A low level implementation of the instruction cache of the C-processor

Pernot, J.G.M.W.

Award date:
1990

January - August 1990.

Master thesis report:

A LOW LEVEL IMPLEMENTATION OF THE INSTRUCTION CACHE OF THE C - PROCESSOR.

By J.G.M.W. Pernot

Supervisor : Prof. Ir. M.P.J. Stevens
Coach : Ir. W.J. Withagen

Eindhoven University of Technology, Department of Electrical Engineering, Digital Systems Group.

ABSTRACT

In this master thesis, a study to the low level implementation of the instruction cache of the C - processor is made. This low level implementation is based on the work of Y.C. Hu which is a description of an IDaSS realization of the Instruction cache.

Not only the biggest sub module of the cache named the fetcher is actually realized in SDA but also a study to problems that have occurred and will occur during hardware realisation and simulation is made.

The problems occurred as a consequence to the unawareness to the possibilities of the SDA system. This includes the not knowledge about Top Down designing in SDA, the IL SDA language and the available translator programs. Other problems are the results of the fact that IDaSS was not especially intended to be a part in a silicon compilation traject.

The conversion of the Fetcher part of the IDaSS design file into a low level implementation is done by hand. So an actual low level realization of an IDaSS design is possible but takes much time and is not very elegant.

Because the fact that there are no simulations in SDA done yet, it is impossible to guarantee the same performance as the IDaSS design in the hardware design. So the real value of the realisation can only turn out of the simulation results. This results can be optimized if some optimalizations in performance and used chip area are completed. But also some extentions of the IDaSS tool, such as the introduction of timedelay into schematics, can make the simulations in IDaSS more interesting in relation to the performance of the hardware realization.

1. INTRODUCTION

At the University of Technology, Eindhoven within the faculty of Electrical Engineering at the Digital Systems Group, a project is started called : ' Structured Analyzes and Structured Design' (SASD). The goal of this project is to develop design tools to handle the complex designs that are made possible thanks to the integration industry. To test the developed design tools of SASD a VLSI project called ' The C - processor ' was started. This C - processor is a dedicated microprocessor for effective execution of programs written in the C programming language or other related high level languages. The C processor must be a microprocessor with a high performance. The rate at which data must be supplied to the processor has become the limiting factor. In order to match the speed of data requests from the processor and data delivery from the memory on chip cache memories are used. One data cache and one instruction cache memory.

My graduation project has its main goal in designing a low level implementation of such an instruction cache. This low level design is derived from an existing IDaSS design, made by Y.C. Hu.[HU]

In this report the difficulties that appear in converting an IDaSS design into a low level implementation are studied and a beginning with the low level implementation is made.

ACKNOWLEDGEMENTS

I would like to thank my coach Ir. Willem-Jan Withagen for his help and support during my graduation project. Further I would like to thank Ir. Ad Verschueren for his explanations of some specialties of the IDaSS CAD tool. And also I would like to thank Ir. Leon Benders who did support me with a lot of usefull discussions about the IDaSS CAD tool.

TABLE OF CONTENTS

LIST OF USED ABBREVIATIONS

ALU	: Arithmic Logical Unit
ASCII	: American Standard Code for Information Interchange
ASIC	: Application Specific Integrated Circuits
CAD	: Computer Aided Design
CAM	: Content Addressable Memory
CPU	: Central Processing Unit
EDIF	: Electronic Design Interchange Format
FF	: FlipFlop
FIFO	: First In First Out
HHDL	: High-level Hardware Description Language
IC	: Integrated Circuit
IDaSS	: Interactive Design and Simulation System
LIFO	: Last In First Out
LRU	: Least Recently Used
MMU	: Memory Management Unit
MOM	: Multiple Output Minimizer
MOM2SDA	: Multiple Output Minimizer to SDA
MOS	: Metal Oxide Semiconductor
PLA	: Programmable Logic Array
RAM	: Random Access Memory
ROM	: Read Only Memory
S.A.S.D	: Structured Analyzes and Structured Design
SDA	: Silicon Design Automation
SDL	: Silvar Lisco's Structured Design Language
SSG	: Schematic to Symbol Generation
TSG	: Text to Symbol Generation
TTL	: Transistor Transistor Logic
ULSI	: Ultra Large Scale Integration
VHDL	: Very-high level Hardware Description Level
VLSI	: Very Large Scale Integration

2. THE CACHE MEMORY

2.1 INTRODUCTION

In modern microprocessor designs is the rate at which data must be supplied to the processor a speed limiting factor. To speed up this data delivering, several techniques are introduced such as pipelining and the use of a cache memory. Both techniques are used in the C-processor design, but in this chapter only an explanation on the general principles of a cache memory are displayed.

2.2 THE CACHE OPERATION PRINCIPLES

Cache memories integrated on chip have been shown to be an effective means of lowering average access time and reducing external bus traffic. The cache is a high-speed memory local to the processor. Traditionally, computers are build with one unified cache memory. All references to memory whether reads, writes or instruction fetches were handled by the same cache. This is the simplest arrangement. This leads also to the most efficient use of a limited resource and thus lower miss ratio compared to a split instruction / data cache. But there are also some disadvantages to a unified cache. Since in a highly pipelined machine, instruction fetches and data reads and writes are largely independent, access time can be shortened by splitting one unified cache into two separate caches respectively an instruction cache and a data cache. However a disadvantage of two separate caches is that the storage size for instructions and data is fixed.

In this project "A design of a C-processor" , two caches will be implemented on chip, respectively a data- and a instruction-cache. Both caches will be implemented on the processor chip so that they will be internal caches. However it is always possible to implement external caches in the C-processor system too.

Studies of modern programming techniques show that programs spend most of the time repetitively executing a few tight loops of code. (This is commonly known as the locality principle) Cache memory speeds up execution by holding loops just executed so that when they are used again they can be accessed much faster than if they were held in main-memory or back-up storage.

A Cache memory can for example be implemented in Bipolar static RAM technology (High Speed) and the Main memory will be implemented in MOS dynamic RAM technology.

The operation of a typical cache memory is relatively simple. When the CPU (or in the

on chip cache case, the instruction unit) requests data from the memory, the following steps are taken place :
(See figure 1)

The cache memory has a Tag RAM in which the first (n - k) bits of the address are held and a Data RAM in which the corresponding Data is held. If the address from the CPU matches an address found in the cache Tag RAM, the corresponding data at the cache Tag address is send to the CPU.
(This is commonly referred as a hit)

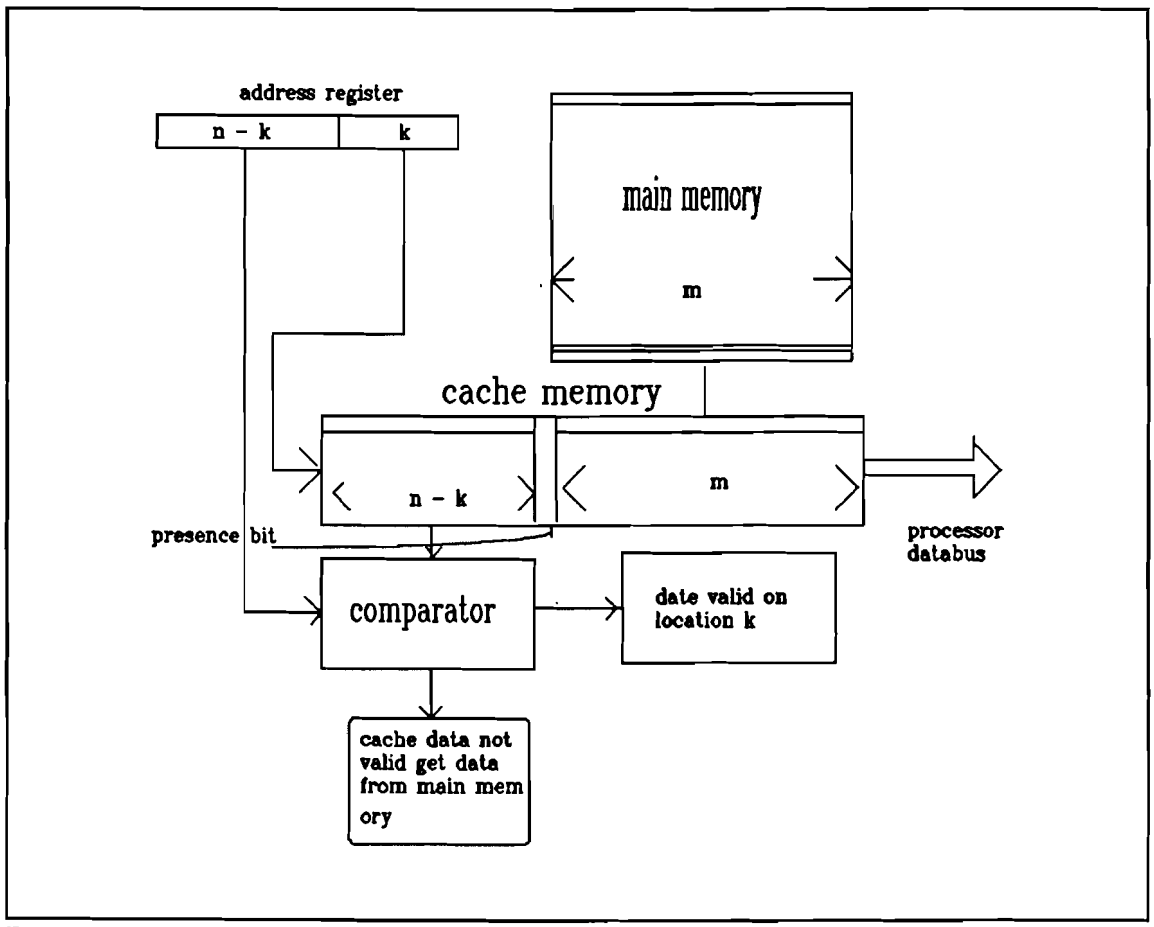


Figure 1 General Model of a Cache memory

When an instruction is fetched from external memory, an entry is also made in the on-chip cache. On making this entry the processor stores also a copy of the upper selection of the addressbus, which is referenced as the Tag field to the actual stored item.

When this address is accessed again, the processor is able to compare the stored Tag with the present addressbus state to determine whether the required instruction is in the

cache. If this comparison is true then the processor reads the instruction directly out of the on-chip cache without the need to go to external memory.

If the processor wants to write data in memory this will be done both in Cache- and main memory. So the main memory has at every time the correct information available. During writing in main memory it is possible to read in cache.

When the Read/Write operations are in the proportion of 6 to 1 there is only a little speed decrease. This 6 to 1 proportion is true in case of register machines but in case of a stack machine this proportion will be 3 to 1 which will led to a less effective use of the cache memory. [GEURTS]

3. THE USED CACHE MODEL

3.1 INTRODUCTION

The base for a low level implementation of the instruction cache of the C -processor, is an existing instruction cache design developed with the IDaSS CAD tool. This chapter describes the chosen architecture, chosen dimensions and chosen protocols. In this report groups of 8, 16 an 32 bits are respectively called bytes, words and quads.

3.2 THE SET ASSOCIATIVE MAPPED CACHE

The cache organisation which is chosen for the instruction cache is the ' Set Associative Mapped cache '. In a set associative mapped cache the memory address is divided into 4 fields respectively : Tagfield, Setfield, Transferblockfield and a wordfield. The chosen two way set associative mapped cache is illustrated in Figure 2.

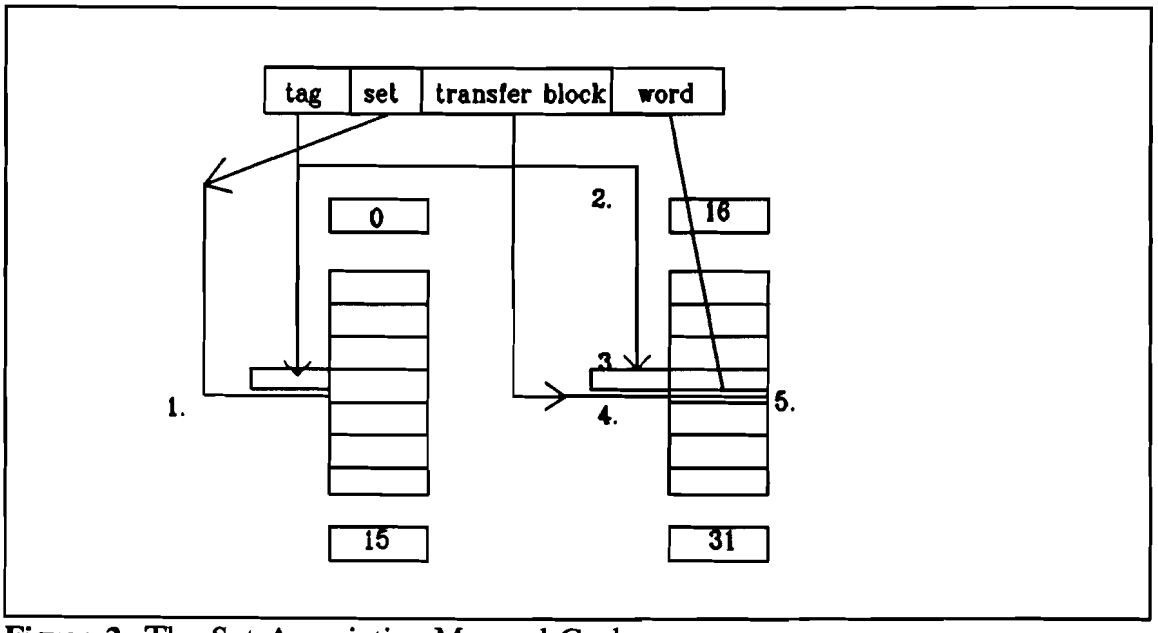


Figure 2 The Set Associative Mapped Cache

In the chosen organisation 5 phases can be distinguished in processing the request for data :

1. The set field points to the set of cache blocks that may contain the block with the requested quad.
2. The Tagfield is compared against the tags of the blocks in the set.
3. If one of the Tags match, the status information of that block is checked.
4. If there's a hit, the transferblock is used to select the transferblock within the block.
5. After the transferblock is selected, the wordfield selects the quad within the transferblock.

3.3 THE CACHE PARAMETERS

The performance of a cache memory is significantly dependent of some cache parameters, respectively :

1. The set size. The larger the set size the lower the miss ratio, but a smaller set size results in a faster and less expensive implementation.
2. The blocksize. The advantage of grouping quads in blocks is that the overhead such as the address tags and the replacement information decreases. The larger the block size the lower the overhead and there can be more adjacent data stored at once, a big block can be read in burst mode and there will be a smaller penalty for cache flushes, which occur in many caches during task switching. The disadvantages are : The larger the block the larger the miss ratio, because a complete block instead of 1 quad has to be fetched from main memory. And there will be quads loaded into the cache which never will be used (= memory pollution)
3. The transfer block size. Dividing blocks into transfer blocks is positive for the miss penalty and the traffic ratio but negative for the miss ratio.

3.4 THE PREFETCH ALGORITHM

There are 2 ways to fetch a block from main memory :

1. Demand fetch : the block is fetched when it is needed.
2. Prefetch : the block is fetched before it is needed.

The need for a prefetch algorithm are the high performance requirements of the instruction cache. Although prefetching always increases the traffic ratio, and also causes memory pollution, if there is too much information prefetched, prefetching makes higher performance possible.
Another problem is that the storage of prefetched quads will delay servicing of processor requests, unless multiported RAM is used.
This multiported RAM makes it possible to access the RAM for two (or more) read operations and one write operation during one clock cycle.
As long as there's no multiported RAM available, an arbiter is needed to control the access to the RAM.

There are several prefetching algorithms known and the one that is chosen is called 'prefetch_lookup'. This method is based on the absence in the cache of the next transfer block. This method however has the disadvantage of one extra cache access. Such extra accesses are likely to increase the average access time. To limit these extra accesses the method can be refined by only prefetching in case of a hit. (prefetch_lookup_on_hits)

3.5 THE REPLACEMENT ALGORITHM

The replacement algorithm has to decide which block in the set will be removed to store a main memory block. The replacement algorithm is the Least Recently Used algorithm. This LRU algorithm is based on the assumption that if a block has been recently (often) used, it is likely to be used in the near future. So if a block is not recently used it is a possible candidate to being moved out of the cache memory.

3.6 ARCHITECTURAL CONSIDERATIONS

The cache has to be able to service quad requests from the instruction unit/processor while it is prefetching. The best way to handle this parallellism is to use two separate modules. One which controls requests from the instruction unit/processor and one which controls the fetching of the quads. The first unit will be called the server and the latter will be called the fetcher.

To limit the increasing of the cache access time due to prefetching, is to decrease the duration of storing a prefetched block. This can be done by placing a buffer between the main memory and the cache memory. This buffer will indeed be used and is called the Fetchbuffer.

Other ways to decrease the cache access time are :

1. to limit the number of cache accesses by the fetcher.
   This is possible by using the fetchbuffer and by using the prefetch_lookup method. The latter method avoids unnecessary prefetches.

2. to limit the number of cache accesses by the server.
   This can be made possible by using a buffer between the cache and the instruction unit. This buffer will be called the readbuffer.

This latter method means that if a request from the instruction unit results in a cache memory hit, the total transferblock is copied in this buffer. The following quads do not require access to the cache memory, as long as they belong to the same transferblock and their valid bits are set. The valid bits are necessary because it is possible that only one part of the transferblock is loaded into the cache memory. But the use of a readbuffer has another, larger advantage and that is the faster read possibility in comparison with the cache RAM itself. This is because of the small size and further it does not need associative search.

Until now we 've seen that the cache consists of a fetcher, a server, a readbuffer, a fetchbuffer, a Data RAM and a Tag/Status RAM. But two more modules are added. First a status register which contains the tranferblock addresses of the transferblock in the readbuffer and in the fetchbuffer. Second a control unit, which collects the status signals from each unit in the cache and distributes them through the system. The schematic of the cache is illustrated in figure 3.

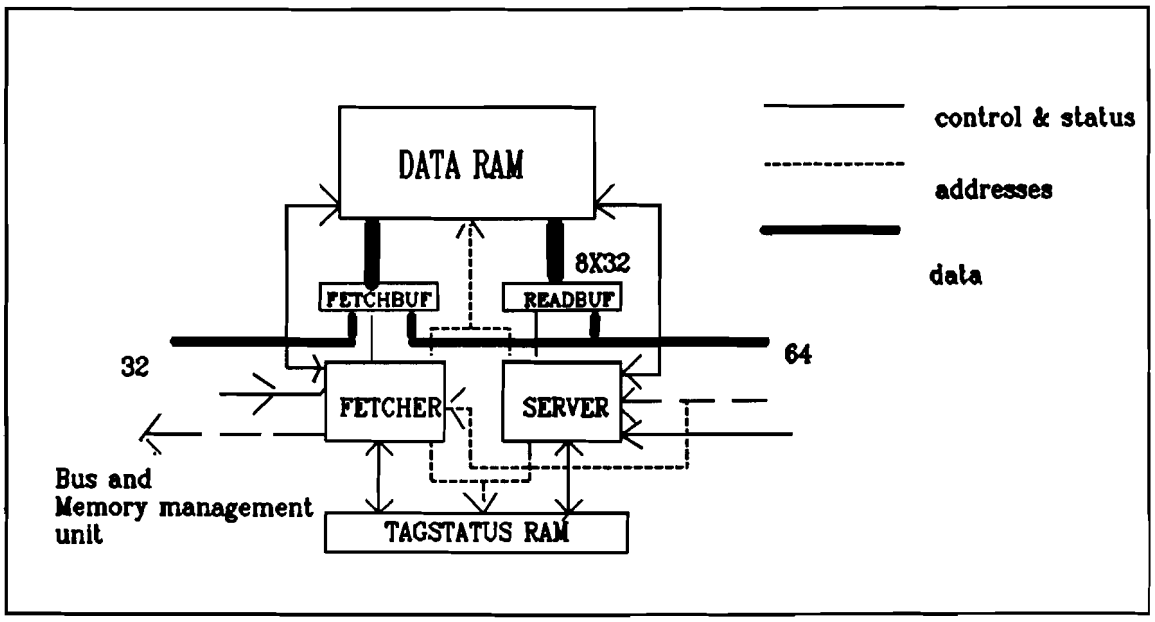


Figure 3 The Decomposition of the Cache in smaller modules

The separated Tag/Status and Data RAM is an effective organisation for several reasons. Because only one tag is required for all quads in a block and some statusinformation, like LRU bits is also required per block. But other status information , like data_valid bits are required per quad. Therefore it is better to use separated Data RAM and Tag/Status RAM. This approach has certain advantages, such as :

1. No wide RAM is necessary
2. The RAMs are smaller and thus faster.
3. Simultaneous access to data (quads) and tag/status information is possible
4. Different architectures can be used for the data and tag RAM.

Two different modules are drawn in Figure 4 to illustrate a two way associative set. The tag and status information of the same set do indeed have to be accessable simultaneously. So different status RAM's could be used. But since these will be very small, it is better to use one status RAM, containing tag and status information of all the blocks of one set in one row. This is depicted in Figure 4 for a two way set associative mapped cache with LRU placement and lookup prefetching.
Quads of different blocks within a set do not have to be accessable at the same time. Thus they can be placed on different rows of one data RAM.
The read- and fetchbuffer are exploited best when complete transferblocks can be copied to or from them. So a wide Data RAM is needed. An example of the data RAM of a cache with set size 2 is depicted in Figure ww. Blocks of 32 quads, divided over 4 transferblocks, are assumed. The address, which has to be supplied to the addressdecoder, succesively consists of the set field, the block field and the transfer block field.

