VLSI DESIGN COURSE

VHDL-Modelling And Synthesis
Of The

DLXS

RISC Processor

Martin Gumm, Dezember 1995

University of Stuttgart
Institute of Parallel and Distributed High-Performance Systems
Dept. of Integrated Systems Engineering

last revision: december 28, 1995

This document (in PostScript format) and all VHDL and assembler source files are available via ftp at :
ftp.informatik.uni-stuttgart.de (directory */pub/vhdl/*)

Contents

1 1 . . . . **The DLXS Processor1**

1.1 External View. 2

1.1.1 The DLXS interface 2

1.1.2 Memory interface and access 2

1.1.3 Reset 5

1.1.4 Halt 5

1.1.5 Error 5

1.2 The Programming Model of the DLXS 6

1.2.1 Accessible Registers 6

1.2.2 The DLXS instruction format .6

1.2.3 The DLXS instruction set 8

1.2.4 Operation modes 12

1.2.4.1 Supervisor mode 12

1.2.4.2 User mode 13

1.2.4.3 Error mode 13

1.2.5 Exception and Interrupt Handling 13

1.2.5.1 Software Interrupts 14

1.2.5.2 Hardware Interrupts 15

1.2.5.3 Exception handling 15

1.2.5.4 Return from Interrupt or Exception 16

1.2.6 Reset 17

1.3 Internal Structure of the DLXS. 17

1.3.1 The datapath 18

1.3.2 The controller. 20

1.3.3 The Basic Steps of Execution 22
1.4 References 23
2 The DLXS Test-Bench 25
2.1 The test-bench structure 25
2.2 Address space 25
2.3 System simulation 27
2.4 References 27
3 The Design Libraries 29
3.1 Predefined libraries 29
3.2 Description of the contents of the libraries 30
3.2.1 Library TEACH_EXMPL 30
3.2.2 Library TEACH_LOGIC_LIB 30
3.2.3 Library IEEE_EXTD 30
3.2.4 Library DLX_BENCH 31
3.2.5 Library DLX 32
3.2.5.1 At the beginning of this design course 32
3.2.5.2 To be developed in Part II of this course 32
3.2.5.3 To be added to the library in Part III of this course 32
3.2.5.4 To be developed in Part III of this course 33
3.2.5.5 To be added in Part IV of this course 35
3.2.5.6 To be developed in Part IV of this course 36
3.3 Selection of useful subprograms and definitions contained in the different packages 36
3.3.1 Library STD 36
3.3.1.1 Package STANDARD 36
3.3.1.2 Package TEXTIO 36
3.3.2 Library IEEE 37
3.3.2.1 Package STD_LOGIC_1164 37
3.3.3 Library IEEE_EXTD 37
3.3.3.1 Package STDL1164_EXTENDED 37
3.3.3.2 Package STDL1164_VECTOR_ARITHMETIC 37
3.3.4 Library DLX (WORK) 38
3.3.4.1 Package DLX_INSTRUCTIONS 38

4 VHDL Programming Guidelines . . . 39

4.1 VHDL Programming . . . 39
4.1.1 Design Unit Names . . . 39
4.1.2 Other Names . . . 40
4.1.3 Types . . . 40
4.1.4 Configurations . . . 40
4.1.5 Processes . . . 40

4.2 VHDL Coding And Documentation Style . . . 41
4.2.1 One Design Unit - One File . . . 41
4.2.2 File Header . . . 42
4.2.3 File Names . . . 42
4.2.4 Source Code Outfit . . . 43

5 Tools . . . 45

5.1 The VHDL Prettifier mvp . . . 45

5.2 The DLX Assembler dlx_asm . . . 46

5.3 References . . . 55

6 Exercises . . . 57

Part I Getting started . . . 58

6.1 SYNOPSYS VSS-Core-Tools and first VHDL models . . . 58
Exercise 6.1.1 A first analyze-simulation run . . . 58
Exercise 6.1.2 A RS-flipflop . . . 59
Exercise 6.1.3 Development of a clock generator . . . 61

Part II DLXS behavioural model . . . 63

6.2 Architecture behaviour_1: . . . 63
Exercise 6.2.1 Preparation of the predefined libraries for simulation . . . 63
Exercise 6.2.2 Creation of an entity for the DLXS . . . 63
Exercise 6.2.3 Development of a procedure to read from the system data-bus . . . 64
Exercise 6.2.4 Development of the architecture behaviour_1 . . . 64
Exercise 6.2.5 First test of the model . . . 67

6.3 Architecture behaviour_2 . . . 68
Exercise 6.3.1 Implementation of the arithmetical and logical instructions . . . 68

Exercise 6.3.2 Test of the new implemented instructions 69

6.4 Architecture behaviour_3: 70
Exercise 6.4.1 Implementation of the jump instructions 70
Exercise 6.4.2 Implementation of the branch instructions 70
Exercise 6.4.3 Implementation of the move instructions 70
Exercise 6.4.4 Test of the new implemented instructions 71
Exercise 6.4.5 Implementation of the six load instructions 71
Exercise 6.4.6 Implementation of the six store instructions 71
Exercise 6.4.7 Test of the new implemented instructions 72

6.5 Architecture behaviour_4 73
Exercise 6.5.1 Implementation of the halt function 73
Exercise 6.5.2 Implementation of the hardware interrupt handling . 73
Exercise 6.5.3 Implementation of the software interrupt 74
Exercise 6.5.4 Implementation of the exception handling 75
Exercise 6.5.5 Simulation of the complete behavioural model 76

Part III DLXS register-transfer-level model 77

6.6 Implementation of a structural model of the DLXS 77
Exercise 6.6.1 Implementation of the three different 32-bit latches . 78
Exercise 6.6.2 Implementation of the bus constant circuit 80
Exercise 6.6.3 Implementation of the 32x2:1 multiplexer 81
Exercise 6.6.4 Implementation of the instruction register 81
Exercise 6.6.5 Implementation of the memory data register 82
Exercise 6.6.6 Implementation of the interrupt control register 84
Exercise 6.6.7 Implementation of the ALU 85
Exercise 6.6.8 Implementation of the register file 86
Exercise 6.6.9 Implementation of the structural model of the datapath 87
Exercise 6.6.10 Development of a configuration for the datapath . . 87
Exercise 6.6.11 Simulation of the RTL model of the DLXS 89

Part IV Synthesis of the datapath. 90

6.7 Synthesis of a gate-level netlist from the RTL model of the datapath . 90
6.7.1 The necessary design steps for synthesis 90
Exercise 6.7.2 Packages dlx_types and dlx_instructions 93
Exercise 6.7.3 Synthesis of the three different 32-bit latches 93

Exercise 6.7.4 Synthesis of the bus constant circuit 93
Exercise 6.7.5 Synthesis of the 32x2:1 multiplexer 93
Exercise 6.7.6 Synthesis of the instruction register 94
Exercise 6.7.7 Synthesis of the memory data register 94
Exercise 6.7.8 Synthesis of the interrupt control register 94
Exercise 6.7.9 Synthesis of the ALU 94
Exercise 6.7.10 Synthesis of the register file 95
Exercise 6.7.11 Synthesis of the datapath 96
Exercise 6.7.12 Synthesis of the core 96
Exercise 6.7.13 Synthesis of the DLXS 97

6.8 Gate-level simulation of the complete DLXS 97

6.9 References 98

7 Concluding remarks and outlook 99

List of Figures

External interface of the DLXS 2
Byte positions in a DLXS word (Big Endian) 3
Memory Read Access 4
Memory Write Access 5
The three DLXS instruction formats 7
The ICR register 14
Internal structure of the DLXS 18
Datapath of the DLXS 19
Structure of the controller of the DLXS 21
The Structure of the Test-Bench for the DLX-Processor 26
Generation of ROM descriptions for the simulation of the test-bench 27
Directory Structure and Design-Libraries for this course 29
Where to declare the libraries 41
File header for the design course 42
Comment style for the design course 44
RS-flip containing two cross-coupled NOR-gates with gate delays 59
Two phase, non-overlapping clock signals and the reset signal 61
Structure of the DLXS behavioural model 65
Structure of the execution part of the behavioural model 68
The design hierarchy of the structural processor model 78
32-bit latch and two 32-bit register 79
Bus constant circuit 80

32x2:1 multiplexer 81
Instruction register (IR) 82
Memory data register 83
Interrupt control register (ICR) 84
ALU 86
Register file 87
The complete datapath 88
Inclusion of compiler switches for synthesis 91

List of Tables

The Memory Operations 3
DLXS Arithmetic and Logic Instructions 9
DLXS Test and Set Instructions 10
DLXS Branch Instructions 11
DLXS Special Instructions 11
DLXS Load and Store Instructions 12
Address space seperation by the most significant bit 13
The ALU Operations 20
The addressing space of the DLXS test-bench 26
The S1 and S2 Bus Constants 81
Coding of the MDR shift operations 83
Coding of the ALU operations 86

1
The DLXS Processor

The DLX processor was defined by Hennessy and Patterson in [HePa90] as an hypothetical RISC machine with a simple 32-bit load/store architecture. This processor is well suited for teaching purposes because of its simple instruction set, its single addressing mode, the simple decoding of its instruction set, and its easy understandable architecture. However , this architecture demonstrates all the major features of the RISC principle.

The instruction set of the DLX contains, among others, instructions for signed and unsigned integer arithmetic and for floating point arithmetic. For this design course, only a subset of the original instruction set will be implemented. On the other hand, our processor model is extended by some features: interrupt- and exception handling, three different operation modes (supervisor , user , error), and one additional addressing mode. A pipeline has not yet been included.

The interrupt and exception handling has been derived from the DLXm model which comes with the Alliance design suite [Gre92] [Baz94] and to some limited extend, from the SP ARC architecture [Sparc92]. The memory addressing has also derived (and simplified) from the SP ARC definition. The timing models for the bus trancactions have been derived from the DP32 processor model described in [Ash90]. The processor description in this chapter refers to [HePa90] and the sections 1.2.3 to 1.2.6 additionally to the DLXm description [Baz94].

Our processor architecture is called “DLXS“ in the following description (where ‘S‘ stands for ‘Stuttgart‘ version).

1.1 External View

1.1.1 The DLXS interface

The external interface of the DLXS is shown in figure 1-1. The DLXS has an address bus (*ADDR*) and a bidirectional data bus (*DATA*), both 32 bits wide. The output lines *RW*, *ENABLE* , and *READY* are needed to handle the memory access (see section 1.1.2). The DLXS has an asynchronous reset input (*RESET*) and a disable input (*HALT*). A two phase, non-overlapping clock signal is expected at the clock inputs *PHI1* and *PHI2* . The *ERROR* output indicates to the outside world that the processor has reached an unrecoverable error state.

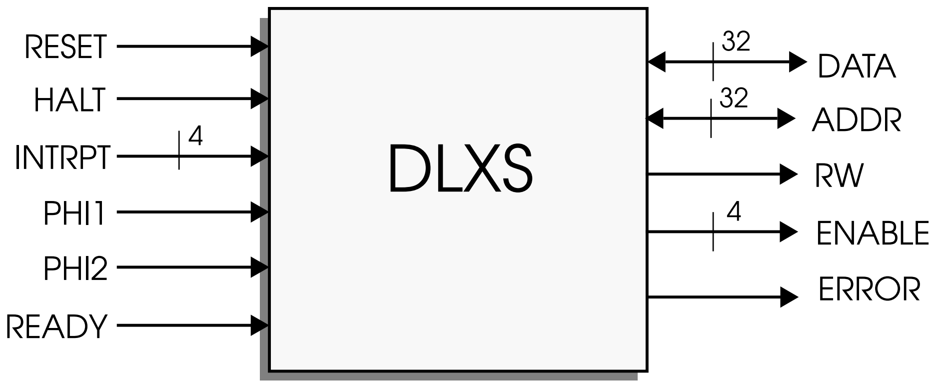

Figure 1-1: External interface of the DLXS

1.1.2 Memory interface and access

A DLXS word is 32 bits long. Memory is byte addressable with a 32-bit address in “Big Endian“ mode, i.e. the byte with the binary address bin“x...x00“ is in the most significant position („big end“) of a 32-bit word (figure 1-2). All memory references are through loads and stores between memory and the general-purpose registers. Accesses can be to byte, to a half word, or to word. In the case of half access the address must be „half-aligned“, i.e. the least significant address bit must be 0, in the case of word access the address must be „word-aligned“, i.e. the two least significant address bits must be 0. All instructions are 32 bits wide and must be word-aligned.

Addressing is only possible in two modes: (immediate displacement or register

Word 0	BYTE 0	BYTE 1	BYTE 2	BYTE 3
Word Bit No.	0(msb) 7	8 15	16 23	24 31(lsb)
Address	bin"x000"	bin"x001"	bin"x010"	bin"x011"
DATA Bus	(0 : 7)	(8 : 15)	(16 : 23)	(24 : 31)

Word 1	BYTE 4	BYTE 5	BYTE 6	BYTE 7
Word Bit No.	0(msb) 7	8 15	16 23	24 31(lsb)
Address	bin"x100"	bin"x101"	bin"x110"	bin"x111"
DATA Bus	(0 : 7)	(8 : 15)	(16 : 23)	(24 : 31)

⋮

Figure 1-2: Byte positions in a DLXS word (Big Endian)

indexed, see section 1.2.2):

```
address = [Rx] + immediate displacement
address = [Rx] + [Ry]
```

During read/write accesses, the four ENABLE outputs and the R W output determine the kind of the bus transaction as stated in table 1-1.

ENABLE(0 : 3)	RW	Transaction	RW	Transaction
1 1 1 1	0	store word	1	load word
0 0 1 1	0	store lower half	1	load lower half
1 1 0 0	0	store upper half	1	load upper half
1 0 0 0	0	store byte 0	1	load byte 0
0 1 0 0	0	store byte 1	1	load byte 1
0 0 1 0	0	store byte 2	1	load byte 2
0 0 0 1	0	store byte 3	1	load byte 3
0 0 0 0	0	idle	1	idle

Table 1-1: The Memory Operations

The timing of the bus read transaction is shown in figure 1-3. During an idle state (T i) the processor places the memory address on the address bus (after the rising edge of phi2) to start the transaction. In the next state (T1), the processor activates the R W and ENABLE lines (after the rising edge of phi1) and waits for the memory to access the data. If the memory has completed its operation during this state (T1) or the following state (T2), it asserts ready and the processor completes the transaction by resetting the ENABLE lines (after the rising edge of phi1) and continues with idle states. Otherwise, the memory leaves the READY line false and the processor repeats T2 states until it detects READY to be true.

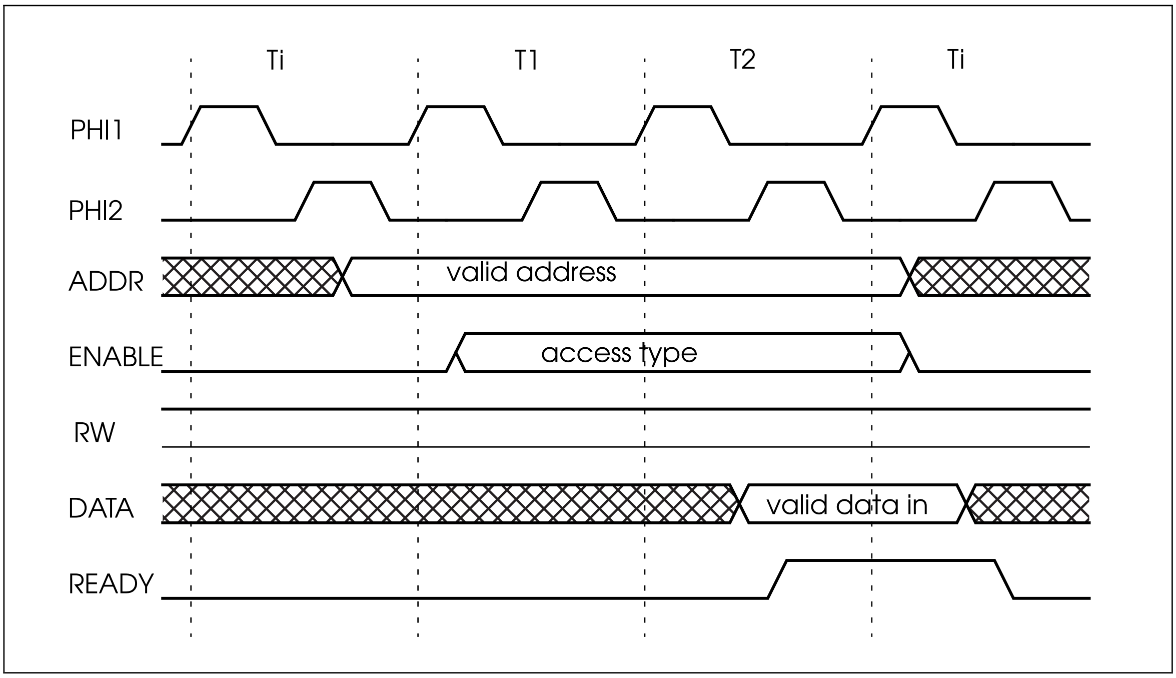

Figure 1-3: Memory Read Access

The timing of the bus write transaction is shown in figure 1-4. During an idle state (T i) the processor places the memory address on the address bus (after the rising edge of phi2) to start the transaction. In the next state (T1) the processor places the data on the data bus (after the rising edge of phi2). In the following T2 state, the processor activates the R W and ENABLE lines (after the rising edge of phi1). If the memory has completed its operation during this state, it asserts ready and the processor completes the transaction by resetting the ENABLE lines (after the rising edge of phi1) and continues with idle states. Otherwise, the memory leaves the READY line false and the processor repeats T2 states until it detects READY to be true.

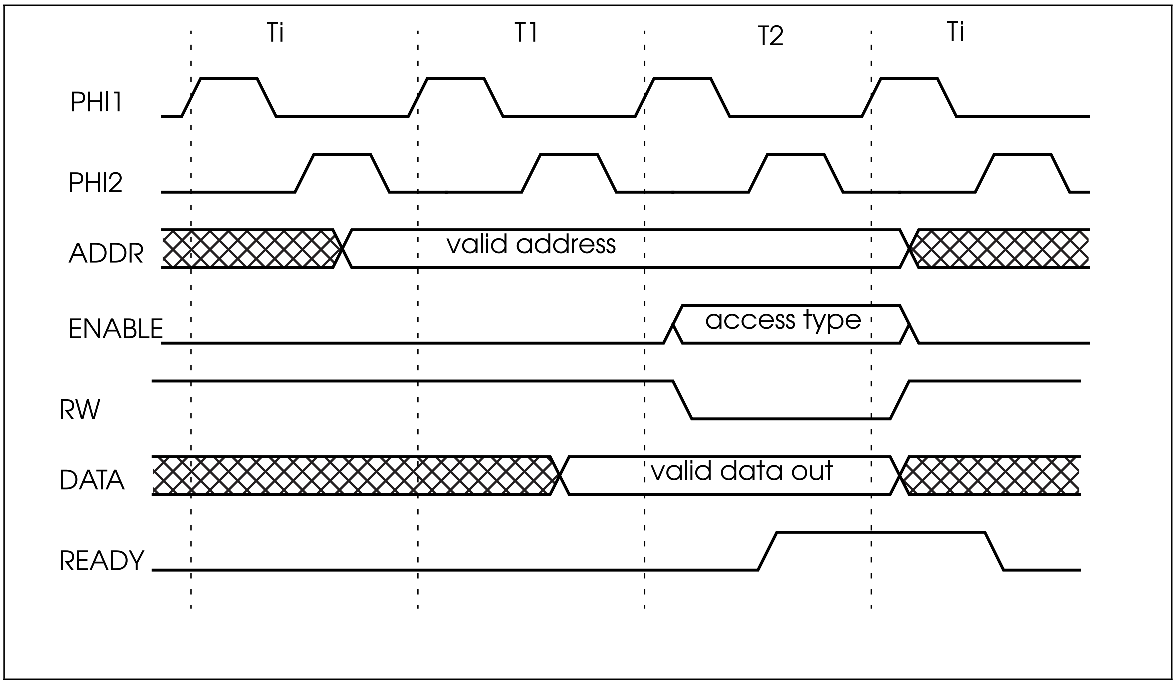

Figure 1-4: Memory Write Access

1.1.3 Reset

An activation of the *RESET* -input (high active) of the DLXS changes the port direction of the bidirectional data-bus *DATA* to input and the outputs *ENABLE* and *RW* are set to zero. The registers affected by the reset are described in section 1.2.6.

1.1.4 Halt

Before fetching an instruction, the DLXS processor checks each time the *HALT* input. If this input is active, the processor changes to the inactive state. All output ports are set to high impedance state and the *DATA* bus is switched to input. The processor stays inactive until the *HALT* signal is set back to zero, and continues afterwards with the normal instruction fetch.

1.1.5 Error

In case of a detection of an error , the processor stops any further operation and changes to the error state (see section 1.2.4 for details). This is indicated by setting the output *ERROR* to one. All output ports are set to high impedance state and the *DATA* bus is switched to input. The processor needs to be restarted (reset).

1.2 The Programming Model of the DLXS

1.2.1 Accessible Registers

The programmer can use the following registers:

- **R0, ..., R31** : 32 general-purpose registers (GPR) of 32 bit. The value of R0 is always 0, i.e. this register can only be read and a write access does not change its content. The register R31 serves also as a address storage when executing call instructions (see section 1.2.2).
- **ICR** : the interrupt control and exception register (see section 1.2.5). The content of this register can be moved to a general-purpose register and vice versa.
- **IAR** : the interrupt address register . This register stores the address of the next instruction when an interrupt is to be executed. The content of this register can be moved to a general-purpose register and vice versa.
- **TBR** : the trap base register . This register stores the base address of the interrupt and exception handling routines in the memory . The content of this register can be moved to a general-purpose register and vice versa.

Other registers are not accessible by the programmer . Among these registers are the program counter (PC) and the instruction register (IR) (see chapter 1.2.2). The DLXS has a load-store architecture, i.e. all arithmetic and logic operations are between GPRs and the memory access is done via those registers.

1.2.2 The DLXS instruction format

All DLXS instructions are 32 bits with a 6-bit primary opcode.There are only three different instruction formats: the J-type, the I-type and the R-type instruction as shown in figure 1-5.

