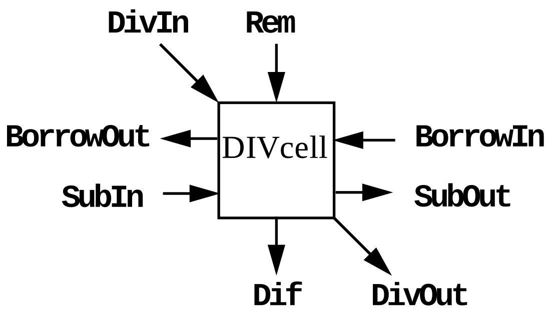


Figure 18-2: Signals of the divider base module DIVcell

BorrowOut =	(**Rem**¯**DivIn**) ∨ (**Rem**¯**BorrowIn**) ∨ (**DivIn** **BorrowIn**)
Dif =	(**Rem** **SubIn**) ∨ (**Rem** **DivIn** **BorrowIn**) ∨ (**Rem** **DivIn** **BorrowIn**)
	∨
	(**Rem**¯**DivIn** **BorrowIn** **SubIn**) ∨ (**Rem** **DivIn**¯**BorrowIn** **SubIn**)

Table 18-1: Specification of the divider base module

The signale of a DIVoutput cell are given in Fig. 18-3 and are specified in table 18-2.

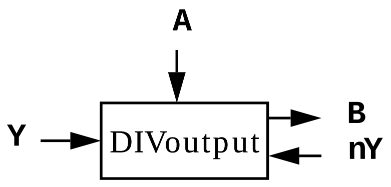


Figure 18-3: Signals of the divider base module DIVcell

Y =	**A** ∨ ¬ **B**
nY =	¬ **Y**

Table 18-2: Specification of the divider interface module DIVoutput

The inputs **Rem**, **DivIn** and **BorrowIn** are used for rest, divisor and carry input. **BorrowOut** denotes the carry output and **SubIn** is a control signal for all cells of the respective row. The signal **Dif** for the computed rest output realizes a function as shown in table 18-1.

Dif =	**Rem** ⊕ **DivIn** ⊕ **BorrowIn**, if **SubIn** = 0
Dif =	**Rem**, if **SubIn** = 1

Table 18-3: Specification of the rest computation

The realization of a cell DIVcell is given in Fig. 18-4.

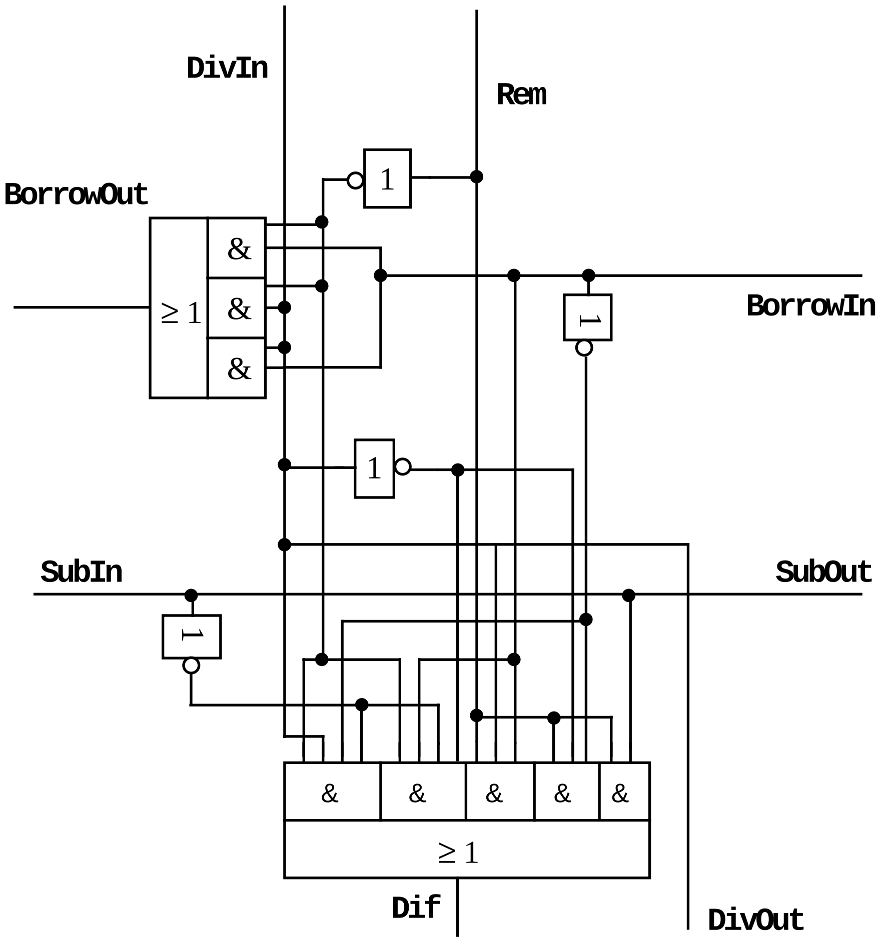


Figure 18-4: Divider base module

18.3 Detailed Implementation Description

For building a divider with an m-bit dividend and a n-bit we need $(m-n)n$ cells. The $(m-n)$ rows of the array are shifted one position right to each other and consist of n cells each. In Fig. 18-1 there are shown 4 unsigned bit vectors: the dividend $N = \{n_5, \ldots, n_0\}$, the divisor $D = \{d_2, \ldots, d_0\}$, the quotient $Q = \{q_2, \ldots, q_0\}$ and the rest $R = \{r_2, \ldots, r_0\}$.

The input **DivIn** (divisor bit in) of the first row the appropriate bit of the divisor is applied, where the leftmost cell gets the most significant bit. Input **BorrowIn** (borrow in) of the i-th cell is connected to the output **BorrowOut** (borrow out) of the (i-1)-th cell $(0 < i \le n-1)$. Input **BorrowIn** of cell 0 is set to 0. Output **BorrowOut** of cell n-1 is connected to an inverter. The inverted signal is “ORed” with most significant bit of the dividend. It is available as a global output and defines the MSB of the quotient. This signal is also inverted again and fed to input **SubIn** (subtract control in) of the (n-1)-th cell. Output **SubOut** (subtract control out) of the i-th cell is connected to **SubIn** of the (i-1)-th cell. Output SubOut of the 0-th cell remains unconnected. All outputs **DivOut** (divisor bit out) of the cells at a particular row are connected to the inputs DivIn (divisor bit in) of the next lower row, according to their indices.

Output **Dif** (difference) of the (n-1)-th row is OR-connected with the inverted **BorrowOut** output of the (n-1)-th cell of the next lower row. All outputs **Dif** of the cells n-2 to 0 are connected to the inputs **Rem (**remainder bit) of the cells n-1 to 1of the next lower rows. The inputs of **Rem** of row 0 get the dividend (MSB left side).

Besides input **DivIn** and **Rem**, all rows are connected as described above for the first row. The j-th row gets a signal via **DivIn** from output **DivOut** of the (j-1)-th row and via **Rem** a signal from the output **Dif** $(0 < j \leq m - n)$. Input **Rem** of the 0-th cell of row j gets the j+1-highest bit of the dividend. Via the outputs **Dif** of the lowest row the rest of the division is available.

Moreover, every row provides a bit of the quotient, according to its row index j and together with the OR-connected and inverted signal of **BorrowOut** of the (n-1)-th cell.

The implementation of a base module DIVcell, the DIVoutput cell and a design using a 6 bit dividend and 3 bit divisor is shown in Fig. 18-5, Fig. 18-6 and Fig. 18-7, respectively.