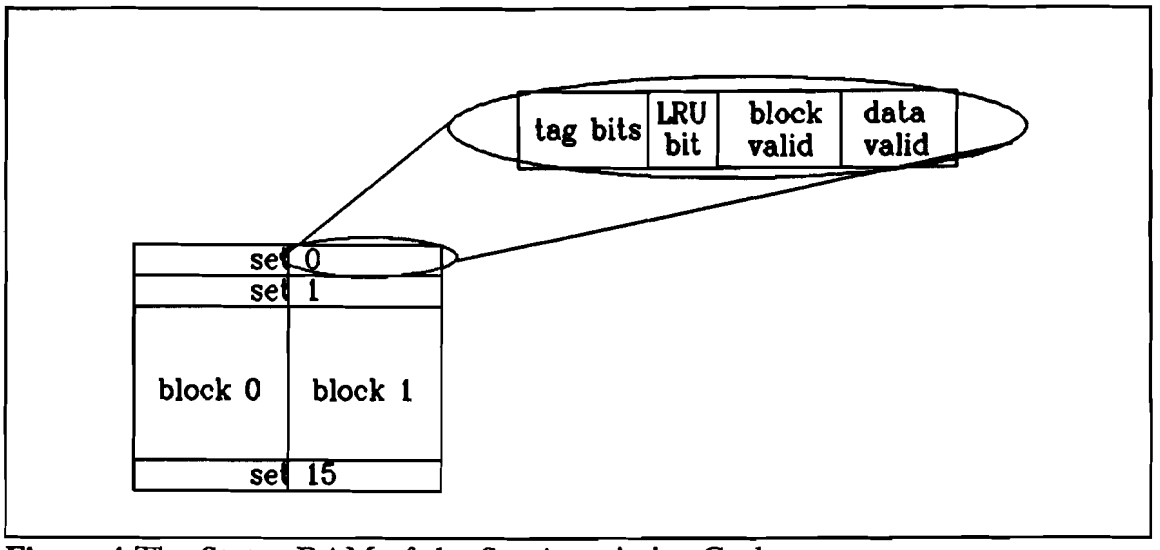


Figure 4 The Status RAM of the Set Associative Cache

3.7 THE USED INSTRUCTION CACHE SIZES

Out of the work of the predecessors the following parameter sizes and options are chosen and they will be used for the current low level implementation :

Cache size	: 1024 quads = 4 KByte
Set size	: 2 sets which contain each 16 blocks.
Block size	: 32 quads.
Transfer block size	: 8 quads.
PreFetch method	: Prefetch_Lookup_on_Hits.
Replacement strategy	: Least Recently Used.

A read buffer and a write buffer are also included to improve the performance of the Cache. [Bormans]

4. THE CONVERSION OF IDaSS OBJECTS INTO A LOW LEVEL IMPLEMENTATION

4.1 THE IDASS OBJECTS

IDaSS describes a design as a tree-like hierarchy of schematics. The schematics contain elements like registers, ALU's, memories, State Machine Controllers and other similar kind of elements. Those elements are entered graphically. In case of a well structured Top down IDaSS design, the following components can be found in the lowest level blocks of an IDaSS design :

1. **State Control**
   (Describes a state machine controller object)
2. **Buffer**
   (Describes an unidirectional tri-state buffer object)
3. **Constant**
   (Describes a constant generator object)
4. **Register**
   (Describes a register object)
5. **Operator**
   (Describes an operator object)
6. **RAM**
   (Random Access (read/write) Memory object)
7. **ROM**
   (Read Only Memory object)
8. **LIFO**
   (Last In First Out (stack) memory object)
9. **FIFO**
   (First In First Out (elastic buffer) memory object)
10. **CAM**
    (Content Addressable (Associative) Memory object)
11. **BUS**
    (Bus object)

Each component has its own problems in case of a conversion into hardware. These problems and their possible solutions will be described in this chapter.

So, this chapter contains some standard solutions (which until now must be executed by hand) for IDaSS objects conversion to a implementation on gate level such as in the SDA environment.

4.2 THE CONVERSION OF OPERATOR BLOCKS

Operators model all combinatorial elements in the schematic. The number of inputs and outputs are unlimited and completely user-defined in IDaSS. Operators can execute one or more user-defined functions. These are entered using a 'Function Editor ' window in which the user types the wanted function. This textual description must be converted into a low level description.

The logical function descriptions can use several expression operators as described in [VERSCHUEREN.2]. There are 3 types of expression operators respectively : Keyword Operators, Binary Operators and Unary Operators. A few examples are listed below. Because of the quantity of available expression operators respectively 17 keyword -, 27 binary - and 21 unary expresion operators, only the most important and most common expression operators are treated in this paragraph. The other expression operator conversions are in some way similar to convert as the examples in this paragraph.

```
address := 0.
addr    := %00010.
ad0     := %00001 inc.
_adres  := accu from:2 to:6.
_temp   := _adres at:0 width:8.
_tmp    := a /\ b  if0: c <>d  if1: c \/ f.
_t      := merge: addr from:0 to:7.
```

To make the functional descriptions less complex and more readable, there is a possibility of using temporary variables. Their names are user-defined, but they must start with a underscore. In case of an 1 on 1 conversion from IDaSS to hardware, this temporary variables can cause some redundancy and thus testing problems because redundancy is not testable. But the use of temporary variables will speed up the design of the low level implementation because the textual description will be less complex. There is a possibility to avoid the redundancy in our low level implementation and that is working out the functional descriptions to bitlevel. But this takes enormous amount of time because this must be done by hand in comparison to advantages that this will give. Other ways to increase the readability of (operator) blocks are the insertion of spaces, tabs and line feeds and also the possibility for the designer to add some textual comments after double quotes.

The operations in the operator blocks are described into one or more functions. Which function is executed during a clock cycle is determined by the controller(s) or a control connector, a default can be defined by the user. (The first function entered automatically becomes the default function) A control connector selects the functions asynchronously, a change in value on the control connector's bus will immediate select another function. Also, if the control connector is given a name, the value on the control connector bus

can be used in the internal calculations. In other words the control connector becomes just another input connector.
This means that the total behaviour of an Operator will be determined by two different, separated sections which even may not be physically connected to the operator block!

It's of course impossible to describe all possible convertion of all possible configurations in IDaSS and therefore are only a few of the most common expressions treated.

Some of the most used expressions are :

at: width:
from: to:
merge: from: to:

all these expressions are in a way convertable to bus tabs as illustrated in the figure below.

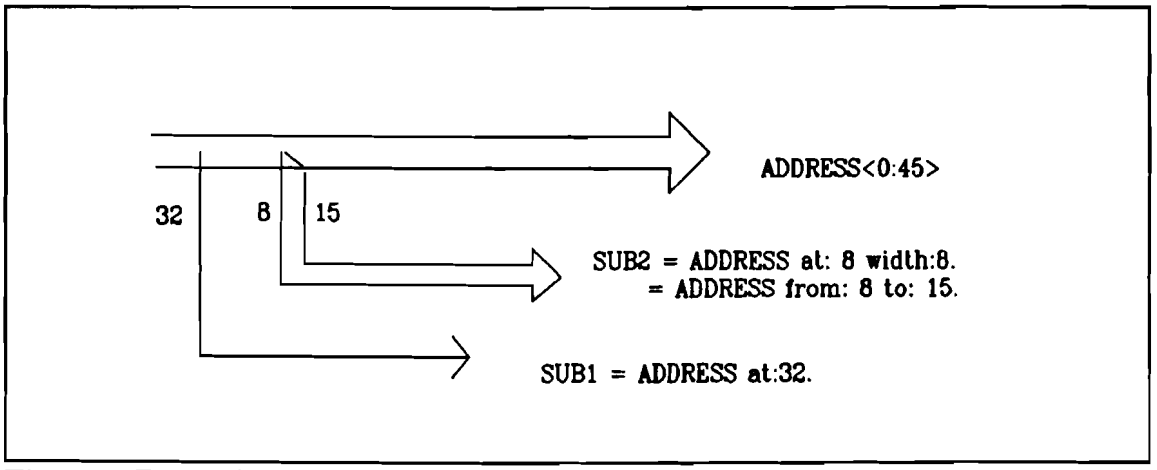


Figure 5 Bus Tabs

Expressions which consist of '/\' , '\/','> <', '< >' and 'Not' can respectively projected on AND-, OR-, XOR-, XNOR- and Invertor gates. Or some other available gates such as NOR, NAND etc.

Other expressions which use equal '=', unsigned compare less '<', unsigned compare more '>', unsigned compare '~=', signed compare less '+ < +', signed compare equal '+ = +', signed compare less equal '+ < = +' or '+ = < +', signed compare more equal '+ = > +' or '+ > = +', signed compare more '+ > +' and signed compare not equal '+ ~ = +' must be converted on real ,eventual 'Sign and Magnitude', comparators.

The description of the logic uses the selection statement **if0: if1:** . This conditional selection statement can be nested and this can lead to complex selecting logic in hardware. This selecting logic consists of multiplexers and eventual some gates or a PLA that realize the control connector function which provide the selecting bits of the multiplexers.

IDaSS operator example:

```
# Operator Voorbeeld
(has 2 functions and is controlled by a control connector)

Control specification

%00     default.
%01,%10 generate.


(Logical description of the 2 functions)

function Default

_in1 := ready
          if0: in1.
          if1: ((in1 + 1) width:5).
_in2 := in2 from:0 to:1 .

out := _in1,_in2.

function generate

_in1 := (in2 = 5)
          if0: 31.
          if1: in2 merge:(2 zeroes) from:2 to:3.

out := in2,_in1.
```

This IDaSS description can be converted to a low level realisation which is illustrated in figure 6. In this example it is obvious that IDaSS needs some extra description of determining the functions in comparison to the low level realisation.

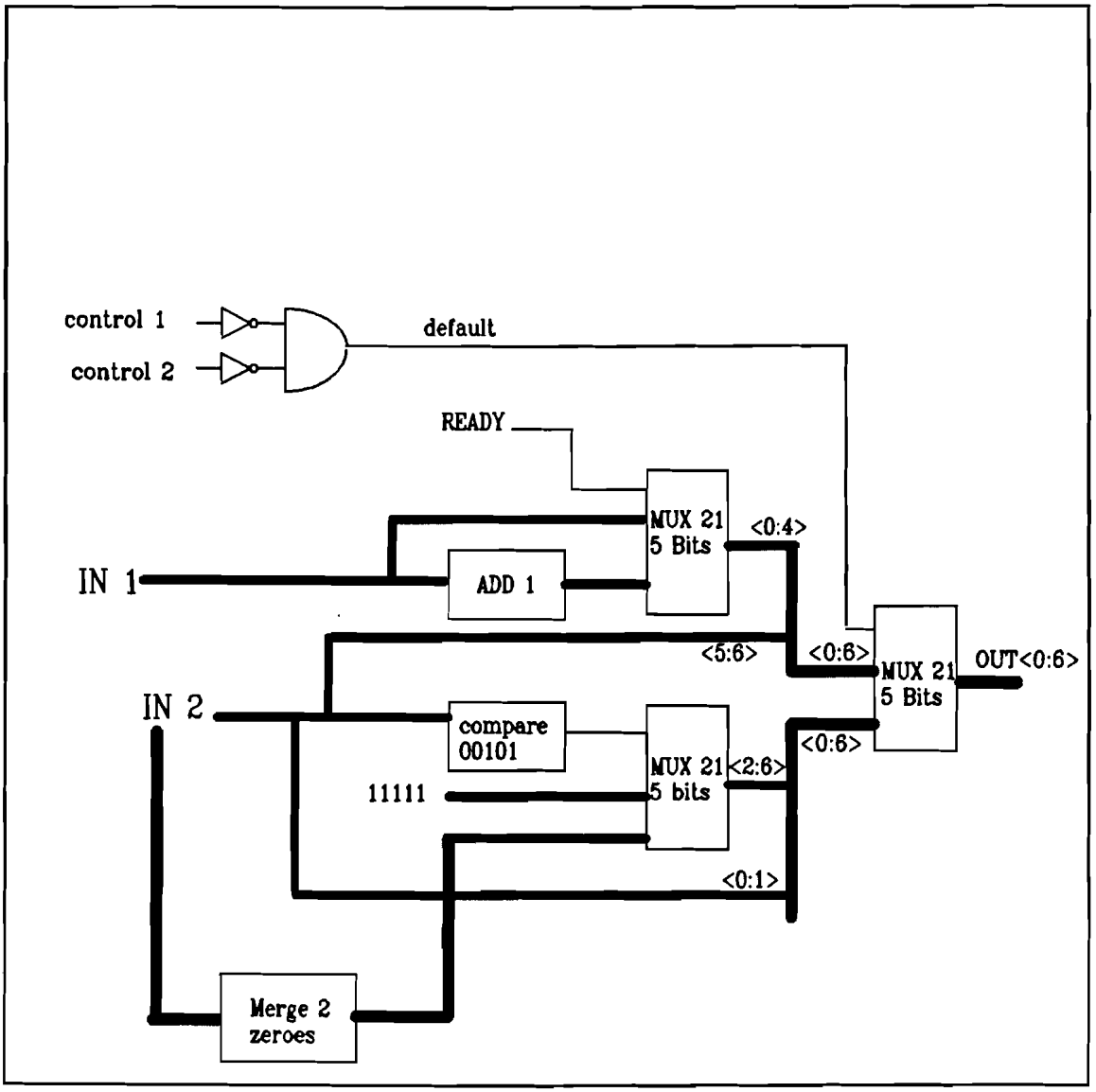


Figure 6 The hardware implementation of the operator example.

4.3 THE CONVERSION OF STATE MACHINES

4.3.1 The State Machines in IDaSS

In general there are two different types state machines respectively the Moore- and Mealy machines. The output of a Mealy machine depents on the current primary inputs and the current state of the state machine. The output of a Moore machine is only dependent of the current state of the state machine.

The state machines in IDaSS (normally) are defined in State Control Blocks. But they are not pure Moore or Mealy machines. Normally a state machine which is described in a StateControl Block is a hybrid Moore/Mealy machine. It has Moore and Mealy Outputs. However the Mealy outputs may be the same as the Moore outputs. Normally spoken are only Moore machines implementable in IDaSS because the inputs of a State Control block must be clocked in via een register or some other clocked object.

It is however possible to create (near) Mealy machines. This 'near' is added because the inputs of a State Control block must be clocked in, for instance via a register.

Because of the disadvantages of the use of Mealy machines we give preference to Moore machines. Because the use of Mealy machines can give cause for critical races and hazards. In the case that the combinational logic of a first Mealy machine is connected directly into combinational logic of a second Mealy machine, whose combinational logic is directy connected into logic of a third Mealy machine whose logic is connected into the logic of the first Mealy machine, then there will be a very long critical path. This is also a serious problem. But should this be the case in an IDaSS design then the simulation would have been stopped automatically. So we know for sure that this is not the case because of the correctness of the IDaSS simulations.

Although we give preference to Moore machines are nevertheless Mealy machines used in the instruction cache design. This is done to get an acceptable performance because Mealy machines can activate (asynchronous) signals within one clockcycle.

For getting a better judgement in an IDaSS description of statemachines, there are a Moore- and Mealy machines defined theoretically, worked out in IDaSS and realized in a low level implementation.

4.3.2 The IDaSS Mealy machine

A Mealy machine can in general be described as illustrated in Figure 7.

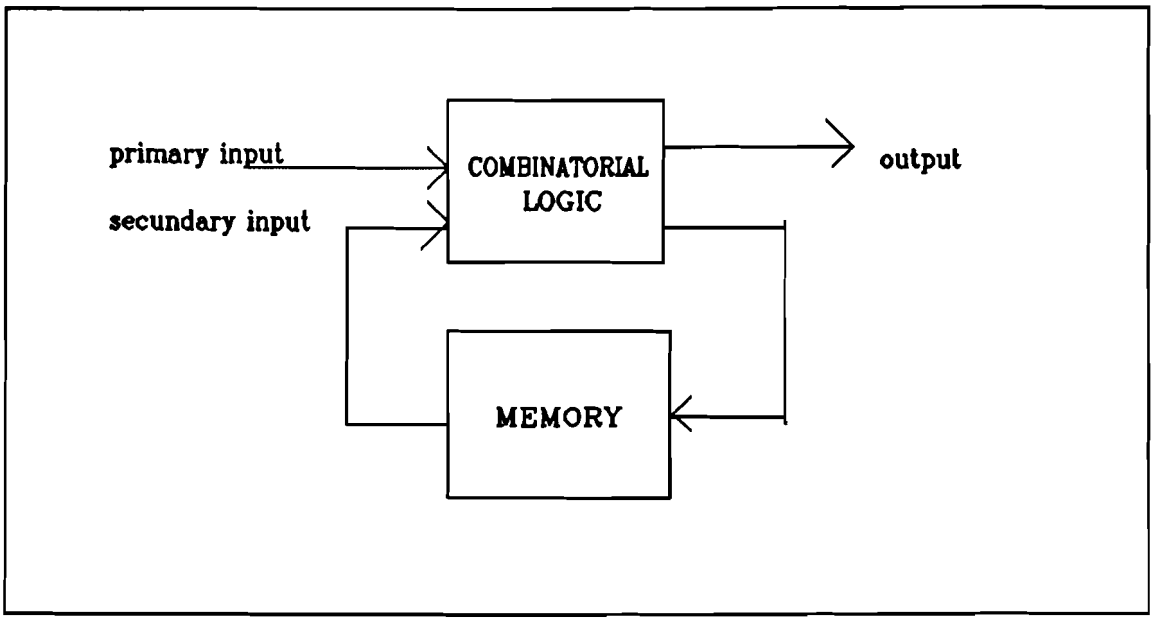


Figure 7 A General Mealy Machine

The first Mealy machine example that will be worked out has 3 states and has state transitions as described in the state diagram (Figure 8) and state table (Table 1.)

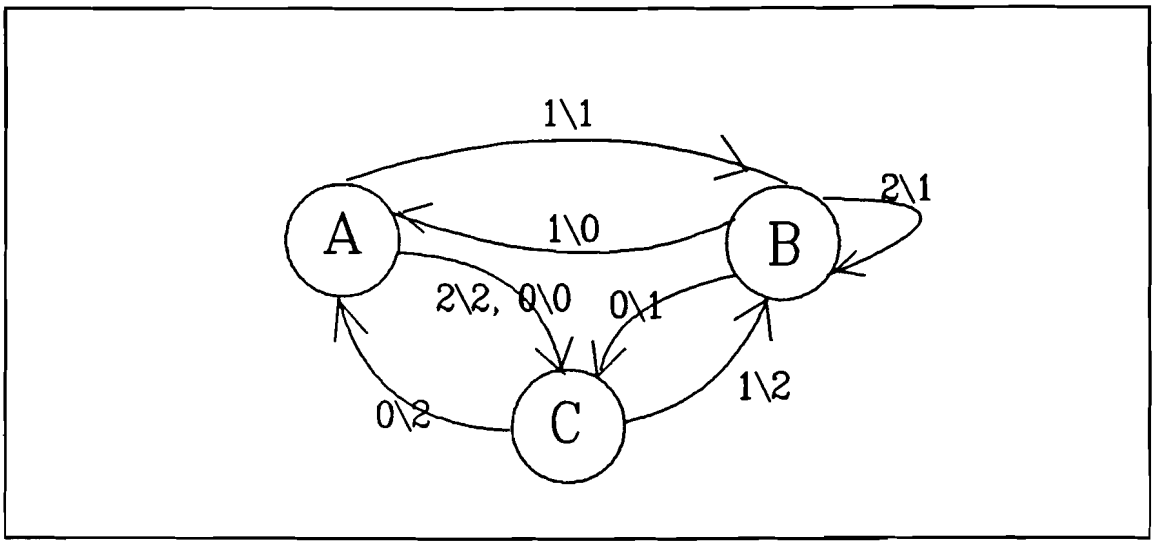


Figure 8 The State Diagram of the Indirect Mealy Machine

Table I The Mealy State Table

		Pi1,Pi2		
		00	01	10
00	A	C,00	B,01	C,10
01	B	C,01	A,00	B,01
10	C	A,10	B,10	-,--

In IDaSS this Mealy machine will be a variation of Figure 7. The memory in Figure 7 is now replaced by a register which only job is to hold the old state. The conversion of this kind IDaSS configuration will led to a normal Mealy machine realisation.

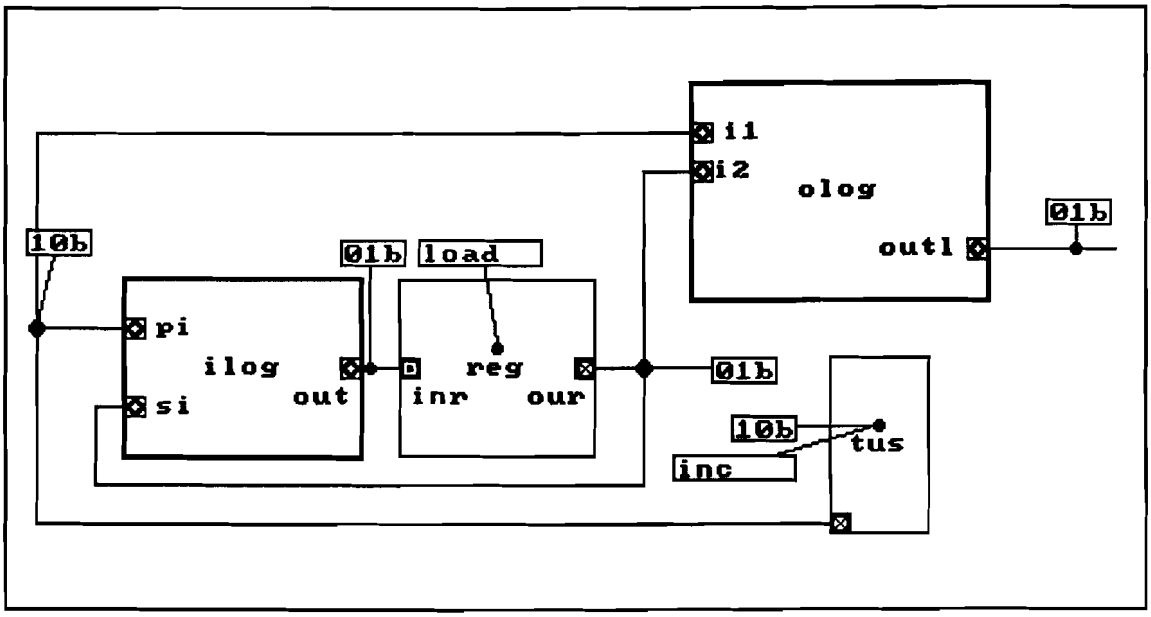


Figure 9 The IDaSS (indirect) Mealy Machine

The state table of the machine is in IDaSS implemented in some operator functions. In this case the output and the input operator blocks consist of 3 functions, respectively named olog and ilog.

