Documentation of the LT16x32 System-on-Chip

Lasse Schnepel, Thomas Fehmel

2015

Contents

Chapter 1

Scope of Delivery

1.1 Products

Containing in the ESyLab-SoC are the the processor core *LT16x32*, a configurable Wishbone bus interconnect module, a minimal interrupt controller and peripheral devices.

An instruction memory is also supplied. It reads a file with machine code during elaboration, so the SoC can both be simulated and synthesized. The assembler to produce that machine code is also supplied in source.

1.2 HowTo: Build Products

The products are delivered as HDL source code. This allows adaptation to any platform, for which the needed tools are available (see sections below).

1.2.1 Processor

The processor is delivered as a set of VHDL files, including one top entity (processor in `processor.vhd`) which can be used in any VHDL synthesis tool. For simulation, the entity `core_tb` (in `core_tb.vhd`) can be used. Both architectures instantiate the core, a memory controller and a interrupt controller. The filename of the used program can be set in these files in the generic map of the memory controller instantiation (see Listing 1.1). Here, also the memory size can be determined (as number of stored words). For details on the memory controller see Section 6.4.

The processor is wrapped in another module, which implements the Wishbone bus-interface for the data memory interface of the processor.

1.2.2 Bus Interconnect

The bus interconnect system is configureable to any number of slave and master modules, the appropriate configuration values (see Listing 1.2) can be found in the file `lib/wishbone.vhd`.

Listing 1.1: Generic Map Extract of Memory Controller

```
memory_inst : component memory
   generic map(filename      => "../programs/test.ram",
               size         => 32,
               [...]
               )
```

All connections can be masked during its instantiation, using the slave and master mask

Listing 1.2: Wishbone interconnect configuration

```
...
constant NWBMST:    integer   :=4;
   --Number of Wishbone master connectors on the Interconnect
constant NWBSLV:    integer :=16;
   --Number of Wishbone slave connectors on the Interconnect
...
```

vectors (see Listing 1.3). They are declared in the top level architecture and are used during instantiation of the interconnection component. The component handles synchronous

Listing 1.3: Wishbone interconnect configuration

```
...
   constant slv_mask_vector : std_logic_vector(0 to NWBSLV-1) := b"1110_0000_0000_0
   constant mst_mask_vector : std_logic_vector(0 to NWBMST-1) := b"1000";
...
```

Wishbone cycle termination.

1.2.3 Assembler

For building the assembler the GNU tools `gcc`, `make`, `bison` and `flex` are needed. With these tools installed, the `make` script can be used to generate an executable called `asm`:

```
($SOCROOT)/assembler/$ make all
```

It is recommended to either copy or dynamically link the assembler executable in a directory in your `$PATH` or directly referenced by in your build tool chain.

Chapter 2

SoC Architecture

The supplied ESyLab-SoC plattform is implemented by the `top.vhd`, which can be found in the `<project root>/soc/top` directory. Its structure is illustrated by Figure 2.1.

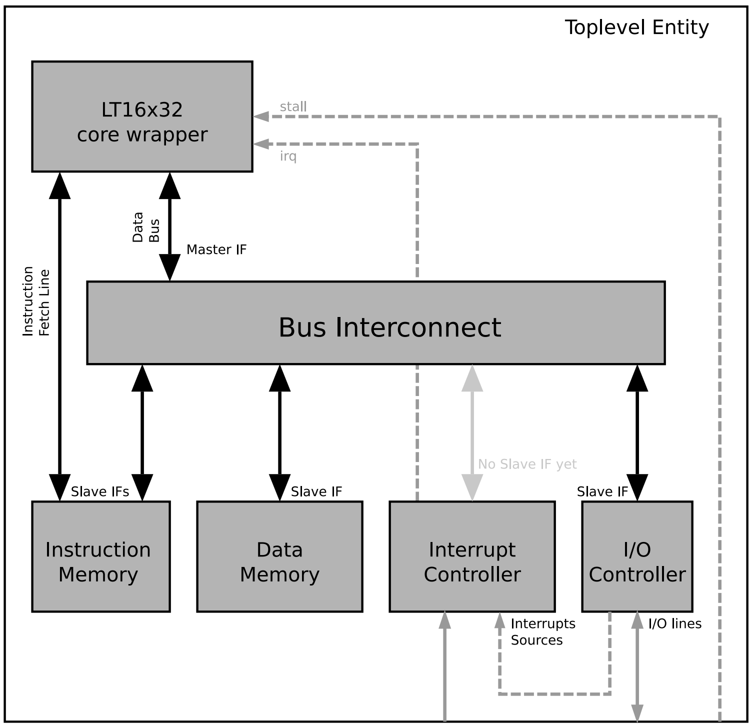

Figure 2.1: Block diagram of the ESyLab-SoC

The interface of the minimal top level entity of the *LT16x32* system is shown in Listing 2.1.