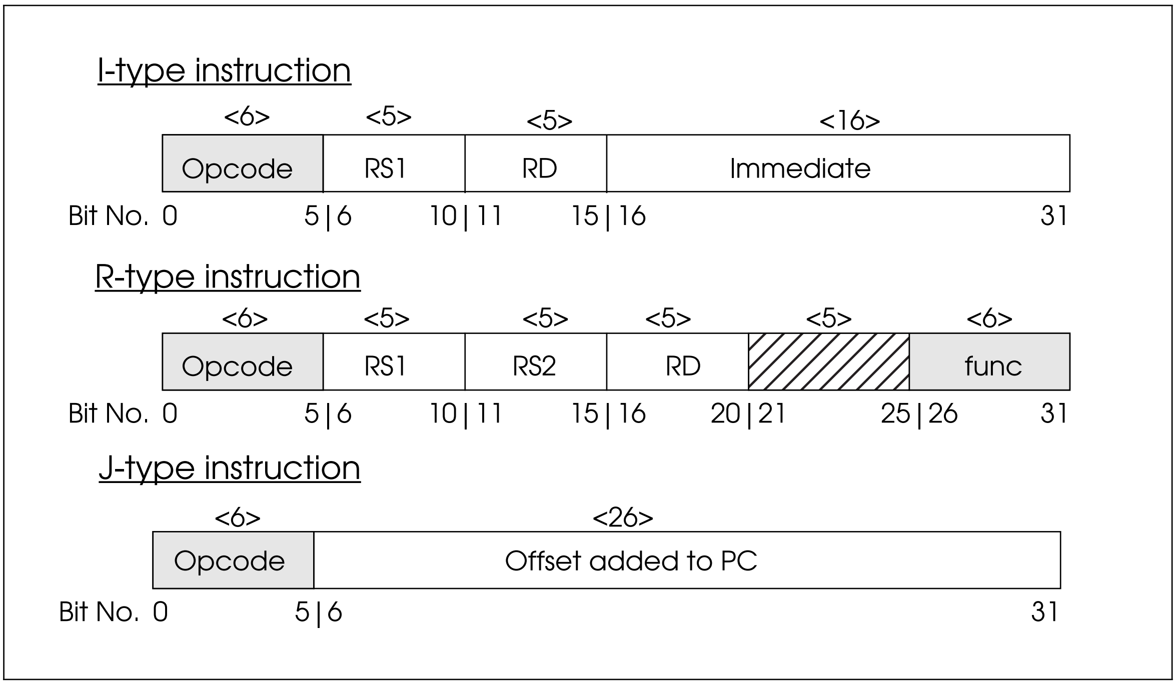

Figure 1-5: The three DLXS instruction formats

The I-type instructions are used to encode the load/store instructions with immediate displacement. In this case, RS1 encodes the GPR which holds the memory address, RD encodes the GPR to read from or to write in respectively , and the 16-bit sign extended immediate is the displacement value. Furthermore, this instruction format encodes the conditional branch instructions where RS1 specifies the GPR, RD is unused and the 16-bit sign-extended immediate specifies the PC-relative destination. Finally , the I-type format encodes the two jump-register instructions where RS specifies the GRP with the absolute jump address and RD and the immediate value are unused.

The R-type instruction encodes the register -register ALU operation where the bit field *func* encodes the ALU operation: $RD \leftarrow RS1\ func\ RS2$ as well as the register indexed load/store instructions. In this case, RS1 encodes the GPR which holds the memory address, RD encodes the GPR to read from or to write in respectively , and RS2 encodes the register which holds the displacement value. Furthermore, the R-type format encodes the two special instructions which move contents from and to special registers IAR, ICR, and TBR where ICR is encoded with bin“00001“, IAR with bin“00010“, and TBR with bin“00100“. RS2 is unused in this case.

The J-type instruction encodes only the unconditional branches where the 26-bit sign-extended immediate is added to the program counter (PC) and two special instructions.

1.2.3 The DLXS instruction set

The DLXS possesses 52 instructions which can be classified as follows:

- 18 arithmetic and logic instructions (including the “no operation“ instruction)
- 12 test instructions
- 6 branch instructions
- 12 memory access instructions
- 4 special instructions

All load/store instructions exist in two formats: addressing with immediate displacement (16-bit) and register indexed, e.g.:

– LW Rd, Rs2(Rs1) (Rd ← Mem[Rs1 + Rs2])

– LW.I Rd, I(Rs1) (Rd ← Mem[Rs1 + I])

The arithmetic and logic operations, except LHI and NOP, are given in two formats: register-register and register immediate, e.g.:

– ADD Rd, Rs1, Rs2 (Rd ← Rs1 + Rs2)

– ADD.I Rd, Rs1, I (Rd ← Rs1 + I)

They are shown in table 1-2. The 16-bit immediate value I is sign-extended for arithmetical instructions and zero extended for logical instructions. The instruction LHI allows to write a 16-bit immediate value in the upper half word of a GRP whereas the lower half word is set to zero.

Assembly instr.		Meaning	Operation	F	Opcode/ RRA-func
ADD	Rd, Rs1, Rs2	addition	Rd ← Rs1 + Rs2 (overflow -> exception)	R	000_000 000_100
SUB	Rd, Rs1, Rs2	subtraction	Rd ← Rs1 - Rs2 (overflow -> exception)	R	000_000 000_110
AND	Rd, Rs1, Rs2	logical AND	Rd ← Rs1 AND Rs2	R	000_000 001_000
OR	Rd, Rs1, Rs2	logical OR	Rd ← Rs1 OR Rs2	R	000_000 001_001
XOR	Rd, Rs1, Rs2	logical XOR	Rd ← Rs1 XOR Rs2	R	000_000 001_010
SLL	Rd, Rs1, Rs2	shift left logical	Rd ← Rs1 logical shifted left by Rs2 (5 lsb of Rs2 are significant)	R	000_000 001_100
SRL	Rd, Rs1, Rs2	shift right logical	Rd ← Rs1 logical shifted right by Rs2 (5 lsb of Rs2 are significant)	R	000_000 001_110
SRA	Rd, Rs1, Rs2	shift right arithmetical	Rd ← Rs1 arithmetical shifted right by Rs2 (5 lsb of Rs2 are significant)	R	000_000 001_111
ADD.I	Rd, Rs1, I	addition	Rd ← Rs1 + I (overflow -> exception) (I : sign extended 16-bit immediate)	I	010_100
SUB.I	Rd, Rs1, I	subtraction	Rd ← Rs1 - I (overflow -> exception) (I : sign extended 16-bit immediate)	I	010_110
AND.I	Rd, Rs1, I	logical AND	Rd ← Rs1 AND I (I : zero extended 16-bit immediate)	I	011_000
OR.I	Rd, Rs1, I	logical OR	Rd ← Rs1 OR I (I : zero extended 16-bit immediate)	I	011_001
XOR.I	Rd, Rs1, I	logical XOR	Rd ← Rs1 XOR I (I : zero extended 16-bit immediate)	I	011_010
SLL.I	Rd, Rs1, I	shift left logical	Rd ← Rs1 logical shifted left by I (5 lsb of immediate I are significant)	I	011_100
SRL.I	Rd, Rs1, I	shift right logical	Rd ← Rs1 logical shifted right by I (5 lsb of immediate I are significant)	I	011_110
SRA.I	Rd, Rs1, I	shift right arithmetical	Rd ← Rs1 arithmetical shifted right by I (5 lsb of immediate I are significant)	I	011_111
LHI	Rd, I	load high immediate	Rd(0:15) ← I Rd(16:31) ← hex0000 (I : 16-bit immediate)	I	011_011
NOP		no operation	<do nothing>	R	000_000 000_000

Table 1-2: DLXS Arithmetic and Logic Instructions

There are 12 test instructions which test a relation (e.g. greater , lower , ...) between either the contents of two GPRs or the content of one GPR and a 16-bit sign-extended immediate. If the result is true, the destination register is set to 1, otherwise it is set to zero. The test instructions are shown in table 1-3.

The DLXS possesses only two conditional branch instructions: “branch on equal

Assembly instr.	Meaning	Operation	F	Opcode/ RRA-func
SEQ Rd, Rs1, Rs2	set if equal	Rs1 = Rs2 : Rd ← hex00000001 else : Rd ← hex00000000	R	000_000 010_000
SNE Rd, Rs1, Rs2	set if not equal	Rs1 /= Rs2 : Rd ← hex00000001 else : Rd ← hex00000000	R	000_000 010_010
SLT Rd, Rs1, Rs2	set if lower than	Rs1 < Rs2 : Rd ← hex00000001 else : Rd ← hex00000000	R	000_000 010_100
SLE Rd, Rs1, Rs2	set if lower or equal	Rs1 <= Rs2 : Rd ← hex00000001 else : Rd ← hex00000000	R	000_000 010_110
SGT Rd, Rs1, Rs2	set if greater than	Rs1 > Rs2 : Rd ← hex00000001 else : Rd ← hex00000000	R	000_000 011_000
SGE Rd, Rs1, Rs2	set if greater or equal	Rs1 >= Rs2 : Rd ← hex00000001 else : Rd ← hex00000000	R	000_000 011_010
SEQ.I Rd, Rs1, I	set if equal	Rs1 = I : Rd ← hex00000001 else : Rd ← hex00000000 (I : sign extended 16-bit immediate)	I	100_000
SNE.I Rd, Rs1, I	set if not equal	Rs1 /= I : Rd ← hex00000001 else : Rd ← hex00000000 (I : sign extended 16-bit immediate)`	I	100_010
SLT.I Rd, Rs1, I	set if lower than	Rs1 < I : Rd ← hex00000001 else : Rd ← hex00000000 (I : sign extended 16-bit immediate)	I	100_100
SLE.I Rd, Rs1, I	set if lower or equal	Rs1 <= I : Rd ← hex00000001 else : Rd ← hex00000000 (I : sign extended 16-bit immediate)	I	100_110
SGT.I Rd, Rs1, I	set if greater than	Rs1 > I : Rd ← hex00000001 else : Rd ← hex00000000 (I : sign extended 16-bit immediate)	I	101_000
SGE.I Rd, Rs1, I	set if greater or equal	Rs1 >= I : Rd ← hex00000001 else : Rd ← hex00000000 (I : sign extended 16-bit immediate)	I	101_010

Table 1-3: DLXS Test and Set Instructions

zero“ and “branch on not equal zero“ which test the content of a GRP for those conditions. Besides the two conditional branch instructions, there are four unconditional instructions. The J (jump) instruction uses a 26-bit sign-extended immediate to determine the jump destination (PC relative) whereas the JR (jump register) instruction uses the content of a GRP as absolute jump destination. Both instructions exist also for procedure calls (JAL, JALR: jump and link, jump and link register) where the address of the next instruction after the call is stored in R31. The branch instructions are shown in table 1-4.

The DLXS possesses four special instructions. Two of them decode moves between a special register, i.e. the ICR the IAR, or the TBR, and a GPR register.

Assembly instr.	Meaning	Operation	F	Opcode/ RRA-func
BEQZ Rs, label	branch if equal zero	Rs = 0 : PC ← PC + 4 + Label Rs = 0 : PC ← PC + 4 (Label : sign extended 16-bit immediate)	I	010_000
BNEZ Rs, Label	branch if not equal zero	Rs /= 0 : PC ← PC + 4 + Label Rs = 0 : PC ← PC + 4 (Label : sign extended 16-bit immediate)	I	010_001
J Label	jump	PC ← PC + 4 + sign_extd(imm26) Llabel : sign extended 16-bit immediate)	J	001_100
JAL Label	jump and link	R31 ← PC + 4 PC ← PC + 4 + sign_extd(imm26) (Label : sign extended 16-bit immediate)	J	001_101
JR Rs	jump register	PC ← Rs	I	001_110
JALR Rs	jump and link register	R31 ← PC + 4 PC ← Rs	I	001_111

Table 1-4: DLXS Branch Instructions

Furthermore, there are two instructions for interrupt or exception handling, see section 1.2.5. The special instructions are shown in table 1-5.

Assembly instr.	Meaning	Operation	F	Opcode/ RRA-func
TRAP Label	trap (software interrupt)	IAR ← PC ICR(0:15) ← ICR(16:31) ICR(7) ← 1 (set S bit) PC ← TBR + Label (Label : 26-bit sign extended immediate)	J	010_011
RFE **(supervisor mode)**	return from interrupt or exception	ICR(16:31) ← ICR(0:15) (no clear of S bit !) PC ← IAR	J	010_010
MOVS2I Rs, S **(supervisor mode)**	move special register to gp-register	Rs ← S (S can be IAR, ICR, or TBR)	R	000_000 00_011
MOVi2S S, Rs **(supervisor mode)**	move gp-register to special register	S ← Rs (S can be IAR, ICR, or TBR)	R	000_000 00_010

Table 1-5: DLXS Special Instructions

The last class of instructions are the 12 load-store instructions as described in section 1.1.2. These instructions are shown in table 1-6.

Assembly instr.		Meaning	Operation	F	Opcode/ RRA-func
LW	Rd, Rs2 (Rs1)	load word	Rd ← M(Rs1 + Rs2) (word aligned address : least two bits = 00)	R	000_000 100_000
SW	Rs2 (Rs1), Rd	store word	M(Rs1 + Rs2) ← Rd (word aligned address : least two bits = 00)	R	000_000 101_000
LH	Rd, Rs2 (Rs1)	load half	Rd(16:31) ← M(Rs1 + Rs2) (Rd sign extended to 32 bit) (half aligned address : least bit = 0)	R	000_000 100_001
SH	Rs2 (Rs1), Rd	store half	M(Rs1 + Rs2) ← Rd(16:31) (half aligned address : least bit = 0)	R	000_000 101_001
LB	Rd, Rs2 (Rs1)	load byte	Rd(24:31) ← M(Rs + Rs2) (I : sign extended 16-bit immediate)	R	000_000 100_010
SB	Rs2 (Rs1), Rd	store byte	M(Rs1 + Rs2) ← Rd(24:31)	R	000_000 101_010
LW.I	Rd, I (Rs)	load word	Rd ← M(Rs + I) (I : sign extended 16-bit immediate) (word aligned address : least two bits = 00)	I	000_100
SW.I	I (Rs), Rd	store word	M(Rs + I) ← Rd (I : sign extended 16-bit immediate) (word aligned address : least two bits = 00)	I	001_000
LH.I	Rd, I (Rs)	load half	Rd(16:31) ← M(Rs + I) (Rd sign extended to 32 bit) (I : sign extended 16-bit immediate) (half aligned address : least bit = 0)	I	000_101
SH.I	I (Rs), Rd	store half	M(Rs + I) ← Rd(16:31) (I : sign extended 16-bit immediate) (half aligned address : least bit = 0)	I	001_001
LB.I	Rd, I (Rs)	load byte	Rd(24:31) ← M(Rs + I) (Rd sign extended to 32 bit) (I : sign extended 16-bit immediate)	I	000_110
SB.I	I (Rs), Rd	store byte	M(Rs + I) ← Rd(24:31) (I : sign extended 16-bit immediate)	I	001_010

Table 1-6: DLXS Load and Store Instructions

1.2.4 Operation modes

The DLXS possesses three different states of operation: the user mode, the supervisor mode, and the error state.

1.2.4.1 Supervisor mode

When the processor is reset, it starts in the supervisor mode and fetches the first instruction at the address hex“0000_0000“ (startup routine). The S (supervisor) bit in the ICR is set to one (see section 1.2.5). In this mode, all instructions are executable, and all addresses of the 32-bit address space are allowed.

The most significant bit of the address space indicates the type of memory as stated in table 1-7.

MSB	address range	memory accessible for
0	hex"0000_0000" - hex"7FFF_FFF"	only supervisor
1	hex"8000_0000" - hex"FFFF_FFFF"	supervisor and user

Table 1-7: Address space seperation by the most significant bit

All interrupt and exception handling is done in supervisor mode, thus the supervisor ROM contains normally the trap table and the concerning routines. Furthermore, the startup routine has to be included in this ROM. The startup routine must place the address of the interrupt and exception handling routine in the TBR and can initialize the interrupt mask bits of the ICR.

If an exception occurs while the DLXS is in this mode, it changes to the error mode whereas interrupts are allowed.

1.2.4.2 User mode

This is the normal operation mode of the DLXS. Except for the TRAP instruction, no special instruction is executable in this mode and such a instruction will cause an exception. A memory access (instruction or data) or a branch which aims the part of the memory reserved for the supervisor mode, will also cause an exception (see section 1.2.5).

1.2.4.3 Error mode

If an exception had occurred while the DLXS was in supervisor mode, the DLXS stops all operation and sets the ERROR output to one. The processor needs to be restarted.

1.2.5 Exception and Interrupt Handling

The DLXS possesses two interrupt types: hardware and software interrupts. They can be controlled by the programmer . Additionally , exceptions can occur during the operation, e.g. overflows or misaligned data accesses. All status information, necessary for the exception and interrupt handling, is contained in the interrupt control register (ICR) which is shown in figure 1-6. The lower half

	saved Intrpt(0:3)	saved Mask(0:3)	saved bits 24-31	Intrpt(0:3)	Mask(0:3)	PRIV	OVAR	OVAD	DAV	IAV	IRRA	IOC	S
Bit No.	0:3	4:7	8:15	16:19	20:23	24	25	26	27	28	29	30	31

Interrupt bits

Intrpt(x) Set to 1 when Intrpt pin no. x is active (at least one cycle)

Mask(x) Mask interrupt no. x (enabled if set to 1)

S 1: Super User
0: User
(set by hardware)

Exception bits (high active)

IOC	Illegal Opcode
IRRA	Illegal RRA-Function
IAV	Instruction Address Violation
DAV	Data Address Violation
OVAD	Address Overflow
OVAR	Arithmetic Overflow
PRIV	Priority Violation

Figure 1-6: The ICR register

word of this register contains the supervisor bit, the seven exception bits, the four interrupt-mask bits, and the four interrupt bits; the upper half word serves as storage of the interrupt and exception information. All bits can be read and written by the programmer (in supervisor mode). The interrupt bits (bits 16-19), the exception bits (bits 24-30), and the S bit (bit 31) are also modified by the hardware. Thus, it is possible to enable an interrupt or exception by writing into the ICR.

In case of an interrupt or exception, the ICR bits 16 to 31 are shifted into the bits 0 to 15 (save) and thereby cleared. Thus, no further interrupt can occur . When returning from an interrupt or exception, the ICR bits 0 to 15 are shifted into the bits 16 to 31 (restore). Thus, before an RFE instruction, the programmer has to take care that all necessary bits (saved interrupt information) in the upper half word are cleared.

The interrupt address register (IAR) stores the old value of the PC in case of an interrupt or exception. The trap base register (TBR) contains the base address of the interrupt and exception handling routine.

1.2.5.1 Software Interrupts

The instruction TRAP allows the execution of a software interrupt, e.g. to simulate step-by-step execution, to set a break point, to execute an IO operation and so on. In this case, the DLXS performs the following operations:

- the content of the PC, i.e. the address of the next instruction to be executed after the interrupt, is stored in the IAR register ,
- the old interrupt information contained in the ICR is saved (as described above) and the S bit is set to one,
- the processor branches to the labelled interrupt routine (the new instruction address is the result of the addition of the TBR content and the 26-bit sign extended immediate).

1.2.5.2 Hardware Interrupts

The DLXS possesses four interrupt inputs (INTRPT[0:3]). If one of these inputs is set to 1 for the duration of one cycle, the corresponding bit in the ICR (Intrpt[0:3]) is set to 1 and keeps its value. If this interrupt is masked, i.e. the corresponding mask bit (Mask[0:3]) is set to 1, the interrupt is taken into account after the complete execution of the current instruction. In this case, the DLXS performs the following operations:

- the content of the PC, i.e. the address of the next instruction to be executed after the interrupt, is stored in the IAR,
- the old interrupt information contained in the ICR is saved (as described above) and the S bit is set to one,
- branch to the address which is stored in the TBR.

Because of the shift operation of the ICR, no further interrupt can be taken into account during the execution of the interrupt handling routine (only exception could occur).

1.2.5.3 Exception handling

Seven incidents can cause an exception:

- an illegal opcode has been detected (IOC),
- an illegal register -register ALU function has been detected

(IRRA),

– an instruction address violation (IA V) has occurred (not a word-aligned address),

– an data address violation (DA V) has occurred (not a word-aligned address in case of word transfer or not a half-aligned address in case of half transfer),

– an overflow has occurred while incrementing the PC (overflow of address, OV AD),

– an overflow has occurred while executing an arithmetic instruction (OV AR),

– a priority violation has occurred while the processor was in user mode, i.e. a special instruction (except TRAP) should be executed or a memory operation tried to access the supervisor segment or a branch target is in the supervisor segment (PRIV).

If one of those incidents has been detected, the execution of the current instruction is stopped immediately to insure that the faulty instruction does no damage to a visible register of the DLXS or to the memory . The DLXS performs the following operations:

– the content of the PC, i.e. the address of the next instruction to be executed after the interrupt, is stored in the IAR,

– the old interrupt information contained in the ICR is saved (as described above) and the S bit is set to one,

– the corresponding exception bit in the ICR is set to 1

– branch to the address which is stored in the TBR.

The exceptions can not be masked.

1.2.5.4 Return from Interrupt or Exception

The instruction RFE allows to resume from the startup routine or the handling of a interrupt or exception. Normally , all exceptions are “deadly“, i.e. it is not possible to continue the regular execution of the program. Only the arithmetic overflow could be ignored. Anyway, it is up to the programmer of the exception handling routine to decide if the processor shall continue executing the faulty program or not. The instruction RFE let the DLXS perform the following instructions:

- the old value of the ICR is restored, i.e. the upper half word is shifted into the lower half word of the ICR. The previous interrupt information is thereby restored so that another interrupt can be processed. The exception bits and the supervisor bit are cleared by the shift.
- branch to the address which is stored in the IAR.

1.2.6 Reset

An activation of the *RESET* -input (high active) of the DLXS resets the following registers to zero:

- PC (program counter)
- TBR (trap base register)

The ICR is set to zero except the S bit which is set to one. No other registers are changed.

1.3 Internal Structure of the DLXS

The internal structure of the DLXS is quite simple and easy to understand. It consists of two main parts: the datapath and the controller as shown in figure 1-7. The datapath executes all operations on data. It contains all registers (including the instruction register), the *ALU*, and the internal data buses for the interconnect (see section 1.3.1). The controller (see section 1.3.2) generates the sequence of control signals which are necessary for the correct flow of the data in the datapath. The signals, exchanged by the two main parts, are separated into five classes. First, the control signals from the controller to the datapath which can change their value after the rising edge of *phi1* ($c(\Phi1)$) or after the rising edge of *ph2* ($c(\Phi2)$). Furthermore, signals which transmit status information from the datapath to the controller. They can change their value after the rising edge of *phi1* ($d(\Phi1)$) or after the rising edge of *ph2* ($d(\Phi2)$).

Finally, there are the signals which transmit the content of the *IR* to the controller.