Table II The ILog Olog Operators Table

Pi	Si	Out	Function	In1	In2	Out	Function
00	00	10	a	00	00	00	f
01	00	01	b	01	00	01	g
10	00	10	a	10	00	10	h
00	01	10	a	01	01	10	h
01	01	00	c	01	01	00	f
10	01	01	b	10	01	01	g
00	10	00	c	00	10	00	f
01	10	01	b	01	10	01	g

The other part of the state transition description is placed in the control connectors of the operators. In this case the description of the control connectors of the both operators is placed below.

Control connector of the operator ILog :

```
%0000, %1000, %0001  oi := 2.
%0100, %1001, %0110  oi := 1.
%0101  oi := 0.
```

Control connector of the operator OLog :

```
%0000, %1000, %0001  oo := 2.
%0100, %1001, %0110  oo := 1.
%0010  oo := 0.
```

The control connector function can be realized in a PLA or some wild logic. This wild logic then contains inverters, AND's and OR's. The register needed to implement the memory function of the described example must have (at least) 2 bits and can for instance be realized with two D FlipFlops. The first FF must be set if 10 should be loaded into the register. The second register must be set if 01 must be loaded into the register.

So the description in IDaSS is complex to understand. This is the result of the split notation of the behaviour of the state machine. The simpliest way to achieve a low level implementation seems to start from the state table. The state table however, is not one unified part in an IDaSS description itself or in the corresponding document file that can be made of an IDaSS design file.

A possible hardware realization is illustrated in Figure 10.

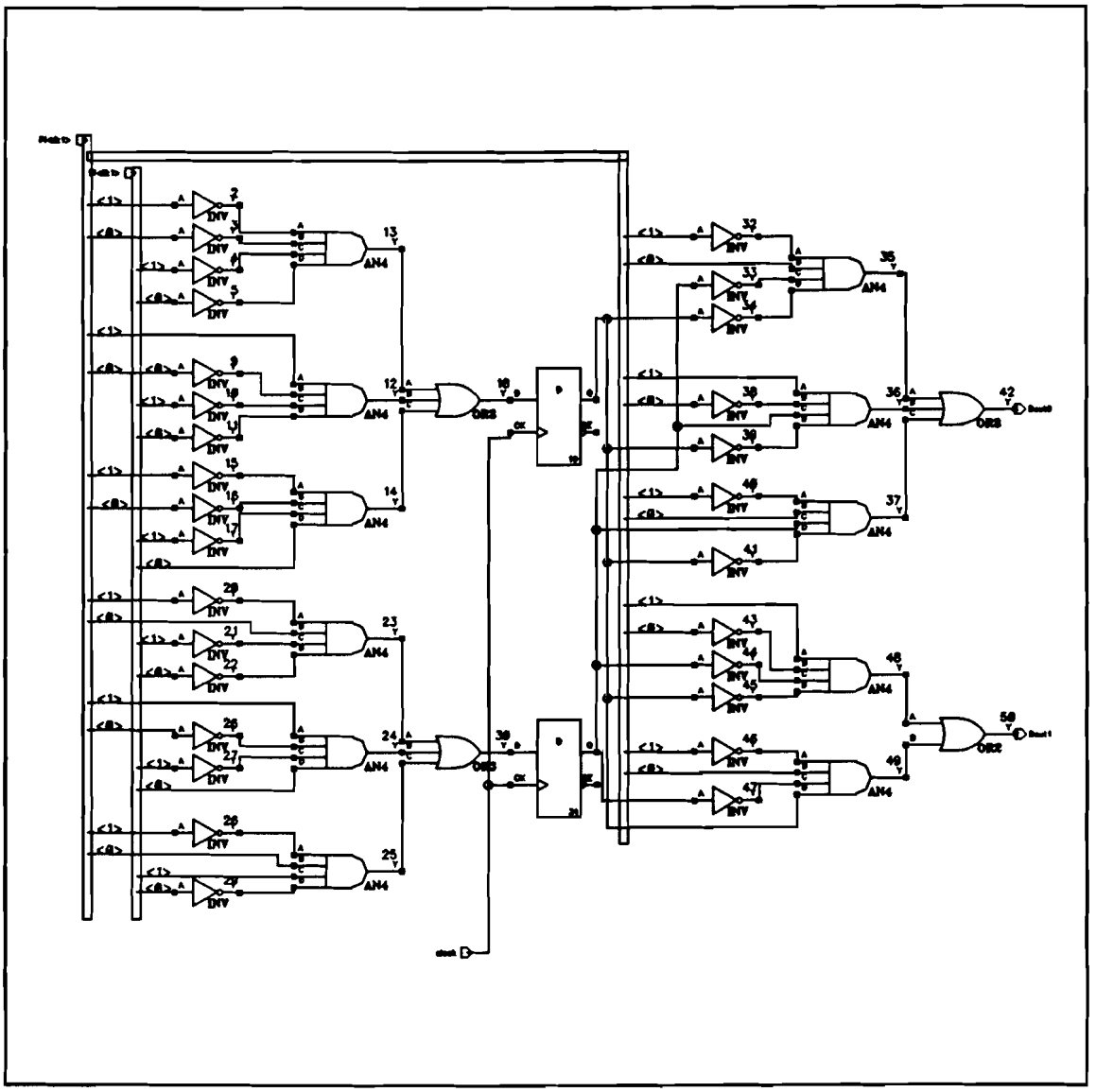


Figure 10 The Mealy Example Machine Implementation

4.3.2 The IDaSS (direct) Mealy machine

This second Mealy machine example is a more direct way of constructing a Mealy machine in IDaSS. Because the first example did not use the State Control block that especially was ment for implementing state machines in the IDaSS environment. This type of Mealy machines is used several times in the instruction cache design, especially in the fetcher. The PreFetcher, DemandFetcher and FController State Control blocks are submodules of the Fetcher and are also implemented in hardware. This type of more direct Mealy machines can be illustrated as in the Figure 11.
The FController is as an example used in this paragraph. The FController has 4 states

respectively Rest, DemF, DemF3 and PreF.

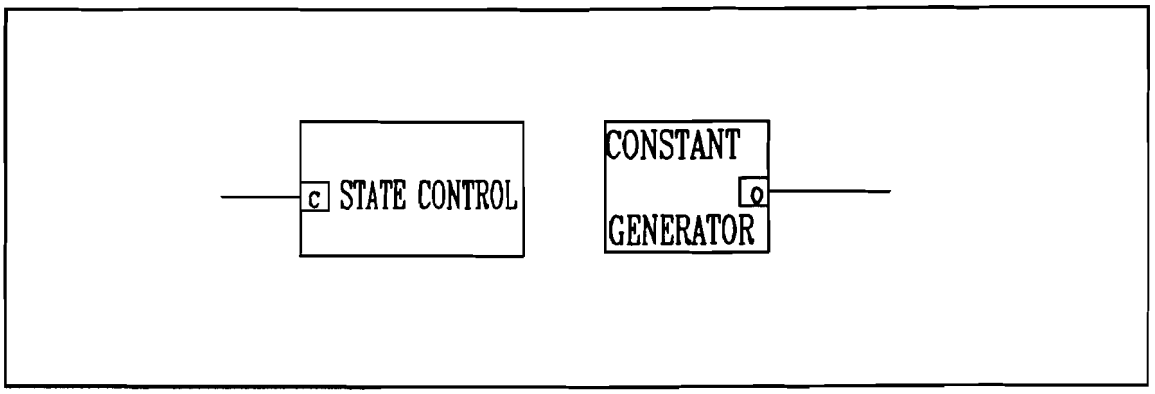


Figure 11 A 'direct' Mealy machine in IDaSS

Within a state a 2 bit Constant Generator named 'DemandorPre' will be given a value. The value that is given to the Constant Generator is equivalent with the code given to the states. In IDaSS there are two possible state transitions with different priorities respectively : State transitions indicated by control connector inputs or state transitions indicated by an clock transition. The latter transition has a lower priority. This means that in case of a simultaneous appearance of both kinds state transitions the state transition that is implied with the control connector will get preferential treatment.

The IDaSS description of this kind of statemachines take place in two separated sections. The first is a control connector section and the second is a State Control block. The description of the FController is as following

Control connectors

```
% X0XX1001010, % X0XX1001100                                  GOTO:Rest.
% X1XXXXXX00X, % X1XXXXXX10X, % X1XX100101X    GOTO:DemF.
% X1XX0XXX01X, % X1XX100001X                                  GOTO:DemF3.
% 100X000000X                                                 GOTO:PreF.
```

State Control Block

```
Rest: DemandorPre setto:00;
      < <
DemF: DemandorPre setto:10;
      < <
DemF3: DemandorPre setto:11;
      --> DemF
PreF: DemandorPre setto: 01;
      < <
```

This can be translated in a state table. To reduce the complexity the control connector inputs that imply state transitions to Rest are named input A. In the same manner are the inputs to DemF called B, to DemF3 are called C and to PreF are called D. The state transitions that occur in case of a clock transition are packed under E which is equivalent with :

E = NOT A and NOT B and NOT C and NOT D.

Table III FController State Table

	A	B	C	D	E
Rest	Rest	DemF	DemF3	PreF	Rest
DemF	Rest	DemF	DemF3	PreF	DemF
DemF3	Rest	DemF	DemF3	PreF	DemF
PreF	Rest	DemF	DemF3	PreF	PreF

The only conversion problem after constructing such a state table is the real hardware implementation. In this case some logic and two Flip-Flops are sufficient to make a low level implementation of such a Mealy machine.
Also in conversion of IDaSS state machines it will appear that IDaSS uses more description than strictly necessary for a hardware realisation. In hardware only minterms,

or optimized derivations have to be realized. In other words the functions in hardware describe not only the situations in which bits must be set but also the situations in which the bits must be reset. IDaSS needs two separate functions to describe when bits must be set and reset. The hardware realisation is used in the SDA implementation of the fetcher and is described in the files fcontsd and fcontsdmw. See Figures 12.
The implementation of the DemandOrPre constant generator is drawn outside the state controller block fcontsd and together they are included in the module fcontsdmw.

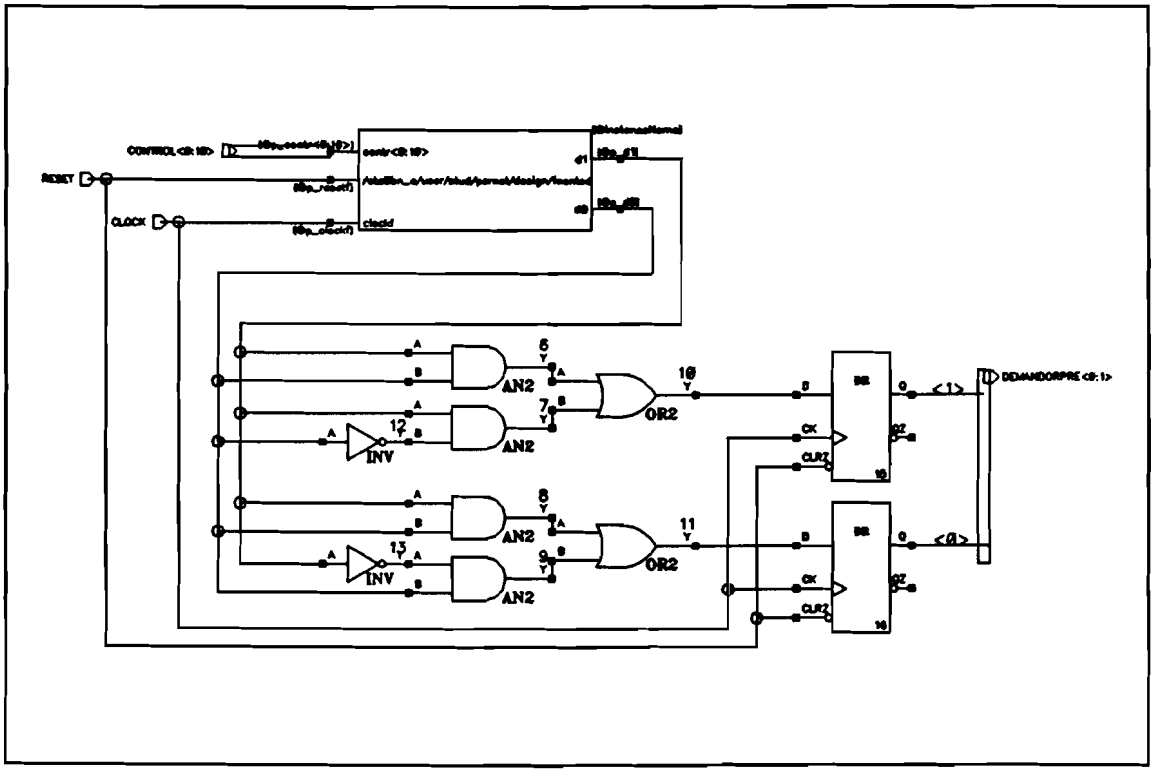


Figure 12 fconsdmw statecontroller with DemandOrPre

4.3.5 Real Moore State machine in IDaSS

A real Moore description in IDaSS has no control connector inputs but only state transitions implied by a clock transition. This looks just like a simplified version of the examples that are described in the previous sections. But a conversion of such Moore state machine can be difficult as will be made clear with the next pure Moore machine.

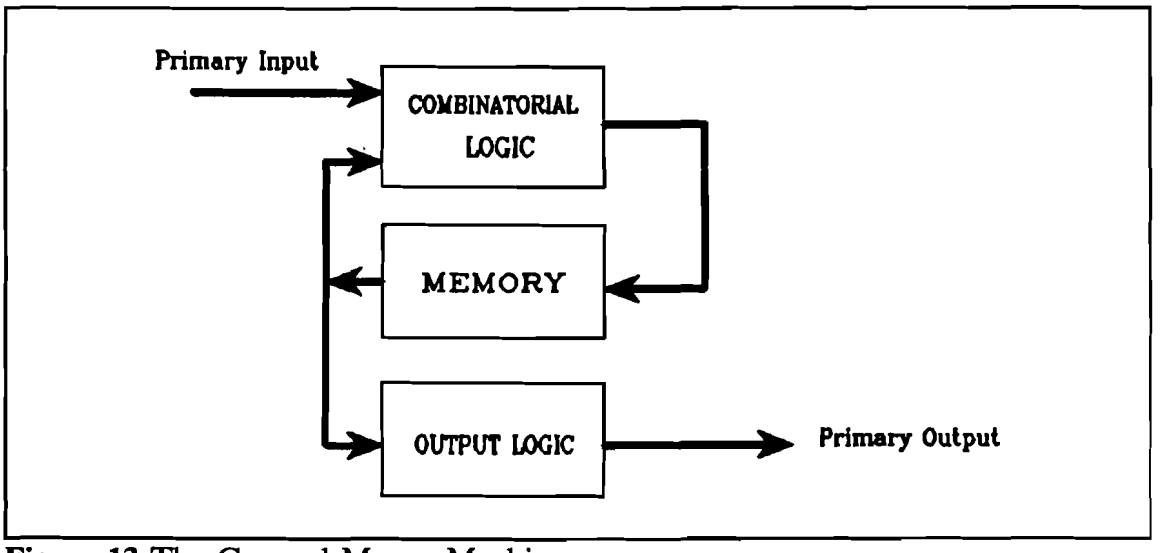


Figure 13 The General Moore Machine

This state machine is a description of a bus arbiter which must control the communication of two systems that will use one shared bus. The IDaSS schematic is described in Figure 14.

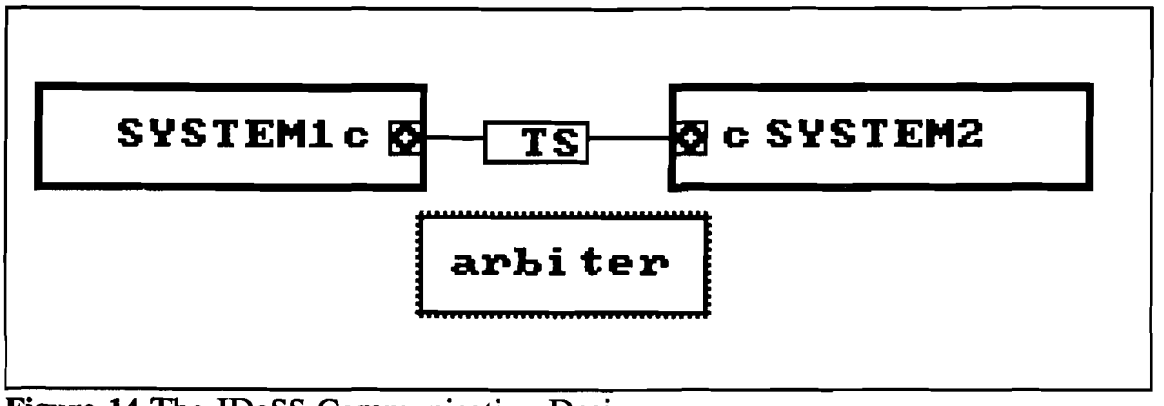


Figure 14 The IDaSS Communication Design

This schematic has no connectors and contains the following blocks :

'Arbiter' is a state machine controller
'System1' is a schematic
'System2' is a schematic

The schematic contains one 8 bits bus shared by the system1 and system2 schematics.

The Arbiter has two states Prio1 and Prio2 which are described as :

```
Prio1:
[SYSTEM1\OUT?? | SYSTEM1\out enable;
               SYSTEM2\in load;
               --> Prio2 ];

[SYSTEM2\OUT?? | SYSTEM2\out enable;
               SYSTEM1\in load ];

<<

Prio2:
[SYSTEM2\out?? | SYSTEM2\out enable;
               SYSTEM!\in load;
               --> Prio1 ];
[SYSTEM1\out?? | SYSTEM1\out enable;
               SYSTEM2\in load ];
<<
```

Communication \SYSTEM1 and \SYSTEM2 are schematics which are equivalent and they each contain the following blocks and buses :
(See Figure .)

'c' is an off schematic connector,
'control' is a state machine controller,
'gen', 'IN', 'OUT' are registers.
'bus', 'gen' buses each 8 bits wide.

The state control 'control' has two states and is described as :

```
generate:
[ gen epty /\ (gen /\ 3) = 0)
  "even parity and not value 00000011 "
| out load;    "out write bit set "
--> wait
];
<<
```

wait:
[out? =1 | gen hold; <<];
--> generate

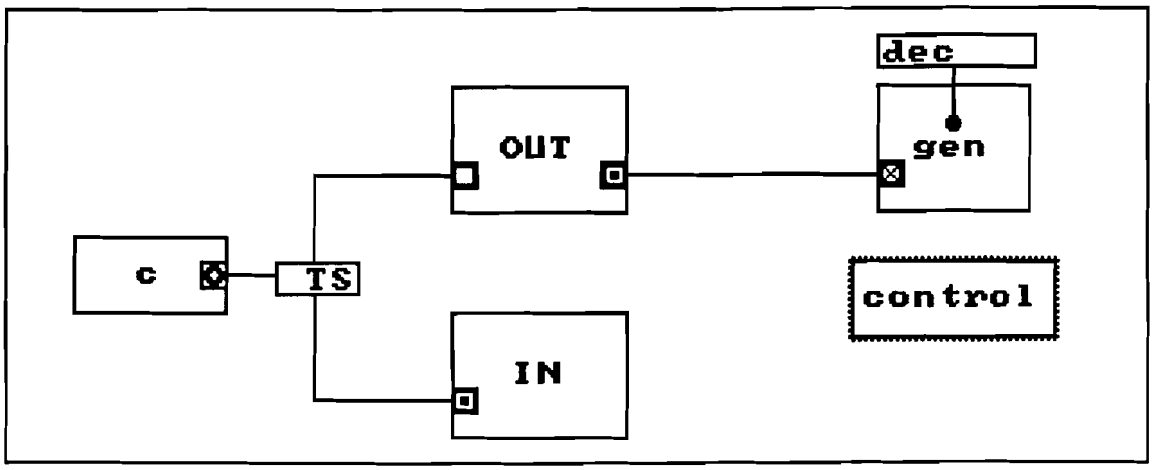


Figure 15 The SYSTEM Schematic

The difficulty in this Moore example is not specific a design hierarchy of two level which both include state control blocks but the two used state control blocks are making a easy low level realisation difficult.

The state control block 'control' has two states which have each two possible state transitions as illustrated in Figure 16.

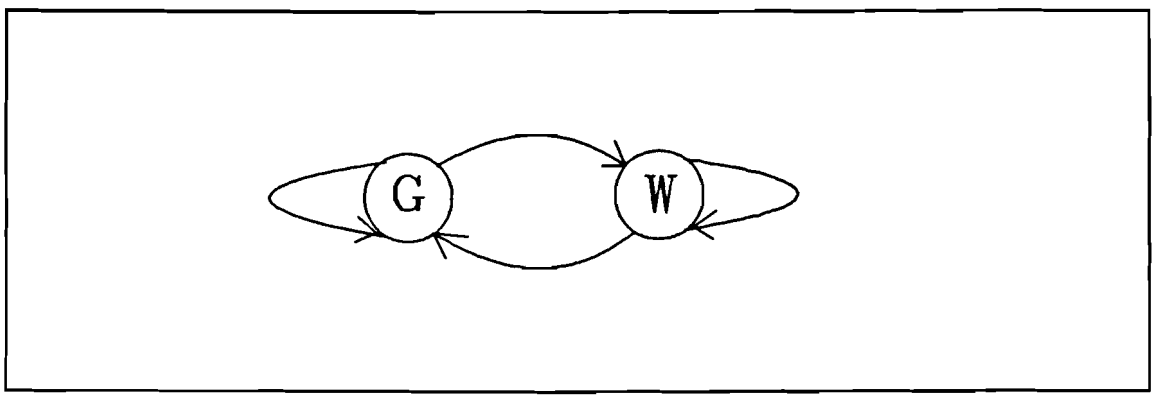


Figure 16 State Table of 'CONTROL'

If the 8 bits registers IN, OUT, GEN are implemented with 8 bits registers wich have a 'Read/Write bit', 'enable output pin' and a 'load' pin then the implementations of both state controllers can be illustrated as in the figures 17 and 18.

The statement <blockPathName>'??' that is used in the form SYSTEM1\Out?? and SYSTEM2\Out'??' is the textual description of the register semaphore with auto reset. This is a kind Read\Write bit. When this bit is set then the register is written in and the

register may be read. So with the help of this bit the 'Arbiter' can determine which register is being written in. After this check the arbiter will set the output enable of that written register and will also set the load data bit of the IN register of the other system.