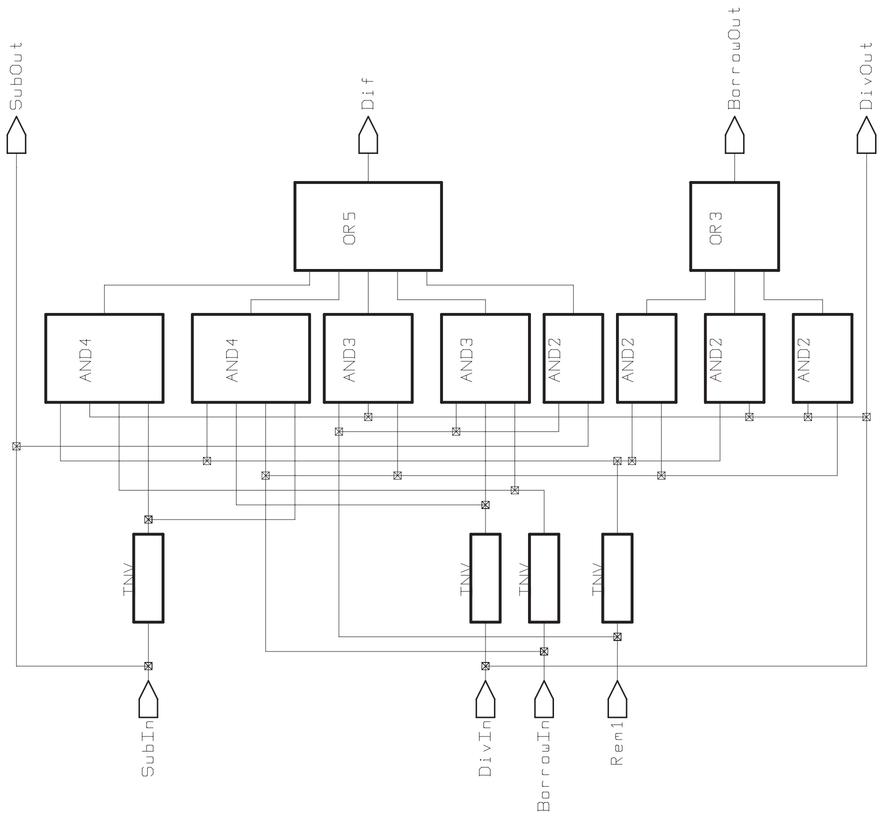


Figure 18-5: Divider base module designed in SYNOPSYS

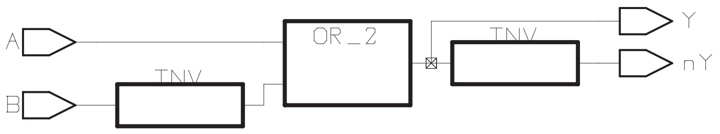


Figure 18-6: Output Cell DIV output

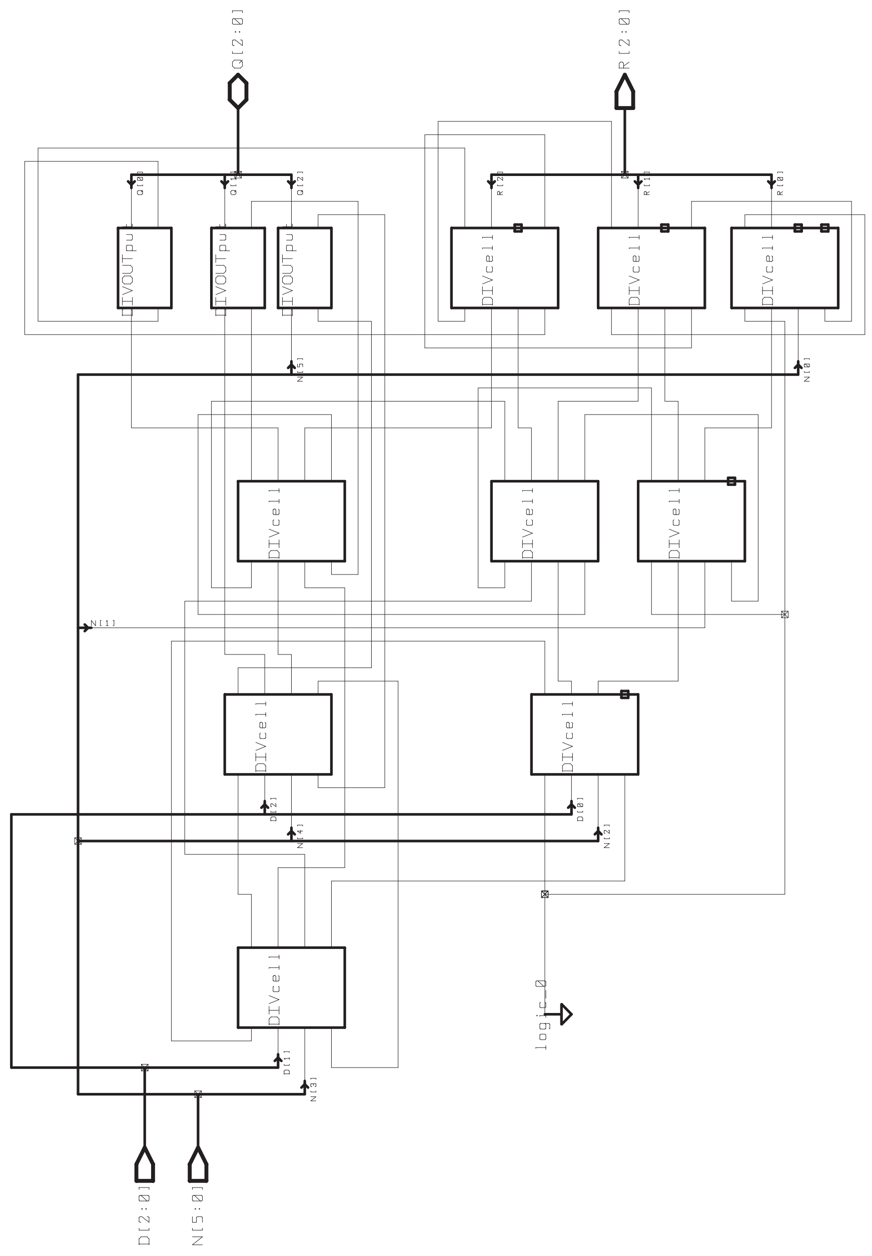


Figure 18-7: Divider designed in SYNOPSYS

Fig. 18-8 shows the simulation waveform..

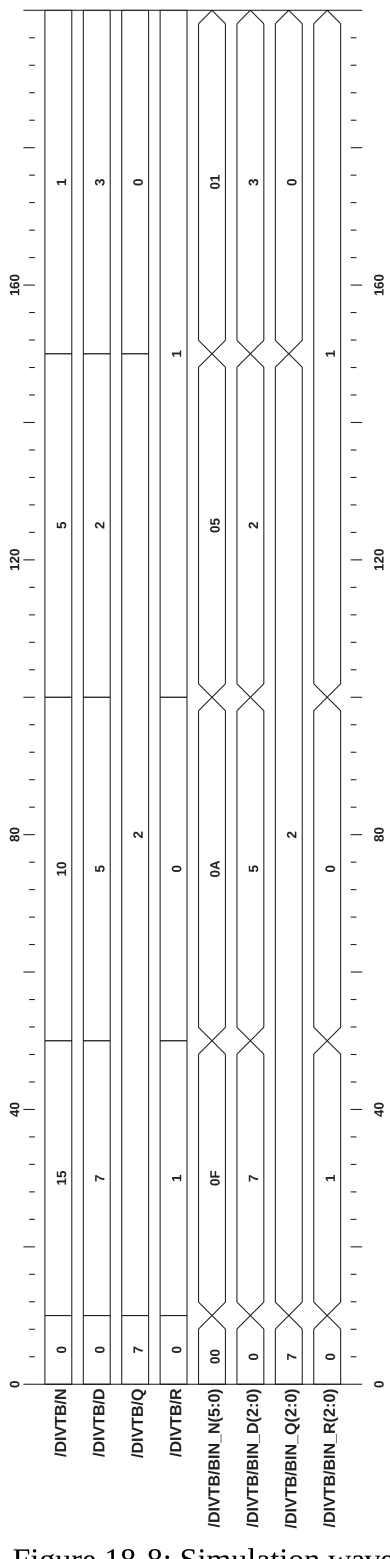


Figure 18-8: Simulation waveform

18.4 Status and Acknowledgments

Thanks to H.-P. Eich and C.-J. Thomas for designing the circuit in a commercial design system.

Thanks to G. Janssen from Eindhoven University for pointing out numerous flaws in the first version of this example.

18.5 Referencee

[Fram92] CADENCE Design Framework II version 4.2a. Reference Manual, February 1992.

19 FIFO

The FIFO storage element has been chosen, since its implementation is completely asynchronous and does not contain any arithmetic or other computational elements. Thus it is well suited to check verification approaches for their asynchronous capabilities. On the other hand, the circuit is specified in a generic way: it may contain *n* storage elements.

19.1 Specification

The specification of the FIFO element is quite simple and follows the first-in-first-out definition.

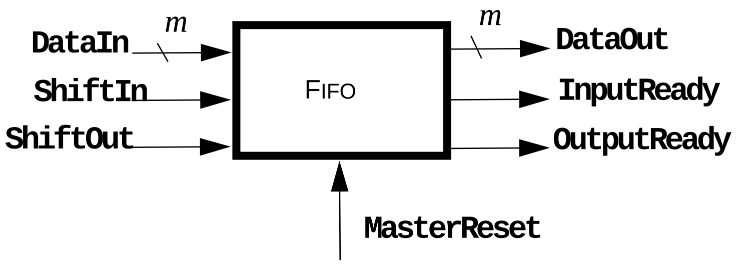


Figure 19-1: Black box view of the FIFO element

Using the input signals given in Fig. 19-1, the following operations should be possible, where “values” denote bit vectors of length *m*:

1. Input a value via **DataIn**, provided that **MasterReset** = 0 and **InputReady** = 1.
2. Output a value via **DataOut**, provided that **MasterReset** = 0 and **OutputReady** = 1.
3. Reset the FIFO element with **MasterReset** = 1.

Hence an overflow of the *n*-place Fifo queue is avoided, if no input is accepted if it is completely filled, i.e. **InputReady** = 0

19.2 Implementation

The implementation is taken from [Schm78]. It is organized using an asynchronous “bubble through” mechanism: input data “fall” automatically to the next free position. It is realized using an asynchronous shift register (Fig. 19-2). Every stage of the shift register consists of a register, i.e. *m* D-flip-flops, responsible for storing the *m* bit vectors. They store data, if a rising edge is applied to their clock signal. Below the shift register cells there is a RS-flip-flop, which indicates, if the respective register contains data (Q =1) or is empty (Q = 0).

Fig. 19-3 and Fig. 19-4 show a realization of one stage and a 4 bit realization, respectively.

Fig. 19-5 shows the simulation of various input and output operations of the FIFO circuit.

19.3 Status and Acknowledgments

Thanks to H.-P. Eich and C.-J. Thomas for designing the circuit in a commercial design system [Fram92].

19.4 Literature

[Schm78] V. Schmidt. *Digitalschaltungen mit Mikroprozessoren*. B.G. Teubner Verlag, Stuttgart, 1979. (in german).