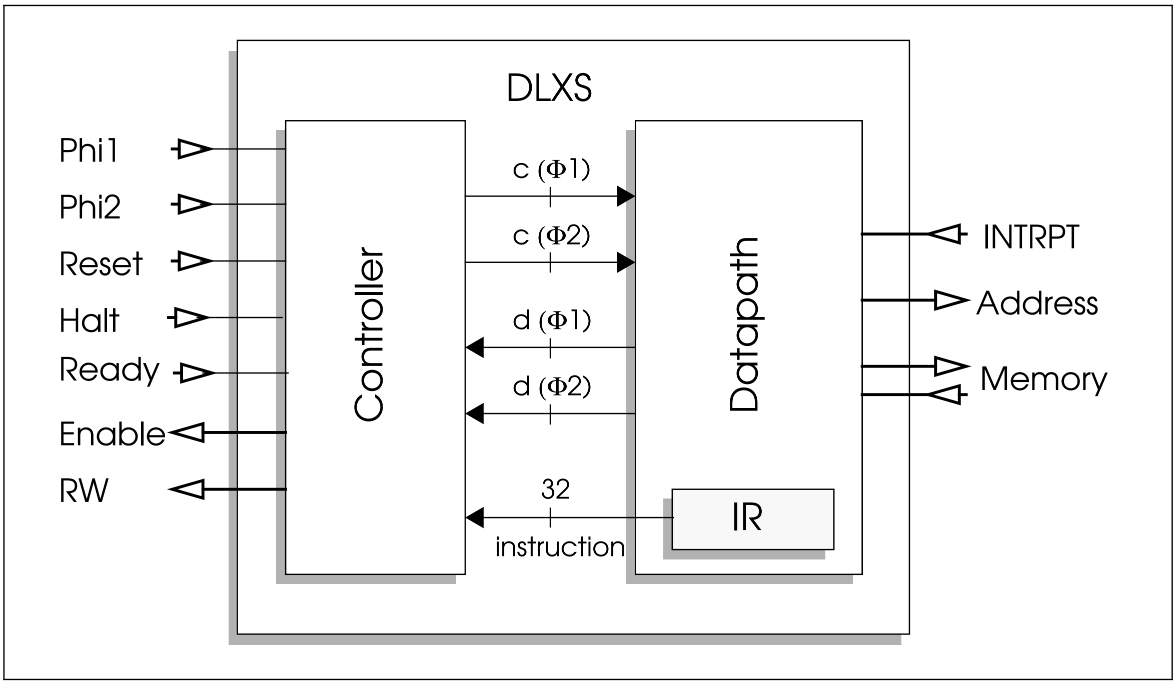

Figure 1-7: Internal structure of the DLXS

1.3.1 The datapath

The structure of the DLXS datapath is depicted in figure 1-8.The GPRs *R0* - *R31* are contained in the register file. The function of the *ICR*, the *IAR*, and the *TBR* has already been described. The *PC* holds the address of the instruction which is to be executed next, the *IR* holds the current instruction. The memory data register (*MDR*) contains the data to be written into the memory in case of a write access or the data read from the memory in case of a read access respectively. The memory address register (*MAR*) contains the address of the concerned memory cell in such cases. The *MAR* can also be used as a temporary register to store intermediate results of a calculation. The processor uses three internal buses: the source1 bus (*S1*), the source2 bus (*S2*), and the destination bus (*Dest*). The fundamental operation of the datapath is reading operands from the register file, operating on them in the *ALU*, and then writing the result back to the register file. Since the register file does not need to be read and written every clock cycle, this sequence is broken into multiple clock cycles to allow for shorter clock periods. Thus, in the datapath there are latches on the two outputs of the register file (called *A* and *B*) and a latch on the input. Two

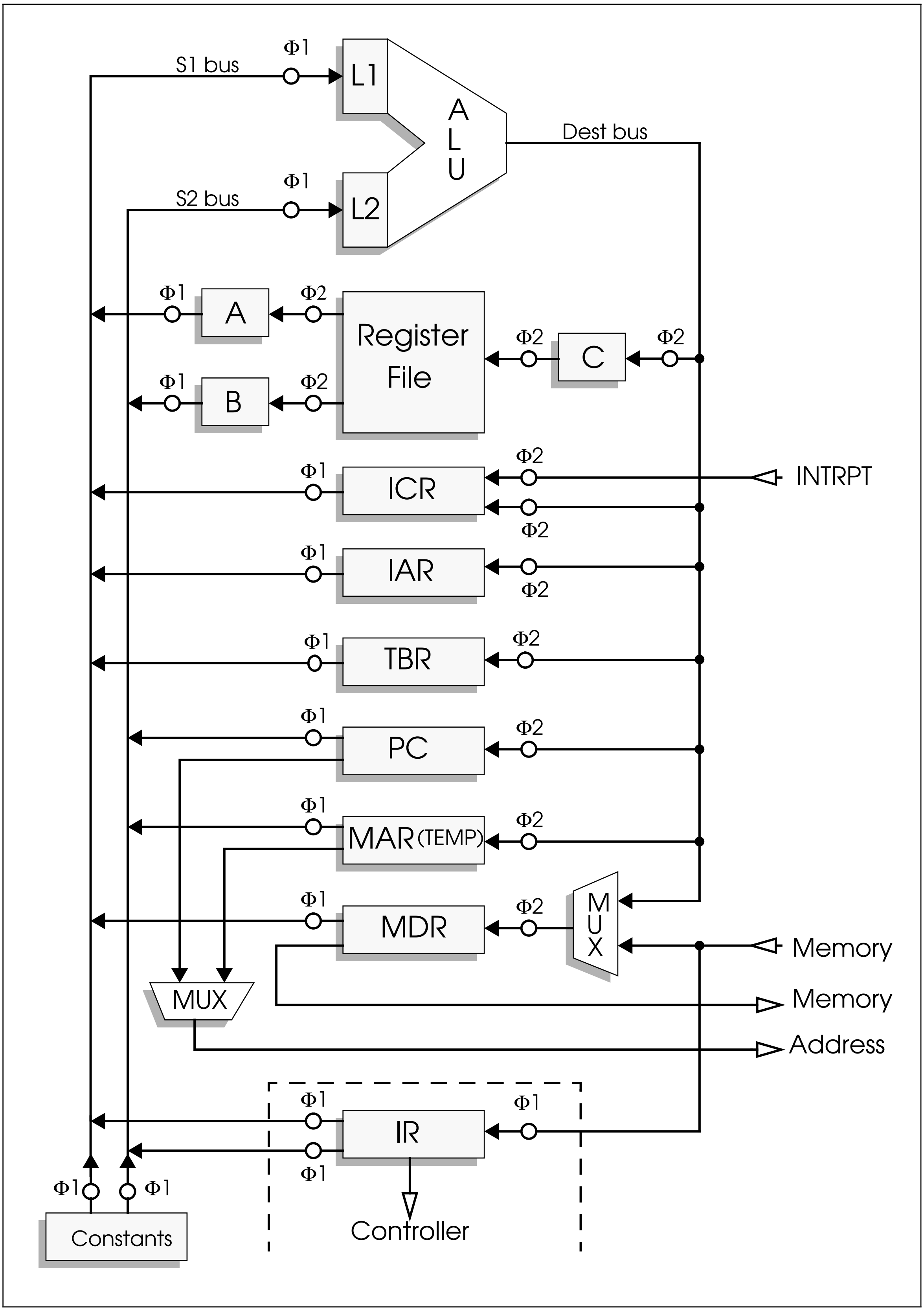

Figure 1-8: Datapath of the DLXS

more latches are on the input of the *ALU* to store the input values of *S1* and *S2*. Note that the only path from the *S1* and *S2* buses to the Dest bus are through the ALU. The ALU can perform the operations shown in table 1-8.

Operation	Operands
pass S1	Dest := S1
pass S2	Dest := S2
signed addition	Dest := S1 + S2
signed subtraction	Dest := S1 - S2
logical shift left	Dest := S1 shifted left logical by S2 (5 least significant bits of S2)
logical shift right	Dest := S1 shifted right logical by S2 (5 least significant bits of S2)
arithmetical shift right	Dest := S1 shifted right arithmetical by S2 (5 least significant bits of S2)
logical and	Dest := S1 AND S2
logical or	Dest := S1 OR S2
logical xor	Dest := S1 XORS2

Table 1-8: The ALU Operations

Furthermore, there are some constants which can be connected to the S2 bus. These constants are needed for the execution of several instructions as explained in chapter 6, table 6-1.

In figure 1-8, the control lines of the input and output of all registers are depicted as circles. The control lines which change their value after the rising edge of *phi1* are tagged with $\Phi1$, those which change their value after the rising edge of *phi2* are tagged with $\Phi2$. In one clock cycle, it is possible to read from the source buses ($\Phi1$), accumulate the result ($\Phi1$), and write back the result ($\Phi2$), e.g. read A and B, add A and B, and write result into C.

1.3.2 The controller

The structure of the DLXS controller is depicted in figure 1-9. It consists of the central finite state machine (*FSM*), the instruction register (IR), three instruction decoders, and some additional logic.

The *FSM* has 64 different states which can change with the rising edge of *phi1*. It generates seven groups of control signals: *rs1_enable, rs2_enable,*

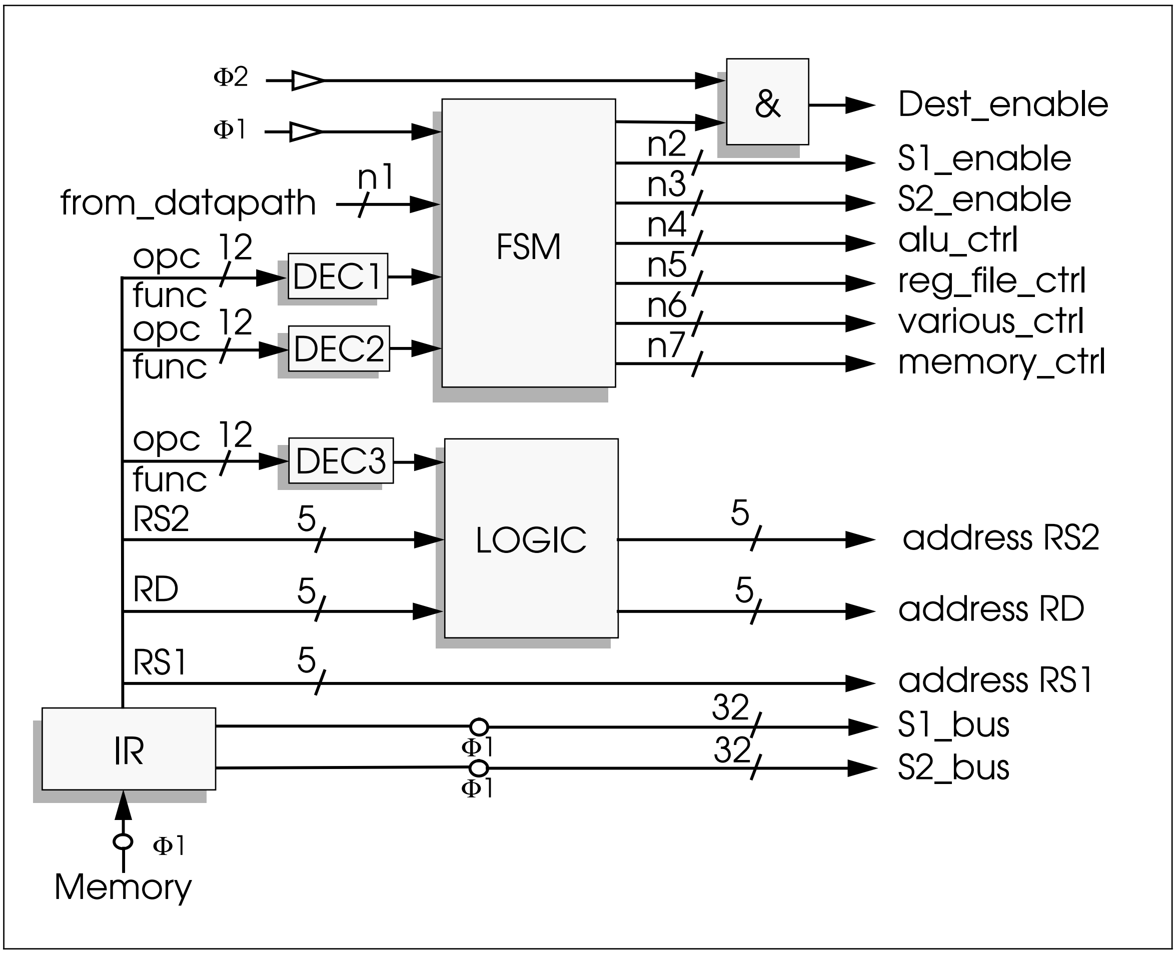

Figure 1-9: Structure of the controller of the DLXS

dest_enable, alu_ctrl, reg_file_ctrl, memory_ctrl, and various_ctrl. The signal groups *rs1_enable* and *rs2_enable* are composed of the output enable signals for all registers which are connected to the *S1* bus and the *S2* bus respectively. The *dest_enable* signals enable the load from the *Dest* bus for all registers which are connected to it. They are gated with *phi2*. The *alu_ctrl* signals select the required ALU function, the *reg_file_ctrl* signals control the load from the *C* register into the register file and the load from it into the *A* and *B* registers. The *memory_ctrl* signals are used for controlling the memory operations. Finally, there are some additionally control signals which are grouped in *various_ctrl*.

The instruction decoder *DEC1* decodes all instructions except for the memory instructions which are decoded in decoder *DEC2*. The decoder *DEC3* is used to generate some control signals for the generation of the register-file addresses. The IR contains the actual instruction, it possesses two outputs connected to the *S1* and *S2* busses for the immediate constants.

1.3.3 The Basic Steps of Execution

For the DLXS instruction set, the instructions can be broken into five basic steps: fetch, decode, execute, memory access, and write result. Each step may take one or several clock cycles.

1. **Instruction fetch step** :
   IR $\leftarrow$ M[PC]
   Fetch instruction from memory .

2. **Instruction decode / register fetch step:**
   A $\leftarrow$ Rs1; B $\leftarrow$ Rs2; PC $\leftarrow$ PC + 4;
   Decode the instruction. Access the register file to read the registers. This can be done in parallel with the decoding because the source registers have always the same location in the instruction formats (fixed-field decoding). Thus the A and B registers are loaded always in this step, regardless if their contents will be used afterwards or not). Increment the PC to point to the next instruction.

3. Execution / effective address step:

   a) Memory reference:
   MAR $\leftarrow$ A + $(IR_{16})^{16}$##$IR_{16..31}$; (immediate displacement)
   MAR $\leftarrow$ A + B (register displacement)
   MDR $\leftarrow$ RD
   The ALU is adding the operands to form the effective address. The MDR is loaded for a store.

   b) ALU instruction:
   C $\leftarrow$ A op B (reg.-reg. ALU instr .)
   C $\leftarrow$ A op $(0)^{16}$##$IR_{16..31}$ (ANDI, ORI, XORI)
   C $\leftarrow$ A op $(IR_{16})^{16}$##$IR_{16..31}$ (other immediate instr .)
   C or Special Reg. $\leftarrow$ Special Reg. or A (MOVI2S, MOVS2I)
   The ALU is performing the specified operation. The result is stored in C or a special register .

   c) Branch/Jump:
   cond $\leftarrow$ A op 0 (cond. branch instr .)
   PC $\leftarrow$ PC + $(IR_{6})^{6}$##$IR_{26..31}$ (uncond. jump instr .)

$PC \leftarrow TBP + (IR_6)^6 \#\# IR_{26..31}$ (TRAP instr .)

In case of a conditional branch, the ALU performs a relative operation (zero or not zero). In case of unconditional jump instruction, the ALU is adding the two operands to form the effective branch address which is stored in the PC. In case of a jump-and-link instruction, the PC is saved in the IAR before. In case of the TRAP instruction, the effective address is the sign-extended immediate. Again, the PC has to be saved in the IAR before.

4. Memory access / branch completion step:

a) Memory reference:
$MDR \leftarrow M[MAR]$; $C \leftarrow MDR$ (load instr .)
$M[MAR] \leftarrow MDR$ (store instr .)
Access memory if needed.

b) Conditional branch:
if (cond) $PC \leftarrow PC + (IR_{16})^{16} \#\# IR_{16..31}$
In case of a conditional branch, add the two operands to form the effective branch address and store the result in the PC if cond is true.

5. Write back step:
$Rd \leftarrow C$
Write the result into the register file.

1.4 References

[HePa90] J.H. Hennessy , D.A. Patterson: *Computer Architecture A Quantitative Approach* , Morgan Kaufman Publishers, 1990

[Ash90] P.J. Ashenden: *The VHDL Cookbook* , First Edition, University of Adelaide, Australia, 1990

[Sparc92] Sparc International Inc: *The SP ARC Architecture Manual (Version 8)* , Prentice Hall, 1992]

[Gre92] A. Greiner et al.: *ALLIANCE A Complete Set of CAD T ools* *for Teaching VLSI Design*, Proc. of the 3. Eurochip Workshop on VLSI Design Training, Grenoble, France, 1992

[Baz94] P. Bazargan et al.: *A Tutorial for Advanced VLSI Course: Designing the 32-bit DLX Microprocessor with the ALLIANCE CAD System*, Proc. of the 5. Eurochip Workshop on VLSI Design Training, Dresden, Germany, 1994

2
The DLXS Test-Bench

2.1 The test-bench structure

A complete system including the socket for the DLXS-processor , supervisor and user RAMs, supervisor and user ROMs, an address-decoder , a clock-generator , and a timer is already predefined and stored as a VHDL-library . The general structure of a one-board computer system has also been used in [Baz94]. It is depicted in figure 2-1 .

Hence, it is possible to "plug-in" your own model of the DLX just by changing the top-level configuration file and to validate it by simulating the entire system.

2.2 Address space

As already mentioned in chapter 1, table 1-7, the address space is separated into a a user segment (which can also be used in the supervisor mode) and a supervisor segment. The detailed partitioning of the address space of test-bench is shown in table 2-1. The entries in the column "Type" indicate whether the address region belongs to the user ('U') and/or to the system address space ('S').

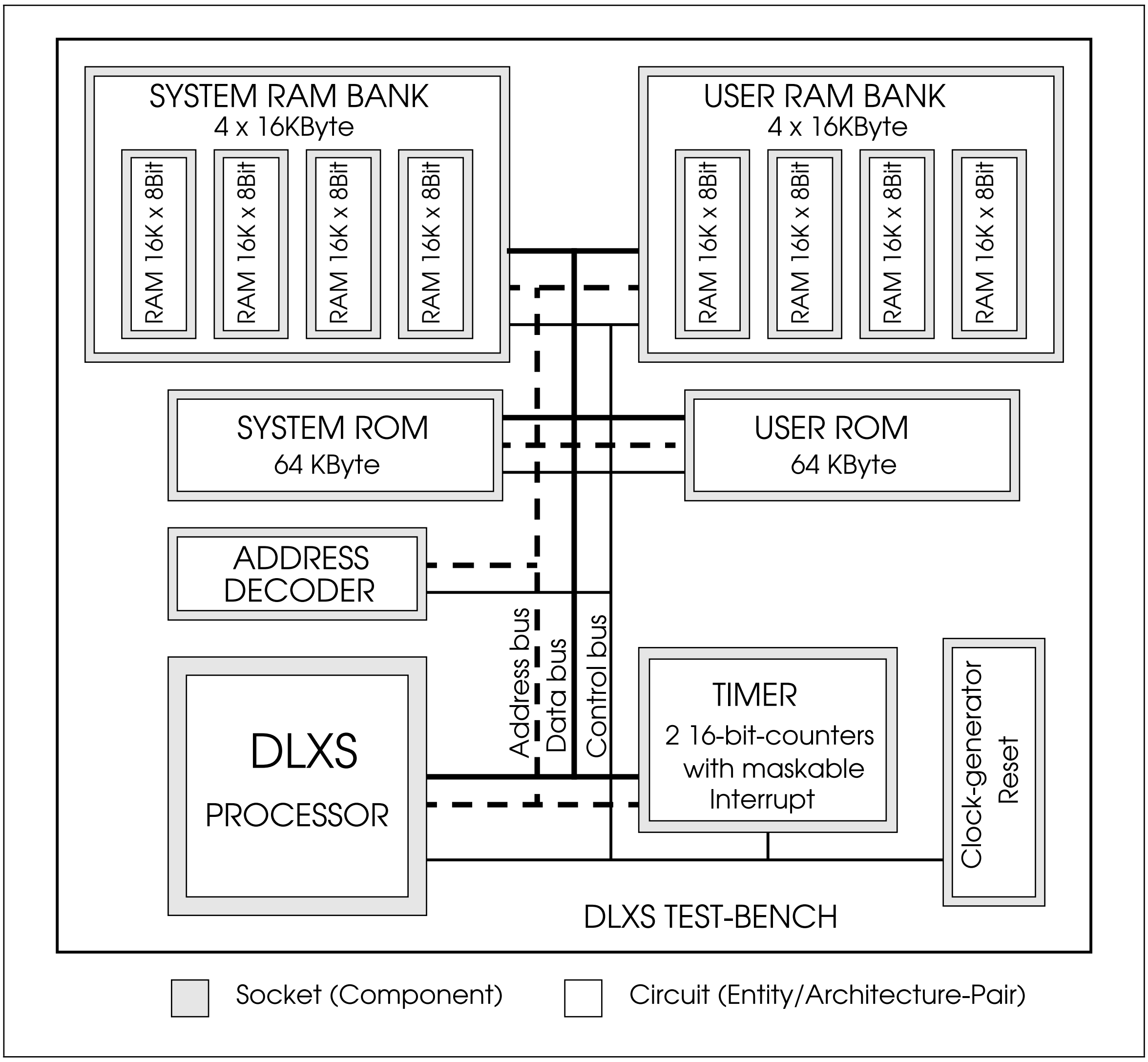

Figure 2-1: The Structure of the Test-Bench for the DLX-Processor

Address (hex)	Device	Type
0000_0000 - 0000_FFFF	System ROM (64K)	S
0001_0000 - 0FFF_FFFF	Off-board system ROM extension (not instantiated)	S
1000_0000 - 1000_FFFF	System RAM (64K)	S
1001_0000 - 6FFF_FFFF	Off-board system RAM extension (not instantiated)	S
7000_0000 - 7000_000B	3 timer register (32 bit)	S
7000_000C - 7FFF_FFFF	additional IO address space (not instantiated)	S
8000_0000 - 8000_FFFF	User ROM (64K)	U, S
8001_0000 - 8FFF_FFFF	Off-board user ROM extension (not instantiated)	U, S
9000_0000 - 9000_FFFF	User RAM (64K)	U, S
9001_0000 - FFFF_FFFF	Off-board user RAM extension (not instantiated)	U, S

Table 2-1: The addressing space of the DLXS test-bench

2.3 System simulation

In chapter 5, the DLX-assembler *dlx_asm* is described. This assembler is an adaption and further development of the dlxasm described in [Baz94]. It allows for the automatically generation of a VHDL architecture-description of the user ROM containing the assembled source code. Thus, the simulation of the written DLXS-model within the test-bench can be performed efficiently. For each feature of the DLXS-model, you can write a small assembler programme, i.e. for the addition instructions, generate the corresponding ROM-architecture and simulate the entire test-bench with this programme just by changing and updating the top-level configuration of the system. This procedure is shown in figure 2-2.

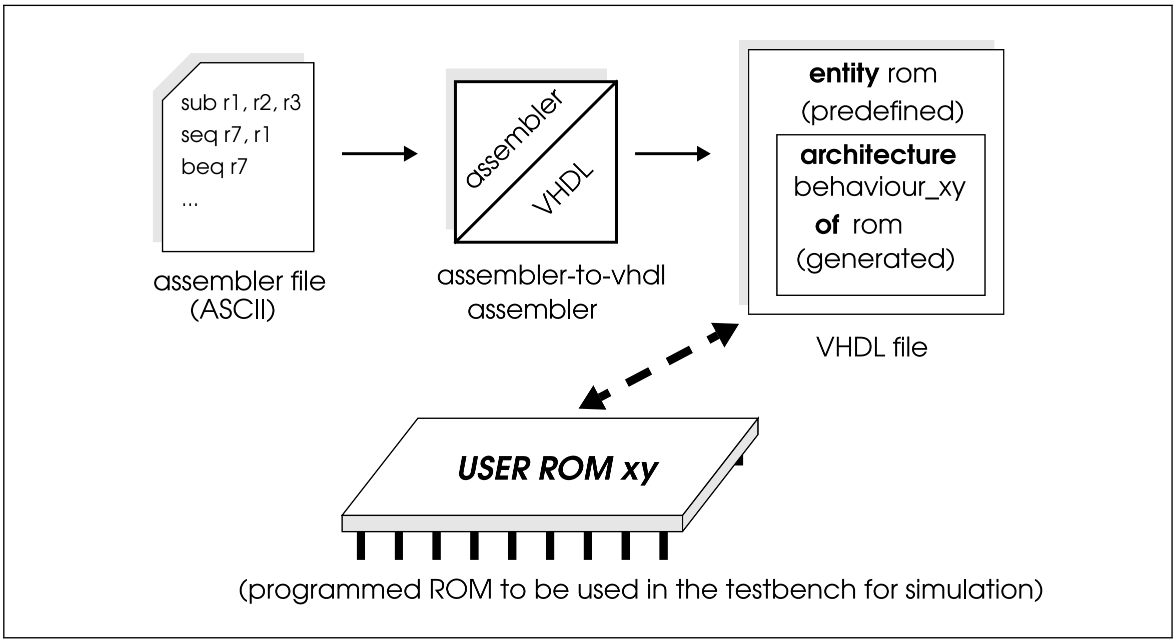

Figure 2-2: Generation of ROM descriptions for the simulation of the test-bench

2.4 References

[Baz94] P. Bazargan et al.: *A Tutorial for Advanced VLSI Course: Designing the 32-bit DLX Microprocessor with the ALLIANCE CAD System*, Proc. of the 5. Eurochip Workshop on VLSI Design Training, Dresden, Germany, 1994

3
The Design Libraries

3.1 Predefined libraries

The predefined directory structure for this design course is shown in figure 3-1. Each subdirectory in the directory *design* contains one VHDL design-library.