The inputs `clk`, which denote the clock input, and `rst`, which is the active high reset, are system-wide signals. The minimal system features only one output signal, which is `led`. This signal should be connected to a set of leds and is connected to a generic

Listing 2.1: Top Level Entity

```
entity lt16soc_top is
generic(
  programfilename : string := "programs/program.ram"
);
port(
  clk       : in  std_logic;
  rst       : in std_logic;

  led       : out std_logic_vector(7 downto 0)
);
end entity lt16soc_top;
```

I/O-controller on the bus.

The individual components of the minimal system are described in the following sections.

2.1 Interconnect

The system utilizes a synchronous Wishbone bus. To ease integration of new bus components, a Wishbone bus interconnect component is provided. The interconnect module currently implements only the synchronous Whisbone protocol. Its interface is given in Listing 2.2.

Listing 2.2: Top Level Entity

```
entity wb_intercon is
generic(
  slv_mask_vector : std_logic_vector(0 to NWBSLV-1) := 0;
  mst_mask_vector : std_logic_vector(0 to NWBMST-1) := 0
);
port(
  clk  : in  std_logic;
  rst  : in  std_logic;
  msti : out wb_mst_in_vector;
  msto : in  wb_mst_out_vector;
  slvi : out wb_slv_in_vector;
  slvo : in  wb_slv_out_vector
);
```

The system is configured to a certain maximum number of slave and master modules as `NWBSLV` and `NWBMST` respectively. The configuration can be found in `config.vhd`.

These constants are used to define a the length of two generic parameters of the module, the mask vectors `slv_mask_vector` and `mst_mask_vector`, and also define how many elements the input and output arrays have. As for the mask vectors, as indicated in the `wishbone.vhd` file. The mask vectors are used to indicate to the interconnect if a module should be connected to the input/output port indicated by the bit position in the mask vector. A '1' at a position in the mask vector indicates that at the respective position in the the input/output arrays, a module should be connected. A '0' at a position in the mask vector will facilitate that the control logic for that position in the respective input/output array is omitted.

The inputs `clk` and `rst` are currently not used, but might be in the future when synchronous wishbone transfer is implemented.

The wishbone bus signals are separated in arrays of master and slave signals. Array of master signals is in the output `msti`, which are the inputs to the connected masters, and the input `msto`, which are the outputs of the connected master. Analogously, `slvi` and `slvo` follow the same principle for the connected slave modules.

2.1.1 Integration of new components

To instantiate a new component in the system, a few steps must be undertaken.

The component declaration should not be put in the top level module, but in the appropriate package. This makes the top level module more readable.

The component instantiation in the top level module must be connected to the bus interconnect system. To do this, the port map must connect its inputs and outputs of its bus interface to a free element of the `slvi` and `slvo` array signals in the top level architecture. Any element in that vector must obviously only be assigned once.

To select to which of those signals you should connect, consult the file `soc/lib/config.vhd`. A number of of slave index constants like shown in Listing 2.3 are defined there. Add your own by adding one to the last defined constant, then use the defined constant to select the element of the bus slave signal arrays, as illustrated in Listing. 2.4.

Listing 2.3: Slave bus indexes

```
-- >> Slave index  <<
constant CFG_MEM  : integer := 0;
constant CFG_LED  : integer := CFG_MEM+1;
constant CFG_DMEM  : integer := CFG_LED+1;
...
```

Listing 2.4: Slave bus connection

```
... port map( ...
wslvi       => slvi(CFG_FOO),
wslvo       => slvo(CFG_FOO)
...
```

The slave masking vector is a constant which configures the interconnect system to only consider certain slave bus interface elements. A slave mask vector `slv_mask_vector` is defined in the top level architecture, see Listing 2.5. To enable any connection on the interconnect, the bit in the slave mask vector must be set to '1' at the same position as the targeted elements in the slave connector arrays.

Setting any bit in the slave mask vector to '0' will disable the corresponding connection to the interconnection in the way that the logic to evaluate the inputs is not generated.

Listing 2.5: Slave bus indexes

```
architecture RTL of lt16soc_top is
...
constant slv_mask_vector : std_logic_vector(0 to NWBSLV-1) := b"1110_0000_0000_0000
...
```

2.2 Processor Core Wrapper

The processor core has a data and instruction memory interface, as per standard Harvard architecture. The instruction memory interface is unchanged, but the data memory interface is wrapped in a piece of control logic translating the processor interface to wishbone signals and takes care of timing.

2.3 Instruction memory

The instruction memory component is configurable to load a file into its memory during elaboration. The loaded file needs to contain legal values of the type std_logic_vector with a length of 32 characters per line. If the content of the file is illegal, an error in the elaboration tool will likely occur.

It has two interfaces, one for instruction read access by the processor, one for a read and write access by the Wishbone bus. The Wishbone interface takes priority over the instruction read. The whole range of the memory can be written, so care should be exercised when writing to the instruction memory, so code or ’constants’ are not unintentionally overwritten.