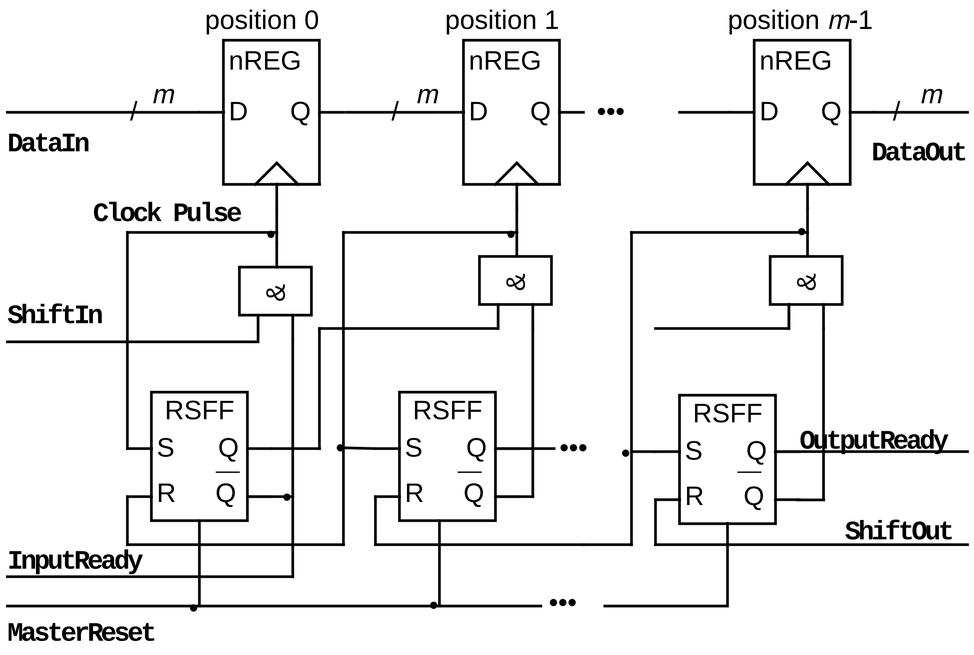


Figure 19-2: Realization of a *m* bit FIFO element with *n* places

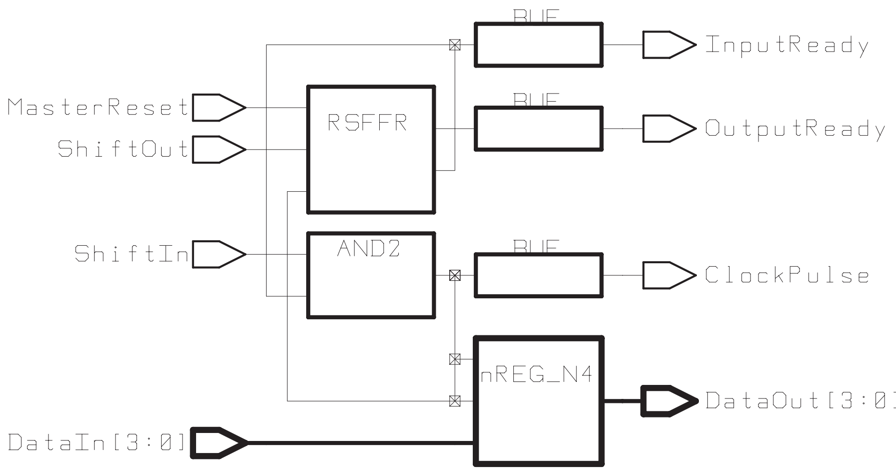


Figure 19-3: Design of one stage of the FIFO element

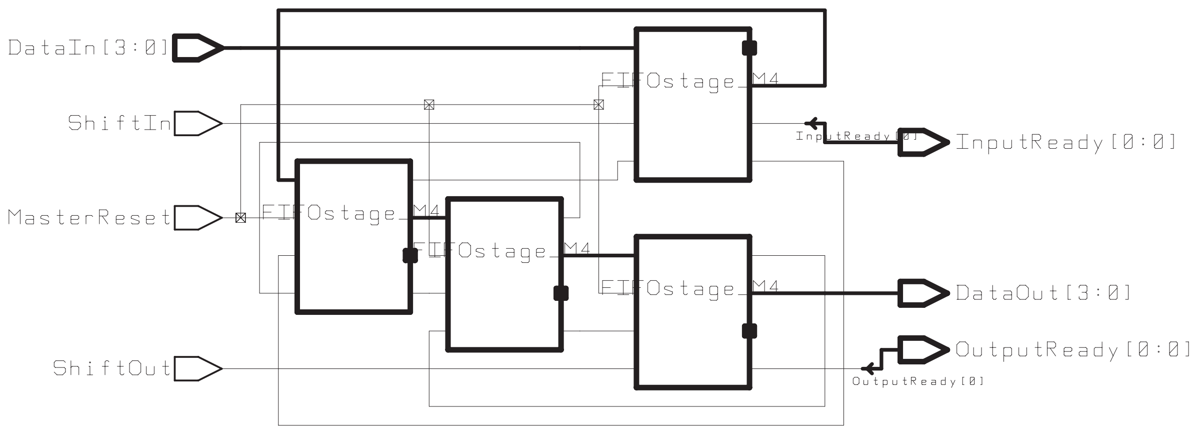


Figure 19-4: Design of a 4-bit FIFO element

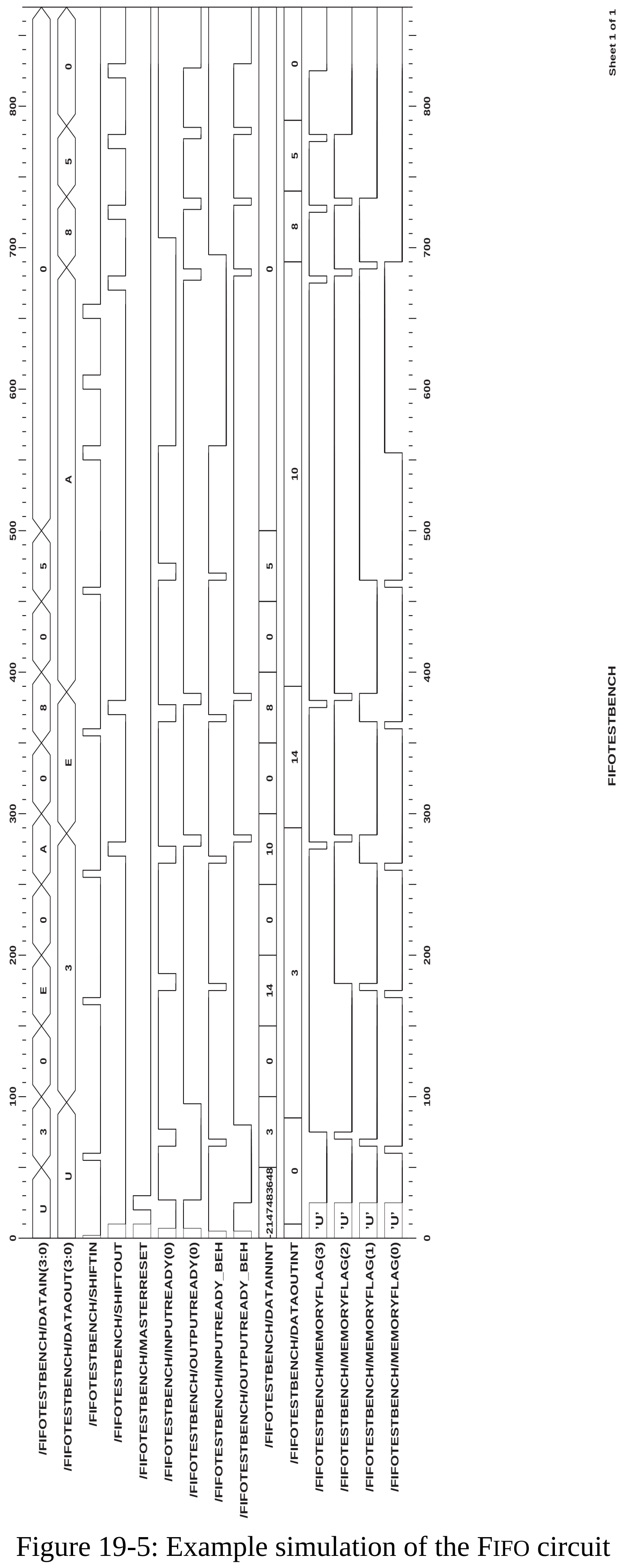
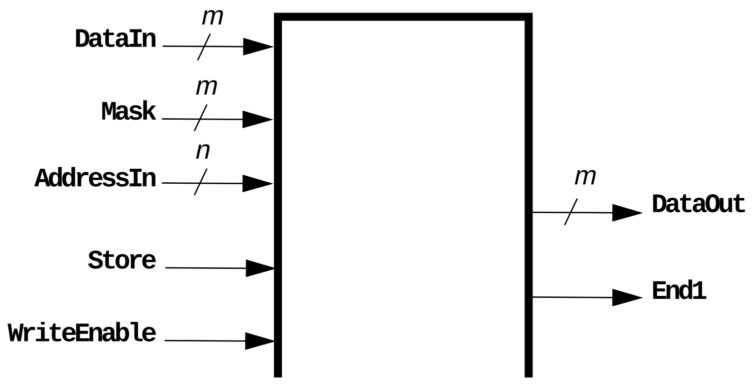


Figure 19-5: Example simulation of the FIFO circuit

20.1.1 General Specification

In an associative memory (also called “content addressable memory”) each stored data wor can be retrieved by using characterizing parts of the data. E.g. having stored persons togethe with their birth dates, when applying a date it is possible that more than one person is store with this date. All of them are “matched” and afterwards “selected”. This matching and selection process is a key part of most associative memories. The data units to be stored are fixed length words. The black box view of the memory is given in Fig. 20-1.



$i \in [0, n-1]$

instant t+1 (rising edge) the result is available at **DataOut** and the signal **End** is set. To start a new request, the controller has to be reset with **Reset** = 1. Using an associative memory, the comparison of data with *n* other words can be performed in O(1).

20.2 Implementation

20.2.1 General architecture

The structure of a simple word-organized associative memory is shown in Fig. 20-2. Each sub-part of such a word may be chosen as a key, i.e. as a search pattern. The bits relevant for a search are marked in the mask register and are compared simultaneously with all stored words. Every matching entry results in a match signal. The “select circuit” then chooses exactly one matching word and produces a “select” signal to retrieve the word from the storage cell array to be written into the output register. There are different strategies for choosing the word like “first hit” etc.

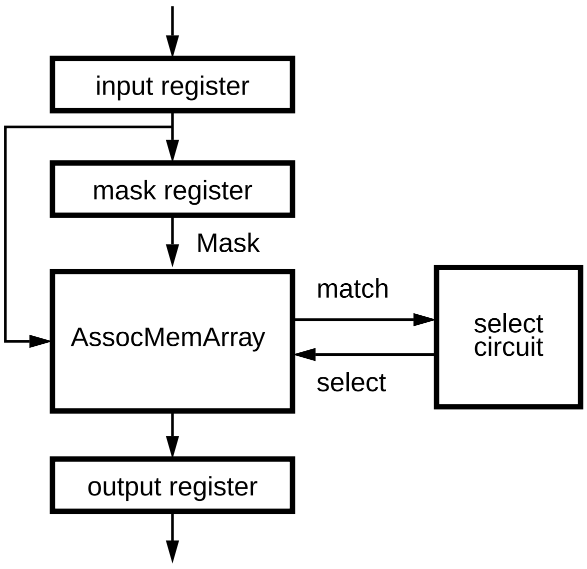


Figure 20-2: Associative memory with fixed word-length

The main module in Fig. 20-2 is the “storage cell array” AssocMemArray. Its black box view is shown in Fig. 20-3 and allows the following operations:

1. Read in a data word, consisting of an input field of length *m* (read in at **DataIn**) and a key field of length *n* (read in at **Mask**).
2. Search for data using a key fed in at **Mask**. **Match** indicates, if there have been found one or more matching entries.
3. Read out a matching data word via **DataOut**, using a key at **Mask** and a selected row (using **Select**).