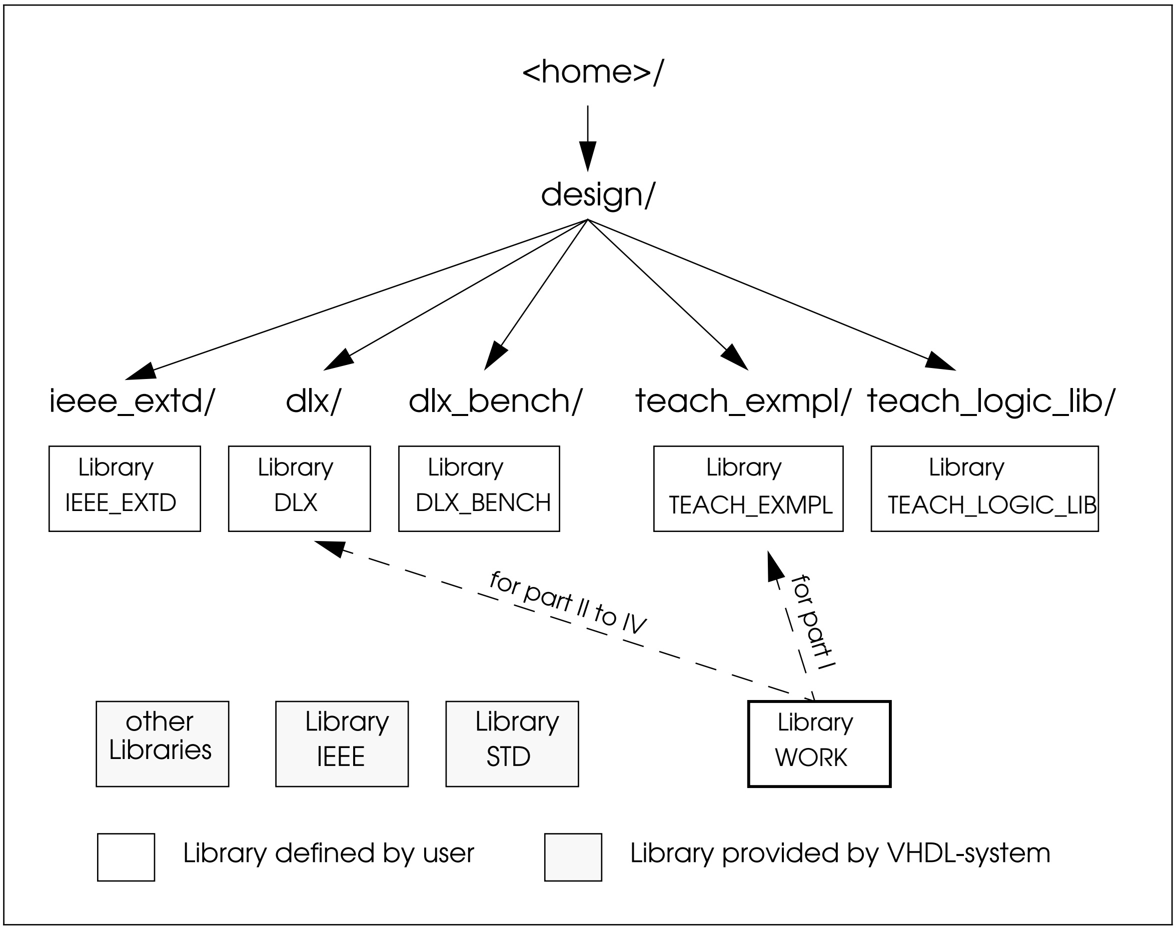

Figure 3-1: Directory Structure and Design-Libraries for this course

The working library *WORK* is mapped to the library TEACH_EXMPL for the first part of the exercises and later on to the library *DLX* for all following parts of the exercises.W ithin each of the user defined libraries, there is always a sub-directory named *vhdl_src* which contains the VHDL source code for the corresponding library units.

3.2 Description of the contents of the libraries

3.2.1 Library *TEACH_EXMPL*

This library contains some easy to understand examples used for the first part of the exercises to introduce VHDL. The contents are:

- a simple text output entity with two different architectures (*hello.vhd, hello-assertion.vhd, hello-stdtextio.vhd*)
- a RS-flipflop (*rs_ff.vhd*) with three different architectures (*rs_ff-behaviour .vhd* , *rs_ff-dataflow .vhd, rs_ff-structural*), its test bench (*rs_ff_test.vhd* , *rs_ff_test-structural.vh* d), and configurations (*rs_ff_test -config1.vhd, rs_ff_test-config1.vhd*).

3.2.2 Library *TEACH_LOGIC_LIB*

This library is an example library containing some logical-gate definitions and a package which declares these components. It is used for the structural models in the first part of the exercises.

3.2.3 Library *IEEE_EXTD*

For all VHDL models in this course, the nine-value logic-system defined in the IEEE_1 164 standard-package is used. This library is provided with the VHDL-analysis and -simulation software. The name of this library is *IEEE* . The library *IEEE_EXTD* , defined for this course, provides extended functions for this logic system in two more packages. The VHDL files contained in this library are:

- *stdl1 164_extended.vhd* declaration of the package containing (among others) overloaded write-

	functions for the type *std_logic*
stdl1 164_extended-body .vhd	package body
• *stdl1 164_vector_arithm.vhd*	declaration of the package containing arithmetical and logical functions for *std_logic* -vectors and conversion functions
stdl1 164_vector_arithm-body .vhd	package body

3.2.4 Library *DLX_BENCH*

All components of the test-bench (except the DLXS processor) are described as VHDL models and are contained in the VHDL design-library *DLX_BENCH*. The VHDL files are:

• *clockgen.vhd*	entity declaration of the clock-generator
clockgen-behaviour .vhd	behaviour architecture
clockgen-dataflow .vhd	dataflow architecture
• *decoder.vhd*	entity declaration of the address decoder
decoder-dataflow .vhd	dataflow architecture
• *dlx_b_components.vhd*	package declaring all components of the library DLX_BENCH
• *ram.vhd*	entity declaration of the RAM bank
ram-structural.vhd	structural architecture
ram-config1.vhd	configuration for the ram bank
• *ram_byte.vhd*	entity declaration of the RAM
ram_byte-behaviour .vhd	behaviour architecture
• *rom.vhd*	entity declaration of the ROM
rom-behaviour_* xy.vhd*	behaviour architecture for ROM (there are several architectures containing different assembler programs)
• *timer .vhd*	entity declaration of the timer

timer-behaviour .vhd	behaviour architecture

3.2.5 Library *DLX*

The VHDL description for the test-bench is contained in the VHDL design-library *DLX*. This library is also your working library which will contain your self-developed design units of the DLXS model. The predefined VHDL files are described below .

3.2.5.1 At the beginning of this design course

- *behav_procedures_ n.vhd* — declaration of the package containing the procedures for the behavioural model of the DLXS
  behav_procedures-body_ n.vhd — package body
- *dlx_types.vhd* — declaration of the package containing the main types for the modelling of the DLXS
- *dlx_instructions.vhd* — declaration of the package containing the DLXS instructions as constants and related types and useful functions
  dlx_instructions-body .vhd — package body
- *system.vhd* — entity declaration of the test-bench
  system-structural.vhd — structural architecture
  system-config.vhd — top-level configuration for the test-bench

3.2.5.2 To be developed in Part II of this course

- *dlx.vhd* — entity declaration of the DLXS
  dlx-behaviour_1.vhd - dlx-behaviour_4.vhd — behavioural architecture descriptions (different stages of completion)

3.2.5.3 To be added to the library in Part III of this course

- *control_types.vhd* — declaration of the package containing the constants for the controller output signals

• *controller .vhd*	entity declaration of the controller
controller-dataflow .vhd	dataflow architecture with instruction decoders and state machine
controller-config1.vhd	configuration of the controller
• *core.vhd*	entity declaration of the processor core
core-structural.vhd	structural architecture containing the datapath and the controller
• *dlx-structural.vhd*	structural architecture of the processor containing the core and the pads
dlx-config_rtl.vhd	configuration for the structural model of the processor
• *pad_in.vhd*	entity declaration of input pad
pad_in-dataflow .vh d	dataflow architecture
pad_inout.vhd	entity declaration of bidirectional pad
pad_inout-dataflow .vh d	dataflow architecture
pad_out.vhd	entity declaration of output pad
pad_out-dataflow .vh d	dataflow architecture
• *fsm.vhd*	entity declaration of the finite state machine (FSM)
fsm-moore_2.vhd	Moore-machine style FSM architecture
• *ir_decode_1.vhd*	entity declaration for the instr . decoder 1
ir_decode_1 -behaviour .vhd	behavioural architecture for instr . dec. 1
ir_decode_2 .vhd	entity declaration for the instr . decoder 2
ir_decode_2 -behaviour .vhd	behavioural architecture for instr . dec. 2
ir_decode_3 .vhd	entity declaration for the instr . decoder 3
ir_decode_3 -behaviour .vhd	behavioural architecture for instr . dec. 3

3.2.5.4 To be developed in Part III of this course

• *alu.vhd*	entity declaration of theALU
alu-behaviour .vhd	corresponding behavioural architecture
• *bus_const32.vhd*	entity declaration of the bus constant

	circuit
bus_const32-dataflow .vhd	corresponding dataflow architecture
• *datapath.vhd*	entity declaration of the datapath
datapath-structural.vhd	corresponding structural architecture
datapath-config.vhd	configuration for the datapath
• *ir.vhd*	entity declaration of the instruction register
ir-behaviour .vhd	corresponding behavioural architecture
• *icr.vhd*	entity declaration of the interrupt control register
icr-dataflow .vhd	corresponding dataflow architecture
• *mdr .vhd*	entity declaration of the memory data register
mdr-behaviour .vhd	corresponding behavioural architecture
• *reg_file.vhd*	entity declaration of the register file
reg_file-behaviour .vhd	corresponding behavioural architecture
• *ir.vhd*	entity declaration of the instruction register
ir-behaviour .vhd	corresponding behavioural architecture
• *word_latch.vhd*	entity declaration of the 32-bit latch
word_latch-dataflow .vhd	corresponding dataflowarchitecture
• *word_mux2.vhd*	entity declaration of the 32x2:1 multiplexer
word_mux2-dataflow .vhd	corresponding dataflow architecture
• *word_reg_1e.vhd*	entity declaration of the 32-bit register with tristate output
word_reg_1e-dataflow .vhd	corresponding dataflow architecture
• *word_reg_1e1.vhd*	entity declaration of the 32-bit register with tristate and one normal output
word_reg_1e1-dataflow .vhd	corresponding dataflow architecture

3.2.5.5 To be added in Part IV of this course

• *cla_a.vhd*	entity declaration of the *cla_a* cell used in the carry-lookahead adder
cla_a-dataflow .vhd	corresponding dataflow architecture
cla_b.vhd	entity declaration of the *cla_b* cell used in the carry-lookahead adder
cla_b-dataflow .vhd	corresponding dataflow architecture
cla_b_end.vhd	entity declaration of the *cla_b_end* cell used in the carry-lookahead adder
cla_b_end-dataflow .vhd	corresponding dataflow architecture
• *demux_1_32.vhd*	entity declaration of the 1:32-demultiplexor
demux_1_32-behaviour .vhd	corresponding behavioural architecture
• *mux_32_1.vhd*	entity declaration of the 32:1 multiplexer
mux_32_1-behaviour .vhd	corresponding behavioural architecture
• *reg_file-structural.vhd*	structural architecture for the register file
reg_file-config_rtl.vhd	configuration for the register file
• *word_adder .vhd*	entity declaration of the 32 bit 2-complement adder
word_adder-ripple.vhd	corresponding riple carry architecture
word_adder-cla_h.vhd	corresponding carry-lookahead architecture
• *word_alu_logic.vhd*	entity declaration of the 32-bit ALU logic block
word_alu_logic-behaviour .vhd	corresponding behaviouralarchitecture
• *word_mux4.vhd*	entity declaration of the 32x4:1 multiplexer
word_mux4-dataflow .vhd	corresponding dataflow architecture
• *word_shifter .vhd*	entity declaration of the 32 bit 2-complement adder
word_shifter-behaviour .vhd	corresponding behavioural architecture

word_shifter-barrel.vhd corresponding barrel shifter architecture

3.2.5.6 To be developed in Part IV of this course

- *alu-structural.vhd* structural architecture of theALU
  alu-config_rtl.vhd configuration for the ALU

3.3 Selection of useful subprograms and definitions contained in the different packages

This section lists some important subprograms and definitions which are contained in the different packages of the libraries. For the subprograms, the **calling statements** are given. Brackets (<>) denote the type of the parameter , (::=) means either (<=) or (:=). For the complete contents of the packages, have a look at their corresponding source-code files.

3.3.1 Library *STD*

3.3.1.1 Package *STANDARD*

(This package contains among others the basic definitions for the types *integer*, *real, boolean, character* , *string, bit, bit_vector* , and *time* . It is included automatically.)

3.3.1.2 Package *TEXTIO*

- ***writeline*** *(output , <line>)*
  write contents of output memory-file to the standard output (screen)

- ***write*** *(<line> , <integer> | <real> | <boolean> | <character> |<string> | <bit> | <bit_vector> | <time>)*
  write the parameter into the output memory-file

3.3.2 Library *IEEE*

3.3.2.1 Package *STD_LOGIC_1164*

(This package contains among others the definition of the types *std_ulogic, std_ulogic_vector* , *std_logic, std_logic_vector* , and subtypes such as *ux01* .)

- *<std_logic_vector> ::=* ***to_stdlogicvector*** *(<bit_vector>)*
  convert a bit_vector to a std_logic_vector (this is useful e.g. to assign a hexadecimal value to a std_logic_vector)
- *<std_logic_vector>* | *<std_logic> ::=* ***to_ux01*** *(<std_logic_vector>* | *<std_logic>)*
  convert a std_logic_vector or std_logic to a std_logic_vector or std_logic which contains only the values U, X, 0, 1 (this should be used for registers which cannot have the value Z)
- *<std_logic_vector> ::= <std_logic_vector>* ***and*** | ***or*** | ***xor*** | ***nand*** | ***nor*** | ***xnor*** | ***not*** *<std_logic_vector>*
  logical functions

3.3.3 Library *IEEE_EXTD*

3.3.3.1 Package *STDL1164_EXTENDED*

- ***write*** *(<line>, <std_logic>)*
  write the parameter into the output memory-file
- ***write*** *(<line>, <std_logic_vector> , X* | *O* | *B , up* | *down)*
  write the std_logic_vector into the output memory-file (X = hexadecimal-, O = octal-, B = binary output style; up = ascending range, down = descending range of the input vector)

3.3.3.2 Package *STDL1164_VECTOR_ARITHMETIC*

- *<std_logic_vector> ::=* ***natural_to_sv*** | ***integer_to_sv*** *(<natural>* | *<integer> , vector_length<natural>)*
  convert a natural or integer value to a std_logic_vector of the length *vector_length*

- *<natural>* | *<integer> ::=* ***sv_to_natural*** | ***sv_to_integer*** *(<std_logic_vector>)*
  convert a std_logic_vector to a natural or integer value

- ***sv_add*** | ***sv_addu*** *(in1<std_logic_vector> , in2<std_logic_vector>, result<std_logic_vector> [, overflow<std_logic])*
  add two std_logic_vector (signed or unsigned) with or without overflow generation

- ***sv_sub*** | ***sv_subu*** *(in1<std_logic_vector> , in2<std_logic_vector>, result<std_logic_vector> [, borrow<std_logic])*
  subtract (signed or unsigned) the std_logic_vector *in2* from the std_logic_vector *in1* with or without borrow generation

- ***sv_sll*** | ***sv_srl*** | ***sv_sra*** *(in<std_logic_vector> , out<std_logic_vector>, shift_factor<std_logic_vector>)*
  shift left logical or shift right logical or shift right arithmetical the std_logic_vector *in* by the factor *shift_factor* and give out the result in *out*

- ***sv_gt*** | ***sv_ge*** | ***sv_lt*** | ***sv_le*** *(in1<std_logic_vector> , in2<std_logic_vector>, result<std_logic>)*
  compare the two std_logic_vector parameters and set *result* to 1 if the condition (greater , greater or equal, lower than, lower or equal) is true, otherwise set *result* to 0

- *<std_logic_vector> ::=* ***sv_sext*** | ***sv_zext*** *(in<std_logic_vector> , extd_length<natural>)*
  sign extension or zero extension of the std_logic_vector *in* to a std_logic_vector of the length *extd_length*

3.3.4 Library *DLX* (*WORK*)

3.3.4.1 Package *DLX_INSTRUCTIONS*

- ***write_instr*** *(<line>, <dlx_word>)*
  decode the given instruction to a text string and write it into the output memory-file

4

VHDL Programming Guidelines

The development of more complex systems in VHDL often leads to large databases with several thousands of VHDL code lines. Hence, a good coding and documentation style becomes a key issue. To ease the maintenance of such databases and to improve the readability and reusability, some guidelines are given in this chapter. These guidelines are concerning the VHDL programming, the creation of the databases, and the documentation style.

4.1 VHDL Programming

4.1.1 Design Unit Names

Choose names for your design units which are clearly understandable, e.g. for a clock-generator entity the name *clockgen* is better understandable than a shorter name like *cg*:

```
ENTITY clockgen IS
...
END clockgen;
```

For architecture names use the following three prefixes for the name: *behaviour*, *dataflow*, and *structural*, e.g.:

```
ARCHITECTURE behaviour_v1 OF clockgen IS
...
END behaviour_v1;
```

If the architecture description contains both, behavioural and dataflow statements, choose the prefix *behaviour*, if it contains both, dataflow and structural statements, use the prefix *dataflow* .

4.1.2 Other Names

Choose significant names for all the signals, variables, constants, and labels. Keep in mind that VHDL doesn‘t distinguish between lower - and uppercase letters. Hence, the names *Clock* , *CLOC* K, and *clock* are all referring to the same object.

4.1.3 Types

For signals, use the data types defined in the IEEE standard multi value logic system (IEEE Std 1 164-1993) which are given in the library IEEE. The following statement has to be included in the source files:

```
LIBRARY IEEE;
  USE IEEE.std_logic_1164.ALL;
```

Library statements in entity or package declarations need not to be repeated in the referring architecture or package-body descriptions. Libraries and packages which are only used for the implementation of one specific architecture of an entity , shall be declared in the architecture, as shown in figure 4-1.

4.1.4 Configurations

Architectures shall not contain any configuration statements. W rite a separate configuration instead.

4.1.5 Processes

Use labels for processes. This can be very helpful while debugging your module. E.g.:

```
do_something: PROCESS
BEGIN
  WAIT UNTIL clk = ‘1‘;
  < more code>
END PROCESS do_something;
```

If a process contains only one wait statement, replace this statement by writing

```
<file dlx.vhd>
LIBRARY IEEE;                -- declare this library and use
  USE IEEE.std_logic_1164.ALL;-- this package for all the
                             -- different architectures

ENTITY dlxis
...
END dlx;
-------------------------------------------------------------------

<file dlx-behaviour.vhd>
   USE WORK.behav_procedures.all;-- use this package only for this
                             -- (behavioural) architecture

ARCHITECTURE behaviour OF dlx IS
...
END behaviour;
```

Figure 4-1: Where to declare the libraries

a process with a sensitivity list. This increases readability (and makes the code synthesizable). The upper example would have to be changed into:

```
do_something: PROCESS (clk)
BEGIN
  IF clk = '1' THEN
    < more code>
  END IF;
END PROCESS do_something;
```

4.2 VHDL Coding And Documentation Style

4.2.1 One Design Unit - One File

The five VHDL design units are: entities, architectures, configurations, packages, and package-bodies. Each design unit shall be kept in its own file, i.e. do not keep more than one design unit in one file. This can significantly reduce the recompilation-time because a VHDL-compiler compiles all the design units found in one file.

4.2.2 File Header

Each file containing a VHDL design unit shall have a file header which indicates the name(s) of the designer(s), the name of the design unit, the corresponding file name, a short description of the purpose or function of the design unit, the simulator name and version, and a list of changes. The predefined header for this design course is contained in the file *header .txt (* figure 4-2).

```
-------------------------------------------------------------------
-- VLSI Design Course / University of Stuttgart / SS 95
-------------------------------------------------------------------
-- Designers :
-- Group No. :
-------------------------------------------------------------------
-- Design Unit Name :
-- Purpose :
--
-- File Name :
-------------------------------------------------------------------
-- Simulator : SYNOPSYS VHDL System Simulator (VSS) Version 3.1.
-------------------------------------------------------------------
-- Date       | Changes
--            |
-------------------------------------------------------------------
```

Figure 4-2: File header for the design course

4.2.3 File Names

The standard file extension for VHDL source code files shall be *.vhd* . For entities and packages use the name of the design unit contained in the file as file prefix, e.g.:

```
clockgen.vhd
my_clock_types.vhd
```

For architectures, use the name of the corresponding entity , followed by a ‘-‘ and the architecture name as file name, e.g.:

```
clockgen-behaviour_v1.vhd
```

For package bodies, use the name of the corresponding package declaration, followed by ‘-body‘ as file name, e.g.:

```
my_clock_types-body.vhd
```

For configurations, use the name of the corresponding entity declaration, followed by ‘-config-‘, followed by the configuration name as file name, e.g.:

```
clockgen-config-test1.vhd
```

Following those guidelines, the database for the clock generator (only VHDL source files, including two different architectures) would look like this:

```
clockgen.vhd
clockgen-behaviour_v1.vhd
clockgen-dataflow.vhd
clockgen-config-test1.vhd
my_clock_types.vhd
my_clock_types-body.vhd
```

4.2.4 Source Code Outfit

Write the VHDL-keywords in uppercase letters. W rite only one VHDL statement per line. Strictly use indenting for your source code (indenting by two or four positions is recommended) to get a readable code. Comment your VHDL source code extensively! This is the most important point! Use the style of the figure 4-3 for your comments:.

```
ARCHITECTURE example OF test IS

  SIGNAL a : std_logic;       -- short comments can be placed here
  SIGNAL b : std_logic;       -- short comments can be placed here
                              -- and can be continued here
  SIGNAL c : std_logic;       -- another short comment
BEGIN
  ---------------------------------------------------------------
  -- Use a box to describe the function of a process, a procedure
  -- or for significant remarks (should be indented)
  ---------------------------------------------------------------
  do_something: PROCESS (clk)
    VARIABLE c : boolean;          -- another short comment
  BEGIN
    --
    -- Use this complete line for longer comments with one leading
    -- and one finishing empty comment-line
    -- (Long comments are also indented)
    --
    IF clk = '1' THEN
      c := TRUE;                   -- another short comment
    ELSE
      c := FALSE;                  -- and another one
    END IF;
  END do_something;
END example;
```

Figure 4-3: Comment style for the design course

5 Tools

In this chapter , some tools for the development and testing of the DLXS processor are described. These are a special VHDL source-code prettifier and a DLX assembler .