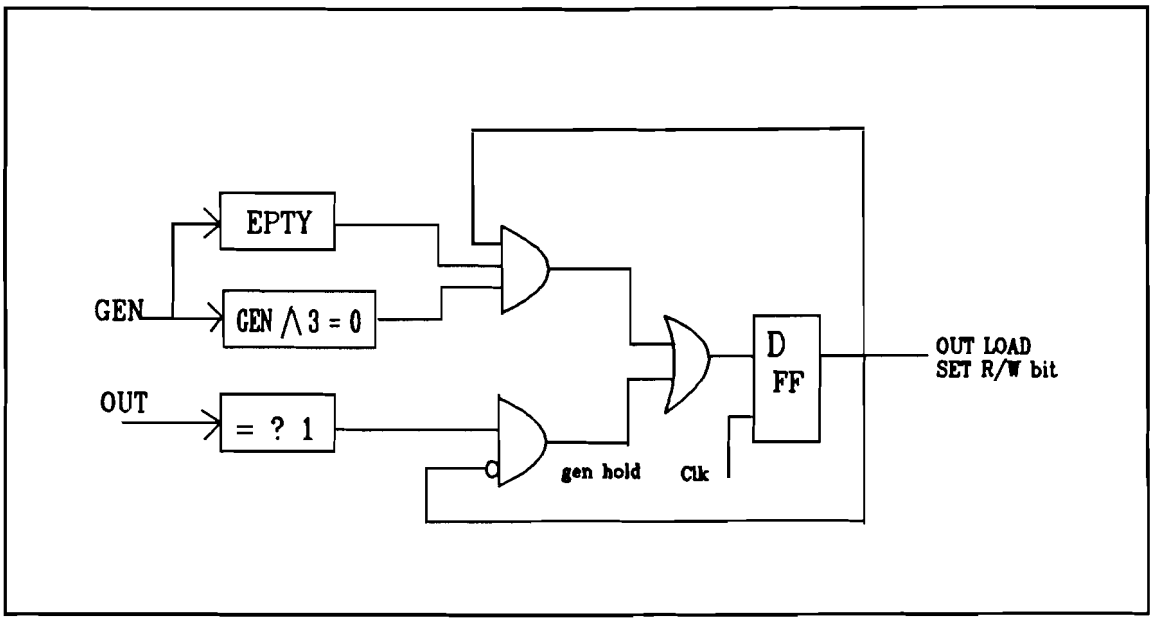


Figure 17 Control State Machine Implementation

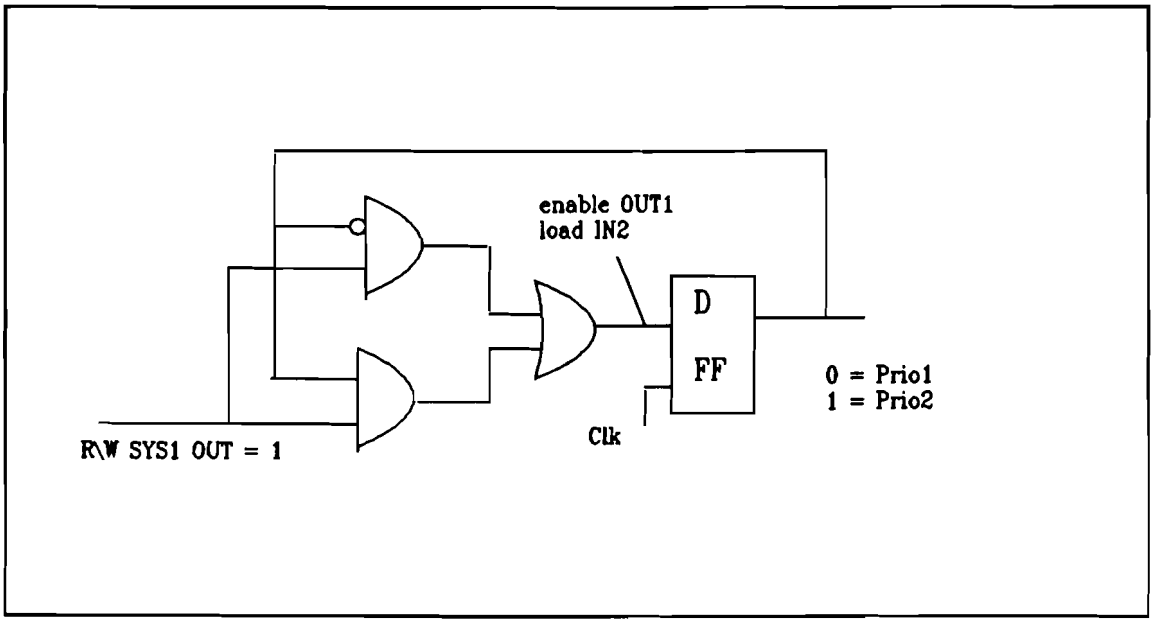


Figure 18 State Machine Arbiter Implementation

4.5 CONVERSION OF SIGNAL BLOCKS

The blocks containing Signals are relatively simple convertable into a low level implementation. Controllers can communicate using a system global dictionary with 'Signals'. These are one bit semaphores which can be set and reset by all controllers in the hierarchy. Signals can be created, removed, inspected and edited by using one or more 'Signal Editor' windows which are opened from the window menu of the toplevel simulator window.

There are currently four kinds of signals operational :

1. **Pulsed Only** signals can be given the command to set themselves high during the next clock period, then they will automatically fall back (if not given another pulse command with !)

2. **Level Only** signals can be given the command to set themselves high ('!!') or reset themselves low ('/') at the next clock (setting takes precedence) If left alone, they will keep their state indefinitely. It is possible to combine testing this signal with giving a reset command (with '??')

3. **Level / Pulse** signals are a combination of the previous two types where the level mode is considered to be the most important one. Giving a pulse command to a signal of this type will not reset the signal following the next clock period, if the signal was set by a set command. Resetting will be done, however, if the signal was set by a previous pulse command.

4. **Pulse / Level** signals are a combination of the types 1 and 2, where the pulse mode is considered the most important. Giving a pulse command to a signal of this type will reset the signal following the next clockperiod, if this signal was set by a set command.

The implementation model for the Level/Pulse and Pulse/Level signal uses two separate Flip-flops. One of them is used to handle the pulsed operations and is called the 'pulse' flip-flop. The other handles the level operations and is called the 'level' flip-flop. The output of these signals is the 'OR' function of both flip-flop outputs.
The pulsed only and the level only types of signals can mange with one flipflop.The implementation models of these 4 possible kinds of signals are shown in the four figures below.

The pulse flip-flops are always rest following system reset. The level flipflops can be put in an user-defined state following system reset. All command inputs coming from controllers are 'OR ed' into signal input lines. And in case of the simultanous occurring of Set and reset in a S-R FF the set input is given a priority.

Al this possible implementations are illustrated in Appendix C.

4.6 CONVERSION OF OTHER IDaSS BLOCKS

The blocks discussed in the previous sections, the State Control and Operator blocks, are one kind of blocks that need to be sorted out. The second kind of IDaSS blocks are the kind that need not to be sorted out that much. So they are almost direct projectable on hardware.

The blocks of the second kind are :

Registers :

The registers are direct projectable on hardware. But if IDaSS control connectors are used some (PLA like) logic must also be included in the hardware design.

Buffers :

Not only registers but also some selecting logic must be included to follow the whole functional description as purposed in the IDaSS object.

Constant Generators :

Can be implemented in a kind of Flip-Flops, registers or only some gate logic. The implementation is depandant of the context in which the constant generator blocks are placed.

Because if a constant generator is used together with a State Control object the constant generator object can be set to a value within a state of the state control block without any physical connection. In this configuration a one bit constant generator can be implemented with some logic which has of course a physical connection to the output of the state machine. The output of the state machine is a representation of the current state of the state controller from which the value that must be given to the constant generator can be evaluated.

Also registers can occur in this kind of configuration with a state control block. This means that some logic must connected to the output of the state control block which must take care about the correct setting of the register bits.

Connectors :

Can be implemented with input and /or output terminals.

RAM and ROM blocks can be implemented in the low level SDA environment with the available RAM and ROM generators.

LIFO and FIFO can be projected directly on sets of registers which must be placed in LIFO or FIFO configurations. And which must be controlled by some included control logic.

The buses in IDaSS are biderectional and can be shared by several 3-state outputs (of which only one may be active at a time) Buses can be converted directly into a hardware realization.

These are almost all problems that occured during the conversion of the fetcher of the instruction cache. This implicates however not that all possible hardware implementation problems have occurred. To get a complete list of all possible IDaSS conversion problems it is neccessary to add the new occurred problems to the list after each conversion that is made until there are no more problems found.

4.7 CONCLUSIONS

In this chapter the most common control expressions that are used in the IDaSS Instruction Cache design are treated. In general, every object showed to be convertable. But this conversion, which was done by hand, is very time consuming. In the near future a lexical scanner/compiler should be developed to reduce this conversion time.

Especially the State Control and Operator blocks took most of the conversion time. Another time consuming convertion factor is to manage with the complexity of an IDaSS design. The document file should care more about the overall views and connections between blocks. Therefore should all redundancy be removed from the document files. All this redundancy does increase the complexity and thus increase the conversion time.

A judgement of Signal conversion is not made because this kind of objects are not used in the IDaSS Instruction Cache design.

5. HARDWARE DESIGN IN SDA

5.1 STRATEGY

The low level implementation of the instruction cache [HU] is derived from an IDaSS implementation because of its availability and functional correctness. Therefore a lexical scanner is written for extraction of usefull information from the IDaSS design file.
This scanner is a small PASCAL program which reads an IDaSS design file and detects from the first characters of a design file respectively : name, width, kind of object, nesting and kind of connector. See Appendix B.

But currently there's another, much more extended version of an IDaSS design file extractor available. This program is written in the SmallTalk environment just as the IDaSS system is under. The latter program expands an IDaSS design file to three times his originally size. In case of the instruction cache the expanded design file, the document IDaSS file, has a size about 600 KB.

This program generates an IDaSS document file which basically contains a list which describes the used objects, schematics, control connectors and their corresponding names and eventually the documents are extended with textual comments added by the IDaSS designer. This all makes a document file easier to read.

It is possible to generate (partial) document files from one (or more) schematics only, but if there is a need for all possible information from a whole design, a document file must be generated at the toplevel of the IDaSS design. This document file will further on be referred as the toplevel document file.
The toplevel document file can be used as a platform to develop a low level SDA implementation of an IDaSS design. Such a toplevel document contains all information of the design. It is also possible to create partial document files which contain just the information on the selected block or schematic you needed.

There are two options in this stadium to derive a low level implementation out of an IDaSS design.

1. Developing an automatic translation and conversion program of an IDaSS design file into an SDA format.

2. The second option has its main task in deriving a low level implementation and eventually will result in the design of some conversion tools. The latter is however not a target on itself.

If the first option was chosen then this master thesis report could have been filled with a set conversion tools. This could more or less have freed the difficult road for successors in the project (or some other project) whose job it will be to make a low level implementation of an IDaSS design.
However, the second option was chosen which makes the low level implementation the main goal.
And this option brings a real hardware realisation possible in the very near future. But this option has also two possible options respectively :

1. Inventarisation and solving of IDaSS conversion problems which occur in this specific IDaSS design of an instruction cache of the C - processor.

2. A real low level implementation of the instruction cache in SDA.

The choice that is made is a mixture of both last options. This can be described as an inventarisation of all conversion problems which occur during the low level implementation of a submodule of the instruction cache of the C - processor. The sub module that for this purpose is chosen is the most complex and biggest submodule of the instruction cache, namely the fetcher. See Chapter 6.

5.2 THE SDA PRIMITIVES

The main goal is to derive a low level implementation of an instruction cache which is already described and simulated on a higher level with the IDaSS CAD tool. This low level should be realised with the Silicon Design Automation software package. This SDA package is installed on a APOLLO Domain Ring network under UNIX. There are currently two versions of SDA available respectively the 2 and 1.5 micron version. This means that the ES2 library, which contains the basic building components, can be projected on respectively 2 or 1.5 micron technology. This ES2 library contains (N)AND's, (N)OR's, Inverters, Buffers, 2 bit adders, 4 bit registers, (De)Multiplexers, several kinds of Flip-Flops and some more complex gates such as AND-OR and OR-AND. Within the ES2 library a ES2CELL2 sublibrary is available which contains most of the TTL family components. This includes fast 4 bit lookahead adders, Sign and Magnitude comparators, buffers, counters etc. These elements of the ES2CELL2 library are realised with the basic building blocks from the ES2 library.

Those ES2 and ES2CELL2 library elements can be used to built low level designs. The standard ES2 library elements of either versions are almost equivalent. But the corresponding elements in those libraries differ in used chip area and timing.

With SDA it is possible to built (hierarchical) low level realisations, to generate test vectors, to simulate networks, circuitry and realisations, and to generate IC-layouts of the designs. There are several simulators available such as SPICE and SILOS. But only SILOS will be used to test a SDA design. The possibility to generate IC layouts is one of the main reasons why SDA was chosen from the other available software packages available in the Digital Systems group. SDA is currently the only software from which chip realisations can be made at the Digital Systems Group.

Further it will take a lot of work to get to some good timing information. This means that timing information can be extracted from simulation results of a (sub) network realization. These simulation results can be re-substituted into the design.

However, SDA shows also some disadvantages on forhand too. Such as the very low level of the primitives from the ES2 library. Some other (dis)advantages of SDA will appear in the remaining part of the report.

5.3 SDA HIERARCHY

In case of an SDA low level implementation of the instruction cache we would like to take advantage of the hierarchical structure of the IDaSS design. A structured Top-Down strategy would be most convenient. In that case the highest levels of the SDA design should be equal to the equivalent levels used in the IDaSS design. Under the lowest level of the IDaSS design other real SDA levels must be added.

In principle it is possible to follow the Top Down strategy in SDA because Top Down designing is a possible feature in SDA. But this feature is not investigated and applied yet at the Digital Systems group. The non ability to designing Top Down made the SDA designing more complex because a Bottom Up strategy had to be followed.

This means that the highest levels are known because of their equality to the levels used in the IDaSS design and the levels which lay underneath the lowest IDaSS level must be first worked out (Top Down) by hand where after a Bottom Up filling in must taken place. This makes the conversion of an IDaSS design to a SDA design unneeded complex and time wasting.

One remark to the used hierarchical level design is that in case of a IC realisation, the SDA design will be projected on one (flat) level. This means that the hierarchy will disappear so that hierarchy is only a helpfull feature during designing. This is because a design will be easier to survey.

In case of the low level implementation of the Instruction Cache, the SDA inputs will consist of hierarchical structured schematics which contain all (one or more) (sub)modules. In this investigation in relation with a possible IDaSS conversion, the highest levels of the low level design will be equal to all the levels used in the IDaSS design hierarchy. But in case of a low level implementation there will be some (hierarchical) layers placed under the lowest IDaSS design layer. In this report only the hierarchical levels of the Fetcher are described and implemented in SDA. This will be described in Chapter 6.

5.4 SDA LANGUAGES

Until now we only have discussed some of the SDA graphical abilities. More precisely, the abilities to make a low level design built up with hierarchical modules which on the lowest level only contain primitives out of the ES2 library in the graphical mode. For this kind of editing the graphics editor SKILL environment is used.

SKILL is an extended version of the 'basic' SDA language IL. SKILL performs the SDA graphical editing and related topics. SKILL uses a special command interpreter that can be invoked directly from the SDA program.

The SDA language IL is a language based on LISP with a C - like syntax. With the use of this 'IL' language, the SDA system cab support the ability to generate textual descriptions of the connectivity of blocks and their properties. These textual descriptions are called 'Netlists'. A netlist is thus a textual description of a design which can be flat or hierarchical. The latter allows the user to describe modules of interconnected models and then use the models again in the Netlist. This allows the user to take advantage of repetitive structures in a design. It is thus possible to create your own user defined library with standard blocks and modules.

The proces of generating a Netlist is called 'Netlisting' and a tool which has the ability to generate them is called a 'Netlister'. To generate your own netlists two things are needed :

1. Create Netlist representations that describe the properties to be used in the netlist and their netlist formats for each block in the library which is currently used.

2. Generate the Netlist

5.5 CONVERSION AUTOMATION

IL language and nestlistings make textual input to SDA possible. A prospective that can be derived from this, is a automated IDaSS to SDA conversion which has the capacity to depict IDaSS design files on SDA netlistings. To automate this conversion a lexical scanner/compiler is needed. The source language of this compiler should be the *.des*, the IDaSS design file format. The target language could be the IL language. Thus as source file an IDaSS design file is most suitable. Because if a document file is used as source then the compiler has to compete with a lot of redundant information and added text. The latter is very practical for understanding by humans but is worthless and a waste of time for computers. As target file a netlist is suitable. But the SDA system includes several translators, so another file format can also be usefull if it fits as input file of one of these available translators. There after the translator can translate a corresponding input file in to a SDA format file. The disadvantage of this possibility is that no graphical edits of the schematic can be made of textual input. And thus is the very helpful feature of adding probes in the design for testability, out of the question.

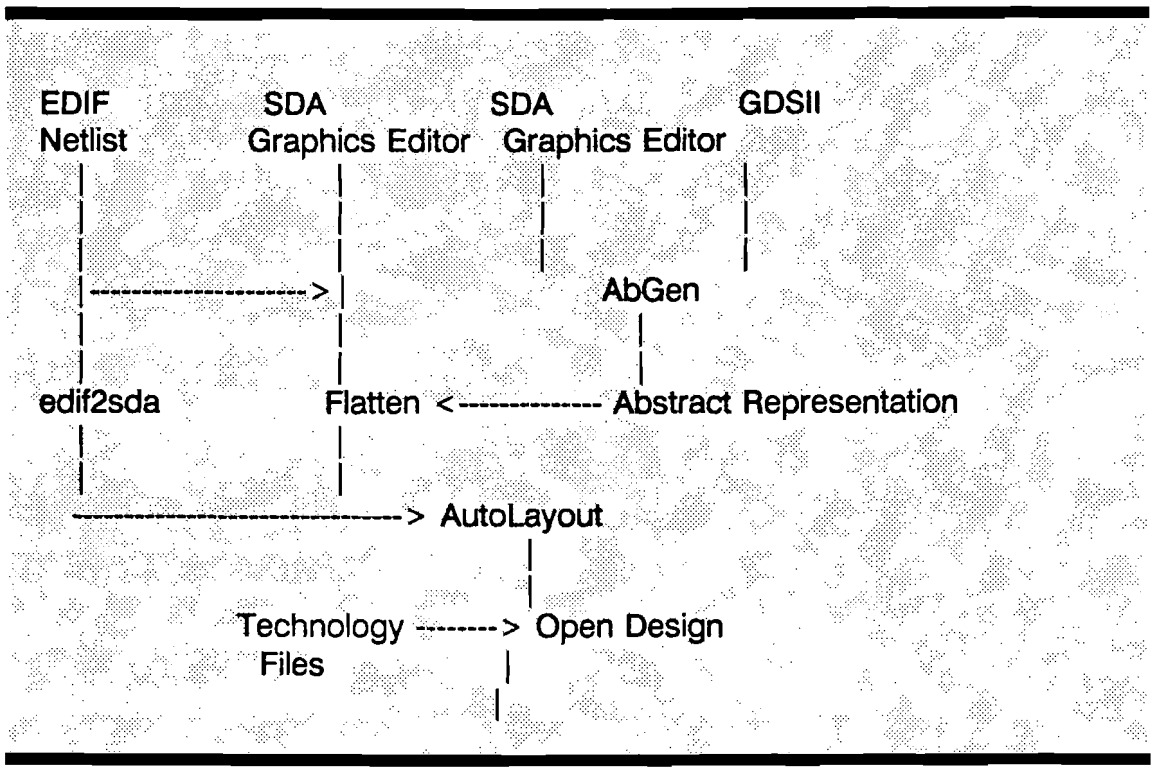
The available translators in the SDA system are :

- Calma GDSII Stream
  sda2stream
  strm2sda

- Electronic Design Interchange Format
  edif2sda

- Other Translators
  cmp2sda
  mif2edif
  sdl2sda

Especially the translators sdl2sda and mif2edif are interesting because they show some relations with the Silvar Lisco software and the Mentor Graphics software. Both simulation software systems are being used at the Digital Systems group. But only Graphic Mentors is still available and operational. Thus the mif2sda translator is the only translator available within SDA which can be used. MIF stands for Mentor's Intermediate Format and this format can be translated with the mif2edif translator in the EDIF format which in turn can be translated with the edif2sda translator into SDA format. After this multistage translation which is described in the SDA reference manual 'Design Framework Vol.1 p.7-46, a valid representation of the schematic should exists and a AutoLayout representation is generated which can be used by Place & Route. See Text Table 1.



1 Design Entries for the SDA System

The layout representation is the master physical representation which is used for the final mask. It is also the representation which is from the abstract representations is generated. The abstract generation is a representation used by the Place & Route system which contains only the pin and boundary information from the layout.

An option that can be a compromise is the development of IL procedures which can depict the parameters that the designer puts in on hardware. (which are textually described)
This option handles with the non flexibility of the SDA system that can not implement a, for example general adder. This general adder block can be given any width whatever the designer wants without having to construct a whole new design of a adder of the wanted width.

Another option is to extract only the useful information out of a design file with the help of an file extractor program.(Already described in section 5.1) The extracted information then must be worked out on to bit level. The bitlevel information can be optimized by a program called MOM. MOM stands for Multiple Output Minimizer. The output file of MOM is optimized and contains no redundancy. This output file can be used as input file for the program MOM2SDA. The output file of this latter program can then in turn be used as an input file of the SDA PLA generator.

The MOM2SDA C program in principle makes only a file with the correct format out of an input file. But this program is extended with some (consistency) check procedures which were already developed and used for and in the MOM program. A program that depicts a MOM output file one to one to a 'Personality' File is in principle sufficient. The data input file for the SDA PLA generator is called a 'Personality' file. The structure of this kind of files consists out 6 sections which are described below. In Appendix C a personality file is illustrated with the corresponding MOM output file.