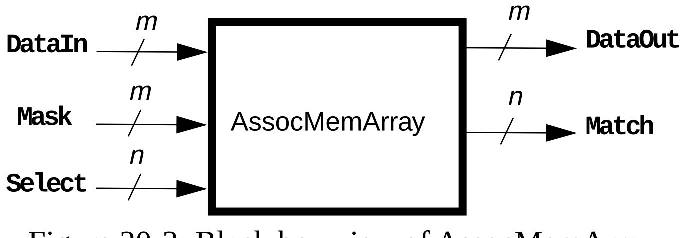


Figure 20-3: Black box view of AssocMemArray

20.2.2 Implementation of one cell of AssocMemArray

Since all words have to be compared with the key, each bit cell AssocMemCell of AssocMemArray needs a compare circuit. The realization of one cell is shown in Fig. 20-4.

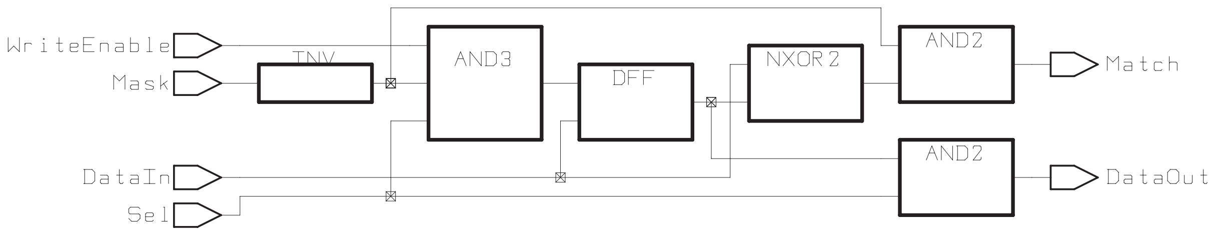


Figure 20-4: One cell of the associative memory (AssocMemCell)

The basic storage element is a D flip-flop. An NXOR-gate performs the actual comparison between the flip-flop content and the input data **DataIn**.The output of the NXOR is AND-connected with the inverted signal **Mask** to form the match signal **Match**. Hence a match only takes place, if the cell is not masked. To read out the flip-flop the select signal **Sel** is AND-connected with its content to get the data output **DataOut**. To store data in the flip-flop, its clock input is AND-connected with the select signal **Sel**, write-enable signal **WriteEnable** and the inverted mask signal **Mask**.

20.2.3 Implementation of a $m \times n$ Storage Cell Array (AssocMemArray)

Given a mask and an input word of length m. For a $m \times n$ memory, $m \cdot n$ cells of type AssocMemCell (Fig. 20-4) are necessary. Cell i, j in the i-th row and j-th column is connected as follows $(0 \le i \le m-1, 0 \le j \le n-1)$. Input **WriteEnable** is connected to a global **WriteEnable**. Data input **DataIn** is connected to the i-th least significant bit of the input word, mask input **Mask** is connected to the i-th least significant bit of the mask word. Select input **Sel** is connected to the j-th lowest significant bit of the select word. Data output **DataOut** and match output **Match** are OR-connected with all respective outputs of the j-th row, i-th column

20.2.4 Implementation of the whole Associative memory (AssocMemCirc)

The whole associative memory is shown in Fig. 20-5.

To perform read and write at definite time instances, three n-bit register for data, mask and result as well as a m-bit register for the address are needed. Read of a data word (mask word) is done via **DataIn** (**MaskIn**) with a storage signal **Store** = 1 and a rising edge of **Clock**. Analogously, an address word is read via **AddressIn** in the address register.

Data input and mask input are connected with the outputs of the data register and mask register. The n-bit “match” output of the associative memory is fed into a “select circuit”. The latter is being realized by a combinational circuit, which selects exactly one hit, if more than one hit has been achieved. The n-bit output is fed into the “1” input of a multiplexer. The multiplexer is selected via **SelectAdrMat**. The n-bit output of the multiplexer is fed into the address input of the associative memory. To be able to read out data at a definite time instant, a control circuit is necessary. Here, **SelectAdrMat** is AND-connected with the inverted signals **Store**, **WriteEnable** and **Reset** and fed into the set input of the RS-flip flop.

The **Reset** line is connected to the reset input. The state of the RS flip flop thus indicates, whether data have to be stored in the output register (state = “1”). The flip flop state also provides the **End** signal. The result of the search process is available at **DataOut**.

A 4×4 memory is given in Fig. 20-7.

As an example, the data as given in table 20-2 is written into the memory. Next, the word to be fetched is written into the memory as 0001. The search mask is set to 1110, such that only

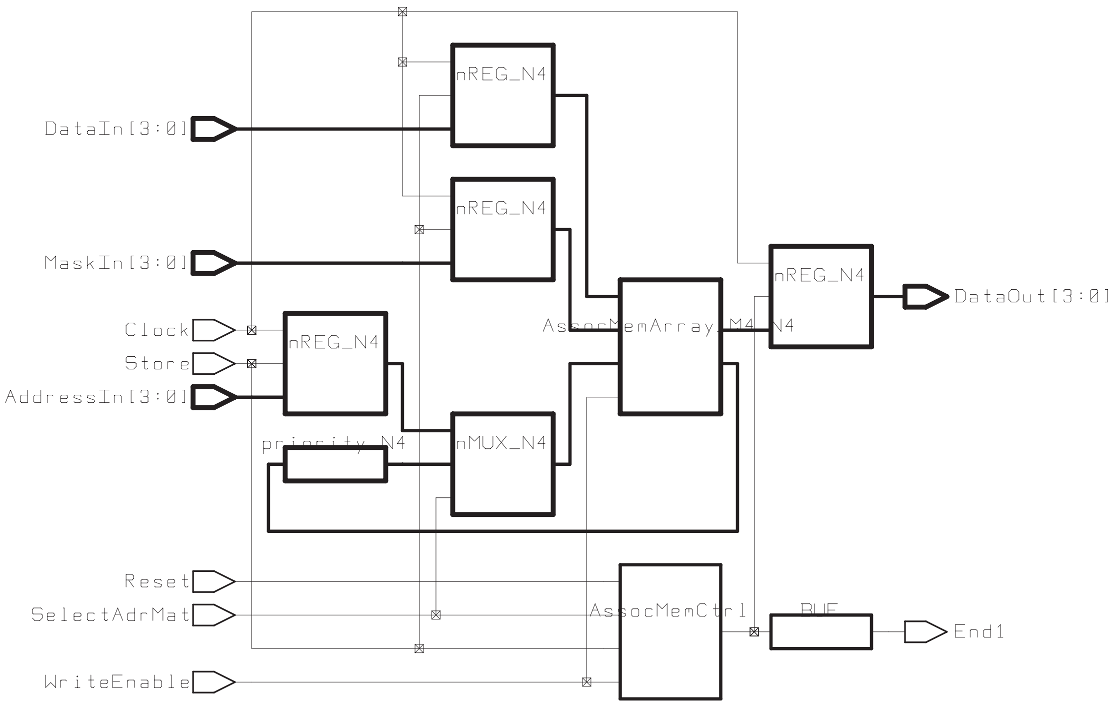


Figure 20-5: Whole associative memory (AssocMemCirc)

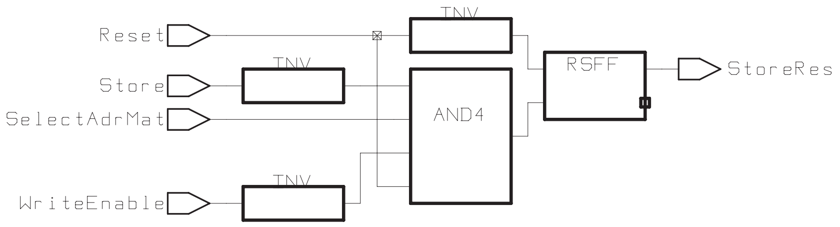


Figure 20-6: AssocMemCntrl of Fig. 20-5

the rightmost bit is used for the search. This leads to a search result 0011, which equals the content of the 4th cell.

The simulation result of this examples is shown in Fig. 20-8 (the suffix “1” denotes signals of the structural description, “2” denotes signals of the behavioral description).

20.3 Status and Acknowledgments

The presented associative memory element is currently used to get a larger hierarchical circuit, which computes the maximum of all stored numbers. This use of the circuit will be added to this description or will constitute a separate verification benchmark in the near future.

Thanks to H.-P. Eich and C.-J. Thomas for designing the circuit in a commercial design system.

20.4 Literature

[Haye88] J.P.Hayes. *Computer Architecture and Organization*. McGraw-Hill, 2. edition, 1988.

[Koho77] T. Kohonen. *Associative Memory*. Communication and Cybernetics. Springer Verlag, 1977.