5.1 The VHDL Prettifier *mvp*

The *mvp* tool is very useful to get formatted VHDL source-code printouts. T ype only *mvp* without any parameters to get the following manual page:

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                         MAKE VHDL PRETTY
                     (print tool for VHDL code)

              Original Version by P.S. Elliot / 1993

             Extended Version (V1.1) by M. Gumm / Dez. 1994

                    "mp" - Printout Style:
                Original version by Steve Holden

              Distribute freely for noncommercial use !
--------------------------------------------------------------------
Usage:

mvp -[ulKIF<int>DB<int>E<int>] -[S <subject>] -[N <name>]
     [infile] [> outfile]

Otions in [] mean:

   ASCII processing:
```

```
          u       Upper case key words (default)
          l       Lower case key words

     PostScript processing (default: no PostScript output):
          K       Bold key words
          I       Bold identifiers
          F<int>  Font size (possible: 6, 8, 10 (default), 12, 14)
          D       HP Lj4 duplex output
          S       Bottom text for all pages (default: <infilename>)
          N       User name (max. 24 letters) in headline (default:
   none)
          B<int>  Begin output with page no. (default: 1)
          E<int>  End output with page no. (default: 1000)
          Z       Output with line numbering
     Files:
          infile  Name of input file; if omitted stdin is read
          outfile Dircet output from stdout to outfile
   ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

One very useful set of options for *mvp* is the following:

mvp -uKF8Z input_file > *output_file*

This will uppercase and bold the VHDL key words, set the font size to 8 pt, and enable the line numbering.

After the execution of this command, the PostScript-file *output_file* can be printed with the normal unix command *lpr*.

5.2 The DLX Assembler *dlx_asm*

The DLX assembler *dlx_asm*, an adaption and further development of the dlx-asm described in [Baz94], is especially made for this design course. It reads a DLX-assembler input file (ASCII), checks for errors, and generates a VHDL file. This VHDL file is a behavioural architecture description of the user ROM which can be directly used for the system simulation. Type only *dlx_asm* without any parameters to get the following manual page:

```
   +------------------------------------------------------------------+
   |  DDDDD    LL       XX   XX          AAA     SSSSSS   MMM    MMM  |
   |  DD  DD   LL        XX XX          AA AA    SS       MMMM MMMM   |
   |  DD   DD  LL         XXX     ===  AAAAAAA   SSSSSS   MM MMM MM   |
   |  DD  DD   LL        XX XX         AA   AA       SS   MM      MM  |
   |  DDDDD    LLLLL    XX   XX        AA   AA   SSSSSS   MM      MM  |
   +------------------------------------------------------------------+
```

```
|  ORIGINAL VERSION v0.1                                             |
|  by Pirouz BAZARGAN SABET                                          |
|  Jan 20 1994                                                       |
|  (C) MASI/IBP/CAO-VLSI                                             |
+------------------------------------------------------------------+
|  MODIFIED VERSION v1.1                                             |
|  -> complete DLX instruction set and ...                           |
|  -> instruction coding as defined by Hennessy/Patterson            |
|                                                                    |
|  by Martin GUMM / IPVR-ISE                                         |
|  Sept 28 1995                                                      |
|  for the use within the                                            |
|  VLSI CAD COURSE                                                   |
|  (C) University of Stuttgart / Dept. of Computer Science           |
+------------------------------------------------------------------+

usage : dlx_asm <-s|u|i|f> <assembly_file> <vhdl_file_ext>

options:
 -s   include only signed integer instructions
 -u   include "-s"-instructions and instructions for unsigned
       integer operations except multipl. and division operators
 -i   include "-s" and "-u"-instructions and
       multiplication and division operators
 -f   include "-s" and "-u"and "-i"-instructions and
       all floating point instructions

example: dlx_asm -u test.asm test
   -> the input file "test.asm" is read
   -> instructions for unsigned integer operations are allowed
   -> the output file "rom-behaviour_test.vhd" is generated
```

For this design course, the DLX assembler has always to be invoked with the *s*-option:

dlx_asm -s assembly_input_file vhdl_file_extension

to restrict the assembler to the instruction subset supported by the DLXS processor . The assembler code supported by dlx_asm (invoked with the *s*-option) is described in chapter 1.

The following pages include the parts of the *dlx_asm* -manual which are relevant for this course:

ORIGIN

The original version (v0.1) belongs to the ALLIANCE CAD system from CAO-VLSI-Team at MASI laboratory, University P. et M. Curie, Paris.

This version (v1.1) was extended and adapted for the use within the VLSI laboratory course held at the department for integrated systems engineering (IPVR-ISE), University of Stuttgart.

NAME

dlx_asm - assembler for the DLX processor defined by Hennessy/Pattersson

SYNOPSIS

dlx_asm <-s|u|i|f> <assembly_file> <vhdl_ext>

DESCRIPTION

The DLX assembler dlx_asm takes assembly language programs, as specified below, and produces a behavioural ROM description in VHDL.

This version (v1.0) supports the complete DLX instruction set due to the definition by Hennessy/Patterson including the floating point instructions. The instruction encoding is also due to the original definition.

The options mean:

-s include only signed integer instructions.

-u include "-s"-instructions and instructions for unsigned integer operations except multipl. and division operators

-i include "-s" and "-u"-instructions and multiplication and division operators

-f include "-s" and "-u"and "-i"-instructions and all floating point instructions

<assembly_file> is the name of the assembler file to be read.

<vhdl_file_ext> is the extension for the default output file name *rom-behaviour_<vhdl_ext>.vhd* and the extension for the default VHDL architecture name *behaviour_<vhdl_ext>*.

ASSEMBLY FORMAT

An assembly program is a set of statements. Each statement must be followed by a line feed. A statement has the form:

<label> <opcode> <operands>

label is a is a string used to tag a specific statement or a portion of the program. A label must be followed by a colon (:) if there is a line feed between the *label* and the *opcode*.

opcode is a string representing either an effective instruction or a synthetic instruction.

An effective instruction is either an instruction from the DLX's instruction set or a synthetic instruction. The assembler expands a macro instruction into a set of DLX's instructions.

INSTRUCTIONS

In the descriptions below ,the following abbrevations are used:

Rd : destination GPR (general purpose integert register) (d = 0 ... 31)

Rs1, Rs2 : source GPRs (s1, s2 = 0 ... 31)

ICR : interrupt control register

IAR : interrupt address register

TBR : trap base register

S : special register (can be either ICR, IAR, or TBR)

The letters R, I, and J describe the corresponding instruction format

available instructions using the -s option:
(signed integer instructions)

aritmetical and logical instructions

ADD	Rd, Rs1, Rs2	signed addition of two registers *Rd ← Rs1 + Rs2*	(R)
SUB	Rd, Rs1, Rs2	signed subtraction of two registers *Rd ← Rs1 - Rs2*	(R)
AND	Rd, Rs1, Rs2	logical AND *Rd ← Rs1 AND Rs2*	(R)
OR	Rd, Rs1, Rs2	logical OR *Rd ← Rs1 OR Rs2*	(R)
XOR	Rd, Rs1, Rs2	logical XOR	

		Rd ← Rs1 XOR Rs2	(R)
SLL	Rd, Rs1, Rs2	shift left logical *Rd ← Rs1 logical shifted left by Rs2* (5 lsb of Rs2 are significant)	(R)
SRL	Rd, Rs1, Rs2	shift right logical *Rd ← Rs1 logical shifted right by Rs2* (5 lsb of Rs2 are significant)	(R)
SRA	Rd, Rs1, Rs2	shift right arithmetical *Rd ← Rs1 arithmetical shifted right by Rs2* (5 lsb of Rs2 are significant)	(R)
ADD.I	Rd, Rs1, I	signed addition of register and immediate *Rd ← Rs1 + I* (I : sign extended 16-bit immediate)	(I)
SUB.I	Rd, Rs1, I	signed subtraction of register and immediate *Rd ← Rs1 - I* (I : sign extended 16-bit immediate)	(I)
AND.I	Rd, Rs1, I	logical AND *Rd ← Rs1 AND I* (I : zero extended 16-bit immediate)	(I)
OR.I	Rd, Rs1, I	logical OR *Rd ← Rs1 OR I* (I : zero extended 16-bit immediate)	(I)
XOR.I	Rd, Rs1, I	logical XOR *Rd ← Rs1 XOR I* (I : zero extended 16-bit immediate)	(I)
SLL.I	Rd, Rs1, I	shift left logical *Rd ← Rs1 logical shifted left by I* (5 lsb of immediate I are significant)	(I)
SRL.I	Rd, Rs1, I	shift right logical *Rd ← Rs1 logical shifted right by I* (5 lsb of immediate I are significant)	(I)
SRA.I	Rd, Rs1, I	shift right arithmetical *Rd ← Rs1 arithmetical shifted right by I* (5 lsb of immediate I are significant)	(I)
LHI	Rd, I	load high immediate *Rd(0:15) ← I ; Rd(16:31) ← hex0000* (I : 16-bit immediate)	(I)
NOP		no operation	(R)

set and test instructions

SEQ	Rd, Rs1, Rs2	set register if equal *Rd ← 1 if (Rs1 = Rs2) else 0*	(R)
SNE	Rd, Rs1, Rs2	set register if not equal *Rd ← 1 if (Rs1 /= Rs2) else 0*	(R)
SLT	Rd, Rs1, Rs2	set register if lower thanl *Rd ← 1 if (Rs1 <Rs2) else 0*	(R)
SLE	Rd, Rs1, Rs2	set register if lower or equal *Rd ← 1 if (Rs1 <= Rs2) else 0*	(R)
SGT	Rd, Rs1, Rs2	set register if greater than *Rd ← 1 if (Rs1 >Rs2) else 0*	(R)
SGE	Rd, Rs1, Rs2	set register if greater or equal *Rd ← 1 if (Rs1 >= Rs2) else 0*	(R)
SEQ.I	Rd, Rs1, I	set register if equal *Rd ← 1 if (Rs1 = Rs2) else 0* (I : sign extended 16-bit immediate)	(I)
SNE.I	Rd, Rs1, I	set register if not equal *Rd ← 1 if (Rs1 /= Rs2) else 0* (I : sign extended 16-bit immediate)	(I)
SLT.I	Rd, Rs1, I	set register if lower thanl *Rd ← 1 if (Rs1 <Rs2) else 0* (I : sign extended 16-bit immediate)	(I)
SLE.I	Rd, Rs1, I	set register if lower or equal *Rd ← 1 if (Rs1 <= Rs2) else 0* (I : sign extended 16-bit immediate)	(I)
SGT.I	Rd, Rs1, Rs2	set register if greater than *Rd ← 1 if (Rs1 >Rs2) else 0* (I : sign extended 16-bit immediate)	(I)
SGE.I	Rd, Rs1, Rs2	set register if greater or equal *Rd ← 1 if (Rs1 >= Rs2) else 0* (I : sign extended 16-bit immediate)	(I)

branch and jump instructions

BEQZ	Rs, label	branch if equal zero *PC ← label if Rs = 0*	(I)
BNEZ	Rs, label	branch if not equal zero *PC ← label if Rs /= 0*	(I)
J	Label	jump *PC ← label*	(J)
JAL	Label	jump and link	

		R31 ← PC ; PC ← label	(J)
JR	Rs	jump register PC ← Rs	(I)
JALR	Rs	jump and link register *R31 ← PC ; PC ← Rs*	(I)
load and store instructions			
LW	Rd, Rs2 (Rs1)	load word *Rd ← M[Rs1 + Rs2]*	(R)
SW	Rs2 (Rs1), Rd	store word *M[Rs1 + Rs2] ← Rd*)	(R)
LH	Rd, Rs2(Rs1)	load half signed *Rd(16:31) ← M[Rs1 + Rs2]* (Rd sign extended to 32 bit)	(R)
SH	Rs2(Rs1), Rd	store half *M[Rs1 + Rs2] ← Rd(16:31)*	(R)
LB	Rd, Rs2(Rs1)	load byte signed *Rd(24:31) ← M[Rs1 + Rs2]* (Rd sign extended to 32 bit)	(R)
SB	Rs2(Rs1), Rd	store byte *M[Rs1 + Rs2] ← Rd(24:31)*	(R)
LW.I	Rd, I (Rs)	load word *Rd ← M[Rs + I]* (I : sign extended 16-bit immediate)	(I)
SW.I	I (Rs), Rd	store word *M[Rs + I] ← Rd* (I : sign extended 16-bit immediate)	(I)
LH.I	Rd, I (Rs)	load half signed *Rd(16:31) ← M[Rs + I]* (Rd sign extended to 32 bit) (I : sign extended 16-bit immediate)	(I)
SH.I	I (Rs), Rd	store half *M[Rs + I] ← Rd(16:31)* (I : sign extended 16-bit immediate)	(I)
LB.I	Rd, I (Rs)	load byte signed *Rd(24:31) ← M[Rs + I]* (Rd sign extended to 32 bit) (I : sign extended 16-bit immediate)	(I)
SB.I	I (Rs), Rd	store byte *M[Rs + I] ← Rd(24:31)* (I : sign extended 16-bit immediate)	(I)

special instructions

TRAP	label	trap (software interrupt) I*AR* ← *PC ; PC* ← *label* (label : 26-bit sign extended immediate)	(J)
RFE		return from interrupt or exception *PC* ← *IAR*	(J)
MOVS2I	Rs, S	move special register to GPR-register *Rs* ← *S* (*S* can be *ICR*, IAR or *TBR*)	(R)
MOVi2S	S, Rs	move GPR-register to special register *S* ← *Rs* (S can be *ICR*, *IAR* or *TBR*)	(R)

available synthetic instructions for all options:

LIW	Rd, I	load 32-bit immediate word *Rd* ← *I* (LHI and ORI instr.)
LIS	Rd, I	load sign-extended 16-bit immediate *Rd* ← *I* (ADDI instr.)
LIU	Rd, I	load zero-extended 16-bit immediate *Rd* ← *I* (ORI instr.)
MOV	Rd, Rs	move GPR values *Rd* ← *Rs* (OR instr. with R0)
CLR	Rd	clear GPR *Rd* ← *0* (AND instr. with R0)
NOT	Rd	invert least significant 16 bit of GPR *Rd{16:31}* ← *NOT Rd[16:31]* (XOR instr.)

PSEUDO_INSTRUCTIONS

In addition to the instruction se, several pseudo-instructions are defined to enhance the readability.

EQU	I	assign the value of an immediate to a label
ORG	I	set the address (of the first statement) of the

		program to the location definbed by the immediate (32-bits)
START	label	marks the first statement of the program to be executed
END		marks the end of the program

NOTES

A comment begins with a semicolon (;) and extends until the end of the line

Immediate values can be expressed using decimal, binary, octal, or hexadecimal representations:

235	decimal value
B“01011“	binary value
O“376“	octal value
H“0f73“	hexadecimal value

A static expression can be used instead of an immediate. The following operations are allowed inside expressions:

+	arithmetic addition
-	arithmetic subtraction
*	arithmetic multiplication
>>	shift logical right
<<	shift logical left

EXAMPLE

dlx_asm -u test.asm test

-> the input file "test.asm" is read

-> instructions for unsigned integer operations are allowed

-> the output file "rom-behaviour_test.vhd" is generated

-> the VHDL architecture name is “behaviour_test“

SUPPORT

Please e-mail to *gumm@informatik.uni-stuttgart.de*

5.3 References

[Baz94] P. Bazargan et al.: *A Tutorial for Advanced VLSI Course: Designing the 32-bit DLX Microprocessor with the ALLIANCE CAD System*, Proc. of the 5. Eurochip Workshop on VLSI Design Training, Dresden, Germany, 1994

6
Exercises

This chapter contains all the exercises for the design course. It is divided into four parts:

- The first part is aimed to give an short introduction to the simulation and debugging tools used in this design course as well as some easy to understand examples to get familiar with some basic features of the VHDL language.
- In the second part, a behavioural model of the DLXS shall be developed and validated step-by-step. During this implementation, more and more VHDL constructs will be used and understood (learning by doing!).
- The third part deals with the step-by-step implementation of a RTL model of the processor (only the datapath shall be implemented, the controller and global structure is already defined).
- The fourth part deals with the synthesis of a gate-level description of the processor with the mapping to a real ASIC library . Again, only the datapath shall by synthesized, a synthesized gate-level description of the controller is available.

Part I Getting started

6.1 SYNOPSYS VSS-Core-Tools and first VHDL models

For the first part of this design course, we use the *SYNOPSYS VHDL-System-Simulator (VSS)* tools. This set of tools allows for the elaboration of VHDL-models using any language construct defined by the original standard IEEE-1076(1988), in contrast to the synthesis tools which are restricted to a synthesizable subset of VHDL. The main *VSS* -tools [SimVSS] are:

- VHDL analyzer:
  - *vhdlan* (shell command)
  - *gvan* (same functionality as *vhdlan* , but with graphical interface for the message output)
- VHDL simulator and debugger
  - *vhdldbx* (debugger with graphical interface)
  - *vhdlsim* (simulator , invoked from *vhdldbx*)
  - *waves* (waveform viewer , invoked from *vhdldbx*)
  - *browser* (allows for the switching of the current hierarchy level, invoked from *vhdldbx*)
- *SYNOPSYS* online documentation (*iview* command)

See The working library for the following exercises is *TEACH_EXMPL* . Thus, the source code for the following exercises is contained in the directory to *<home>/design/teach_exmpl /vhdl_src* .

Exercise 6.1.1: A first analyze-simulation run

A) Start a text editor and have a look at the following files: *hello.vhd* , *hello-assertion.vhd* , and *hello-stdtextio.vhd* . The two architectures are a small examples which demonstrate two possibilities for text output in VHDL. Analyze the three files by typing:

```
gvan –w teach_exmpl <filename>
```

Afterwards, invoke the VHDL-Debugger (the debugger comprises both, the simulation tool and interactive debugger) by typing:

```
vhdldbx&
```

Simulate the two architectures.

B) In the file *hello-assertion.vhd*, change the second assertion statement to:

SEVERITY WARNING;

Reanalyze the file and simulate again. What has changed? Try the same with the two other severity-levels *ERROR* and *FAILURE*. What happens?

Exercise 6.1.2: A RS-flipflop

The structure of the RS-flipflop containing two cross-coupled NOR-gates, is depicted in figure 6-1.

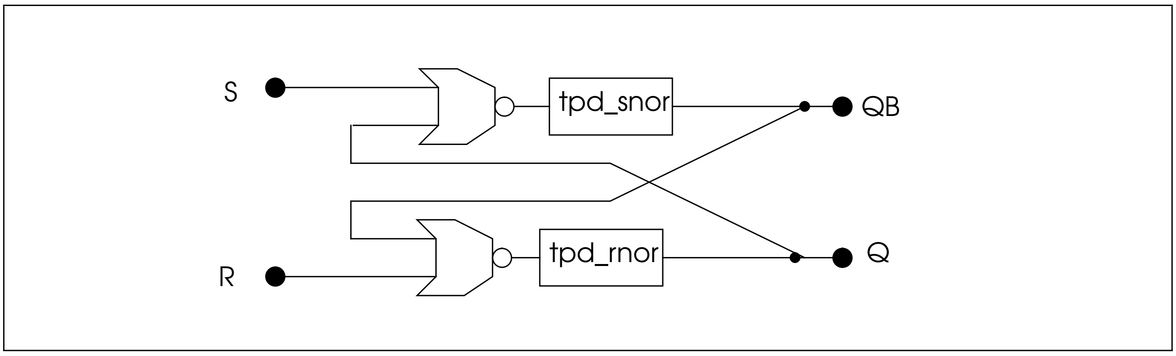

Figure 6-1: RS-flip containing two cross-coupled NOR-gates with gate delays

A) Have a look at the files:
rs_ff.vhd (entity-declaration of the RS-flipflop),
rs_ff-dataflow.vhd (dataflow architecture of the RS-flipflop),
rs_ff_test.vhd (entity-declaration of a test bench for the RS-flipflop),
rs_ff_test-bench.vhd (architecture of the test bench),
rs_ff_test-config1.vhd (configuration for the test bench).
Analyze these files according to the given order and invoke the debugger. Get familiar with the basic functions of the waveform viewer and the hierarchy browser. Simulate the configuration. Validate the function of the flipflop by examining the simulation output with the waveform viewer. Try out the various debugging facilities of the *vhdldbx* tool

such as setting breakpoints on source lines and signals, monitoring and evaluating of signals, step by step simulation etc.

B) Reset the simulation and simulate again using the step function. Watch the stated simulation time and delta time at each source line. Explain the simulation results.

C) In the file *rs_ff_test-bench.vhd*, set the first version of the stimuli generation statements into comments and remove the comments of the second version of the stimuli generation statements. Reanalyze the file (and the configuration file) and restart the simulation. Explain the simulation results. Proceed in this way with the third, fourth and fifth version while explaining each time the simulation results.

D) In the file *rs_ff_test-bench.vhd*, use the sixth version of stimuli generation statements. Reanalyze the file (and the configuration) and restart the simulation. What happens at the simulation time *24 ns*? Explain the simulation results. Change the generic parameters in the configuration file *rs_ff_test-config1.vhd* subsequently to:

tpd_snor <= 0 ns, *tpd_rnor <= 1 ns*
tpd_snor <= 1ns, *tpd_rnor <= 0 ns*
tpd_snor <= 0 ns, *tpd_rnor <= 0 ns*

and simulate each configuration. Explain the simulation results.

E) Have a look at the file *rs_ff-behaviour* and analyze it to your working library. It represents a behavioural description of the flipflop. To link this architecture version to the test bench, you have to change the *use*-statement in the configuration file *rs_ff_test-config1.vhd* and reanalyze it. Simulate this configuration. It should produce exactly the same results. (Use the first and sixth version of the stimuli generation statements).

F) Have a look at the file *rs_ff-structural* and analyze it to your working library. It represents a structural description of the flipflop. Within this architecture, two logic gates are used which are stored in the library *TEACH_LOGIC_LIB*. This library contains also a package (file *logic_components.vhd*) where all available gates are declared. Have a look at the file *rs_ff_test-config2.vhd*. It represents a complete configuration for all levels of the design hierarchy . To link this architecture version to the test bench, you have to analyze this file to your working library . Simulate this configuration. It should produce exactly the same results. (Use the first and sixth version of the stimuli generation statements).

G) Write an structural architecture for a RS-flipflop which contains two cross-coupled NAND-gates (RS-flipflop with low active inputs). Modify the test-bench to generate the required input-stimuli signals. Modify the configuration file and simulate the new architecture.

Exercise 6.1.3: Development of a clock generator

In this Exercise, a clock generator for a two phase, non-overlapping clock with power -on reset signal, as depicted in figure 6-2, shall be developed.