.INPUTS	Number of input pins (Integer)
.OUTPUTS	Number of output pins (Integer)
.MINTERMS	Number of productterms (Integer)
.INNAMES	List of unique input names
.OUTNAMES	List of unique output names
.PROG	The connectivity matrix for the PLA. Legal characters are [input plane] 0,1 and '-' (= don't care) and [output plane] 0,1. Each row represents a product line the first colomn refers to the input (NOR) plane while the second refers to the output plane.

2 The Personality File Format

After generation the PLA block may be instanced in a schematic for system design in conjunction with other Standard Library components in the Graphical Editor environment.

This approach can however only be applied with IDaSS Operator - and State Control objects and Control Connectors. Another disadvantage may be the restriction of the program MOM which can only handle 32 inputs and 32 outputs. Also is a PLA implementation in most cases not an optimal implementation of the operators. So the optimization achieved by MOM could be nullified. But an advantage of a PLA implementation is the possibility to implement several operators within one PLA.

Further is a lexical scanner/parser indispensable. This scanner must have the ability to distinguish the several kinds of blocks and to convert these blocks or objects with the correct conversion procedures. This conversion procedures take care of the working out on to bitlevel of the usefull extracted textual IDaSS design file descriptions.
But as long as there is no such scanner/compiler available the conversion must be done by hand and this is very time consuming.

5.6 CONCLUSIONS

SDA is the only simulation system at the Digital Systems group from which a real hardware realisation can be made. Despite of some the inflexibilities is SDA a complete package with a lot of (helpful) features. The biggest disadvantage of SDA in the Digital Systems group is the lack of knowledge about the SDA system. This includes Top Down design, the SDA languages IL and SKILL and the available translators.

Another recommendation is to take the placement information in the design file into account. This is useful in the way that in case of Top Down designing the highest level(s) can be based on the same layout and sub modules as used in the IDaSS levels. So only within the levels below the lowest IDaSS level the designer himself must make a suitable layout and hierarchy.

The blocks that are used in the IDaSS design but which are not explicitely necessary in a low level implementation, are implemented because of the transparancy with the original IDaSS design.

A scanner/compiler can be developed to automate the conversion from IDaSS to SDA. A precise description of the semantic and syntax of the IDaSS format are needed and are already described in [Verschueren.1]. There are several possibilities for the target language of this compiler. The best choice has still to be sorted out.

6. HARDWARE IMPLEMENTATION OF THE FETCHER

6.1 THE CACHE ENVIRONMENT

Within the work of Hu, not only the cache is described but also the Cache environment. (See Figure 1) This test environment is just a substitute environment which is only used and sufficient for simulation and testing of the functional behaviour of the Cache. The normal environment of an on chip cache includes a Bus Unit, a Memory Management Unit and an Instruction Unit.

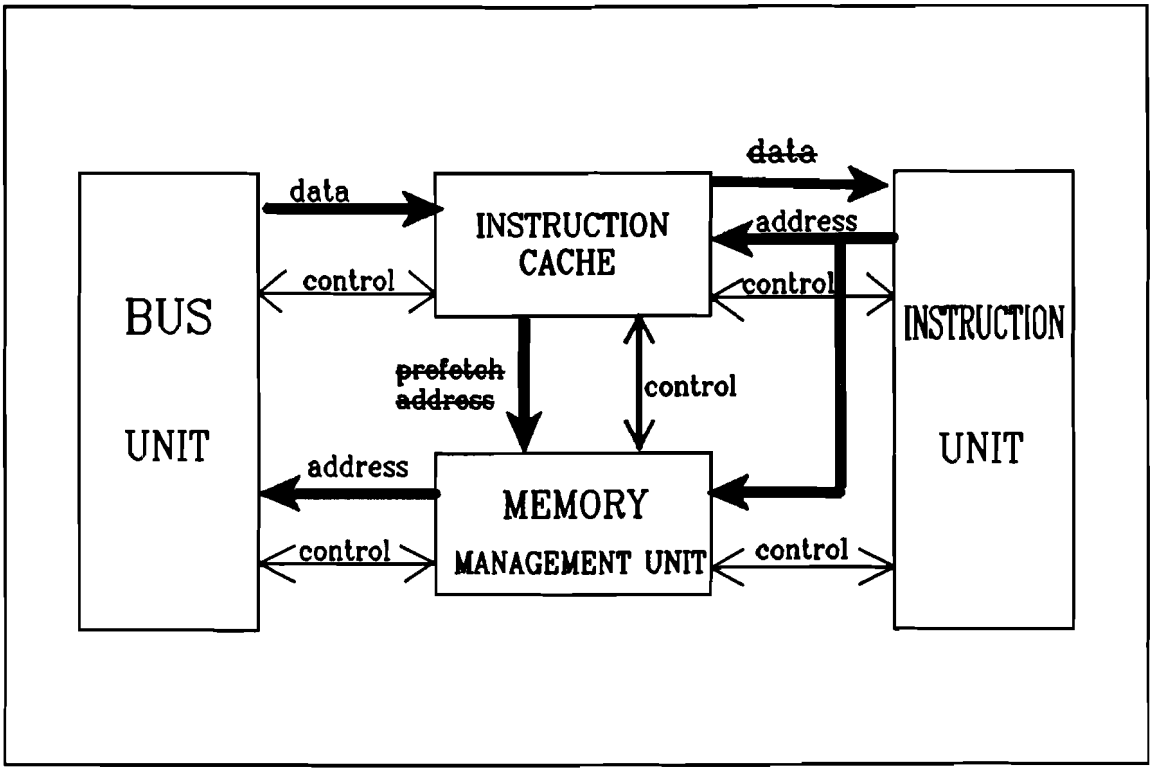


Figure 19 The Test Environment of the Instruction cache

The bus unit is the interface to the outside world. Besides requests from the Instruction Cache it receives also requests from the Data Cache. The bus unit is able to serve these requests simultaneously. The instruction bus is 32 bits wide. The handshake signals between the instruction cache and the bus unit are TransferBlock, Valid Count and Cancel which are the outputs of the cache. And Data, Ready and TimeOut which are inputs to the cache.

The instruction Unit is the real processor element in which all the operations on data will be executed. Therefor a 64 bits data bus is used to transport the instructions between the Cache and the instruction unit. Instead of specifying the address of 64 bits, the instruction unit specifies the address of the first of the two needed quads. The second quad is always placed on the next memory location. The handshake signals between the

instruction unit and instruction cache are Ready1, Ready2, Request, Address, Ack and Data signals eventually pagefault and Timeout signals. The Address, Ack and Request signals are the inputs to the cache and the others are the outputs. The pagefault and timeout signals are however not implemented in the IDaSS design.

The Memory Management Unit (MMU) will translate the virtual addresses into real addresses. But the MMU is still a subject which must be studied. Only the prefetch addresses and status signals of the Cache are implemented. The DemandPre signal which is controlled by the Cache indicates in which mode the Cache wants to fetch data. The MMU knows by polling this signal what the current state the of the Fetcher is. The DemandPre signal is two bits wide and gives cause to the following implementations :

00 : The Fetcher and MMU are not active.
10 : The Fetcher is demand fetching and the MMU has to translate the address delivered by the instruction unit.
01 : The fetcher is prefetching and the MMU has to translate the prefetch address delivered by the instruction cache.
11 : The Fetcher is demand fetching, the prefetcher is updating the cache status and the MMU has to translate the address delivered by the instruction unit.

After the Cache and MMU receive a request from the instruction unit, the Cache will search for the requested data and the MMU will translate the virtual addresses into real addresses. If the data is in the Cache and the prefetching is not active then the MMU will not proceed. If the data is not in the cache then the MMU will deliver the translated address to the bus unit. When data is present in the cache then the MMU will skip the translation of the virtual addresses delivered by the instruction unit and it starts to translate the virtual prefetch addresses delivered by the instruction cache. After translation the MMU will deliver the real address to the bus unit.

The test environment which is used by Hu is shown in Figure 6.2.

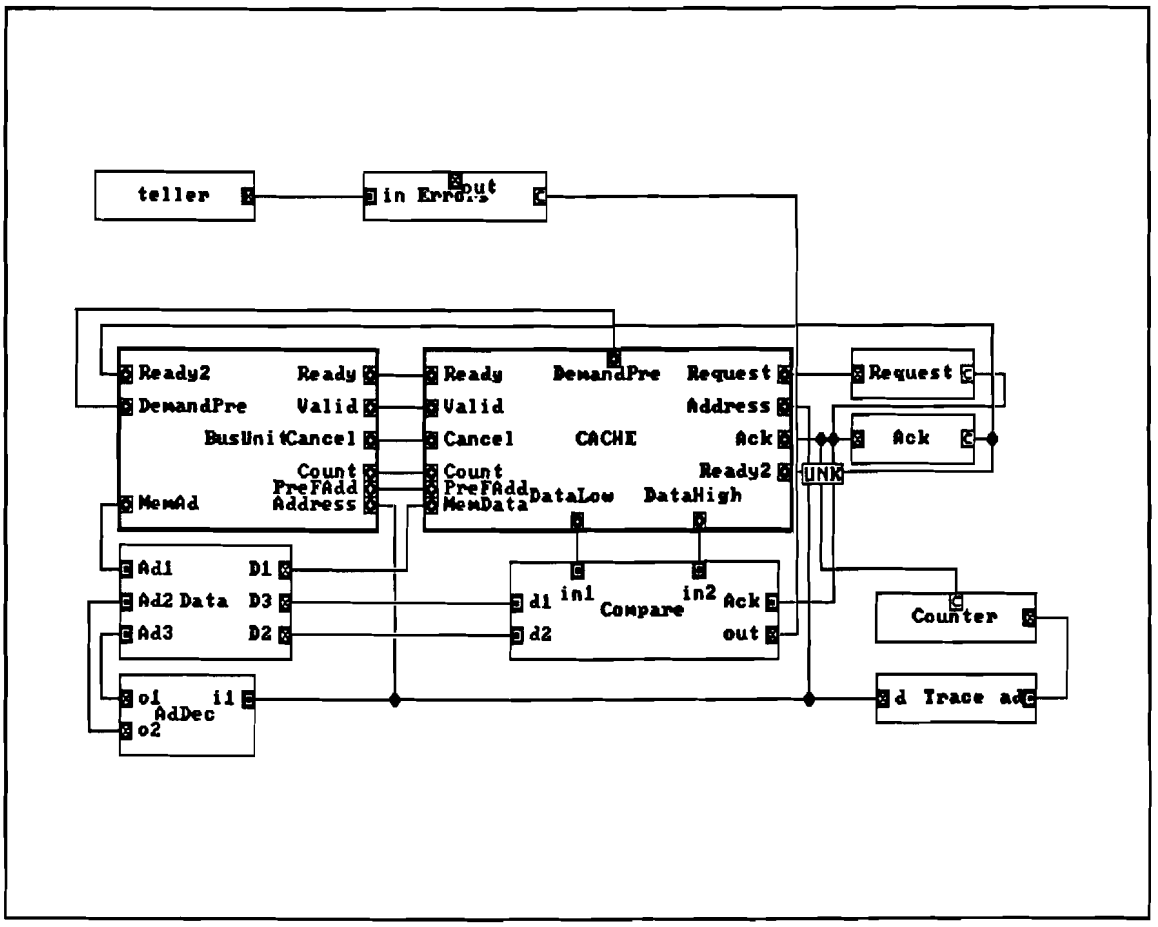


Figure 20 The IDaSS Cache Design Toplevel

The 1 bit constant generator ACK is used to simulate the asynchronous acknowledge signal of the instruction unit. Ack is set in case the instruction unit has taken data from the 64 bits instruction bus.

The used test address decoder, abbreviated AdDec translates the virtual address supplied by the instruction unit to a memory address with width of 11 bits.

The Bus Unit simulates both the Bus Unit and the MMU. It calculates the memory address (MemAd) from the address delivered by the instruction unit in case of a demand fetch.

Request is a register which simulates the request signal of the instruction unit. When the register is set the instruction unit requires data from the cache.

The 32 bits register 'Teller' is used as counter. The value of the counter represents the number of the clock. During a mismatch of the data from the cache with the data from the ROM, the clocknumber is saved in a FIFO memory called 'Error'. This FIFO can contain 8 words of 32 bits.

Trace is a ROM which contains 7168 words of 46 bits each and is used to simulate the behaviour of the instruction cache.[BORMANS]

DATA is a ROM which contains 2048 words of 32 bits each. The contents is used as data from the main memory. The ROM is filled with its own addresses. So the data is equivalent to the address on which it is placed.

The 13 bits register Counter is used as an address counter for the Trace ROM.

The block 'Compare' checks the outputs of the cache with the data in the Data memory. It checks only in case that the instruction unit received the data. (Ack)

After description of the toplevel blocks which are only added for testing the instruction cache we can start to describe the instruction cache itself.

6.2 THE DECOMPOSITION OF THE CACHE

The instruction cache sub modules are respectively : Data RAM, Prefetch Buffer, Read Buffer, Control Bus Unit, Tag RAM, Status Registers, Server and the Fetcher.

The Tag-, Status- and Data RAM are separated in the cache. The memories in IDaSS have the restriction that they can not be wider than 64 bits. Therefore the status memory in the instruction cache design is divided in separate memories for the tag of the first block, the tag of the second block, the status of the first block and the status of the second block. They are respectively named : TAG0, TAG1, STATUS0 and STATUS1. The tag memories are 37 bits wide and the status memories are 34 bits wide. Both memories have a depth of 16 words.

The data RAM is 8 * 32 = 256 bits wide which is equivalent with the transfer block size and has a depth of 128 words.

The simultaneous access to the Data RAM by the Server and Fetcher is not implemented due to the non-availability of multiported RAM. To handle this simultaneous accesses an arbiter is installed to control the access to the RAM. Normally First Come First Serve is applied, but in case of a simultaneous access of the Fetcher and the Server, the Server has the highest priority.

The Read- and Fetch buffer are used to avoid the repeatingly access to the data RAM. The Read buffer copies the transfer block with the required quads and also the status of the transfer block. So the read buffer is built up with data registers containing quads and status registers which contain the data_valid bits. The fetch buffer is also constructed with data- and statusregisters.

The Server has to find the required quads and has to take the appropriate actions to deliver them to the instruction unit. The quads can be found in 3 different places : 1. in the Fetch Buffer, 2. in the Read buffer and 3. in the Cache RAM. The Server is divided into 3 main blocks respectively : Comparators, combinational block1 and combinational block 2. The function of the combinational blocks is to set the control signals for the corresponding quads. The function of the comparators block is to detect the validity of the required quads.

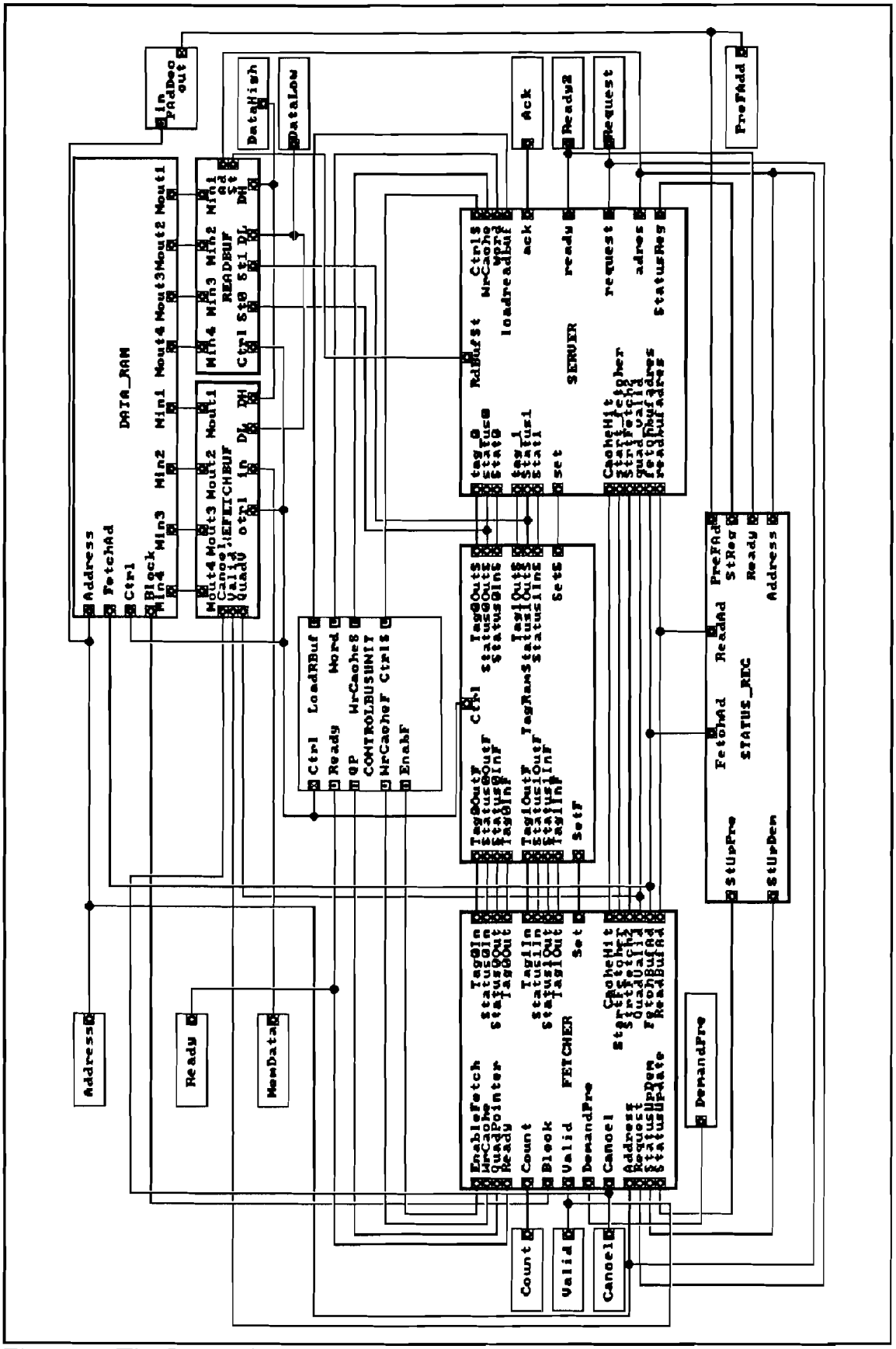


Figure 21 The Instruction Cache

6.3 THE IMPLEMENTATION OF THE FETCHER

6.3.1 Introduction to the Implementation of the Fetcher

The task of the Fetcher is to fetch the quads from main memory. This is however not as simple as seems to be. The Fetcher is the biggest and most complex sub module of the total Instruction Cache. The Fetcher takes almost half of the total IDaSS code of the total Instruction Cache design.

The Fetcher delivers instructions in two different modes respectively : the demand fetch mode and the prefetch mode.

In the demand mode the instructions are delivered at demand from the Instruction Unit. The prefetch mode will occur in case of a cache hit but only if and only if it is not already in the cache. This 'Prefetch_Lookup_on_Hits' algorithm has the advantage that the number of cache accesses is minimized.

The fetcher sub module is subdivided in to 4 main parts respectively :

1. Prefetcher
2. Demandfetcher
3. Comparators
4. Rest

These four parts will be described in the next paragraphs. The replacement and fetch algorithms are illustrated in [Hu p 50 - 51]. We can see that both submodules only differ slightly.

6.3.2 Implementation of the Comparator Module

To implement the prefetch algorithm some comparators are needed. Those are used for an existency check of the next transfer block in the cache. The comparators schematic contains three 43 bits, two 37 bits and two 34 bits comparators. The comparators are implemented with EXOR and OR gates. The EXOR gates compare two corresponding bits and when they are not equal the output will be '1'. After the first column of EXOR gates OR gates are used to collect the 'ones' if there are any. The output of a comparator will be '0' if the two compared words are equivalent and the output will be '1' if the words are unequal. This can create some confusion. Because IDaSS uses : *(Expression) if0: .. if1: ..)* which means that the statement(s) after *if0:* is (are) executed if the expression is false and if the expression is true the statement(s) after *if1:* is (are) executed. Thus IDaSS uses *0* = *false* and *1* = *true* and the comparators which are

implemented in SDA, use *0* = *true* and *1* = *false*.

This EXOR / OR implementation is neither optimized in performance nor in used chip area. This means that they consist of EXOR gates followed by 3 or more columns OR gates. The one bit 'TransferBlockHit' output of the comparators block will be 'high' if and only if the next transfer block is in the Cache. But if the fetcher is busy the output of the comparators block will be ignored. The other outputs respectively : _ReadHit, Cacheit and fetchHit are only implemented for testability.

The 'Fetblockcomparators' is an implementation of the compare operator in IDaSS. The compare operator description uses about 30 lines of code and the SDA implementation uses about 206 EXOR's, 121 OR's, 22 two bit ADDers and 16 MUX41 ES2 primitives.

6.3.3 The Implementation of the Prefetcher

The Demand and prefetcher sub modules have to fetch data from main memory. The sequence of the fetching operation can be divided into 2 parts. The first part initializes the interface protocol between the fetcher and the bus unit. The second part is the follow up part. This part takes only care of guiding the quads to the correct place in the fetch buffer. And when the last quad is fetched, the cache status is updated by the fetcher. For guiding quads to the correct place in the buffer a register called 'QuadPointer' is used. The contents of this register is used to address the data registers of the fetch buffer. The quadpointer register is 4 bits wide and can be implemented with an REG4 ES2 primitive. Registers of more than 4 bits wide must be build up with several REG4 blocks parallel.

The replacement and fetch algorithms are illustrated in [Hu p 50-51]. We can see that both CacheUpDate submodules of the fetcher and server only differ slightly. Therefore a description of the prefetcher Cache Update is worked out into detail and the description of the demand CacheUpdate submodule fetcher is cut down to a review of the dissimilarities with the prefetcher submodule.

The prefetcher module fetches quads if the fetcher is in the prefetch mode. A state controller {pfsc} is used to implement the prefetcher flow diagram. Besides the state controller, an operator is included. This operator updates the status of the fetched status block at the end of a fetch operation. It also updates the tag bits of the transfer block in case of a block replacement. This operator is activated by an active LoadCache signal.

The prefetcher state controller is initialized by an active Start signal. The start signal will be set if :

1. The server did not start the fetcher with the signal startfetcher.

and 2. The instruction unit is requiring data with the request signal.

and 3. The next transferblock is not present in the cache which is indicated by the trblock signal.