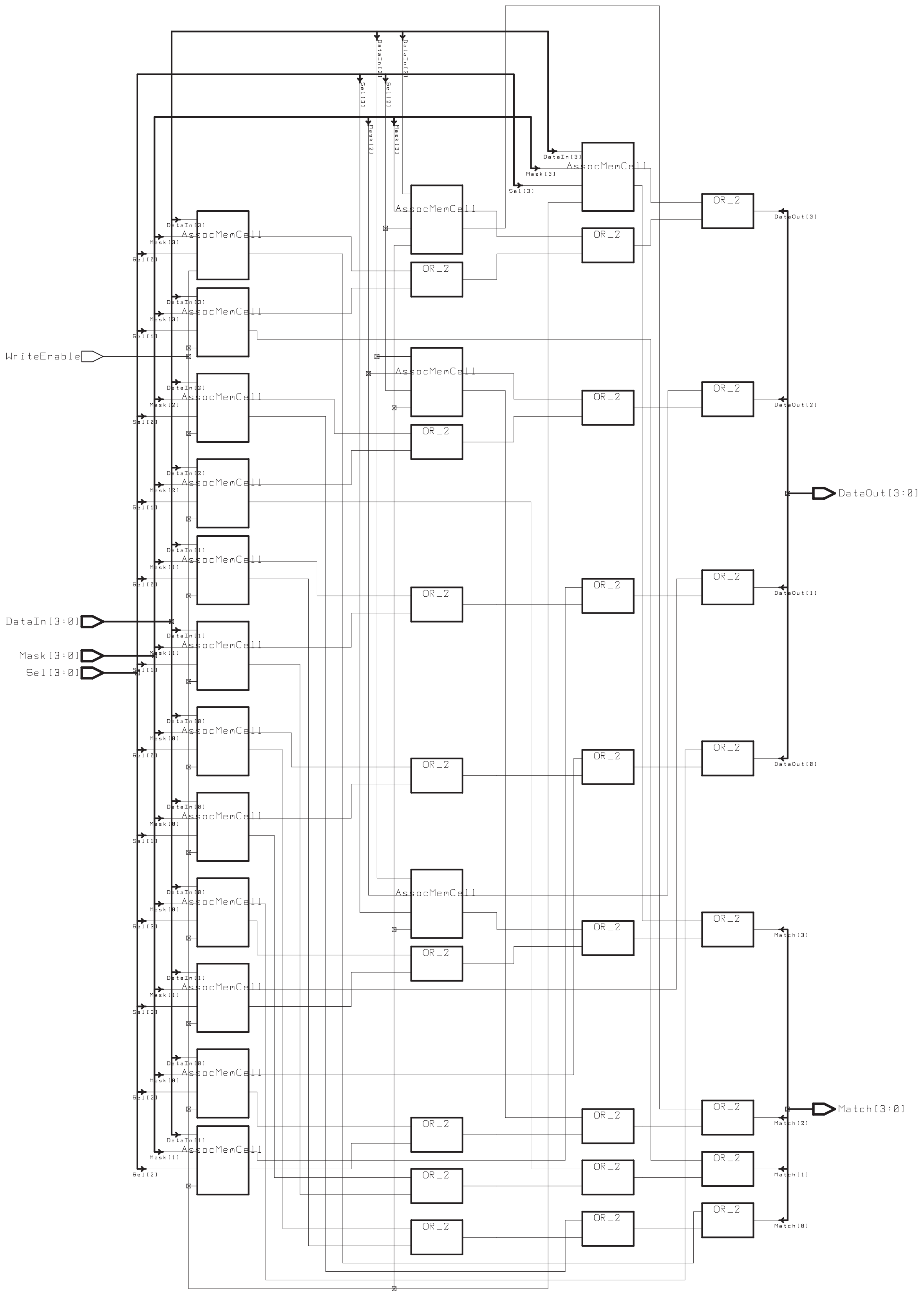


Figure 20-7: Realization of a 4×4 memory (AssocMemArray of Fig. 20-5)

cell #	**data (binary)**
1	0110
2	1000
3	left empty
4	0011

Table 20-2: Example Data written into the memory

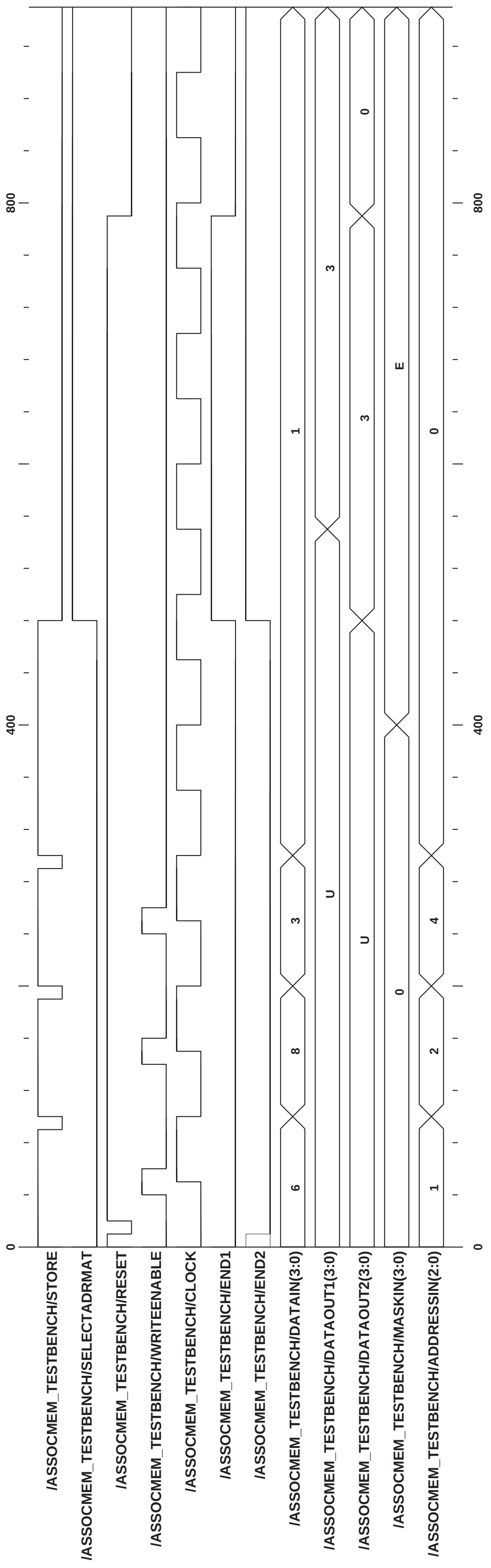


Figure 20-8: Simulation waveform of the example

21 1Syst (Filter)

21.1 Introduction

The filter circuit of this section and the matrix multiplier of section 22 have been chosen as witnesses of one-dimensional and two-dimensional systolic architectures. The notion of “systolic arrays” has first been introduced by Kung and Leierson [KuLe78]. Basics on these specialized regular architectures can be found in [Kung82]. The circuit to be presented here is a filter element also taken from [Kung82].

Systolic arrays require additional efforts for specifying the intended behavior, since here data have to be applied at different times in a special order to achieve a correct functioning of the circuit. Moreover, more-dimensional systolic arrays are a good illustration of more-dimensional generic circuits.

21.2 Specification

The circuit processes a stream of input values $\{x_1, x_2, \ldots, x_n\}$, $x_i \in N$, $1 \le i \le n$. These input values have to be multiplied with a list of weights $\{w_1, w_2, \ldots, w_k\}$, $w_j \in N$, $1 \le j \le k$. Each y_i of the resulting output stream $\{y_1, y_2, \ldots, y_{n+1-k}\}$ is computed as in equation 21-1.

$$y_i = w_1 x_i + w_2 x_{i+1} + \ldots + w_k x_{i+k-1} \tag{21-1}$$

An example computation for $k = 3$ and $n = 5$ is given in equation 21-2.

$$\begin{aligned} y_1 &= w_1 x_1 + w_2 x_2 + w_3 x_3 \\ y_2 &= w_1 x_2 + w_2 x_3 + w_3 x_4 \\ y_3 &= w_1 x_3 + w_2 x_4 + w_3 x_5 \end{aligned} \tag{21-2}$$

In the implementation given below, first the weights w_i are fed serially into the input **`StreamIn`**. After k weights have reached their respective position in the k cells, they are stored by setting **`StoreWeight`** to true. Afterwards the input values are fed serially into the **`StreamIn`**. The values y_i are computed and serially shifted out on output **`ResultOut`**.

This specification can be verified for concrete values of k weights or as a generic circuit.

21.3 Implementation

21.3.1 General architecture

The implementation is realized using overlapping additions and multiplications. In the first cycle t_1 the input value is multiplied with the weight stored in the respective stage. In the second cycle t_2 this product is to be added to the now available intermediate result y_i. In order to achieve this, the input values x_i must be separated by two clock ticks (Fig. 21-1).

During an initialization phase the weights w_j are loaded into the stages by using the first k input values of the input stream as weights. In the following, inputs values are treated as *n* bit integers and output values as *m* bit integers (Fig. 21-2).

Let w a weight and z an internal variable. If there is a rising edge at $t1$, then $x^{t+1} = x^t$ and $z = x^t \cdot w$ is computed. If there is a rising edge at $t2$, then $y^{t+1} = z + y^t$ is computed.

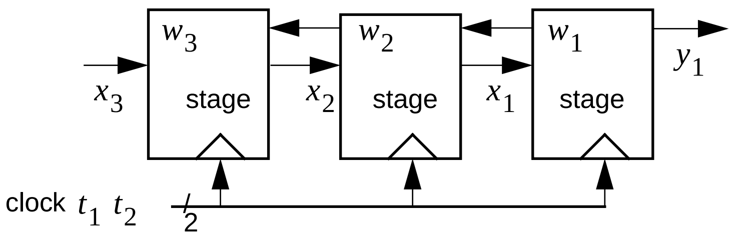


Figure 21-1: Basic array structure computing the first result

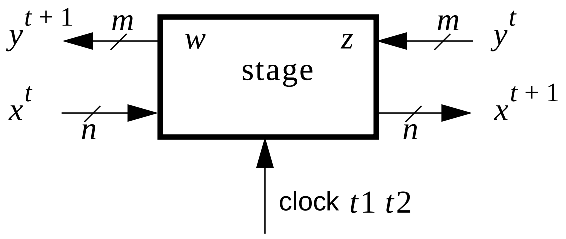


Figure 21-2: Black box view of one stage

21.3.2 Implementation of one stage

The stage has two global inputs **StreamIn<n-1:0>** and **ResultIn<m-1:0>.** Input **StreamIn<n-1:0>** is fed via a 1:2 *n* bit demultiplexer into two *n* bit registers. Using the control signal **SelectWgtStr** of the demultiplexer either the register for storing the weight (**SelectWgtStr** = 0) or the register for the input value (**SelectWgtStr** = 1) is selected. Both register store their respective inputs, if **StoreWgt** = 1 (**StoreStr** = 1) and there is a rising edge at t1.

The content of the input register is available at the output **EStreamOut<n-1:0>** and the input of an *n* bit multiplier. The multiplier also gets the weight.[1] The 2*n* bit output of the multiplexer is stored in a register if **StoreRes** = 1 and there is a rising edge at clock t2 (Fig. 21-3).