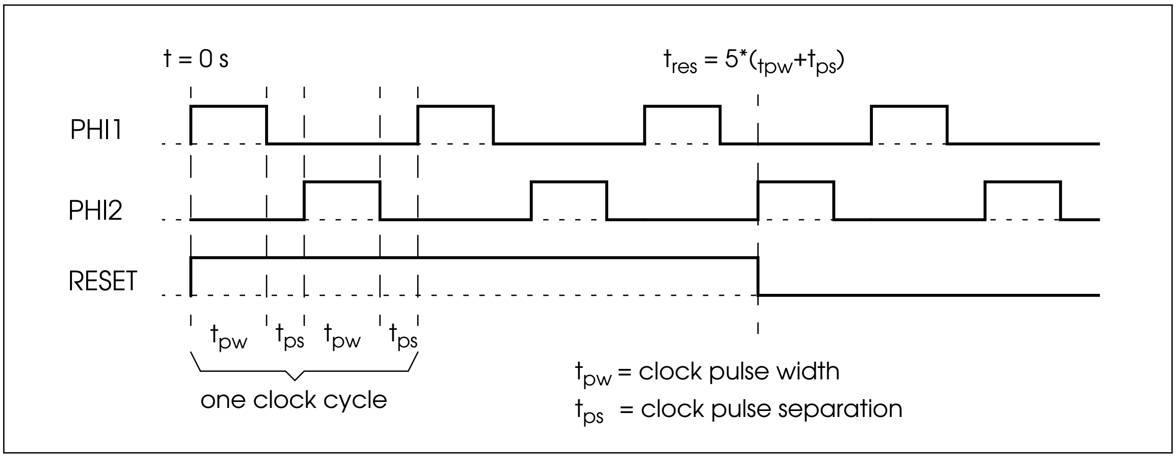

Figure 6-2: Two phase, non-overlapping clock signals and the reset signal

A) Write an entity declaration for the clock generator named *clockgen* . The entity shall have no input ports, three output ports, and two generic parameters, *tpw* and *tps* .

B) Develop a behavioural architecture for the clock generator named *behaviour* . Use one single process for the modelling.

C) Write an entity declaration for a test bench (named *clockgen_test)* for the clock generator . The entity shall have no input and output ports. Develop an architecture for the test bench named *bench* . Within this architecture, use the clock generator as a component. Connect the out-

puts of the clock generator to signals declared in the test bench architecture.
(Note: for the example of the clock generator, the development of a test bench is only for teaching purposes. It is not really necessary, because the clock generator has no inputs which need to be stimulated.)

D) Write a configuration for the test bench (named *clockgen_config_1*) with its subdesign and debug your design with the VHDL simulator *vhdldbx*.

E) Develop a architecture (named *dataflow*) for the clock generator which contains only concurrent (dataflow) statements. You can use initial values for internal signals. Change the configuration of the test bench and debug your design. Use the step function for simulation and watch the stated simulation time and delta time at each source line.

F) In this exercise part, a structural model for the clock generator shall be developed. You should use only the logic modules defined in the library TEACH_LOGIC_LIB. For this, you have to declare this library in your architecture description (see file *rs_ff-structural.vhd)*. Do not use initial values for internal signals. Each logic module of the library can be configured (in part G of this exercise) with a user-definable input-to-output delay, so that the modelling of the clock signals and the reset signal is possible.

G) Write a configuration for the test bench (named *clockgen_config_2*) in which the structural model and its subdesigns are configured (see file *rs_ff_test-config2.vhd* for the correct syntax). Assign the required delay times to the instantiated logic modules. Debug your design with the VHDL simulator *vhdldbx*.

Part II DLXS behavioural model

6.2 Architecture behaviour_1:

First steps to implement a behavioural model of the DLXS

Exercise 6.2.1: Preparation of the predefined libraries for simulation

The libraries and their contents are described in chapter 3.

A) Prepare the library *IEEE_EXTD* for simulation, i.e. analyze the files in its corresponding directory (see section 3.2.3) by typing:

```
gvan -w ieee_extd <filename>
```

The files should analyze (according to the given order) without errors.

B) Prepare the library *DLX* for simulation, i.e. analyze the files (except the *system* files) in its corresponding directory (see section 3.2.5.1):

```
gvan <filename>
```

The files should analyze (according to the given order) without errors (the option *-w ...* is not necessary because the default working-library *WORK* is mapped to the library *DLX*).

C) Prepare the library *DLX_BENCH* for simulation, i.e. analyze the files in its corresponding directory (see section 3.2.4) by typing:

```
gvan -w dlx_bench <filename>
```

The files should analyze (according to the given order) without errors.

Exercise 6.2.2: Creation of an entity for the DLXS

A) Write an entity declaration for the DLXS. The entity shall contain two generic parameters, *tpd_behav* of type time (general clock-to-output delay) and *debug* of type boolean (text output during simulation when set to *true*). Include the packages *std_logic_1 164* of the library *IEEE* and *dlx_types* and *dlx_instructions* from the working library *WORK* .

The names of the ports of the entity shall be: *a_bus* , *d_bus* , *enable* , *rw*, *ready* , *reset* , *halt* , *error*, *intrpt* , *phi1* , and *phi2* . Take care of the required port directions according to the DLXS interface. For the ports, use the types defined in the two mentioned packages! Analyze your entity description to check for errors.

Exercise 6.2.3: Development of a procedure to read from the system databus

Have a look at the package *behav_procedures_123* (declared in the files *behav_procedures_123.vhd* and *behav_procedures_123-body .vhd).* in your working library . At this time of the design course, it contains only the procedure *bus_write* which implements the bus-write transaction for the DLX according to the timing specified in chapter 1.

A) Extend this package by a procedure *bus_read* which shall implement the bus-read transaction for the DLX according to the timing specified in chapter 1. The prototype of the procedure has to be declared in the package declaration, the procedure itself in the package body . Analyze the extended package to check for errors.

Exercise 6.2.4: Development of the architecture *behaviour_1*

In this exercise, the basic structure of the behavioural model of the DLXS shall be implemented and simulated. The architecture for the DLXS named *behaviour_1* shall have the structure given in figure 7-1. This architecture contains only one process. W ithin this process, there is one endless loop, which implements the fetch-decode-execute cycle of the processor . This loop is only exited when a reset has been detected. In this case, some of the variables which represent the processor registers, are initialized and the process starts again.

A) For all the following steps don‘t forget to write comments in your VHDL-code!

B) Write an architecture “skeleton“ as shown in figure 7-1. Make the following packages visible in your architecture: *stdl1 164_extended* and *stdl1 164_vector_arithmetic* from the library *IEEE_EXTD* , *behav_procedures123* from your working library *WORK* , and *textio* from the standard library *STD* .

```
<make visible required packages>
ARCHITECTURE <architecture_name>
<any declarations needed ?>
BEGIN                       -- of architecture
  <process_label> : PROCESS -- only one process in architecture
  <declaration of variables>
  BEGIN                     -- of process
    <loop_label> : LOOP     -- endless loop only exited when
                            -- reset is active
      <check for reset active, exit loop in this case>
      <fetch instruction from bus>
      <increment program counter>
      <decode instruction:  write instruction text / decode>
      <execute instruction: empty for exercise 2>
    END LOOP <loop_label>

    <check: loop exited correctly (ASSERT)>

    IF <reset active> THEN
      <initialize variables>
      <wait until reset no longer active>
    END IF

  END <process_label>       -- end of process (start from beginning)

END <architecture_name>     -- end of architecture
```

Figure 7-1: Structure of the DLXS behavioural model

C) Define V ariables within the process. Each variable shall represent a register of the processor except the register -file which shall be declared as an array-variable. Use the types defined in the package *dlx_types* .
To ease the instruction decoding, define *ALIASes* for the instruction register which represent the defined parts of the DLXS instruction word.
Use the types defined in the package *dlx_types* .
Declare variables of type *natural* or of subtype *reg_index* (package *dlx_types*) which shall contain the decoded register addresses.

D) <check for reset active>: Use the EXIT command to exit the loop in case of an active reset signal.

E) <fetch instruction from bus> : When the *debug* -parameter is true, a message shall be written to the standard output:

"> instruction fetch from address : <instruction address>".

Use the procedure *bus_read* to read the instruction. Use the (overloaded) procedure *write* from the package *stdl_1 164_extended* for writing a variable of the type *std_logic_vector* .

F) <increment program counter> : When the *debug* -parameter is true, a message shall be written to the standard output:

"> incrementing program counter".

To increment the program-counter variable, use the addition procedure from the package *stdl1 164_vector_arithmetic.*

G) <decode instruction>: When the *debug* -parameter is true, a message shall be written to the standard output:

"> decoding instruction: <instruction>".

Use the procedure *write_instr* from the package *dlx_instructions* to write out the (decoded) instruction. Use the conversion functions of the package *stdl1 164_vector_arithmetic* to assign values to the variables which contain the decoded register addresses.

H) <execute instruction>: When the *debug* -parameter is true, a message shall be written to the standard output:

"> executing instruction".

(for this exercise, no instruction execution will be implemented).

I) <check: loop exited ...>: Use an ASSERT statement to check whether the main loop was exited correctly . The message shall be:

"DLXS-behaviour: main loop exited unexpectedly !"

The severity level is ERROR.

J) <initialize variables> : When the *debug* -parameter is true, a message shall be written to the standard output:

"> reset detected, resetting processor".

Set the required variables to zero (see section 1.2.6) and set the required processor outputs to the defined values. The process shall be stopped until the reset signal is cleared.

Exercise 6.2.5: First test of the model

A) Analyze your behavioural model and check for errors.

B) Adapt the top-level configuration in the file *system-config.vhd* to simulate the architecture *behaviour_1* .

C) Analyze the three files *system.vhd, system-structural.vhd* , and *system-config.vhd* .

D) Simulate the whole system (test bench). What are the simulation results?

E) Generate another ROM from the set of given assembler programs as described in chapter 2. Change the top-level configuration to simulate the new generated ROM and perform another simulation.

Note: Once you have analyzed all the VHDL units of your design, you can generate a makefile for it to ease further modifications. In the directory *~/design/dlx* enter the command:

simdepends <top-unit name> *> makefile*

where *<top unit name>* stands for the VHDL name (and not the file name) of the top-level unit in your design. In our case, it is *system_config* .

Afterwards, you can update your design every time you made changes just by typing:

make

and restart the simulation.

6.3 Architecture behaviour_2

Extension of the behavioural model by arithmetical-, logical-, and test instructions

A) Copy your file *dlx-behaviour_1.vhd* containing the behavioural model developed in the previous session to a file named *dlx-behaviour_2.vhd* and change the architecture name to *behaviour_2* .

Exercise 6.3.1: Implementation of the arithmetical and logical instructions

The structure of the execution part of the behavioural model is shown in the figure 7-2.

```
...
<execute instruction: >
  {compare opcode with all allowed choices}

    {register-register ALU-function ?}
      {compare function with all allowed choices and execute it}
      {not a allowed function: report error}

    {other opcode :}
    {compare with other allowed choices and execute operation}

    {not an allowed opcode: report error}

<end of execution>
...
```

Figure 7-2: Structure of the execution part of the behavioural model

A) Which (sequential) VDHL statement is the best suited to implement the comparison between the actual instruction and the allowed choices?
Use the constants defined in the package *dlx_instructions* from your working library *WORK* for the comparison.

B) Implement the arithmetical and logical instructions according to table 1-2. Use the procedures defined in the package *stdl1 164_vector_arithmetic* from the library *IEEE_EXTD.*
Do not take into account arithmetic overflows. Keep in mind that 16-bit

immediates for arithmetical operations are sign-extended whereas those for logical operations are zero-extended.
Use comments !

C) Implement the test-an-set instructions according to table 1-3. Use the procedures defined in the package *stdl1 164_vector_arithmetic* from the library *IEEE_EXTD*.
Use comments !

D) Use assert-statements to write out a warning in case that a non-defined opcode or register -register ALU-function is detected.

Exercise 6.3.2: Test of the new implemented instructions

A) Analyze and simulate your new architecture. To do this, write small assembler programs to test your model. Use the assembler *dlx_asm* to generate your specific ROM desription, change the top-level configuration *system_config* to “plug-in“ your ROM and simulate your model.
Because no load instructions are implemented so far, you have to load immediate values into the registers which shall serve as source operands to test an arithmetic or logical instruction, e.g:

```
or.ir1, r0, X"ab03"     ;load r1 with hex(ab03)
or.ir2, r0, X"3fe8"     ;load r2 with hex(3fe8)
add r3, r1, r2          ;add r1 and r2 put result in r3
```

There are some predefined assembler programs in the directory *~/asm_src*.

6.4 Architecture behaviour_3:

Extension of the behavioural model by load-, store-, branch-, jump-, and move instructions

A) Copy your file *dlx-behaviour_2.vhd* containing the behavioural model developed in the previous session to a file named *dlx-behaviour_3.vhd* and change the architecture name to *behaviour_3* .

Exercise 6.4.1: Implementation of the jump instructions

A) Extend your model by the four jump instructions. This can easily be done by adding the sign-extended immediate to the PC (for this exercise, you do not have to check for overflows) or by loading the value of the concerned register into the PC respectively .
For the two jump-with-link (subprogram call) instructions, the old content of the PC has to be saved in the predefined register before the PC is changed.

Exercise 6.4.2: Implementation of the branch instructions

A) Extend your model by the two branch instructions. Keep in mind that the offset added to the PC is signed. When the *debug* -parameter is true, a message shall be written to the standard output. In case of a taken branch, the message shall be:

"> branch taken"

otherwise it shall be:

"> branch not taken".

Exercise 6.4.3: Implementation of the move instructions

A) Extend your model by the two move-to/from-special-register instructions. They are Rtype-instructions. In the instruction word, the register IAR is encoded with the register address 1(bin"00001"), the ICR with the address 2(bin"00010"), and the TBR with the address 4(bin"00010").
In case of a move-to-special-register , this address is coded in the field RD, in case of a move-from-special-register it is coded in the field RS1 (see figure 1-1).

Exercise 6.4.4: Test of the new implemented instructions

A) Write small assembler programs to test the new implemented instructions. Use the assembler *dlx_asm* to generate your specific ROM description, change the top-level configuration to “plug-in“ your ROM and simulate your model.

Exercise 6.4.5: Implementation of the six load instructions

A) For all load instructions, the first step is to calculate the memory address and to store it in the MAR.

B) The next step is to read the data from the system data bus into the MDR by calling the predefined procedure. Within this procedure, the reset line is checked after each clock cycle, and the procedure is exited in case of an active reset signal. Thus, after the procedure call, an conditional exit statement has to be implemented to finish the loop in case of an active reset signal.

C) The final step depends on the very instruction. For the load-word instruction, the implementation is finished by assigning the value of the MDR to the destination GPR.
For the load-half instruction, either the upper or the lower half word contained in the MDR has to be sign extended and assigned to the destination GPR. The required half word is defined by the second least significant bit of the address contained in the MAR (see also figure 1-1).
For the load-byte instruction, one of the four bytes contained in the MDR has to be sign extended and assigned to the required GPR. The required byte is defined by the two least significant bits of the address contained in the MAR (see also figure 1-1).

Exercise 6.4.6: Implementation of the six store instructions

A) For all store instructions, the first step is to calculate the memory address and to store it in the MAR.

B) The second step depends on the very instruction. For the load-word instruction, the MDR is loaded with the destination GPR.

For the store-half instruction, either the upper or the lower half word of the MDR has to be loaded with the least significant half word of the destination GPR. The required half word of the MDR is defined by the second least significant bit of the address contained in the MAR (see also figure 1-1). The other bits of the MDR shall be set to zero.
For the store-byte instruction, one of the four bytes of the MDR has to be loaded with the least significant byte of the destination GPR. The required. The required byte of the MDR is defined by the two least significant bits of the address contained in the MAR (see also figure 1-1).

C) For all store instructions, the final implementation step is equal. The predefined procedure to write on the system bus has to be called. Within this procedure, the reset line is checked after each clock cycle, and the procedure is exited in case of an active reset signal. Thus, after the procedure call, an conditional exit statement has to be implemented to finish the loop in case of an active reset signal.

Exercise 6.4.7: Test of the new implemented instructions

A) Write small assembler programs to test the new implemented instructions. Use the assembler *dlx_asm* to generate your specific ROM description, change the top-level configuration *system_config* to “plug-in“ your ROM and simulate your model.

6.5 Architecture behaviour_4

Extension of the behavioural model by halt-, interrupt-, and exception handling

A) Copy your file *dlx-behaviour_3.vhd* containing the behavioural model developed in the previous session to a file named *dlx-behaviour_4.vhd* and change the architecture name to *behaviour_4* .

B) Copy the file *behav_procedures_123.vhd* to a file named *behav_procedures_4.vhd* and change the package name to *behav_procedures_4* . Copy the file *behav_procedures_123-body .vhd* to a file named *behav_procedures_4-body .vhd* and change the package-body name to *behav_procedures_4* .

Exercise 6.5.1: Implementation of the halt function

A) Include a new procedure named *check_halt* in this package. It shall check whether the halt input of the processor is active or not. When this signal is active and the *debug* -parameter is true, a message shall be written to the standard output:

“> halt detected, freezing processor“.

All bidirectional and output ports of the processor shall be set to high impedance state until the halt signal is set to zero again. When the *debug* -parameter is true, a message shall be written to the standard output:

“> resuming from halt“.

B) This new procedure shall be called at the beginning of the fetch-decode-execute loop. Y ou can test this new feature by setting the halt signal of the system to ‘1‘ during simulation.

Exercise 6.5.2: Implementation of the hardware interrupt handling

A) To improve the readability of your code, define *ALIASes* for the interrupt-control register which represent the interrupt bits and the mask bits.

B) The behavioural model of the processor has to check in each cycle whether an interrupt input has become active. This shall be done during the ‘1‘-phase of *phi2*. If at least one interrupt input is active, the concerning interrupt bit in the ICR has to be set. Extend the package *behav_procedures _4* by a procedure named *intrpt_check* which implements this behaviour.

C) The only timed behaviour of our model is implemented in the two read and write procedures in the package *behav_procedures_4*. Thus, they have to be extended. After each wait statement which waits on the clock *phi2* to become ‘1‘, the above defined procedure shall be called.

D) To implement the interrupt handling in your processor model, the interrupt bits have to be checked after each instruction fetch (before the incrementing of the PC). If at least one interrupt bit is set as well as its corresponding mask bit, the processor shall behave as described in section 1.2.5.2. The saving of the interrupt information in the ICR can easily be done by shifting its least significant half-word into the most significant half-word. This will also clear all mask bits and prevent another hardware interrupt to be taken into account during the execution of the current interrupt. Afterwards, the program execution is continued at the new address with a fetch operation...

E) Extend your model by the instruction *rfe* (return from interrupt/exception). The processor shall behave as described in section 1.2.5.4. If the *debug* parameter is true, a message shall be written to the standard output stating the new address in the PC:

“> return from int./except.: resuming from address: X“....“ “.

Exercise 6.5.3: Implementation of the software interrupt

A) Extend your model by the instruction *trap* (software interrupt). The processor shall behave as described in section 1.2.5.1. When the *debug* parameter is true, a message shall be written to the standard output stating the new address in the PC:

“> trap detected, jumping to interrupt address: X“....“ “.

Exercise 6.5.4: Implementation of the exception handling

A) To improve the readability of your code, define *ALIASes* for the interrupt-control register which represent the seven different exception bits and the supervisor bit.

B) In your architecture, declare two new variables (e.g. *exc_flag*, *prev_exc_flag*) which shall serve as flags. The first shall be set when an exception has occurred during the actual execution, the second when an exception has occurred during a prior execution.

In all cases described below, the fetch-decode-execute loop has to be exited immediately if an exception is detected.

C) The IOC bit has to be set when the opcode has shown to be illegal after the comparison with all legal possibilities.

D) The IRRA bit has to be set when the register-register-ALU function code has shown to be illegal after the comparison with all legal possibilities.

E) The IA V bit has to be set when a new calculated instruction address is not word aligned. This can happen during the execution of the four jump instructions, the two branch instructions, and the trap instruction.

F) The DA V bit has to be set when a new calculated data address is not word aligned (load word / store word instructions) or not half aligned (load half / store half instructions). This can happen during the execution of the twelve load/store instructions.

G) The OV AD bit has to be set when the incrementing of the PC has caused an overflow.

H) The OV AR bit has to be set when an arithmetical operation has caused an overflow. This can occur during the execution of the two addition and the two subtraction instructions.

I) The PRIV bit has to be set, when the processor is in user mode and a data or instruction address had been calculated which is in the supervisor address segment or a privileged instruction (*movi2s, movs2i, rfe*) was detected.

J) Extend the ASSERT statement which checks whether the fetch-decode-execute loop was exited correctly according to the new exit condition for the fetch-decode-execute loop in the architecture.

K) Extend the model (outside the fetch-decode-execute loop) which implements so far only the reset execution by the exception execution according to section 1.2.4 and section 1.2.5 (note that if the fetch-decode-execute loop had been exited while the processor was already handling an exception, it changes into the error mode).

When the *debug* parameter is true and an exception occurred, a message shall be written to the standard output:
“> exception detected“.
In case of an error , a message shall be written to the standard output:
“> error detected, waiting for reset“.

Don‘t forget to extend the reset routine to clear the two exception flag variables.

Exercise 6.5.5: Simulation of the complete behavioural model

The behavioural model of the DLXS processor is now complete.

A) To simulate and validate your model, use the given assembler programs (directory *~/asm_src*). Each program tests a subset of the processor instructions. The normal structure of such a program is the use of the concerning instructions and a test of the result. In case of a expected result, the program continues or branches to a address labelled “good“, otherwise the program branches to an endless loop or to an address labelled “bad“. Y ou can also write your own assembler programs. Use the assembler *dlx_asm* to generate the specific ROM description, change the top-level configuration *system_config* to “plug-in“ your ROM and simulate your model.

Part III DLXS register-transfer-level model

6.6 Implementation of a structural model of the DLXS

The next design step on the way to a gate-level description of the processor is the implementation of register -transfer -level (RTL) model. The global structure of this model and the controller are already predefined. The task of this part is to implement the datapath and its subdesigns according to figure 1-8. The complete design hierarchy of the DLXS is depicted in figure 7-3. The two lowest hierarchy levels will be used only in the synthesis part (Part IV) of this course and are not of interest for this part of the exercises.

The structural model consists of the top-level architecture (*dlx*) of the DLXS which contains the processor core (*core*) and the pads (*pad_in, pad_out, pad_io*). The processor core consists of two parts, the controller (*controller*) and the datapath (*datapath*).

The architecture of the controller uses an instanced finite state machine (*fsm)* and three instruction decoders *(ir_decode_1, ir_decode_2, ir_decode_3).*

The datapath consists of the ALU (*alu*), the register file (*reg_file*), the interrupt control register (*icr*), the instruction register (*ir*), the memory data register (*mdr*), a 32-bit latch (*word_latch*, used for the c-register), a 32-bit latch with one tri-state output (*word_reg_1e*, used for the a-register, the b-register, the interrupt address register, and the trap base register), a 32-bit latch with one tristate and one direct output (*word_reg_1e_1*, used for the program counter and the memory address register), a 32x2:1-bit multiplexer (*word_mux2*), and a bus constant circuit with two tristate outputs (bus_*const32*).

A) Analyze the new source files of the library DLX (according to section 3.2.5.3) to your working library. You must analyze the design units first which are at the lowest level of the design hierarchy, e.g. the finite state machine has to be analyzed before the controller etc.