The initialization of the protocol is done by setting the valid and count signal. The valid signal indicates the validity of the delivered address. The count signal represents the number of required quads by the fetcher. And the quadpointer points to the first address of the first wanted quad in the fetch buffer.

Immediate after initialization the startfetcher and the ready signals are polled. If the startfetcher signal is set then the prefetch actions must be stopped. (Stop prefetch) And the contents of the fetch buffer is written to the data RAM. So if the bus unit delivers a quad to the cache, the bus unit sets the ready signal to indicate the presence of a valid quad. If the ready signal is set, the quadpointer points to the next place in the fetch buffer. This is done by incrementing the contents of the quadpointer register.

In IDaSS (default) functions can be given to a register such as increment, decrement,

hold and load. The hold and load functions can be implemented by a load signal. The increment and decrement functions have to be implemented in new hardware blocks that must be included in the hardware design. Therefore standard blocks can be developed.

The check which includes the comparison of the contents of the quadpointer register to 9 is an implementation of a fetch buffer full test. Because if the quadpointer contents is equal to 9 the quadpointer is reaching beyond the end of the fetch buffer.

So a new transfer block can be fetched and the status of the fetch buffer has to be stored in the Tag/Status RAM. The data in the Fetchbuffer is stored in the Data RAM.

The already mentioned quadpointer register is used as pointer to the correct place in the fetch buffer. The increment function of this register is only applied if and only if a quad is received. (Ready) The quadpointer register is 4 bits wide and can be implemented with an REG4 ES2 primitive. registers which are more then 4 bits wide must be constructed with several REG4 primitives parallel.

An operator is used to implement this increment function and is called 'Multiplexer'. The control signals of all interface protocols are implemented as constant generators, which are controlled by the prefetch controller. This is because the restriction in IDaSS that no output connectors on state controllers are available.

The SDA implementation of the prefetcher sub module is constructed with 4 main blocks, two AND gates and 3 D-FF's. The biggest block is the cacheupdate block. {pfocuto} This prefetchercacheupdate block contains the cupprt1 block which is sub divided into 5 blocks. The exact hierarchy of the used sub module not only of the prefetcher but also of the whole SDA design is described in Appendix E. Most of the time the names are restricted to a maximum of 8 characters. This is because of a possible want to use the plotfile in a DOS environment.

The SDA implementation of the prefetcher is constructed with 4 main blocks, two AND's and 3 D FlipFlops.

The block 'Fpfcaupd1' block compares the 'Fetchbufferaddress' bits <6:42> with the 37 bits inputs Tag0In and Tag1In. This function is executed by two 37 bits comparators. The outputs of these comparators are used as select bit for MUX21 elements. Those MUX21 elements take care of the propagation of the correct signals to the outputs _WrCache0 and _WrCache1.

```
_Tag := FetchBufferAddress

_WrCache0 := (_Tag = Tag0In)
        if1:1
        if0:((_tag = Tag1IN)
            if1:0
```

3 The IDaSS Description of _WrCache

6.3.4 The Implementation of the Demand fetcher

The demfet sub module contains 5 blocks and two REG4 primitives. The five blocks are respectively called dfcuptot, fdfscmw, fdemfetquadpointer, fdemandfetchmultitotal and fdemandfetchstatusupdate. The exact hierarchy is described in Appendix E. So only a few blocks are in an extended way presented in this paragraph.

The fdemandfetchopermulti block contains only some wild logic to determine the outputs S0def+multi and S1multi+wait. Originally 3 signals respectively 'default', 'multi' and 'wait' are used to select the corresponding operator function. But to fit in a multiplexer with only two select bits available they are put together 2 by 2 .

The fdemandfetchmultirest module describes a 3 bits 2-complement substraction. First a two complement notation must be made from the input. This is done by inverting all input bits and after this 001 is added. In the figure below all three possible functions are multiplexed. Also in the fdemendmultirest2 block a 2-complement substraction must be implemented. A 2-complement substraction should (if possible) be avoided by an IDaSS designer because this is a costly operation. But in this 1 to 1 conversion the used 2-complements substractions are implemented without questioning the use of such substraction.

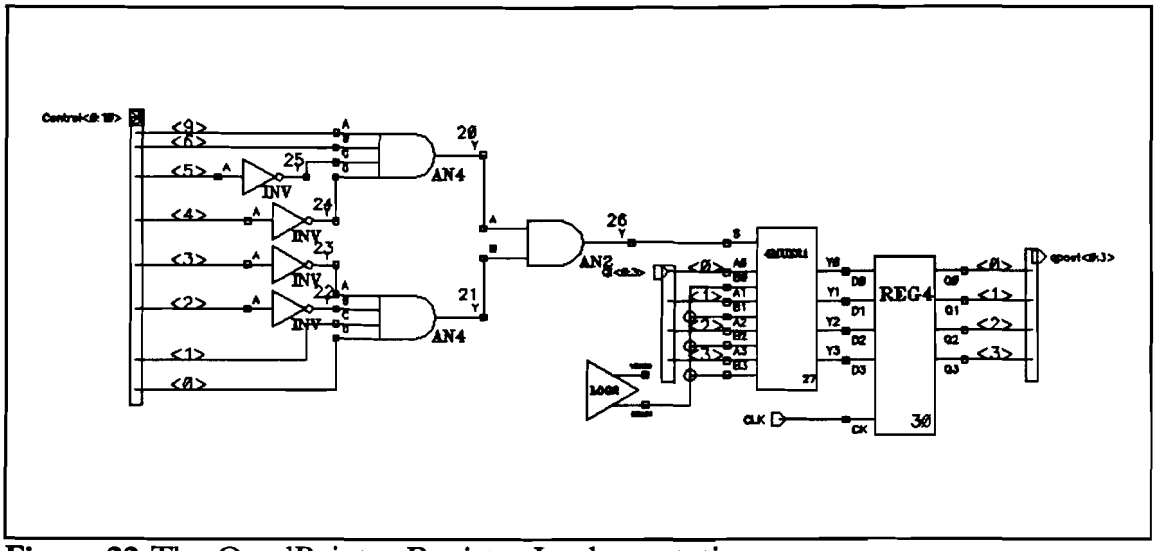


Figure 22 The QuadPointer Register Implementation

The fdfcuptr contains also an interesting sub module. In this fpfqvsel block a 8 bits block must be substituted into 8 bit banks in SUBSTAOUT. The area in which the byte must be placed is determined by two select signals respectively _ADDR3 and _ADDR4. After this substitution a 34 bits 2 to 1 multiplexer takes care of the 34 bits from which 26 are old and 8 are new.

6.3.5 Implementation of the Rest Part

The 'Rest' part of the fetcher includes 3 operators, 1 state controller and 1 register. The operators are needed to merge the control signals of both the prefetcher and the demandfetcher together. This 'Rest' part includes 4 blocks which are respectively called : 'ContrMerge', CountMerge', 'SignalMerger' and 'Merge2'.

The blocks Merge2 and SignalMerger merge the control signals of both fetch sub modules together. This is done by some multiplexers. In case of prefetching, the multiplexers select the 'prefetch' sub module and in case of demandfetching the multiplexers select the control signals of all other sub modules. The used multiplexers are mostly of the 2 to 1 type. The ES2 primitives that can be applied are the MUX21 and the 4MUX21 blocks. The 2 to 1 multiplexers have widths of respectively 37, 34 and 4 bits and are composed with the mentioned ES2 primitives. The Merge2 operator contains a 4 bits 3 to 1 multiplexer. So 3 signals of 4 bits have to be multiplexed on 1 output signal of 4 bits wide. Because there is no ES2 primitive that fits exactly the MUX31 function, the ES2 MUX41 primitives are used. In this particular case 4 parallel placed MUX41 primitives are needed to

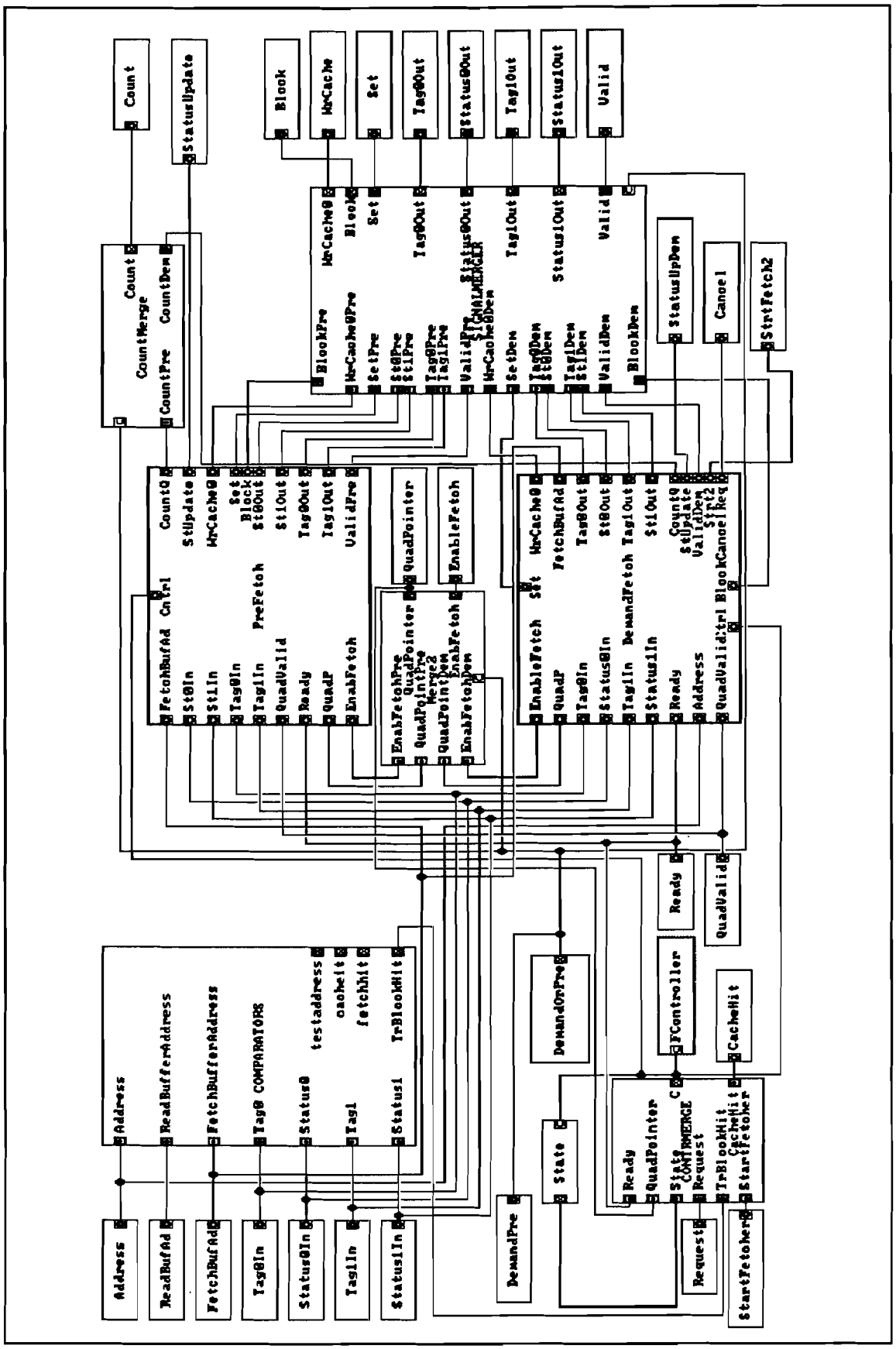


Figure 23 The Sub Module Fetcher

implement the 3 to 1 multiplexer function. Another specialty of this fmerge2 block is the use of the logic level ES2 primitive which can implement the logic levels '0' and '1'. But also the gnd and vdd ES2 primitives can be used. (see Figure FMerge2)

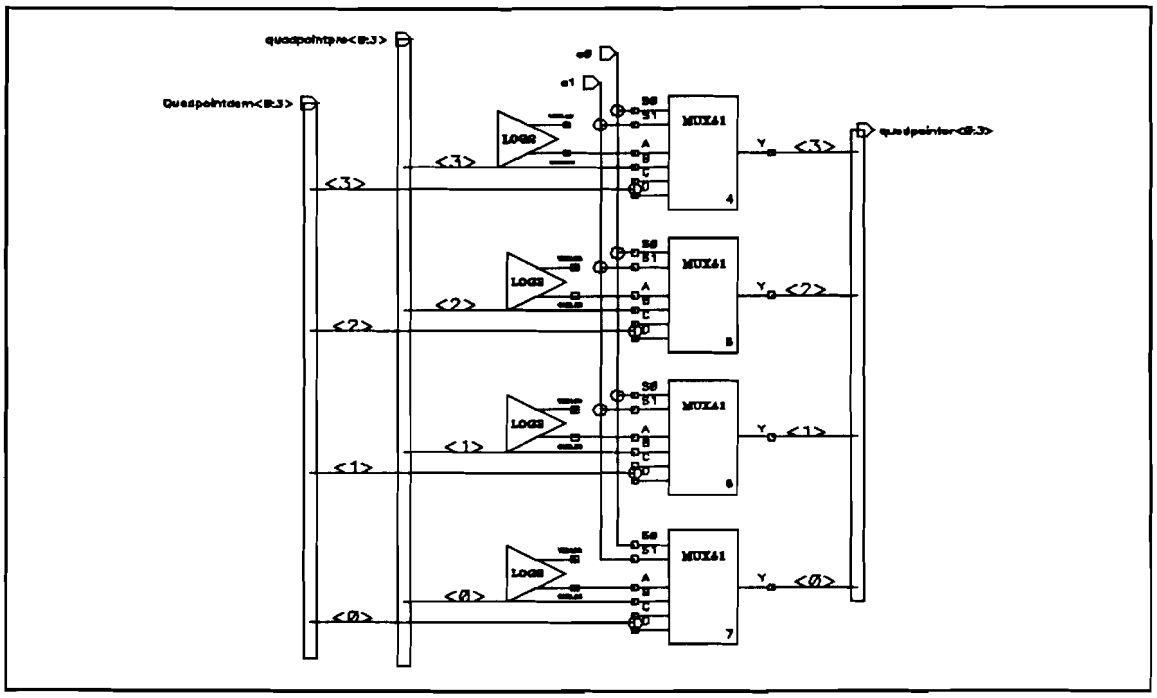


Figure 24 The Fetcher Merge2 Block Implementation

The highest level of the SignalMerger SDA module contains 6 blocks, one MUX21 primitive and one logic level primitive. The 6 blocks are two fsigme3, two fsigme2, one fsigme1 and one fsignmer block. They all implement multiplexer/ merge functions. The multiplexers select either the signals coming from the prefetcher or the signals coming from the demand fetcher. The output signals of this SignalMerger module are outputs of the instruction cache.

The 'ContrMerge' operator is a typical IDaSS block. This block merges the input signals respectively Request (1 bit), StartFetcher (1 bit), TransferblockHit (1 bit), Ready (1bit), QuadPointer (4 bits), State (2 bits) and CacheHit (1 bit) onto one bus called 'C' of 11 bits wide. This block is in principle unnecessary in a hardware realisation and can be implemented a bus tabs. (See Figure 25).

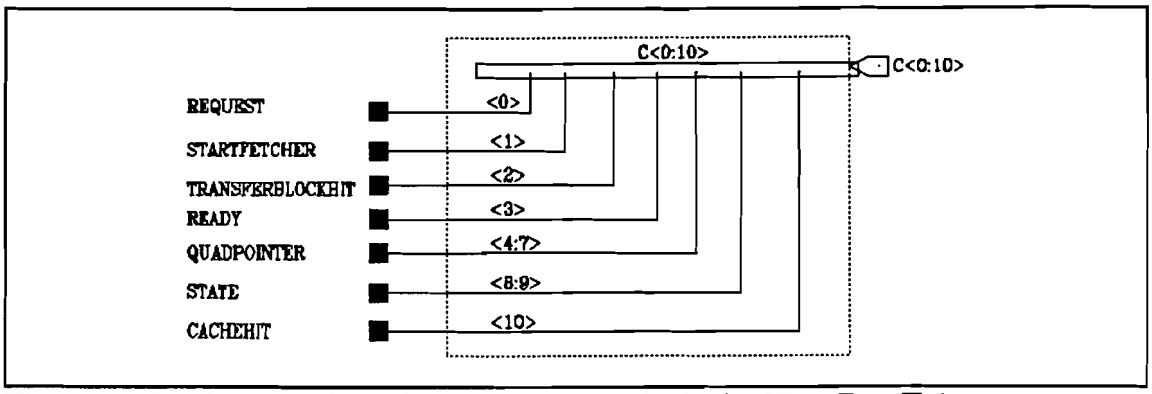


Figure 25 Implementation of IDaSS Merge Blocks in SDA Bus Tabs

The SDA implementation of a block as shown with the discontinues line and the outcoming I/O pins is however not possible. So another SDA feature must be applied and that is the so called 'patch cord'. This ES2 primitive must be used in case that two buses with two different names must be connected. But also a patch cord doesn't bring a solution to the problem above because patch cords can only be used in case that the two to be connected buses not both are input or output terminals. This is the reason why the ContrMerge block is omitted and replaced by the bustabs as shown by the bus tabs shown inside the discontinues block of Figure 25.

Another sub module of the rest part of the fetcher is the block 'FController' which includes a State Controller. This state Controller has 4 states and controls the DemandOrPre Constant Generator. The setting of the two DemandOrPre bits is dependant of the current state of the state controller. The transitions to a next state can only be dependant of the old state but can also only be dependant of the current control connector input bits.

The demand and prefetcher can not be active simultaneously so the fetchcontroller arbitrates the 2 fetch modules. Further is the controller used for indicating the mode in which the fetcher operates. This latter is important to the MMU and the rest of the fetcher sub modules.

The state controller has in IDaSS no physical connections to the Constant Generator DemandOrPre. In the SDA realisation this phantom connection has to made physical. It occurs several times in the instruction cache that within a state of a state control block an constant generator or register is being manipulated. (Read/Write) This can be done without any physical connections to these blocks. The physical connection must be made and mostly the constant generators or registers are added into one block together with the corresponding state controller. (See fcontsdmw, prefscmw and demscmw which contain the FController and DemandOrPre, the prefetcher an his to be manipulating constant generators and the DemandFetcher and his to be manipulating constant generators)

The SDA module fcontsdmw incudes the state controller 'FController' as well as the implementation of the fuction of the constant generator 'DemandOrPre'. These modules are already described in chapter 4.4.

The IDaSS register State is the last schematic in the 'Rest' part. The function of this schematic is to hold the present state of the total fetcher. The state of the prefetch state controller, demand fetch state controller and fetch controller.

After implementation of all fetcher sub modules a fetcher symbol is generated which is illustrated in the Figure 26.

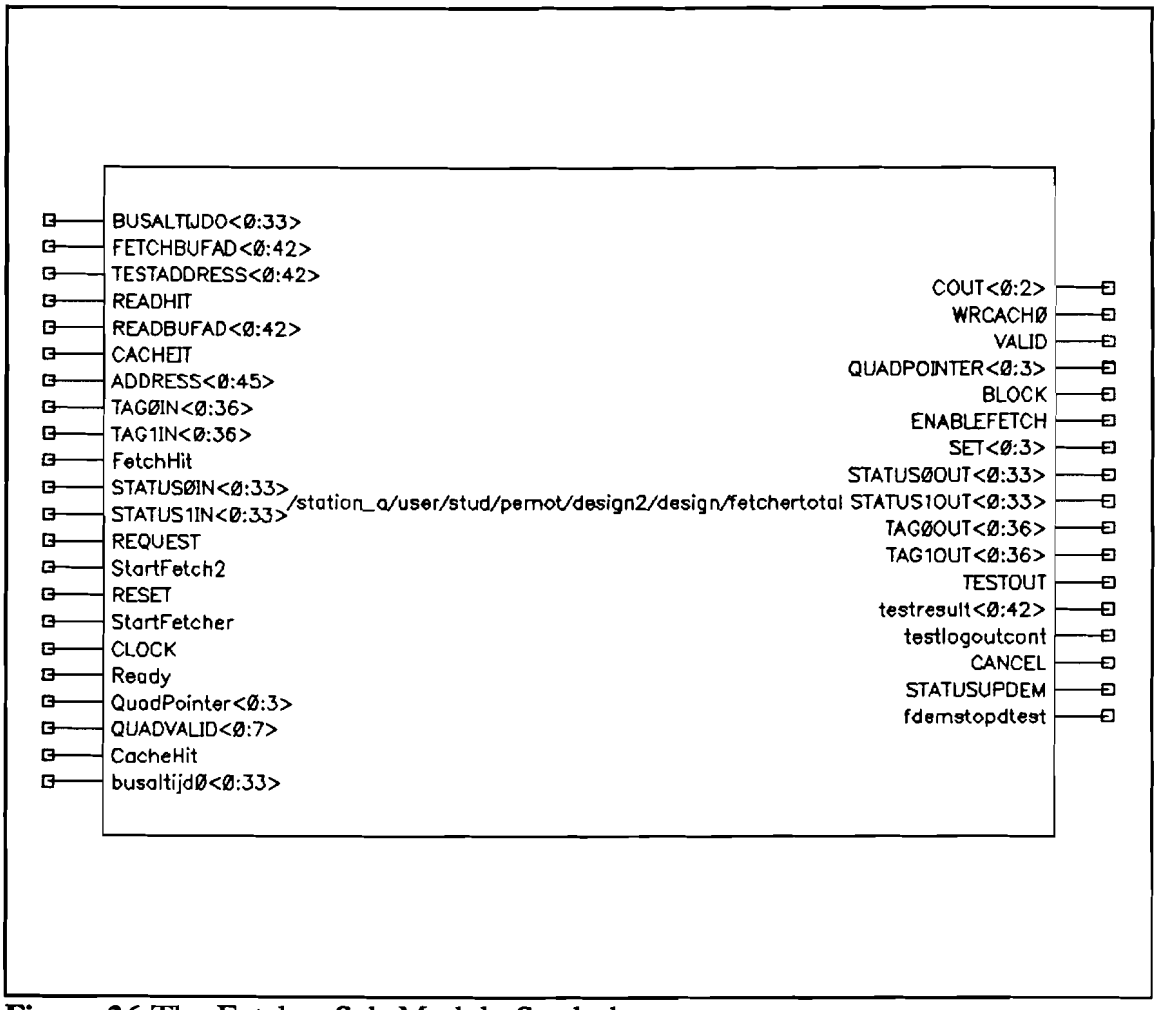


Figure 26 The Fetcher Sub Module Symbol

7. INTRODUCTION TO THE SIMULATION OF THE FETCHER

7.1 INTRODUCTION

As electronic designs increase in complexity, the design simulation process evolves to keep in pace. Simulation is defined here as using a computer program (the simulator) to exercise and analyze a representation of an electronic component or system. Simulation offers the designer the advantage of allowing analyses of a design without having to go through the effort, expense and time to build a prototype. More time will be required for the design phase by reducing the number of prototype iterations in the design cycle.