Intermediate results, available at input **ResultIn<m-1:0>** are stored in a m bit register if **StoreRes** = 1 and a rising edge at t2. The content of this register as well as the multiplication results are added via an adder[2] to get the result at **ResultOut<x:1>** with $x \geq max(n, m) + 1$.

The general clocking scheme must be such that the rising edge of t1 occurs before the rising edge of t2.

The realization of a stage in a commercial design system is given in Fig. 21-3 and Fig. 21-4.

Fig. 21-5 shows an example computation with $w1 = 4$, $w2 = 2$, $w3 = 1$ and $x1 = 9$, $x2 = 8$, $x3 = 7$, $x4 = 6$ and $x5 = 5$, respectively. The output of a behavioral VHDL description (the “specification”) produces the output stream 59, 52, 45, the structural VHDL description (the implementation) produces $3B$, 34, $2D$ (hexadecimal).

21.4 Status and Acknowledgments

Thanks to H.-P. Eich and C.-J. Thomas for designing the circuit in a commercial design system.

21.5 Literature

[Kung82] H.T. Kung. Why systolic architectures. *IEEE Computer,* pages 37–46, January

1. The multiplier is not further specified here since its realization is arbitrary. Is is possible to use e.g. the multiplier of section 17.
2. For the adder the same holds as for the multiplier.

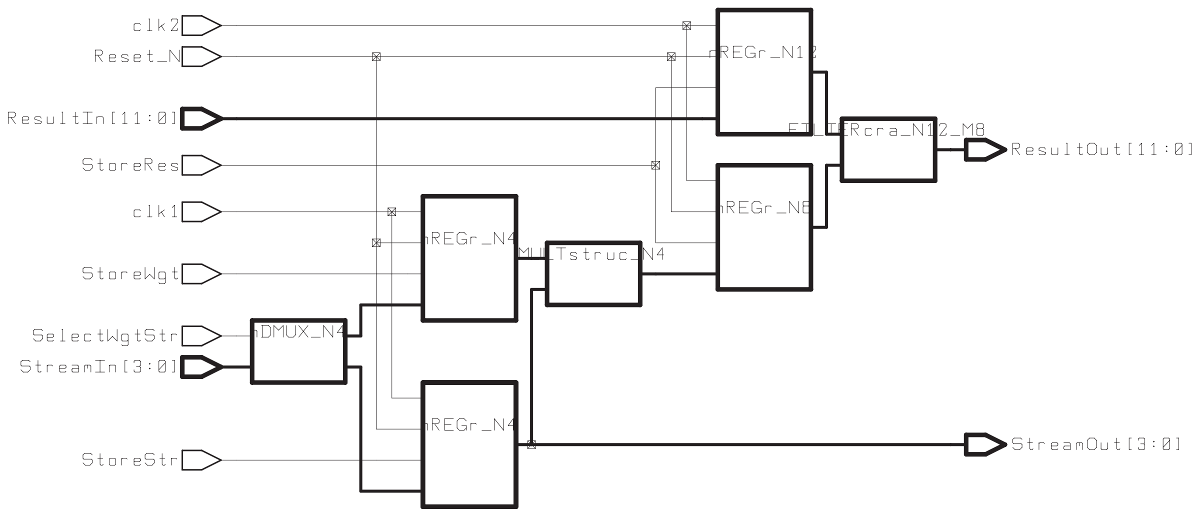


Figure 21-3: SYNOPSIS realization of one stage

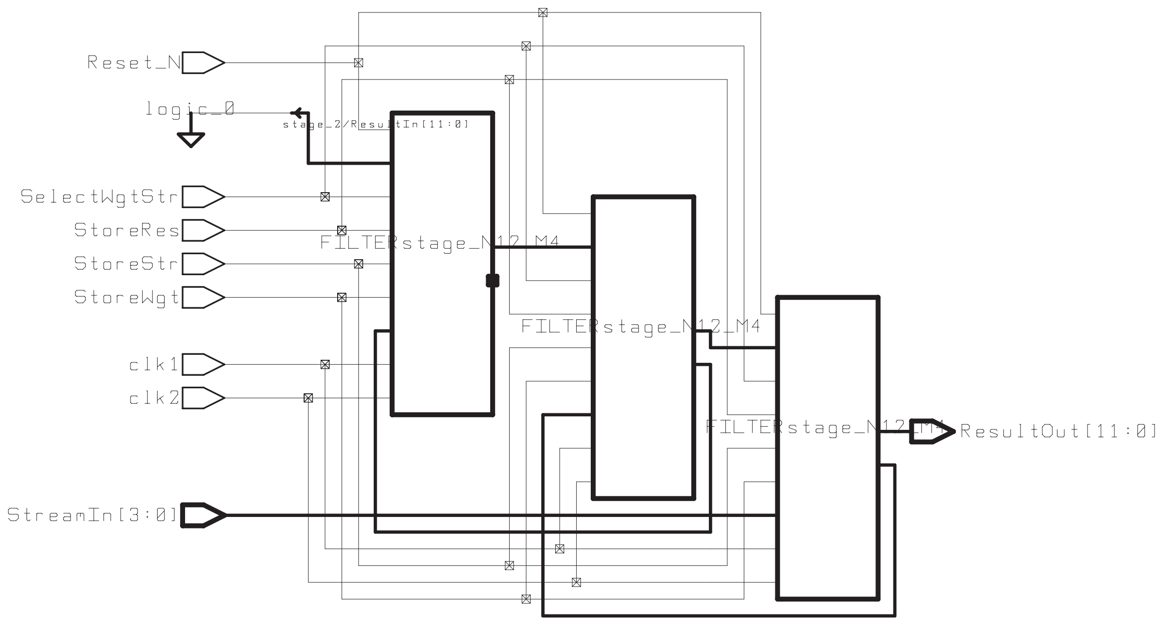


Figure 21-4: SYNOPSIS realization of a complete structure

1982.

[KuLe78] H.T. Kung and C.E. Leierson. Systolic arrays (for VLSI). In *Sparse Matrics Proceedings*, pages 256–282. Society for Industrial and Applied Mathematics 1979, 1978.

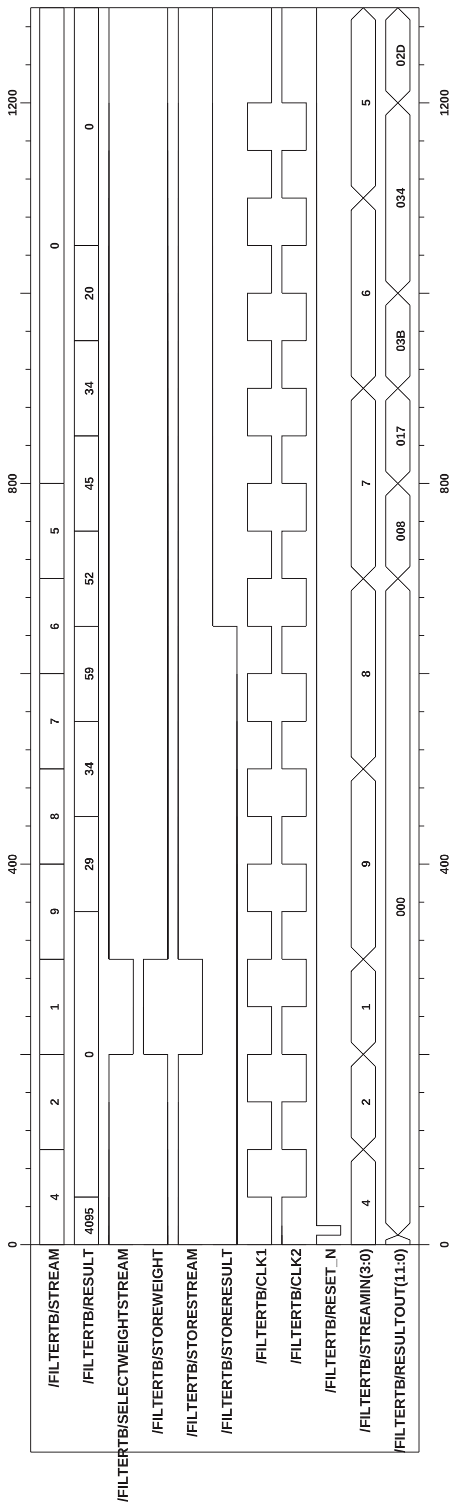


Figure 21-5: Detailed timing diagram

22 2Syst (Matrix Multiplication[1])

The filter circuit of this section and the filter of section 21 have been chosen as witnesses of one-dimensional and two-dimensional systolic architectures. The notion of “systolic arrays” has first been introduced by Kung and Leierson [KuLe78]. Basics on these specialized regular architectures can be found in [Kung82].

Systolic arrays require additional efforts for specifying the indented behavior, since here data have to be applied at different times in a special order to achieve a correct functioning of the circuit. Moreover, more-dimensional systolic arrays are a good illustration of more-dimensional generic circuits.

22.1 Specification

22.1.1 General Specification

The circuit to be presented here is a two-dimensional “hexagonal” systolic array. The circuit has been taken from [MeCo80] and is intended to multiply two matrices.

The multiplication of two $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$ results in a matrix $C = (c_{ij})$, where the multiplication is defined by the recursive equation 22-1.

$$\begin{aligned} c_{ij}^{(1)} &= 0 \\ c_{ij}^{(k+1)} &= c_{ij}^{(k)} + a_{ik}b_{kj} \\ c_{ij} &= c_{ij}^{(n+1)} \end{aligned} \tag{22-1}$$

However, the matrix multiplier must only be able to multiply so-called “band” matrices. These are special matrices where certain matrix positions must carry zeros. The multiplication of two 3×3 and two 4×4 matrices *A* and *B*, having width $w = 3$ and $w = 4$ are given in equation 22-2 and equation 22-3, respectively (note that A and B have the zeros at “mirrored” places).

$$\begin{bmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & a_{23} \\ 0 & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & 0 \\ b_{21} & b_{22} & b_{23} \\ 0 & b_{22} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} \tag{22-2}$$

$$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & a_{42} & a_{43} & a_{44} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} & 0 \\ b_{21} & b_{22} & b_{23} & b_{24} \\ 0 & b_{32} & b_{33} & b_{34} \\ 0 & 0 & b_{43} & b_{44} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix} \tag{22-3}$$

1. The VDHL files often refer to this circuit as MATRIX.

22.1.2 Timing Requirements

The black box diagram of the circuit for a 4×4 multiplication is shown in Fig. 22-1.