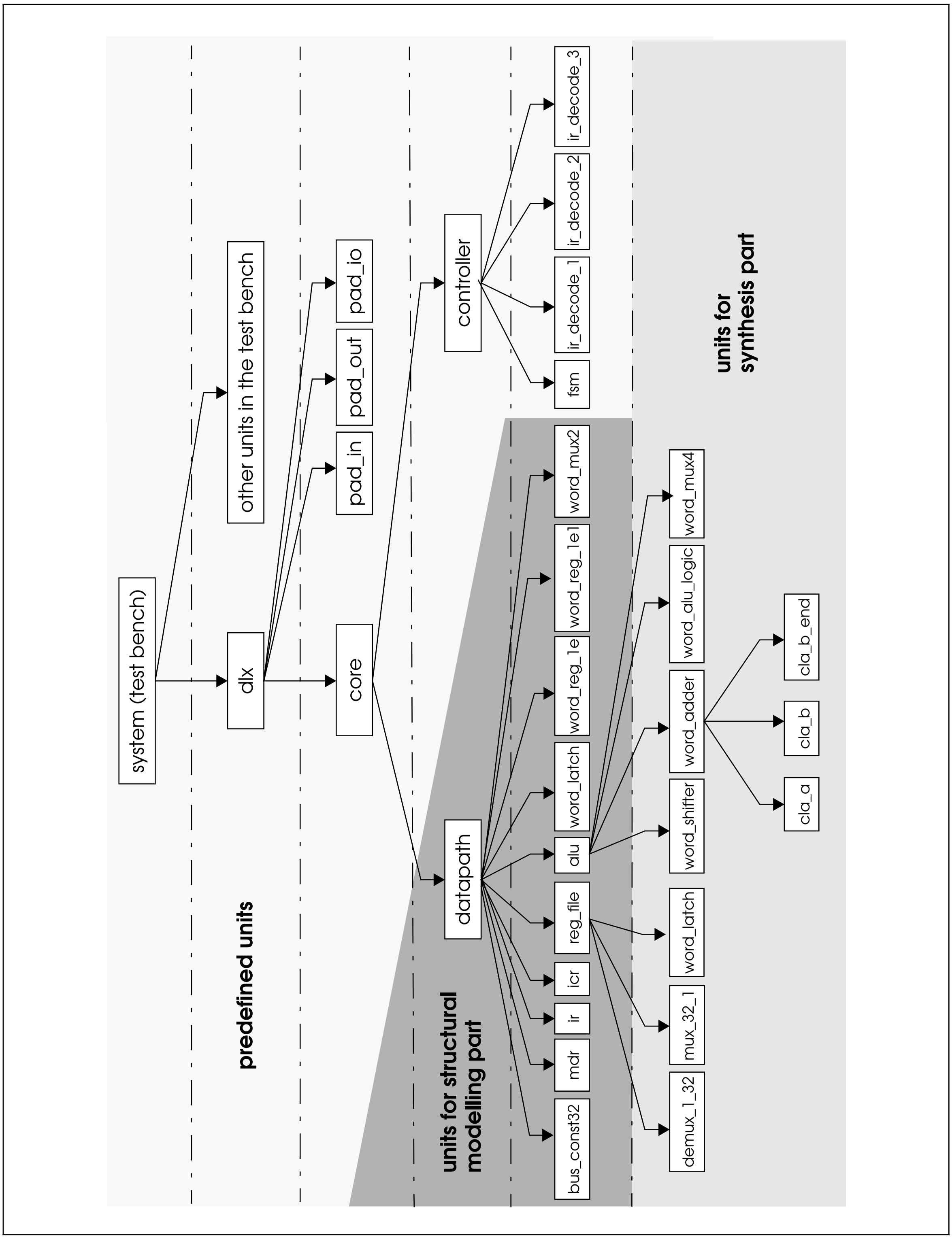

Figure 7-3: The design hierarchy of the structural processor model

Exercise 6.6.1: Implementation of the three different 32-bit latches

A) Write an entity declaration for the 32-bit latch (*word_latch)* according to figure 7-4-a. The delay time between *d* and *q* shall be defined by a generic parameter *tpd_latch*.

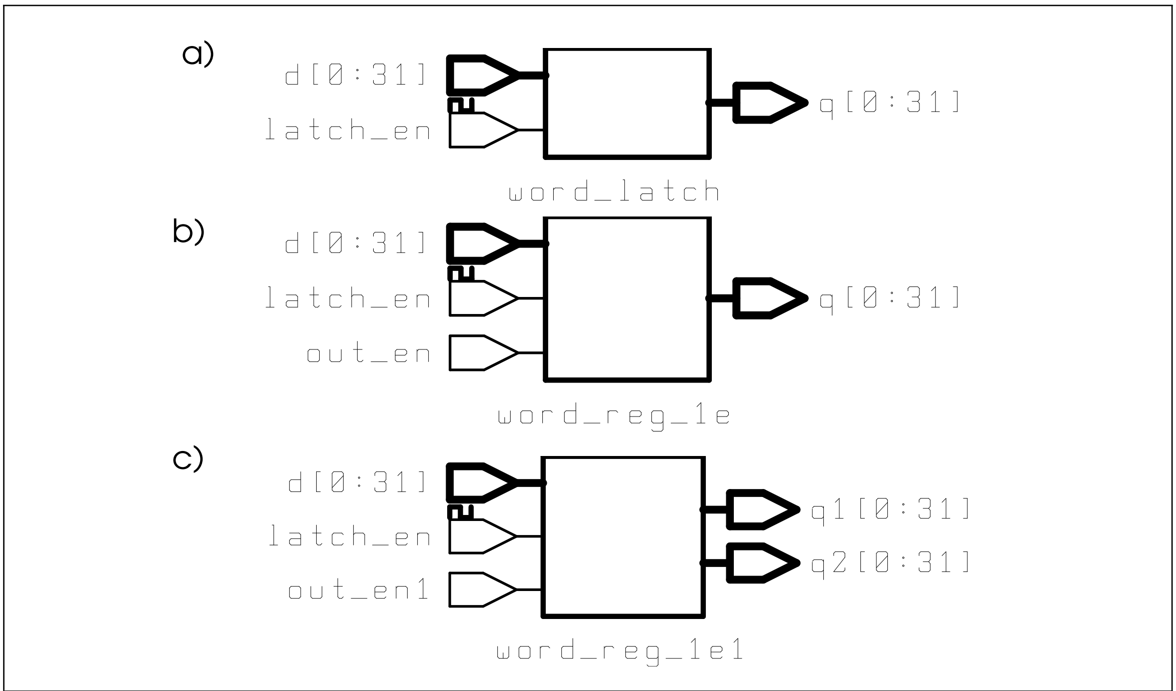

Figure 7-4: 32-bit latch and two 32-bit register

B) Develop an architecture for this entity . The function of the latch can be implemented in one process with a sensitivity list. The latch is loaded when the input *latch_en* is zero (low active signal). Keep in mind that a register value can‘t be ‘Z‘.

C) Write an entity declaration for the 32-bit register with one tristate output (*word_reg_1e*) according to figure 7-4-b. The delay time between *d* and the internal latch output shall be defined by a generic parameter *tpd_latch,* the output delay time between the internal latch output and *q* by the generic parameter *tpd_out*.

D) Develop an architecture for this entity . The function of the latch can be implemented in one process with a sensitivity list. The latch is loaded when the input *latch_en* is zero (low active signal). Keep in mind that a register value can‘t be ‘Z‘. The stored value is written to the output q when *out_en* is zero (low active signal). The output function can be implemented with concurrent statements.

E) Write an entity declaration for the 32-bit register with one tristate output and one direct output (*word_reg_1e1*) according to figure 7-4-c. The delay time between *d* and the internal latch output shall be defined by a generic parameter *tpd_latch,* the output delay time between the internal latch output and *q1* by the generic parameter *tpd_out* .

F) Develop an architecture for this entity . The function of the latch can be implemented in one process with a sensitivity list. The latch is loaded when the input *latch_en* is zero (low active signal). Keep in mind that a register value can‘t be ‘Z‘. The stored value is written to the output *q1* when *out_en1* is zero (low active signal) and directly connected to the output *q2*. The output function can be implemented with concurrent statements.

G) Write a small test bench (entity/architecture pair with corresponding configuration) for the three register types developed in this exercise. Within the test-bench architecture, declare appropriate test pattern (waveforms) for the input signals to validate your designs.

Exercise 6.6.2: Implementation of the bus constant circuit

A) Write an entity declaration for the bus-constant circuit (*bus_const32)* with two tristate outputs according to figure 7-5. The output delay time between *out_en1(2)* and *q1 (q2)* shall be defined by the generic parameter *tpd_out* .

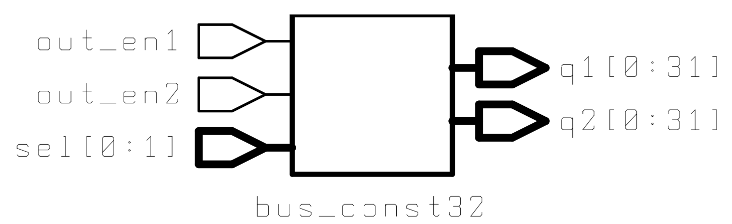

Figure 7-5: Bus constant circuit

B) Develop an architecture for this entity . To implement the functionality , refer to table 6-1 for the required constant values and their encoding by the input *sel*. The corresponding constants are written to the output *q1* when the input *out_en1* is zero (low active signal). This is also valid for *out_en2* and *q2* respectively .

Constant	sel(0:1)	Connected to	Needed for
hex"0000_0000"	00	S1 Bus	test and set instructions
hex"0000_0001"	01	S1 Bus	test and set instructions
hex"0000_0004"	10	S1 Bus	PC increment
hex"0000_0000"	00	S2 Bus	test and set instructions
hex"0000_0010"	01	S2 Bus	shift for load of signed byte/half
hex"0000_0018"	10	S2 Bus	shift for load of signed byte/ half

Table 6-1: The S1 and S2 Bus Constants

C) Write a small test bench (entity/architecture pair with corresponding configuration) for the two bus-constant circuits developed in this exercise. W ithin the test-bench architecture, declare appropriate test pattern (waveforms) for the input signals to validate your designs.

Exercise 6.6.3: Implementation of the 32x2:1 multiplexer

A) Write an entity declaration for the 32x2:1 multiplexer (*word_mux2)* according to figure 7-6. The output delay time between *in0 / in1* and *q* shall be defined by the generic parameter *tpd_out* .

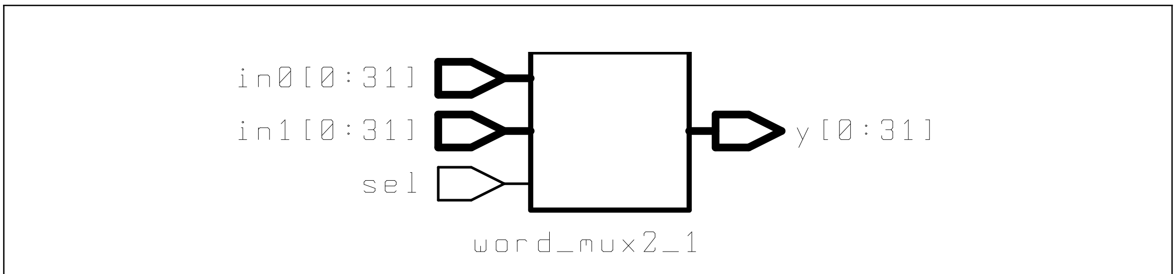

Figure 7-6: 32x2:1 multiplexer

B) Develop an architecture for this entity . When *sel* is zero, the input *in0* shall be connected to the output *y*, otherwise the input *in1* .

C) Write a small test bench (entity/architecture pair with corresponding configuration) for the multiplexer developed in this exercise. W ithin the test-bench architecture, declare appropriate test pattern (waveforms) for the input signals to validate your design.

Exercise 6.6.4: Implementation of the instruction register

A) Write an entity declaration for the instruction register (*ir)* according to figure 7-7. The delay time between *d* and the internal latch output shall be defined by a generic parameter *tpd_latch,* the output delay time between the internal latch output and the two outputs *immed_out1* and *immed_out2* by the generic parameter *tpd_out.* .

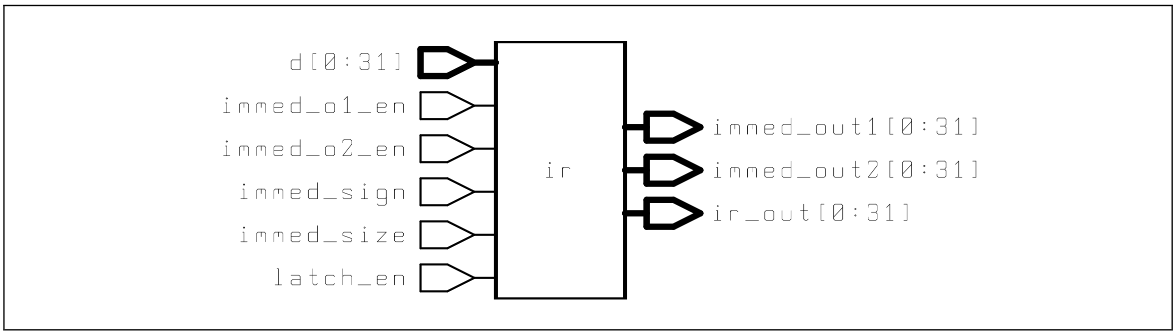

Figure 7-7: Instruction register (IR)

B) Develop an architecture for this entity . The load function can be implemented in one process with a sensitivity list. The addressed latch is loaded when the input *latch_en* is zero (low active signal). Keep in mind that a register value can't be 'Z'. The output *ir_out* is directly connected to the latch (no tristate).

The output function can be implemented in another process with a sensitivity list. The outputs *immed_out1* and *immed_out2* are tristate (the corresponding output-enable signals *immed_o1_en* and *immed_o2_en* are low active). When the input *immed_size* is zero, these outputs are 16-to32 bit extended immediates, otherwise 26-to-32 bit extended. When the input *immed_sign* is zero, the immediates are zero-extended, otherwise sign-extended (see also section 1.2.2).

C) Write a small test bench (entity/architecture pair with corresponding configuration) for the instruction register developed in this exercise. Within the test-bench architecture, declare appropriate test pattern (waveforms) for the input signals to validate your design.

Exercise 6.6.5: Implementation of the memory data register

A) Write an entity declaration for the memory data register (*mdr*) according to figure 7-8. It corresponds to the *word_reg_1e1* design unit with a special shift capacity . The delay time between *d* and the internal latch output shall be defined by a generic parameter *tpd_latch,* the output delay time between the internal latch output and *q1* by the generic parameter *tpd_out* . Another generic parameter , *tpd_logic* , shall describe the delay of the internal shift logic;

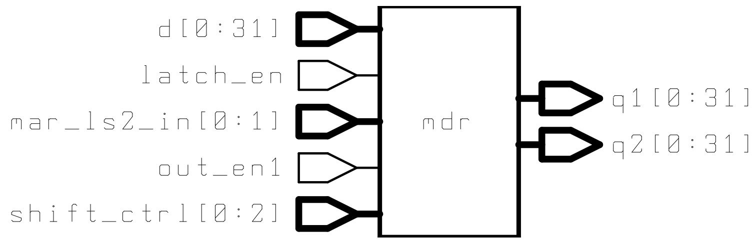

Figure 7-8: Memory data register

B) Develop an architecture for this entity . The shift logic can be implemented in one process with a sensitivity list. The shift function depends on the inputs *shift_ctrl* and *mar_ls2_in* according to table 6-3. The output of the shift block shall be written to a temporary 32-bit signal.

Operation	shift_ctrl(0:2)	Shift Operation
load / store word	0 - -	none
load half	1 0 0	MDR(0:15) ← d(0:15) if MAR(30) = 1 else MDR(0:15) ← d(16:31)
load byte	1 0 1	MDR(0:7) ← d(0:7) if MAR(30:31) = 00 else MDR(0:7) ← d(8:15) if MAR(30:31) = 01 else MDR(0:7) ← d(16:23) if MAR(30:31) = 10 else MDR(0:7) ← d(24:31)
store half	1 1 0	MDR(0:15) ← d(0:15) if MAR(30) = 0 else MDR(0:15) ← d(16:31)
store byte	1 1 1	MDR(0:7) ← d(24:31) if MAR(30:31) = 00 else MDR(8:15) ← d(24:31) if MAR(30:31) = 01else MDR(16:23) ← d(24:31) if MAR(30:31) = 10 else MDR(24:31) ← d(24:31)

Table 6-2: Coding of the MDR shift operations

The function of the latch can be implemented in one process with a sensitivity list. The latch is loaded with the temporary value when the input *latch_en* is zero (low active signal). Keep in mind that a register value can‘t be ‘Z‘. The stored value is written to the output q1 when *out_en1* is zero (low active signal) and directly connected to the output *q2*. The output function can be implemented with concurrent statements.

C) Write a small test bench (entity/architecture pair with corresponding configuration) for the memory data register developed in this exercise. Within the test-bench architecture, declare appropriate test pattern (waveforms) for the input signals to validate your designs.

Exercise 6.6.6: Implementation of the interrupt control register

The ICR and all its bits are described in figure 1-6.

A) Write an entity declaration for the interrupt control register(*icr)* according to figure 7-9.The delay time between *d* and the internal latch output shall be defined by a generic parameter *tpd_latch,* the output delay time between the internal latch output and the outputs *q, super* , and *intrpt* by the generic parameter *tpd_out.*

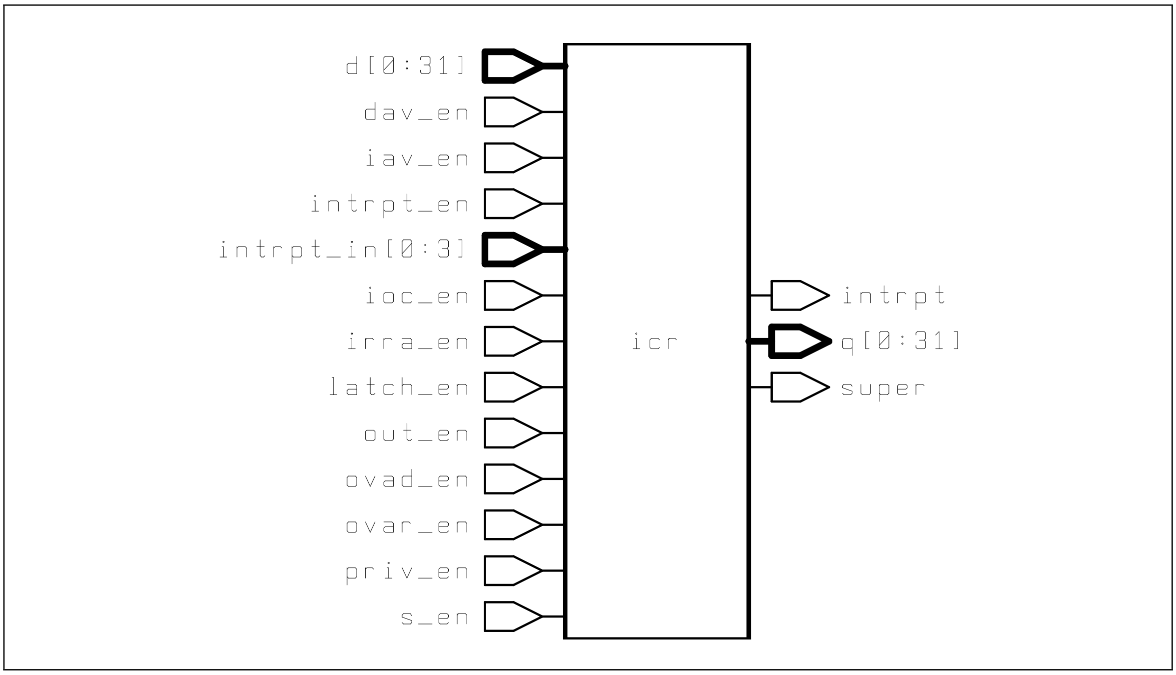

Figure 7-9: Interrupt control register (ICR)

B) Start the implementation of the architecture by defining a 32-bit signal for the value to be stored.

C) The S, IOC, IRRA, IA V, DAV, OVAD, OVAR, and PRIV bits can be implemented each in a process with a sensitivity list. When the correspond-

VLSI Design Course by 1995 M. Gumm, University of Stuttgart, 1995

ing load-enable input (e.g. *ioc_en*) is zero (low active signal), the bit is set to one. Otherwise, if the input latch_en is zero (low active signal), the bit is loaded with the value of the corresponding bit of the input *d*. Keep in mind that a register value can't be 'Z'.

D) The four interrupt bits can be implemented each in a process with a sensitivity list. They are set to one when the input *intrpt_en* as well as the corresponding bit from the interrupt input are one. Otherwise, if the input latch_en is zero (low active signal), the bits are loaded with the value of the corresponding bit of the input *d*. Keep in mind that a register value can't be 'Z'.

E) All other bits can be implemented together in one process with a sensitivity list. They are loaded with the value of the corresponding bit of the input *d* when the input *latch_en* is zero (low active signal). Keep in mind that a register value can't be 'Z'.

F) The contents of the register are connected to the output q when the enable input *out_en* is zero (low active signal). The *intrpt* output is set to 1 if at least one interrupt bit and its corresponding mask bit are 1. The *super* output is set to 1 if the S (supervisor) bit is one.

G) Write a small test bench (entity/architecture pair with corresponding configuration) for the interrupt-control register developed in this exercise. W ithin the test-bench architecture, declare appropriate test patterns (waveforms) for the input signals to validate your design.

Exercise 6.6.7: Implementation of the ALU

A) Write an entity declaration for the ALU (*alu)* according to figure 7-10. The delay time between the inputs *s1 / s2* and the internal latch outputs shall be defined by a generic parameter *tpd_latch,* the output delay time between the internal latch outputs and *result / overflow / zero / negative* by the generic parameter *tpd_out*.

B) Develop an architecture for this entity. The ALU shall contain two 32-bit input latches which are loaded when the input *latch_en* is zero (low active signal). Keep in mind that a register value can't be 'Z'. The latch

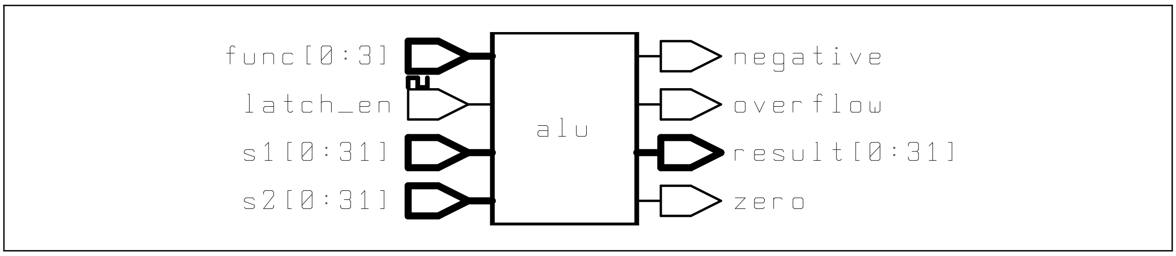

Figure 7-10: ALU

function can be implemented in one process with a sensitivity list.

The ALU function can be implemented in another process with a sensitivity list. The input *func* determines the ALU function according to table 6-3. The output *zero* shall be set to one when the calculated result is zero, the outputs negative and overflow shall be set when the result is negative (msb = 1) or in case of an arithmetic overflow respectively. Use the procedures and functions declared in the package *stdl1 164_vector_arithmetic*.