Functional requirements and functional specifications of the behaviour of the system or in other words what outputs responses to what inputs are most appropriate described in a formal specification language. In this case, the design is directly in IDaSS implemented and this is not a so called formal description. But in IDaSS the functional behaviour can be simulated. The instruction cache is also simulated but simulation can never give a 100% guarantee of correct functioning. Simulation in IDaSS is also a functional test simulation which doesn't deliver any information about the fault coverage and is also very restricted and time consuming.

When a one to one conversion of an IDaSS design onto a SDA implementation is made then the same functional behaviour should be expected if and only if the timing requirements are not taken into account. But timing and synchronisation aspects are that are not implemented (yet ?) in IDaSS. Signals in operator objects which are asynchronous are, whatever the length of the critical path is, propagated through the operator block within one clock cycle. Even in a 43 bits incrementer or comparator signals can propagate in one clockcycle through the operator object. In real hardware/SDA this is impossible due to the delay in the switching elements, buses and gates. These dissimilarities in propagation time in logic circuits are able to originate dynamic hazards, which of course should be avoided.

Another subject that must be avoided is complexity. Therefore we try to design for testability. This is the main reason why an IDaSS design should be divided into several small circuits and hierarchies. The complexity is then reduced and the testability will be improved and faster.

Another effect which occurs during an one to one IDaSS to SDA conversion is the production of redundant logic. IDaSS uses a more complete description of logic functions. In case of the 1 to 1 conversion to hardware this will cause some redundant logic because in hardware realisation of the max- or minterms is sufficient. redundant logic should be avoided because redundancy can cause problems to the test pattern generator. The test pattern generating programs keep trying to find a test pattern for something that can't be tested. The time that these programs take to conclude that such test pattern doesn't exists is enormous in relation to test pattern generation of non

redundant logic. Another effect of redundancy is the possibility of fault masking. This means that an error as it were extinguishes due to an existing error in redundant hardware. Another situation is created when two or more errors occur simultaneous. In this situation this can led to non testability of the circuit with the used test set.

In the description of the conversion of the fetcher in chapter 6, the redundant hardware is as far as possible removed.

In the beginning of the implementation no testability approaches were used. But during separated conversion of all the used IDaSS blocks I did automatically introduce some partitioning. Also are the test pins used by the IDaSS designer taken into account. So the insight of the IDaSS designer is used. But as always designing should be done with testing in mind. In general a few systematic test methods are popular.

1. Build In Test
2. Scan Test
3. Hierarchical Testing

7.2 THE BUILD IN TEST

When the first method is applied the logic must be divided in separated combinational- and sequential blocks. Because with the use of so called BILBO's (Built In Logic Block Observer), the sequential blocks are rebuilded to test pattern generators and/or response evaluating elements. This method is in relation with the implementation of the instruction cache not advisable. Because there's a problem to find a optimal partitioning with the BILBO's which are in fact Modified Linear Feedback Shift Registers. This method uses also extra hardware and will decrease the performance of the circuit. This method may however be applicable in relation of the PLA's and ROM's in the instruction cache. It also costs additional chip area, additional I/O pins, a longer design time and a longer test time.

7.3 THE SCAN TEST

To test circuits in SDA the use of the digital simulator SILOS will be used. But to increase the testability of the designed circuits, additional I/O pins can be applied. With these pins a scan can be made of the behaviour of the circuit. So this Scan Method allows a test mode which can be initiated through the use of a reserved pin. Therefore a user-defined library can be designed which contains basic blocks which can be completely tested via the extra pins. Two simple examples of elements which could be a member of such user defined library will be illustrated here.

example 1.

The first example looks like an extended version of a D FlipFlop. See Figure .

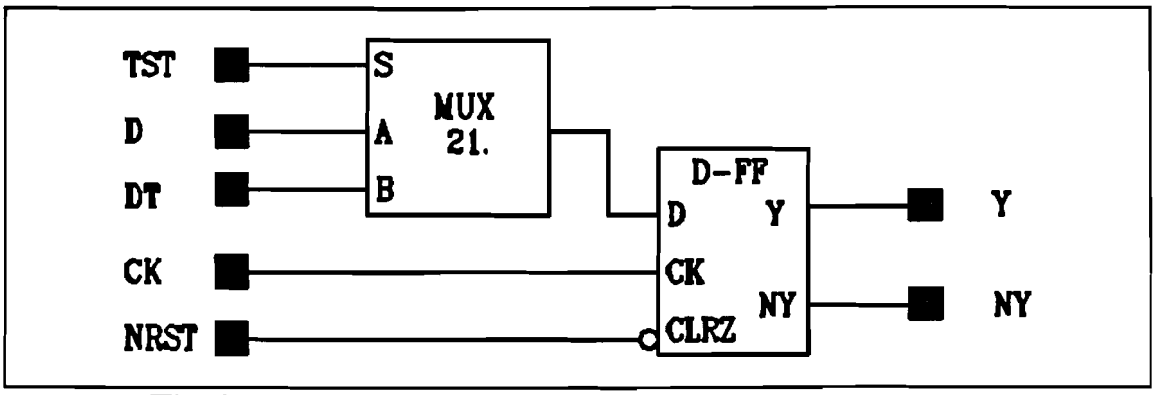


Figure 27 The Scan FlipFlop

TST : pin is an active high input pin which sets the FF in the scan mode or sets the FF in the non scan mode. If TST = 1 then Scan Mode.
D : Signal input of the FF which will be let through if TST is low.
DT : Scan input of the FF which will be let through if TST is high.
CK : Clock input
Y,NY : Normal respectively Inverted output FF.
NRST : Low active input for resetting the FF.

example 2 :

This second example is an extended version of example 1 and is illustrated in Figure .

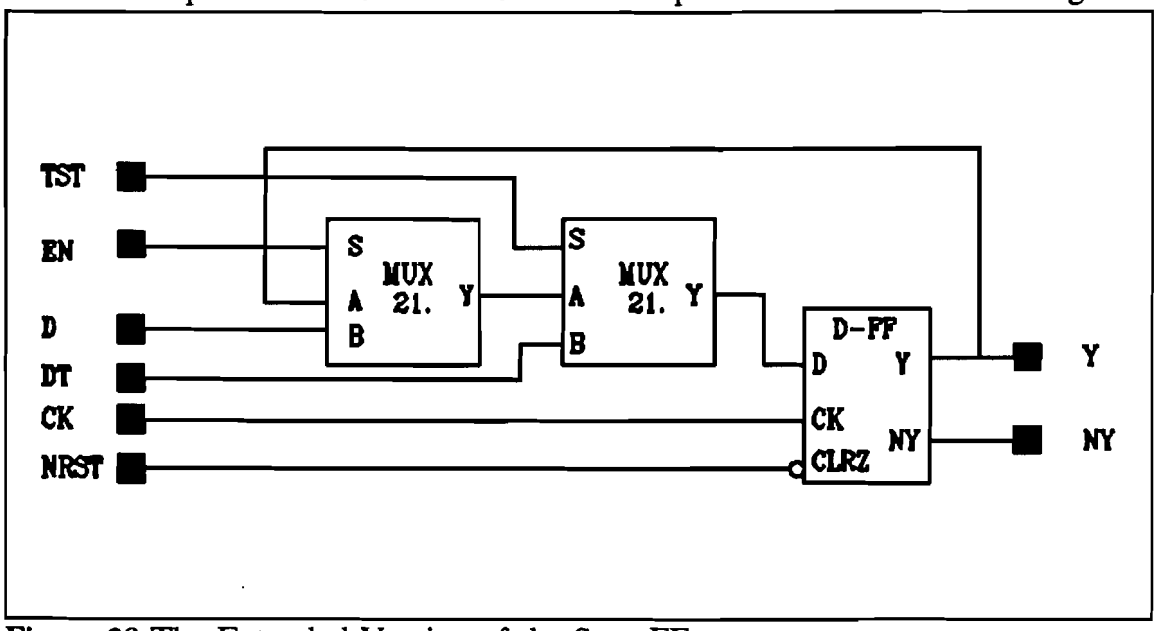


Figure 28 The Extended Version of the Scan FF

D : Is the signal input of the FF. Data can be clocked in when EN is high.
DT : Is the scan input of the FF. If TST is high the data will be clocked in on this input. The status of D and EN are don't care.
CK : Is the clock input of the FF.
Q,NQ : Are respectively the non inverted and inverted output of the FF.
EN : Is an active high signal which activates the FF. When EN is low the output of the FF will propagate to the input of the first multiplexer.
TST : Is an active high input signal which determines whether the FF is in scan mode or not.

Within this method all FF's can be made scannable.
The advantages of this method are :

- An Automatic test pattern generation is possible.
- 100 % fault coverage is possible

The disadvantages are however :

- A synchronous design
- A scan path needed
- Additional chip area needed
- Additional I/O pins needed
- Additional delay in MUX

The expectation is that in the first place this method will not be used because the extra scan path pins are not taken into account in the already made implementation of the fetcher SDA design. But when serious simulation problems or severe designing errors occur this method might be advisable to use at the points where the problems occur.

In case of testing a PLA block there is a must to ensure that the PLA will perform correctly. It is the designers responsibility to ensure that the proposed test vectors can be presented to the PLA inputs and the PLA outputs can be observed at the output pins. A typical test strategy is to allow a test mode which is to be initiated through the use of a reserved test pin. A test signal would divert signals from user input pads directly to the PLA via multiplexers, and the PLA output pads via decoders. This allows direct access during the PLA testing phase with the only penalty of one extra pin. Another test method for PLA's is the use of one extra row in the OR plane which is connected to all minterms in the AND plane. With this row are all used minterms wired and can be tested. The latter method can not be used in case of PLA testing in the SDA system. Because there is no possibility to add an extra pin on a PLA that is generated by the SDA PLA generator.

7.4 HIERARCHICAL TEST OF A DESIGN

This testing method defines first a global test strategy and then a sub block test strategy, a sub block test generations, an overall test pattern assembling and a test program creation. The problem is that test pattern generators generate tests for modules assuming 100 % controllability and observability of inputs and outputs. This method is also only applicable if and only if the sub block are testable individually.
This method seems to be the best fitted simulation strategy but must be worked out yet.

Performance requirements determine the functionality of the ASIC. The simulation should also investigate input and output rate, response time, accuracy, used chip area and also the fault coverage of the test patterns. Only after the simulation results of these requirements are available a realistic judgement can be made. This judgement will determine whether or not the implementation of an IDaSS design has been successful.

7.5 DELAY AND PERFORMANCE IN A HARDWARE DESIGN

In the IDaSS instruction cache design are several big comparators, adders and incrementers used with widths of 34, 37, 43 and 48 bits. Those kind of blocks will decrease the performance of the instruction cache enormously.

The first implementation of the fetcher sub module was just a realisation of the logic behaviour of the Fetcher. So no optimization of the logic has taken place. Only the redundancy in of the IDaSS description is thrown away.

The comparators are thus realised with simple EXOR gates which is not the most optimal realisation. In principle a 2 bit comparator built with only 3 transistors is possible.

But if an adder of 43 bits should be made with a concatenation of the basic ES2 2 bit adders the result will be an enormous performance decrease. This is because of the so called 'Ripple Carry' effect. This means that the most significant 2 bit adder block in case of a let say 44 bit adder, must wait (44 / 2) * TdelayCout before the adder is capable to calculate the two most significant bits. The ES2 two bit adder takes 60 transistors and thus a 32 bits adder build with 16 of these blocks 1920 transistors and the total carry calculations take 92 nano seconds. But to decrease this delay faster adders can be used. Such as adders with look ahead carry. This principle of using look ahead carry generators can also be applied with comparators.
A very fast 32 bits adder is already designed in [PAALMAN]
But also in case of the adders a direct implementation is made thus with the ripple carry effect.

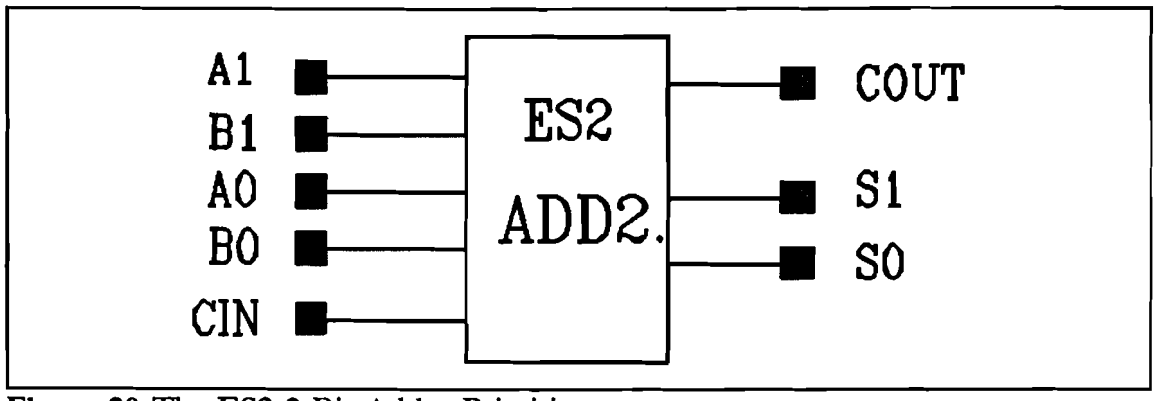


Figure 29 The ES2 2 Bit Adder Primitive

The incrementers that are used in the IDaSS design can be designed as adders with are always adding 1. This implementation is being applied in the fetcher. But also this implementation is very overdone. This will be illustrated in the section below where a 4 bits incrementer with look ahead is described.

With the help of a functional table (see below) two optimal functions can be derived, one for the sum bit and one for the carry bit.

Table IV A 2 Bits Adder Functional table

Ai	Bi	Ci	Si	Ci+1
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	1
1	0	0	1	0
1	0	1	0	1

The extracted optimal function for the Sum bit and the Carry bit are illustrated in the equations below.

$$C_i = A_i \cdot B_i + (A_i \oplus B_i) \cdot C_i$$
$$= A_i \cdot B_i + (A_i + B_i) \cdot C_i$$

$$S_i = (A_i \oplus B_i) \oplus C_i$$

The term *Ai.Bi* does implement the outgoing transport of the carry bit. (*Ci+1 = 1 if and only if Ai and Bi are 1*). this transfer is implicated in the add section itself whatever the value of the carry may be.
The sub function *Ai.Bi* is called the 'Carry Generate' Function.(Gi)
The term *(Ai + Bi)* is equivalent with the condition that an incoming transport of a carry bit = '1' is arrives and this implicates an outgoing transport of *Ci+1 = 1.*
this sub function *Ai + Bi* is called the 'Carry Propagate' Function.(Pi)

So,

$$C1 = A_0 \cdot B_0 + (A_0 \oplus B_0) \cdot C_0$$
$$= G_0 + P_0 \cdot C_0$$

We are however not designing a carry ahead adder but a carry ahead incrementer. In this specific case :

$$C_1 = G_0 + P_0 C_0 = P_0 C_0$$
$$C_2 = G_1 + P_1 G_0 + P_1 P_0 C_0 = P_1 P_0 C_0$$
$$C_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 = P_2 P_1 P_0 C_0$$
$$C_4 = P_3 P_2 P_1 P_0 C_0$$

So, the carry bits can all being expressed with carry 0. This means that the carry bits are faster available and therefore is the adder capable to calculate faster without having to wait on the slow ripple carry signal. this means also that the look ahead carry generator of an x bits incrementer is less complex then a look ahead carry generator of an x bits full adder. The showed principle can also be extended to x > 4 bit incrementers.

$$C_4 = P_3 P_2 P_1 P_0 C0 = P_{0-3} C_0$$
$$C_8 = P_{4-7} P_{0-3} C_0$$
$$C_{12} = P_{8-11} P_{4-7} P_{0-3} C_0$$
$$C_{16} = P_{12-15} P_{8-11} P_{4-7} P_{0-3} C_0 = P_{0-15} C_0$$

An implementation of a 4 bits incrementer block is illustrated in Figure 30.

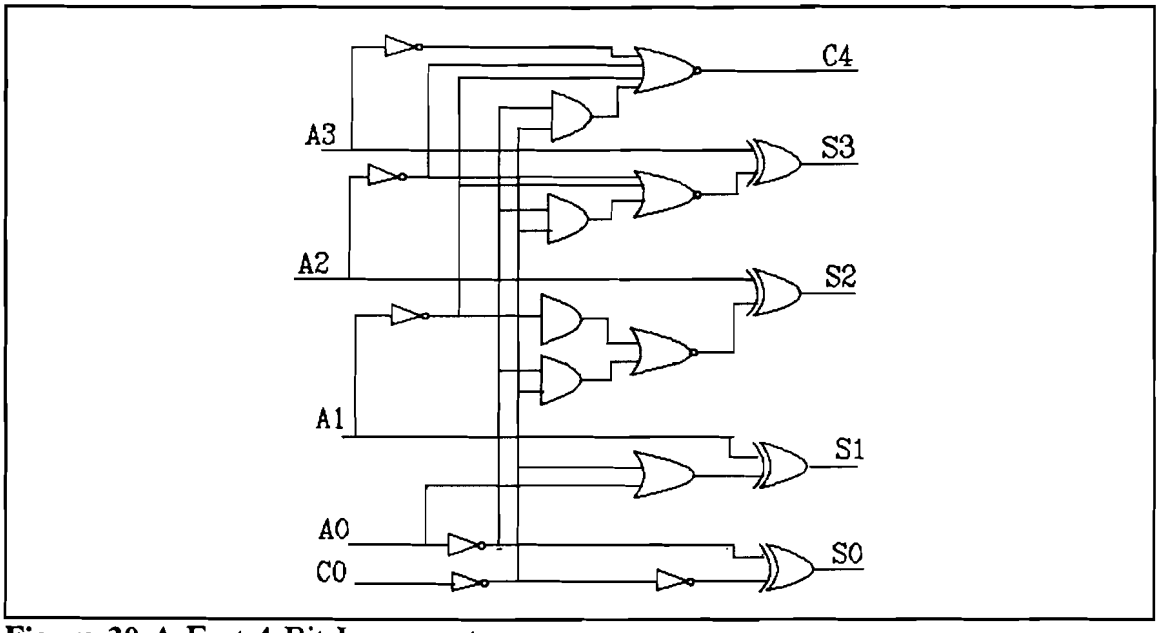


Figure 30 A Fast 4 Bit Incrementer

The maximum delay of the Carry4 bit of this incrementer in relation to a ripple carry Carry 4 bit is illustrated in the next equation.

$$\frac{\textit{C4 adder carry 4}}{\textit{C4 incrementer carry 4}} = \frac{\tau 8 \textit{ equivalent NAND gates}}{\tau 3 \textit{ equivalent NAND gates}} = 2.67$$

7.6 CONCLUSIONS

So far 2 different kinds of models are passed through. The first model IDaSS is a hybrid functional/ behavioral model and the second system SDA is a gate/hardware model. The two models do have a small overlap. A conversion from IDaSS to SDA is possible and implemented. This implementation however is not optimized and represents only the functional logic of what is described in the IDaSS model. Some testing strategies are described in this chapter.

But on the question whether or not IDaSS is a good candidate as preprocessor to a SDA implementation, some precaution should be taken. When an automation can and will be made the answer should be yes. But also when the conversion must made by hand then the answer on the question could be yes. Because this is very much dependent whether or not the timing requirements can be taken on an easy way into account.

The latter is not tested yet and is a possible subject to a successor in the project. To make the instruction cache low level design complete the IdaSS design can be worked out in smaller sub modules just like the fetcher. Those sub modules can independently realised and tested. When all sub modules are realised and tested they can be put together and only after this a performance testing can be made and an answer on the question above can be ensured.

8. CONCLUSIONS AND RECOMMENDATIONS

8.1 CONCLUSIONS

The IDaSS CAD tool was not specific developed to take part in a silicon compilation traject. But IDaSS does have some strong relationship to hardware and in principle is an IDaSS design convertible into a low level implementation such as a SDA implementation. The conversion is currently done by hand, but if some knowledge about IL, the SDA language, or the translators in SDA can be achieved an automated conversion of most of the IDaSS objects is in principle possible. This automation needs also a intelligent scanner/compiler which translates the IDaSS format into a IL format or some other format that can be translated with the in SDA available translators into SDA format.

The current SDA implementation of the fetcher sub module of the instruction cache was very time consuming and not very elegant. Because the only way to construct a SDA hardware implementation is, to use a bottom up strategy which is very uncomfortable in case of an IDaSS into SDA conversion. The Top Down strategy is less complex and when the knowledge about Top Down designing in SDA is achieved, the SDA designing can be done more rapidly.

A one to one conversion of the fetcher is made. This means that no optimization is applied neither in relation to the performance nor in relation with the used chip area. So the expectation is that the performance will be poor but can be improved by using faster subcircuits. This implies that with substitution of faster adders, comparators and incrementers the performance can significantly be improved.

The used chip area can be minimized by making a clever IDaSS design which uses as less as possible over complete descriptions and double tests. The IDaSS logic could also be minimized after it is worked out onto bitlevel, by a program such as MOM.

8.2 RECOMMENDATIONS

To make a conversion from IDaSS to hardware faster and less complex, some knowledge about the SDA should be achieved. In particular the Top Down designing and programming in the IL SDA language are very welcome.

Further could an examination of the available translators in SDA bring another fast way to achieve SDA implementations out of an IDaSS description. Especially the translators SDL2EDIF and EDIF2SDA should be studied.

Another recommendation could be the development of automatic conversion programs which can convert IDaSS design files into netlistings or some other format that can be translated with the just mentioned translators into the SDA format.

The study of the restrictions that could be followed by IDaSS designers to make the work of the SDA designer less complex, can also speed up the hardware conversion process.