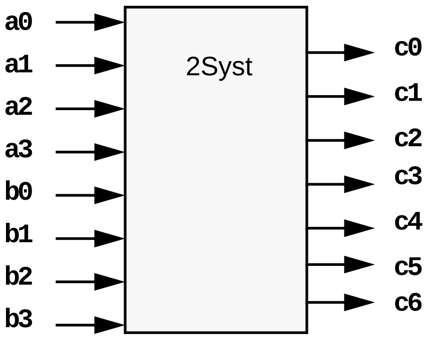


Figure 22-1: Black bock view of 2Syst

For performing a 4×4 multiplication with a band width $w = 4$, the values a_{ij} and b_{ij} have to be fed into the inputs **a0** to **b3** at the time instants as indicated in table 22-1.

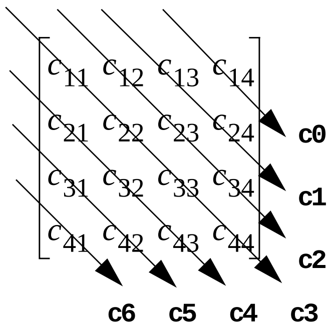
t	a3	a2	a1	a0	b3	b2	b1	b0	c6	c5	c4	c3	c2	c1	c0
0	0	0	0	0	0	0	0	0	-	-	-	-	-	-	-
1	0	0	a_{11}	0	0	0	b_{11}	0	-	-	-	-	-	-	-
2	0	a_{21}	0	0	0	b_{12}	0	0	-	-	-	-	-	-	-
3	a_{31}	0	0	a_{12}	b_{13}	0	0	b_{21}	0	0	0	c_{11}	0	0	0
4	0	0	a_{22}	0	0	0	b_{22}	0	0	0	c_{21}	0	c_{12}	0	0
5	0	a_{32}	0	0	0	b_{23}	0	0	0	c_{31}	0	0	0	c_{13}	0
6	a_{42}	0	0	a_{23}	b_{24}	0	0	b_{32}	c_{41}	0	0	c_{22}	0	0	c_{14}
7	0	0	a_{33}	0	0	0	b_{33}	0	0	0	c_{32}	0	c_{23}	0	0
8	0	a_{43}	0	0	0	b_{34}	0	0	0	c_{42}	0	0	0	c_{24}	0
9	0	0	0	a_{34}	0	0	0	b_{43}	0	0	0	c_{33}	0	0	0
10	0	0	a_{44}	0	0	0	b_{44}	0	0	0	c_{43}	0	c_{34}	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	c_{44}	0	0	0
13	0	0	a_{11}	0	0	0	b_{11}	0	0	0	0	0	0	0	0
14	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…

Table 22-1: Input/output timing scheme

Obviously one of the input matrices is fed column by comun, the other row by row into the circuit. The diagonal elements a_{ii} and b_{jj} are fed into the $\left\lfloor \frac{w-1}{2} \right\rfloor$-th input (**a1** and **b1** in case of $w = 4$, **a2** and **b2** in case of $w = 5$). The time distance between input columns and rows is 2. Output values are available after 2 time instances. The mapping of output matix values c_{ij} to output signals ck ist given in Fig. 22-2. The distance between consecutive output values is 2.

The total length of the output stream equals the length of the input stream, hence it is finished 2 time instances after the last input value has been applied.

The outputs c0 to c6 produce the result c_{ij} according to the scheme given in table 22-2.



c_{11} c_{12} c_{13} c_{14}
c_{21} c_{22} c_{23} c_{24}
c_{31} c_{32} c_{33} c_{34}
c_{41} c_{42} c_{43} c_{44}
c0
c1
c2
c6
c5
c4
c3

Figure 22-2: Output variable correspondence

22.2 Implementation

22.2.1 General architecture

The overall hexagonal architecture of the systolic array for an 4×4 example is given in Fig. 22-3.

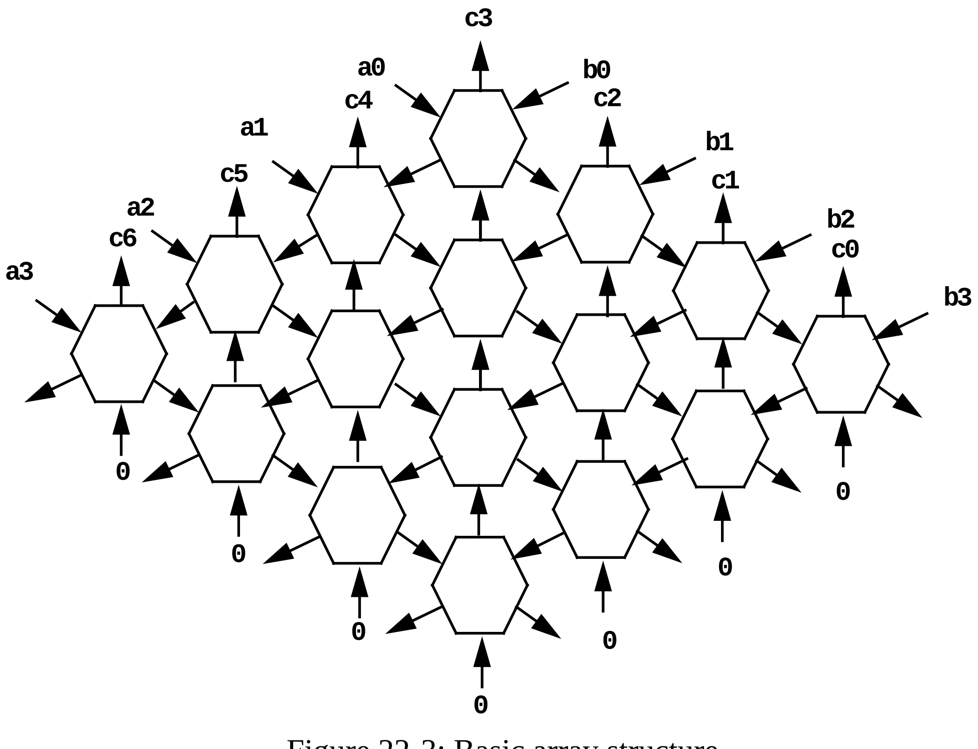


Figure 22-3: Basic array structure

22.2.2 Implementation of one cell

Each cell of Fig. 22-3 has three inputs **A_In<n-1:0>**, **B_In<n-1:0>** and **C_In<m-1:0>** and three outputs **A_Out<n-1:0>**, **B_Out<n-1:0>** and **C_Out<m-1:0>** (Fig. 22-4).

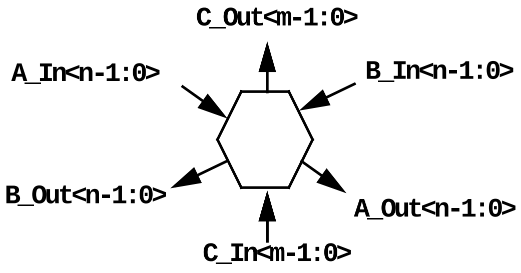


Figure 22-4: Black box view of one cell

The inputs **A_In<n-1:0>** and B**_In<n-1:0>**are connected to a *n* bit register and the input of a multiplier. The 2*n* bit multiplier output is connected to an adder. The second input of the adder is fed by **C_In<m-1:0>**. The adder output is fed in a register. The register outputs provide the cell outputs **A_Out<n-1:0>**, **B_Out<n-1:0>** and **C_Out<m-1:0>** (Fig. 22-5).

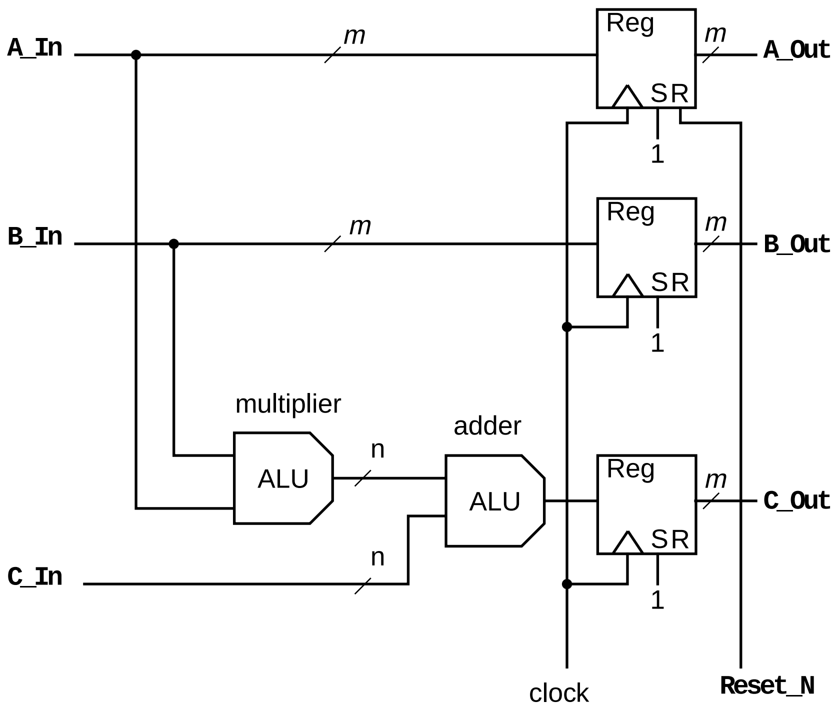


Figure 22-5: Realization of one array cell

22.2.3 Actual Realization and Example

A 4×4 realization of the basic cell (Fig. 22-5) and the whole systolic array (Fig. 22-3) is given in Fig. 22-6 and Fig. 22-7, respectively.