Operation	func(0:3)	Operation	func(0:3)
pass S1	00 -0	logical shift right	10 10
pass S2	00 -1	arithmetical shift right	10 11
signed addition	01 -0	logical and	11 00
signed subtraction	01 -1	logical or	11 01
logical shift left	10 0-	logical xor	11 1-

Table 6-3: Coding of the ALU operations

C) Write a small test bench (entity/architecture pair with corresponding configuration) for the ALU developed in this exercise. Within the test-bench architecture, declare appropriate test pattern (waveforms) for the input signals to validate your design.

Exercise 6.6.8: Implementation of the register file

A) Write an entity declaration for the register file(*reg_file)* according to figure 7-11. The delay time between *d* and the internal latch output *s* shall be defined by a generic parameter *tpd_in,* the output delay time between the internal latch outputs and the two outputs *q1* and *q2* by the generic parameter *tac*.

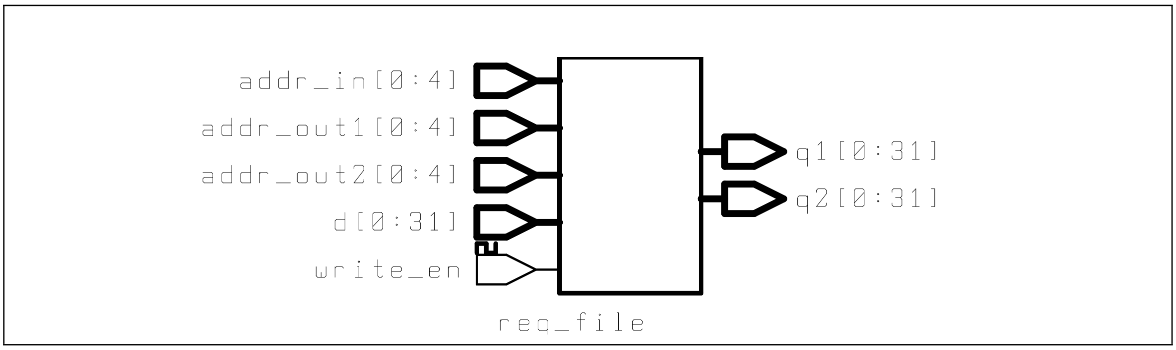

Figure 7-11: Register file

B) Develop an architecture for this entity. The load function can be implemented in one process with a sensitivity list. The addressed latch is loaded when the input *latch_en* is zero (low active signal). Keep in mind that a register value can't be 'Z'. The output function can be implemented in another process with a sensitivity list. The addressed latches are connected directly to the two outputs *q1* and *q2* are direct (no tri-state).

C) Write a small test bench (entity/architecture pair with corresponding configuration) for the register file developed in this exercise. Within the test-bench architecture, declare appropriate test pattern (waveforms) for the input signals to validate your design.

Exercise 6.6.9: Implementation of the structural model of the datapath

A) Write an entity declaration for the datapath (*datapath)* according to figure 7-12. You can copy the port list from the component instantiation in the file *core-structural.vhd*. The port names of the entity and their corresponding function are also explained in this file.

B) Develop an structural architecture for this entity. This architecture shall contain only the different components of the datapath (according to figure 1-8) and their connections.

Exercise 6.6.10: Development of a configuration for the datapath

A) Develop an configuration for the datapath named *datapath_config1*. Assign for all generic time parameters a value of 2 ns except for the register file (4ns) and the ALU (2ns and 10 ns).

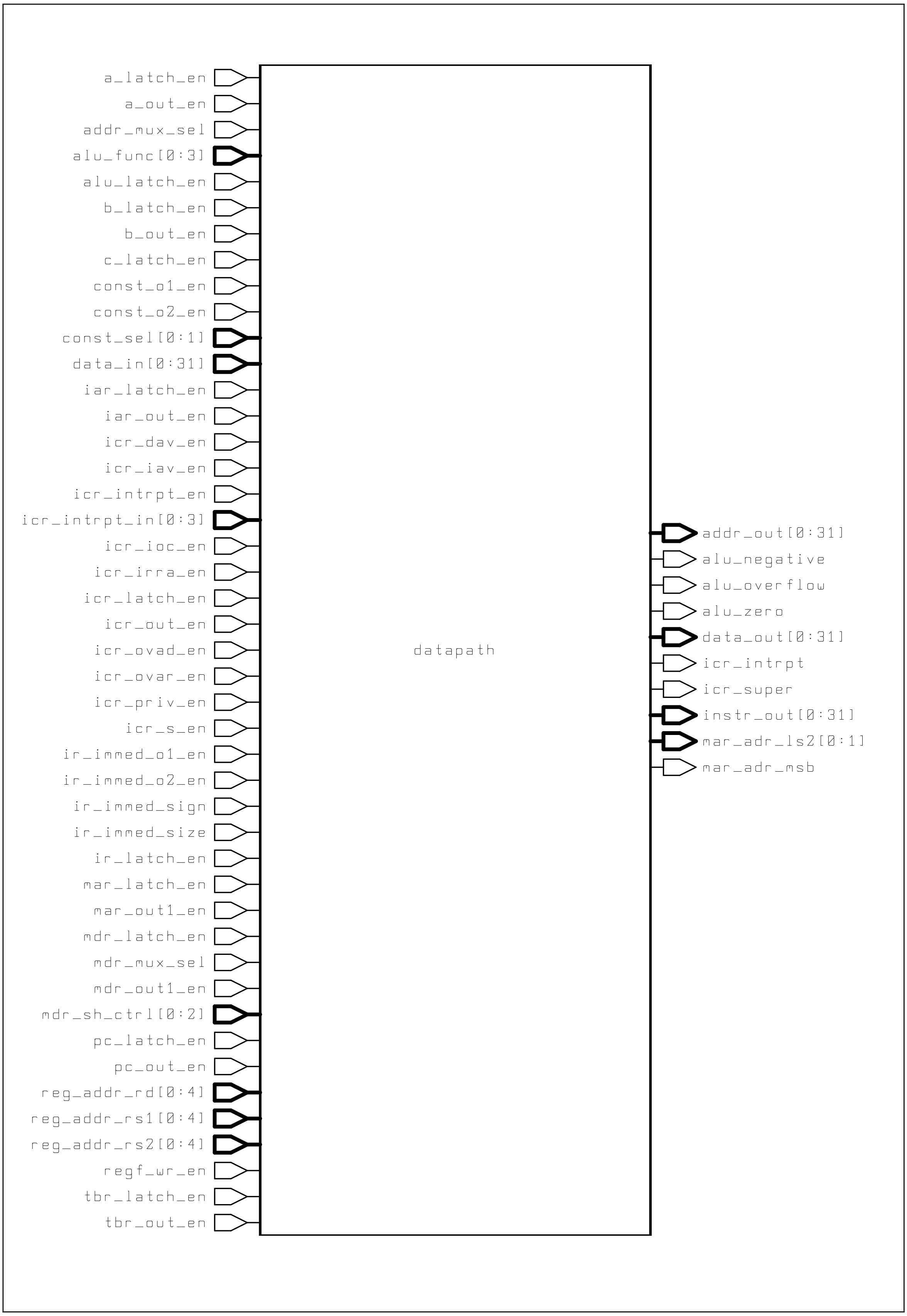

Figure 7-12: The complete datapath

Exercise 6.6.11: Simulation of the RTL model of the DLXS

The RTL model of our processor is now complete and ready for sythesis (if the VHDL statements you employed are contained in the subset which is supported by the SYNOPSYS synthesis tools; this should be the case).

A) Before you can simulate the datapath, you have to modify the system-configuration file (*system-config.vhd*). Instead of the statement

FOR test_dlx USE entity WORK.dlx(behaviour_4);

you have to insert

FOR test_dlx USE configuration WORK.dlx_config_rtl1;

B) To simulate and validate your model of the datapath with the structural model of the DLXS, proceed as described in section 6.5.5. You should test your model extensively and try to run all given assembler programs to be able to find all the bugs of your datapath model (there will be quite a a lot of them!).

Note: Once you have analyzed all the VHDL units of your design, you can generate a makefile for it to ease further modifications. In the directory *~/design/dlx* enter the command:

simdepends <top-unit name> *> makefile*

where *<top unit name>* stands for the VHDL name (and not the file name) of the top-level unit in your design. In our case, it is *system_config* .

Afterwards, you can update your design every time you made changes just by typing:

make

and restart the simulation.

Part IV Synthesis of the datapath

6.7 Synthesis of a gate-level netlist from the RTL model of the datapath

For this part of this design course, we use another set of the *SYNOPSYS* tools:

- *Design Analyzer* [SynDA] (invoked with the command *design_analyzer*)
- *Design Compiler* [SynDC] [SynDCT ut] (main synthesis tool, invoked from *Design Analyzer* or by command shell)
- *VHDL Compiler* [SynVHDL] (invoked from Design Analyzer)
- Timing Analyzer (invoked from Design Analyzer)

Note: Commands from the Design Analyzer menus are described in a hierarchical way . For example |*File*|*Save*| means the *Save* command from the *File* menu.

Note: In this section when VHDL descriptions are to be "anayzed", we refer to the command |*File*|*Analyze*| of the Design Analyzer and not to the VSS tool *vhdlan* .

6.7.1 The necessary design steps for synthesis

In this section we briefly describe the different steps necessary to get from a VHDL description to a synthesized gate-level netlist. It is not meant to be a tutorial. Refer to [SynDCT ut] for a good introduction.

1. Define the target library for synthesis. This can be done with the command |*Setup*|*Defaults*| . We use in this course the IMS library *gfd200* .
2. Analyze your VHDL-descriptions with the command |*File*|*Analyze*| . It invokes the VSS *vhdlan* command with the synthesis policy checking option, i.e. the descriptions are checked for VHDL statements, which are not supported for synthesis by the VHDL Compiler .

This command generates the normal VSS-simulator design library files (*<name>.sim*) and design library files (*<name>.syn)* with synthesis related information.

Note: The execution of this command will produce an error when you try to analyze an entity description with generic parameters of the type *time* and warnings when the tool detects delayed signal assignments. You can ignore the warnings. To fix the error, you have to set the generic list within two special SYNOPSYS compiler directives as shown in figure 7-13. They will cause the synthesizer to ignore the VHDL statements in between whereas the VSS simulator treats the compiler directives like normal comments.

```
ENTITY inverteris
-- synopsys synthesis_off
GENERIC (tpd_inv : time)
-- synopsys synthesis_on

PORT (I : IN std_logic; Y : OUT std_logic;
END inverter;
```

Figure 7-13: Inclusion of compiler switches for synthesis

3. Invoke the VHDL Compiler with the command |*File*|*Elaborate*| to generate an intermediate design description (which is not mapped to any real library). If you want to load an already elaborated or synthesized design, you can use the |*File*|*Read*| command.

4. Check your design for unconnected ports, short circuits etc. with the command |*Analysis*|*Check Design*|.

5. Set design constraints and attributes. In detail you have to execute the following commands:

   – set the wire load model (|*Attributes*|*Operating Environment*|*Wire Load*|)

   – set the operating conditions, i.e. the delay model (we use *typ*ical delays) (|*Attributes*|*Operating Environment*|*Operating Conditions* |)

– set the drive strength for all inputs of your design (|*Attributes*|*Operating Environment*|*Drive Strength*|*).* Enter the command *"drive_of (gfd200/ni01d2/Z)"* , i.e. the driving strength of the output *Z* of the buffer cell *ni01d2* of the library *gfd200* , in the value field.

– set the output load for all outputs of your design (|*Attributes*|*Operating Environment*|*Load*|). Enter the command *"load_of (gfd200/ni01d2/I)"* , i.e. the capacitance load cause by the input *I* of the buffer cell *ni01d2* of the library *gfd200* , in the value field.

– set area constraints if required (the first synthesis run should be done without area constraints) with the command (|*Attributes*|*Optimization Constraints*|*Design Constraints* |)

– set timing constraints (this is very important as we will try to optimize our design for speed). If the design contains only combinational logic, you can set input-to-output delay time constraints with the command |*Attributes*|*Optimization Constraints*|*Timing Constraints*| . If your design contains sequential logic, you have to specify a clock signal for your clock input (|*Attributes*|*Clocks*|*Specify*|), input delays for the input signals (|*Attributes*|*Operating Environment*|*Input Delay*|), and output delays for the output signals (|*Attributes*|*Operating Environment*|*Output Delay*|).

– set optimization directives with the command |*Attributes*|*Optimization Directives*|*Design*| and constrain the fanout of all inputs to 4 (| *Attributes*|*Optimization Directives*|*Input Port*|).

6. Save your design with all the added information done in step 5, before compiling (|*File*|*Save As*|*).*

7. Start the design optimization and the mapping to our target library with the command |*Tools*|*Design Optimization*| .

Note: If your design contains subdesigns which are instantiated more than once (e.g. the structural model of the ALU contains two times the subdesign *word_latch*), you have to execute the command

|Edit|Uniquify|Hierarchy| to assign to each instantiated subdesign a unique name.

8. Analyze the synthesized result with the command *|Analyze|Report|* to check the achieved area and speed (delay) of the design, the net loads, the delay between selected points in your design, and much more. If the results are not satisfactory, change the timing or area constraints and carry out another synthesis run.

9. If your results are satisfactory, save your data.

Exercise 6.7.2: Packages *dlx_types* and dlx_instructions

A) Analyze the packages *dlx_types* and *dlx_instructions* to generate the synthesis design library files.

Exercise 6.7.3: Synthesis of the three different 32-bit latches

A) Analyze the entity and architecture descriptions of the three different register types which you developed in exercise 6.6.1.

B) Synthesize the circuits as described in section 6.7.1.

Exercise 6.7.4: Synthesis of the bus constant circuit

A) Analyze the entity and architecture descriptions of the bus constant circuit which you developed in exercise 6.6.2.

B) Synthesize the circuit as described in section 6.7.1. The delay between all inputs and outputs should be inferior to 5 ns.

Exercise 6.7.5: Synthesis of the 32x2:1 multiplexer

A) Analyze the entity and architecture descriptions of the 32x2:1 multiplexer which you developed in exercise 6.6.3.

B) Synthesize the circuit as described in section 6.7.1. The delay between all inputs and outputs should be inferior to 5 ns.

Exercise 6.7.6: Synthesis of the instruction register

A) Analyze the entity and architecture descriptions of the instruction register which you developed in exercise 6.6.4.

B) Synthesize the circuit as described in section 6.7.1. The delay between all inputs and outputs should be inferior to 8 ns.

Exercise 6.7.7: Synthesis of the memory data register

A) Analyze the entity and architecture descriptions of the memory data register which you developed in exercise 6.6.5.

B) Synthesize the circuit as described in section 6.7.1. The input delay between all inputs and the internal latch should be inferior to 8 ns.

Exercise 6.7.8: Synthesis of the interrupt control register

A) Analyze the entity and architecture descriptions of the interrupt control register which you developed in exercise 6.6.6.

B) Synthesize the circuit as described in section 6.7.1.

Exercise 6.7.9: Synthesis of the ALU

A) Analyze the entity and architecture descriptions of the ALU which you developed in exercise 6.6.7.

B) Set the following constraints:

- assign inverted clock of 50 ns period to the input latch_en,
- set no area constraint,
- assign input delays of 3 ns to the inputs *s1* and *s2*,
- assign output delays of 10 ns to all outputs
- set an timing constraint of 40 ns between the input *func* and all outputs.

C) Synthesize the circuit as described in section 6.7.1.

Note: On a SP ARC10 with 96 MB memory the optimization task needs about 5 hours to complete !

D) After analyzing the result (|*Analyze*|*Report*|), you will realize that they are not very satisfactory . This is caused by several reasons:

- the procedures for the vector addition and subtraction are based on the ripple carry principle. Ripple-carry adders are compact but the slowest alternative.
- the circuit of the ALU is already quite complex with its two 32-bit latches and all the logic. This has a direct influence to the achievable synthesis results.

E) To overcome these problems, you shall develop a structural architecture for the ALU. You can use the following predefined units: a 32-bit twos-complement adder (*word_adder* , use the carry-lookahead architecture *cla_h*), a 32-bit left-right shifter (*word_shifter*) for logic and arithmetic shift, a 32 bit logic block for AND,OR, and XOR (*word_alu_logic*), and a 32x4:1-multiplexer (*word_mux4*). The carry-lookahead adder is composed of some subdesigns as depicted in figure 7-3.

F) Analyze the corresponding files to your design database (see section 3.2.5.5 for the file names). Write a structural architecture for the ALU. What additional logic is necessary to implement the addition and subtraction operations with one adder? Write a configuration for the ALU and adapt the configuration for the datapath to include the ALU configuration instead of the ALU behavioural model.

G) Synthesize and optimize each component of the ALU separately following the procedure described in section 6.7.1. Afterwards, synthesize the structural model of the ALU using the same constraints as in part B) of this exercise. Analyze the results (|*Analyze*|*Report*|). The delay of the circuit should have decreased significantly .

Exercise 6.7.10: Synthesis of the register file

The register file is a highly regular circuit which is mostly composed of sequential cells (the flipflops for the latches). We can easily calculate their number to 32 registers x 32 bit/register x 1 flipflop/bit = 1024 flipflops. This high number of sequential elements will result in a very long run time for synthesis (Elaborate + Optimization takes approximately 12 hours on a SPARC 10 with 96 MB memory!) which would exceed the limits of this course. Hence, we will restrict ourselves to a register file with 9 registers. Furthermore, we will use a structural model for the register file which is composed of some subdesigns as depicted in figure 7-3.

A) Analyze the entity description of the register file which you developed in exercise 6.6.8.

B) Analyze the corresponding files to your design database (see section 3.2.5.5 for the file names). In the architecture of the register file, change the number of registers to 9 (registers *r0* to *r7* and the link register *r31*).

C) Synthesize and optimize each component of the register file separately following the procedure described in section 6.7.1. Afterwards, synthesize the structural model of the register file.

Exercise 6.7.11: Synthesis of the datapath

Now we have synthesized all subdesigns which are instantiated in the DLXS datapath (see figure 7-3).

A) Analyze the entity and architecture descriptions of the datapath which you developed in exercise 6.6.9.

B) Synthesize the datapath as described in section 6.7.1.

Exercise 6.7.12: Synthesis of the core

The core of the DLXS is composed of the controller and the datapath as depicted in figure 7-3.

A) The controller is already synthesized. You just have to load the database file *controller.db*.

B) Analyze the two files *core.vhd* and *core-structural.vhd*.

C) Synthesize the core described in section 6.7.1.

Exercise 6.7.13: Synthesis of the DLXS

The DLXS is composed of the core and the pad circuits as depicted in figure 7-3.

A) The pad circuits are already synthesized. You just have to load the database files *pad_in.db, pad_out.db,* and *pad_io.db*.

B) Load the database file for the core you synthesized in exercise 6.7.12.

C) Analyze the two files *dlx.vhd* and *dlx-structural*.vhd

D) Synthesize the DLXS described in section 6.7.1.

The DLXS processor is now completely synthesized. The design should have around 10,000 gates which is already a quite complex circuit. The DLXS with the full 32-register file would count approximately 15,000 gates. For comparison: state-of-the-art 64-bit processors of today (1995) count up to 2,500,000 gates when large onchip-RAMS as cache are included.

E) Analyze your results (|*Analyze*|*Report*|). Find out the number of gates in your design and the maximal clocking frequency which can be derived from the path with the maximal delay.

6.8 Gate-level simulation of the complete DLXS

Before you can simulate the synthesized gate-level model of the DLXS, you have to carry out the two following steps:

A) Save the synthesized DLXS database in VHDL format. This can be done with the design_analyzer command |*File*|*Save As*| with the file format set to *VHDL*.

For the gate-level simulation, we employ a special library (FTGS, Full Timing optimized Gate level Simulation) which contains the VHDL models with exact timing information of all basic logic cells in our synthesis library. Our complete DLXS model now comprises about 15,000 gate equivalents (one gate equivalent is the number of transistors of a CMOS NAND gate, i.e. four transistors). The simulation will be quite time expensive and we have to restrict ourselves in this design course to a short simulation run with a few test vectors just to give you an idea about it. Hence, we will only simulate the arithmetical instructions as contained in the assembler source file *arith1 .asm* .

B) You have to change the top-level configuration *system_config* to include this VHDL gate-level model in the test bench.

C) Include the mentioned user ROM in the test bench.

D) Set the clock frequency to 1MHz for the first simulation run by changing the generic parameters for the clock generator .

E) Reanalyze (with the *vhdlan /gvan* VSS analyze *r*) the top-level configuration.

F) Simulate the test bench. Change the clock frequency step by step to the maximum value which results from the delay time of the critical path computet in exercise 6.7.13. Find out the maximum clocking frequency for your DLXS model.

6.9 References

[SimVSS] SYNOPSYS: *VSS Expert Core Programs Manual* , V3.2, 1992 (Online documentation)

[SynDA] SYNOPSYS: *Design Analyzer Reference Manual* , V3.2, 1992 (Online documentation)

[SynDC] SYNOPSYS: *Design Compiler Reference Manual* , V3.2, 1992 (Online documentation)

[SynDCT ut] SYNOPSYS: *Design Compiler T utorial* , V3.2, 1992 (Online documentation)

[SynVHDL] SYNOPSYS: *VHDL Compiler Reference Manual* , V3.2, 1992 (Online documentation)

7
Concluding remarks and outlook

During this course, we focused to a modern VLSI design style. Therefore, we first introduced to the hardware description language VHDL by exercising some easy examples. Coming up from a given design specification, you learned how to develop a behavioural description which serves a as reference for a later architectural (register transfer) level model. Then we showed how to implement and later refine such a model by developing the detailed datapath.

We introduced to logic synthesis and exercised all necessary steps to prepare a VHDL description for synthesis. We let you synthesize your VHDL models and analyze your resulting netlists. Finally, you carried out the gate-level simulation of your final design. This was the end of the scope of this design course.

The steps which would be necessary to get the final layout, are briefly described below:

1. Conversion of the synthesized gate-level netlist into an exchange format, e.g. EDIF (electronic data interchange format), which can be read by the placement and routing CAD tool. This can be done with SYNOPSYS.
2. Our target library for synthesis was a proprietary gate-forest library from the IMS. Using this library, the next step is to generate the placement of the chip (the distribution of the library cells contained in our netlist over the chip area) and the routing (the interconnection of the library cells).
   Note: normally, one would use for the generation of the layout of very regular macro cells like adders or register files specialized tools (generators) or library macros which deliver more compact and faster blocks. However, this is not subject of this course.
3. From the generated layout data, the exact values for the net capacitances, wire resistances, and the resulting additional delays are calculated (this is called backannotation). This information is used

to simulate the design with much more accuracy .
A detailed timing analysis is necessary to detect timing violations like setup- or hold time violations of flipflops. Furthermore, the nets which capacitances exceed the maximal fanout limit of their driving cells, have to be redesigned, e.g. by including additional buffers.
The backannotated timing information can also be used as design attributed for a resynthesis of the circuit if necessary .

4. Step 3 has to be repeated until the final simulation shows that no more timing violations occur , no net capacitances are exceeded, and the overall timing and chip area requirements are met.

5. The next step would be a fault simulation which is not explained here in detail. This simulation is necessary to generate a set of test vectors which are to be used for the test of the produced chips. This test vector set must assure that nearly all possible production faults on every net can be detected (normally , 95% fault coverage is required in industry).

6. Afterwards, the work for the design engineer is done and he can go take some holidays or , what is more likely , start with the next project (time is money and money is rare). The layout data can be given to the production line to produce the required masks and finally , the circuits itself.

It is evident that we could not cover during this course all aspects of the VHDL language, all possibilities of synthesis based design, or all the features of the employed CAD tools. However , we hope to have given you a good overview of this very interesting field.

M.G.