Further a very helpful feature could be build in IDaSS and that is the possibility to add delay into the used IDaSS blocks. This means that also some timing into the functional simulations in IDaSS can be added. This means also that the IDaSS design already is designed with some timing requirements.

LITERATURE

[BINK] Bink, S.
Implementatie en Simulatie van de control unit van een TST - netwerk
Stage Report EB 240
Technical University Eindhoven, 1990

[BORMANS] Bormans, J.E.H.M,
Instruction cache for the C - processor
Master Thesis report EB 205
Technical University Eindhoven, 1989

[BURNS] Burns, D. and Jones, D.
68020 cache design
Electronics & Wireless World, vol 92,(1986) p. 727 - 30

[GEURTS] Geurts, A.
Technical University Eindhoven
Besturingsprogrammatuur voor Digitale Systemen I en II
Coarse number 5N300 and 5N330

[HU] Hu, Y.C.
An instruction cache in IDaSS
Master Thesis report EB 223
Technical University Eindhoven, 1989

[PAALMAN] Paalman, J.
Snelle Optellers
Stage report EB 230
Technical University Eindhoven, 1990

[SMITH] Smith, A.J.
University of California, Berkeley
Cache memory design : an evolving art.
IEEE Spectrum december 1987, p. 40-4

[VERSCHUEREN.1] Verschuren, A.
Technical University Eindhoven
The IDaSS file formats
march 19, 1990

[VERSCHUEREN.2] Verschueren, A.
Technical University Eindhoven
The Interactive Design and Simulation System (IDaSS)
january 15, 1990

[WILSON] Wilson, I.
Cacheing the chips
Electronics & Wireless world, vol 95, (1989) p. 75 - 7

[WITHAGEN] Withagen, W.J.
Definition and high-level description of the C - processor, Master Thesis report EB 168
Technical University Eindhoven, 1988

APPENDIX A : IDASS TOPLEVEL ORGANISATION

SUPERBLOCK TOPLEVEL

MODULE ROM Trace
OPERATOR Compare
MODULE FIFO Errors
REGISTER Teller

SUPERBLOCK Busunit (lower level schematic)
OPERATOR FetchDec
OPERATOR AdDec
OPERATOR Merge
{einde BusUnit}

{einde register Teller}

CONSTANT GENERATOR Ack
MODULE ROM Data
OPERATOR AdDec
REGISTER Request
REGISTER Counter

SUPERBLOCK Cache

SUPERBLOCK PreFetchBuf (lower level schematic)
OPERATOR Merge
OPERATOR Multihigh
OPERATOR FetchBufCtrl
OPERATOR exp4
OPERATOR exp2
OPERATOR exp3
OPERATOR Multilow
OPERATOR exp1

{einde PreFetchBuf}

OPERATOR ControlBusUnit

SUPERBLOCK TagRam
OPERATOR Extract
OPERATOR Merge6
OPERATOR Merge1
OPERATOR Merge5
OPERATOR Merge2
OPERATOR Merge4
OPERATOR Merge3

{einde TagRam}

SUPERBLOCK DATA_Ram
OPERATOR Arbiter

{einde DATA_Ram}

SUPERBLOCK Fetcher (lower level schematic)
OPERATOR SignalMerger
Function Prefetching
Function DemandFetching
Function DemandPreFetching
OPERATOR Comparators
Function Compare
OPERATOR ContMerge
Function ContrMerge
OPERATOR CountMerge
Function Merge
Function Pre
Function Demand

SUPERBLOCK PreFetch

OPERATOR CacheUpdate
Function Default
Function Update
OPERATOR Multiplexer
Function Default
Function setto
CONSTANT GENERATOR Count
CONSTANT GENERATOR LoadCache
MACHINE CONTROLLER PreFetcher
REGISTER QuadPointer
CONSTANT GENERATOR ValidPreFetch

{einde PreFetch}

SUPERBLOCK DemandFetch

OPERATOR StatusUpd
Function default
OPERATOR CacheUpdate
Function default
Function Update
OPERATOR Merge
Function default
OPERATOR Multi
Function default
Function Multi
Function Wait
MACHINE CONTROLLER DemandFetcher
REGISTER QuadPointer

{einde DemandFetch}

CONSTANT GENERATOR State
CONSTANT GENERATOR StatusUpdate
OPERATOR StatusUpd
Function default
CONSTANT GENERATOR DemandOrPre
MACHINE CONTROLLER FController
OPERATOR SMerg2
Function default
Function Demand
Function Pre
Function DemandPre

{einde Fetcher}

OPERATOR PAdDec

SUPERBLOCK Server

OPERATOR SignalMerger
Function default
OPERATOR SignalMerge
Function default

SUPERBLOCK Comparators
OPERATOR Extract
Function Extract
OPERATOR Merge
Function Merge
OPERATOR Quad1Finder
Function Compare1
OPERATOR Quad2Finder

Function Compare2
OPERATOR LRU
Function default

{einde comparators}

OPERATOR Multi
Function Multi
OPERATOR Comb_3
Function Always
OPERATOR Comb_2
Function cachehit
Function default
Function fetchhit
Function memory
Function readhit
Function wait1
function wait2
OPERATOR Comb_1
Function cachehit1
Function cachehit2
Function cachehit3
Function default
Function fetchhit1
Function fetchhit2
Function fetchhit3
Function memory1
Function memory2
Function memory3
Function readhit1
Function readhit2
Function readhit3
Function wait1
Function wait2
Function wait3
Function wait4
CONSTANT GENERATOR CtrlState
CONSTANT GENERATOR Next
MACHINE CONTROLLER Scontroller
CONSTANT GENERATOR State

{einde Server}

SUPERBLOCK ReadBuf

SUPERBLOCK Quad5
OPERATOR Mux

SUPERBLOCK Quad1
OPERATOR Mux

SUPERBLOCK Quad8
OPERATOR Mux

SUPERBLOCK Quad4
OPERATOR Mux

OPERATOR Merge

SUPERBLOCK Quad7
OPERATOR Mux

OPERATOR ReadBuf Control
OPERATOR Multihigh

SUPERBLOCK Quad3
OPERATOR Mux
OPERATOR Exp4
OPERATOR Exp2
OPERATOR Multilow
OPERATOR Exp3

SUPERBLOCK Quad6
OPERATOR Mux
OPERATOR Divide
OPERATOR Exp1

SUPERBLOCK Quad2
OPERATOR Mux

OPERATOR StatusDetect

{einde Read_Buf}

SUPERBLOCK Status_Reg (lower level schematic)
OPERATOR TrBlockAd
OPERATOR TrBlockAd2
OPERATOR FetchBufControl

{einde Status_Reg}

{einde Cache}

{einde Top_Level}

Remark 1 :

Only functions that are included with operators of the Fetcher and the Server are illustrated.

Remark 2 :

The fetcher contains 25 functions that must be implemented. This needs some puzzling when these functions are on to bit level searched. The Fetcher contains 3 state machines which are with help of a standard procedure relatively simple convertable in hardware. There are however some functions which have only use in the IDaSS SmallTalk environment. These functions are : Merge2, Multiplexer, SignalMerger. Those functions are strictly taken not necessarily in a low level implementation but by means of a one to one conversion those functions will also being realised.

The Server contains about 35 functions and 1 state controller, but is contrast with the higher number of functions less complex and much smaller then the Fetcher.

APPENDIX B AN IDaSS DESIGN FILE INTERPRETER

The IDaSS design file (.des) format is not a standard format and is especially developed for the IDaSS CAD tool. Only memory contents for the RAM, ROM and CAM objects are using a standard format namely the 'Intel HEX Format'. The IDaSS design file is a textual description of :

1 the used blocks and their hierarchy, inputs, outputs and names.
2 the used objects in the schematics and their names, connectors ,buses and their (functional) descriptions and widths.
3 Information about the topographical place on the monitor.

This IDaSS design file interpreter is a PASCAL program which extracts some useful information out of an IDaSS design file which are described above under 1 and 2.

The PASCAL interpreter determines from the first characters of a design file line which kind of translation must be made. This is possible because each design file line starts with a 'switch' character. Following this a second and eventually a third 'switch' character may be present, depending on the menaing of the line. Optionally one or more data items may be placed following the switch character(s). The first data item follows immediate the 'switch' character.
The most common switch characters are :

'#' This switch character starts the description of any major object in the system and is always followed by two names. The first name gives the type of the object, the second gives the name of the object. A corresponding '.' line will fence off the block. So, the '#' and '.' lines form pair of braces and between those 'guards', the block can be filled with some corresponding block parameters and eventually other (nested) blocks. When blocks are nested the overall block is called a 'Schematic'.

'/' Starts a line which contains information used only to draw the schematic. This includes information about the logical origin and size of the schematic and used blocks, connectors, nodes and segments. These lines will be skipped.

''' The single quote starts a line containing textual information which must be compiled. These compiled texts are used for the following objects :

- Control Connector
- Functions of a function block
- States of a state machine controller

The other important and most common switch characters are :

- #Superblock
- #SuperConnector
- #Bus
- #StateControl
- #Signal
- #Buffer
- #Constant
- #Register
- #Operator
- #RAM
- #ROM
- #CAM
- #FIFO
- #LIFO

APPENDIX C: LOW LEVEL REALISATIONS OF SIGNAL OBJECTS

In this appendix are the hardware implementations of the four different types of Signals illustrated.

A note must be made here because :

'Pulse' Flip-Flops are always reset after a system reset and 'Level' Flip-Flops can be put in a user-defined state following system reset.

Furher are all command inputs coming from controllers OR'ed into Signal input lines.

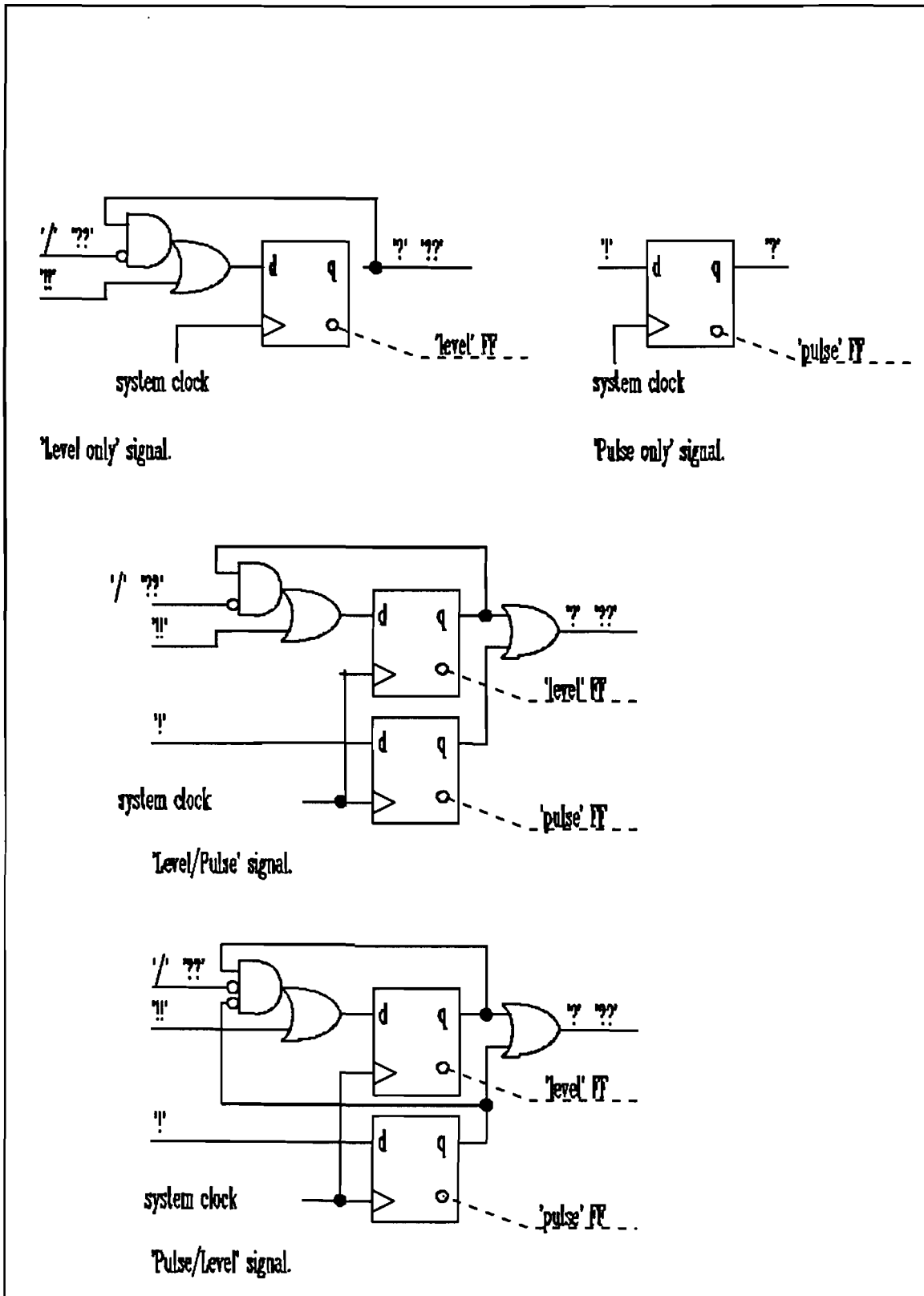


Figure 1 Low Level Realisations of Signal Objects

APPENDIX D : PLA GENERATION FILES

This appendix contains files that are generated during SDA PLA generation. Respectively a MOM input file, MOM output file, MOM2SDA output file and the generated PLA symbol.

The example that is this Appendix is carried out in an illustration of steps that must be taken to achieve a PLA schematic in SDA.

We start from an imaginary IDaSS Control Connector of 8 bits wide. The control connector controls for example a register of 4 bits. (See Table 1)

```
CONTROL CONNECTOR

% 00000001      REGISTER SETTO : 1;
% 10011001      REGISTER SETTO : 3;
% 10000001      REGISTER SETTO : 3;
% XXXX1100      REGISTER SETTO : 7;
% XXXX1110      REGISTER SETTO : 7;
% 011000XX      REGISTER SETTO : 13;
% 100100XX      REGISTER SETTO : 15;
% X11XX00X      REGISTER SETTO : 9;
```

It is evident that this control connector description contains some redundancy. The description can thus be optimized. This can be done by using a program such as MOM. The input file of MOM (**.def**) will be optimized and an output file (**.min**) is created and contains a minimized description. The **.min** output file can be used as input file of the program MOM2SDA. The output file of this latter program (**.pla**) has a personality file format which can be used as an input file for the SDA PLA generator. This generator creates a PLA symbol that can be instanced in onther SDA modules or designs.

```
)00001 | ...A
   10011001 | ..AA
   10000001 | ..AA
   XXXX1100 | .AAA
   XXXX1110 | .AAA
   011000XX | AA.A
   100100XX | AAAA
   X11XX00X | A..A
```

```
Min-cover, found 7 essential productterms
" The function contains 16 don't cares and 16 actives

 x0000001 | ...a
 100x0001 | ..a.
 1001x001 | ..aa
 100100xx | aaaa
 011000xx | aa.a
 x11xx00x | a..a
 xxxx11x0 | .aaa


"FILE: controlconnect.def,
"minimized by MOM version 4.33, Multiple Output Mode
```

```
.INPUTS
8
.OUTPUTS
4
.MINTERMS
7
.INNAMES
in01 in02 in03 in04 in05 in06 in07 in08
.OUTNAMES
out01 out02 out03 out04
.PROG
----11-0 0111
-11--00- 1001
011000-- 1101
100100-- 1111
1001-001 0011
100-0001 0010
-0000001 0001
```

APPENDIX E : THE SDA IMPLEMENTATION HIERARCHY OF THE FETCHER

FETCHERTOTAL :

1. fstate (12 inv, 7 AND, 3 OR, 2 Latch)

    Input : Control<0:10>, clockload
    Output : State<0:1>

2. fcontsdmw (2 inv, 4 AND, 2 OR, block fcontsd)

    Input : Control<0:10>, reset, clock
    Output : DemandOrPre

    fcontsd State Controller FController
    (33 inv, 22 and, 8 or, 2 DFF with reset)

3. fetblockcomparators (2 and, 2 or, 1 level, 1 MUX21, 8 blocks)

    Input : fetchBufferAddress<0:42>, testaddress<0:42>, readbufferaddress<0:42>, address<0:45>, Tag0In<0:36>, tag1In<0:36>, Status0In<0:33>, Status1In<0:33>
    Output : _ReadHit, cacheit, transferblockhit, fetchhit

    3.1 Comparator 43 bits 3* (43 exor, 25 or)
    3.2 Comparator 37 bits 2* (37 exor, 19 or)
    3.2 Fadd43bitsd (22 add2)
    3.4 fetblockcompselect 2* (8 MUX41, comparator8bits = 8 exor + 4 or)

4. fcountmerge (3 MUX41, 3 level)

    Input : C<0:1>, CountDem<0:2>, CountPre<0:2>
    Output : Cout<0:2>

5. fsmtot (1 level, 1 MUX21, 6 blocks)

    5.1 fsigme3 (1 or, 9 4MUX21, 1 MUX21)
    5.2 fsigme2 (2*) (1 or, 8 4MUX21, 2 MUX21)
    5.3 fsigme1 (2*) (1 or, 1 4MUX21)
    5.4 fsignmer (9 inv, 9 and, 3 or, 3 MUX21)

6. fmerge2 (5 level, 5 MUX41)

7. prefet (2 and, 1 inv, 3 D FF, 4 blocks)

- 7.1 fpfqupo (3 and, 4 inv, 1 4MUX21, REG4)
- 7.2 fpfscmw (3 inv, 4 and, 1 or, 3 blocks)
  - 7.2.1 camparator4bits (4 EXOR, 3 OR)
  - 7.2.2 fprefetcherstatecontrol (22 inv, 13 and, 2 or, 2 D-FF)
- 7.3 fmultp (2 ADD, 1 MUX21, 1 4MUX21)
- 7.4 pfocutot (4 level, block)
  - 7.4.1 cupprt1 (4 level, 4 blocks)
    - 7.4.1.1 fpfcaupd3 (2*) 9 4MUX21, MUX21)
    - 7.4.1.2 fpfcaupd1 (2 inv, 1 and, 4 MUX21, 2 blocks)
      - 7.4.1.2.1 Comparator 37 bits (2*) (37 EXOR, 19 OR)
    - 7.4.1.3 fdfcuppr (4 MUX21, 2 blocks)
      - 7.4.1.3.1 fdfcuptp (2 34MUX21, 2 comparator37)
      - 7.4.1.3.2 fdfcuptr (2 * qvsel, 2 34MUX21)
    - 7.4.1.4 fdfcupt1 (1 or, block)
      - 7.4.1.4.1 fdfcupto (3 level, 4MUX21, 2 34MUX21, 2 37MUX21)

8. demfet (2 REG4, 5 blocks)

- 8.1 fdemfetquadpointer (REG4)
- 8.2 fdemandfetstatusupdate (7 inv, 6 and, 4 or, 1 latch)
- 8.3 fdemandfetchmultitotal (3 blocks)
  - 8.3.1 fdemandfetchopermulti (33 inv, 20 and, 8 or)
  - 8.3.2 fdemandfetchmultirest (3 inv, 6 ADD, 1 4MUX21, 3 MUX41)
  - 8.3.3 fdemandmultirest2 (6 ADD, 3 level, 3 4MUX21, 4 MUX41, comparator 4bits)

8.4 fdfscmw (11 inv, 7 and, 3 or, 2 * comparator 4bits, block)

8.4.1 fdemandfetchsd (25 inv, 21 and, 7 or, 3 D-FF)

8.5 dfcuptot (2 blocks)

8.5.1 fadd43bitsd (22 add2)

8.5.2 cupprt1 (5 blocks)

8.5.2.1 fpfcaupd3 (2*) (9 4MUX21, 1 MUX21)

8.5.2.2 fpfcaupd1 (3 inv, 1 and, 4 level, 4 MUX21, 2 comparator37bits)

8.5.2.3 fdfcuppr (4 MUX21, 2 blocks)

8.5.2.3.1 fdfcuptp (2 34MUX21, 2 Comparator37bits)

8.5.2.3.2 fdfcuptr (2 34MUX21, 2 qvsel = 2*(2 MUX21, 8 4MUX21, 8 inv, 4 and)

8.5.2.4 fdfcupt1 (1 or, 1 block)

8.5.2.4.1 fdfcupto (3 level, 4MUX21, 2 34MUX21, 2 37MUX21)

The number of transistors which are achieved by counting the ES2 primitives in all sub modules by hand are described in text box number 1.
The second text box contains the number of transistors which are achieved by running the **netlist + simulator** which creates under the SILOS run directory a **si.llog** file. This file includes facts about fanout , fanin, number of loaded nets, number of loaded transistors and number of gates.

NUMBER OF USED ES2 PRIMITIVES :

	NUMBER	TRANSISTORS	CHIP AREA	TOTAL #TRANSTORS
INV	185	4	14.4 * 76.0	740
AND	116	8	34.4 * 76.O	928
OR	372	7	34.4 * 76.0	2604
EXOR	602	11	44.0 * 76.0	6622
LEVEL	31	4	16.4 * 76.0	124
ADD	58	62	178.8 * 76.0	3596
D FF	10	26	92.4 * 76.0	260
LATCH	3	11	40.4 * 76.0	456

The Total Number Primitives and Their included Transistors and Chip Area

With the Netlist + Simulator some problems already ocurred in the SDA design. Such as in the Fetblockcomparator in which the simulator is unable to connect all instances. An in some other sub blocks some problems occurred with the fanout. It is a known problem of SDA which ocuurs at places where bus tabs are added. Only the fcontscdmw block seemed to have no errors at this stage.

The gates in Table 2 are not real gates but equivalent NAND gates which are always a factor 4 smaller than the number of transistors.

	LOADED NETS	# TRANSISTORS	# GATES
PREFET	907	8188	2047
FSMTOT	327	1810	452
FCONTSDMW	85	541	135
FMERGE2	17	170	42
DEMFET	1151	11305	2826
FCOUNTMERGE	11	102	25
FSTATE	32	151	37
FETBLOCKCOMPARATORS :			
FCOMP43BITS	153	657	164
FCOMP37BITS	119	512	128
FADD43BITSD	68	1364	341
FBCOMPSELECT	61	390	97
REST		44	11

3 Netlist + Simulator Calculations