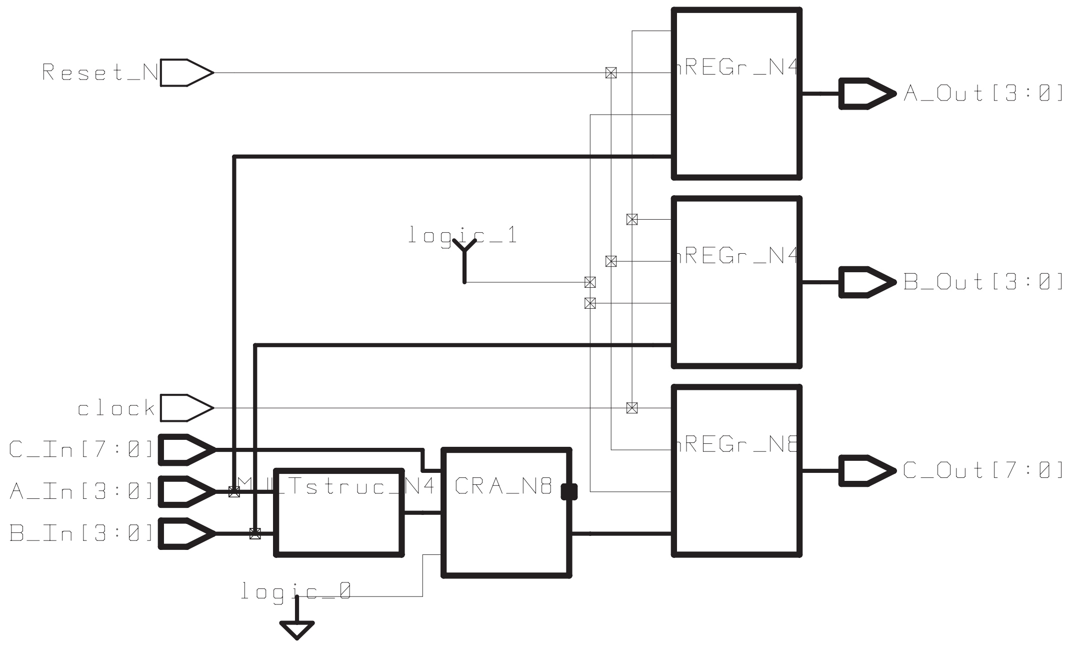


Figure 22-6: SYNOPSYS realization of a basic cell

The following waveform (Fig. 22-8) shows the result of computing the multiplication given in equation 22-4.

$$\begin{bmatrix} 3 & 2 & 0 & 0 \\ 5 & 1 & 3 & 0 \\ 7 & 4 & 2 & 5 \\ 0 & 2 & 1 & 8 \end{bmatrix} \cdot \begin{bmatrix} 4 & 2 & 1 & 0 \\ 5 & 3 & 1 & 2 \\ 0 & 6 & 2 & 7 \\ 0 & 0 & 4 & 5 \end{bmatrix} = \begin{bmatrix} 22 & 12 & 5 & 4 \\ 25 & 31 & 12 & 23 \\ 48 & 38 & 35 & 47 \\ 10 & 12 & 36 & 51 \end{bmatrix} \qquad (22\text{-}4)$$

The busses **StrA0In** to **StrA3In** and **StrB0In** to **StrB3In** carry the coefficients of the matrices *A* and *B*.

22.3 Status and Acknowledgments

Thanks to H.-P. Eich and C.-J. Thomas for designing the circuit in a commercial design system.

The documentation of 2Syst has been completely revised September 1995 due to valuable comments from Scott Hazelhurst, University of Brithish Columbia, Canada.

22.4 Literature

[MeCo80] C. Mead and L. Conway. *Introduction to VLSI Design*. Addison-Wesley, 2. edition, October 1980.

[Fram92] CADENCE Design Framework II version 4.2a. Reference Manual, February 1992.

[Kung82] H.T. Kung. Why systolic architectures. *IEEE Computer*, pages 37–46, January 1982.

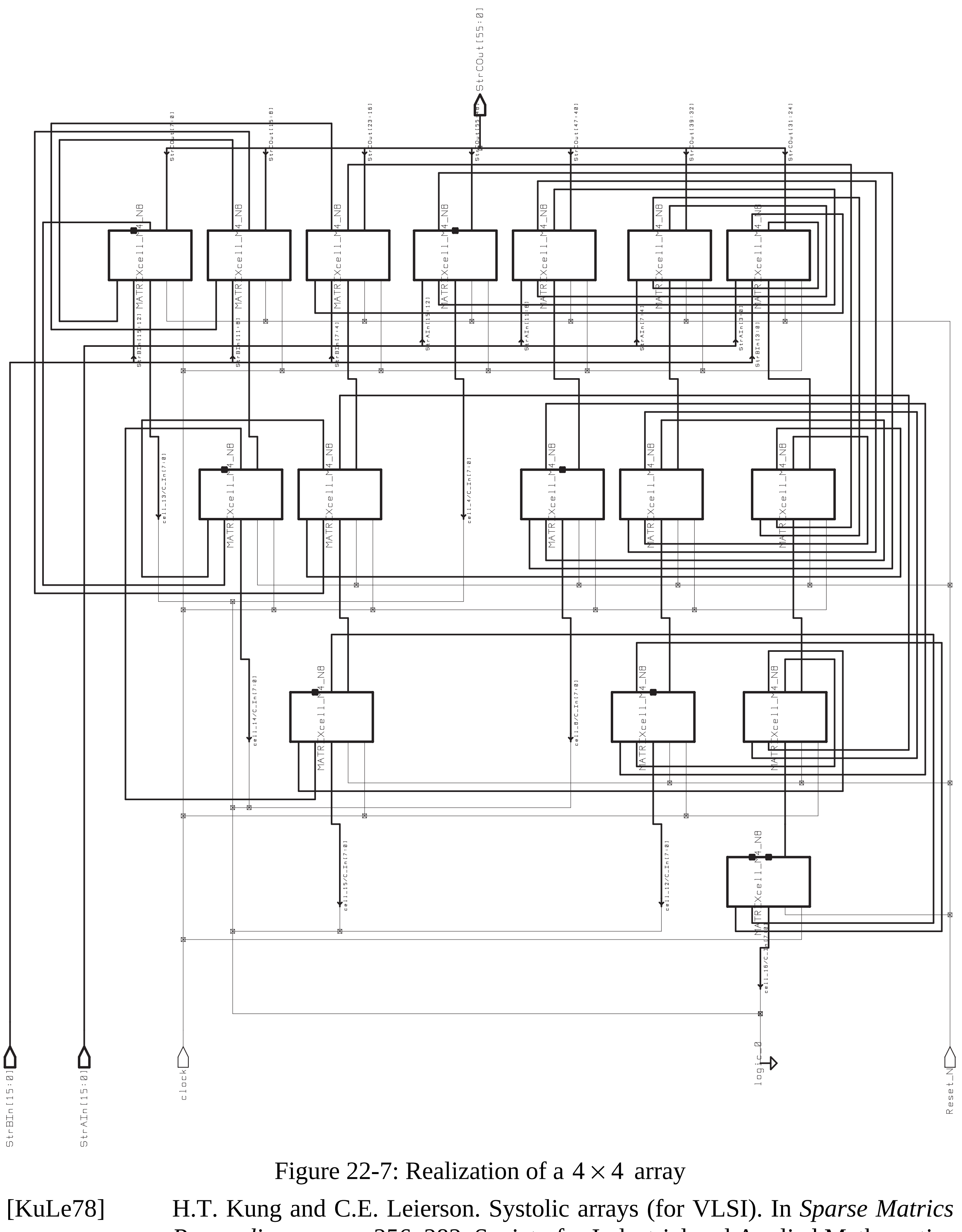


Figure 22-7: Realization of a 4×4 array

[KuLe78] H.T. Kung and C.E. Leierson. Systolic arrays (for VLSI). In *Sparse Matrics Proceedings*, pages 256–282. Society for Industrial and Applied Mathematics 1979, 1978.

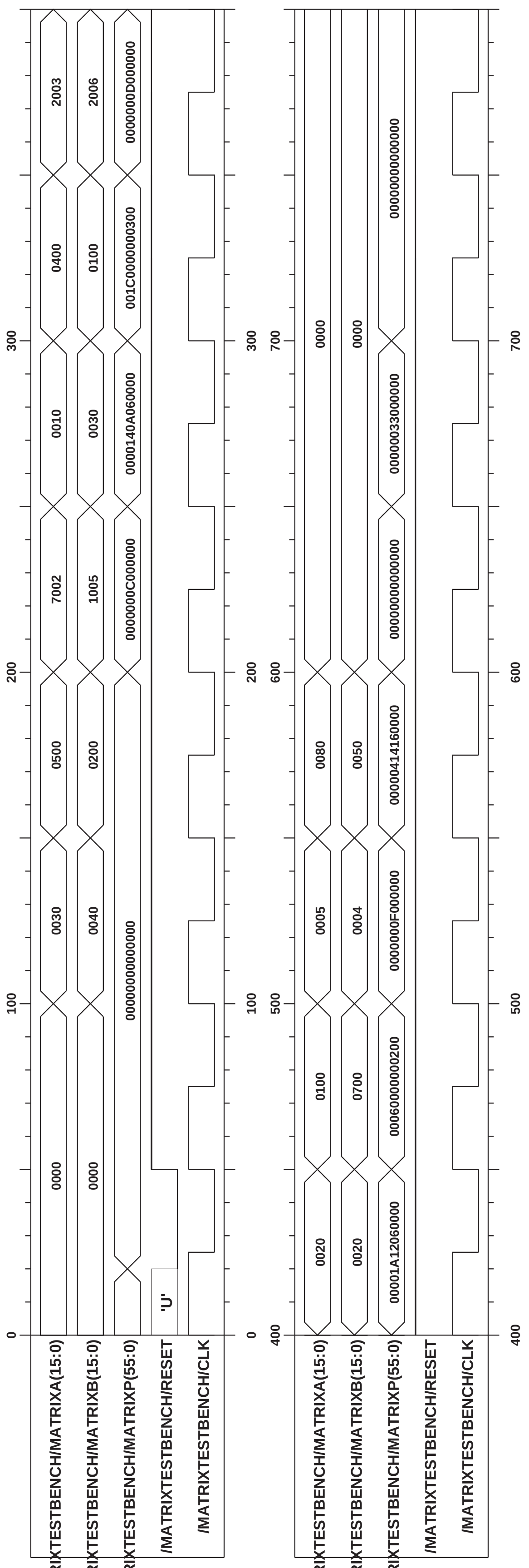


Figure 22-8: Waveform of computing equation 22-4