Chapter 3

Processor Architecture

3.1 Block Diagram

A simplified block diagram of the *LT16x32* can be found in Figure 3.1. A diagram of the datapath can be found in Figure 3.2

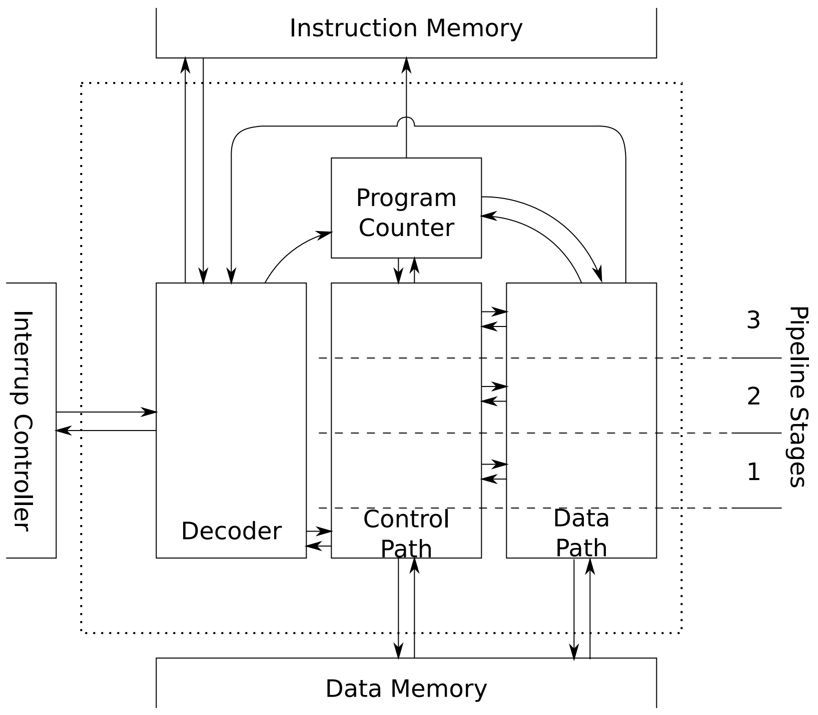

Figure 3.1: Simplified Block diagram of core architecture with surrounding entities

3.2 Configuration

Easy ways to configure the *LT16x32* can be found in package `lt16x32_internal` (see Listing 3.1) and in package `lt16x32_global` (see Listing 3.2).

Listing 3.1: Possible Configurations in internal package

```
-- execute branch delay slot. if set to true,
-- the first operation behind any type of branch
-- is executed. if set to false, stalls are inserted
constant execute_branch_delay_slot : boolean := TRUE;
-- register width
constant reg_width                 : integer := 32;
-- pc width (should be smaller or equal to register width)
constant pc_width                  : integer := 32;
```

Listing 3.2: Possible Configurations in global package

```
-- width of the vector holding the interrupt
-- number, maximum 7 due to processor architecture
constant irq_num_width             : integer := 4;
-- width of the vector holding the interrupt
-- priority, maximum 6 due to processor architecture
constant irq_prio_width            : integer := 4;
```

Currently, three values can be configured:

Branch delay slot If the constant `execute_branch_delay_slot` is set to `true`, the instruction behind a branch instruction is always executed. If the constant is set to `false`, a `nop` is inserted if a branch is performed. By default, the branch delay slot is always executed.

Pay attention that even if the branch is not yet taken in the branch delay slot, the program counter may contain the new address. Hence, instructions using the program counter should not be placed in the branch delay slot. This also means, that storing the current address for function returns can not be done in the branch delay slot. Instead, use the `call` instruction for this purpose.

Register width With the constant `reg_width` the width of the internal registers can be set. Valid values are 8, 16 and 32, by default 32bits are used.

PC width With the constant `pc_width` the width of the program counter can be set. Valid values are 8, 16 and 32, it must be smaller or equal to the register width. By default, 32bits are used.

Interrupt Number Width This constant holds the width of the interrupt number, values between 1 and 7 are valid. This determines the number of possible interrupts. By default, 4bits are used, allowing $2^4 = 16$ interrupts.

$$\text{Possible Interrupts} = 2^{(\text{Width})}$$

Interrupt Priority Width This constant determines the width of the runtime/interrupt priority, values between 1 and 6 are valid. By default, 16 levels of priority are allowed. One additional bit is supported by the use of non-maskable interrupts (NMI, see Section 5.2).

3.3 Pipeline Stages

As shown in Figure 3.2, the processor has three pipeline stages. Due to this scheme, there is a delay between reading the instruction and changing data in the (register) memory of three clock cycles.

Stage 1: Decode/Setup In the first stage, the instruction is fetched from the instruction memory and decoded. Also, the needed register contents are loaded from the register file.

Stage 2: Load and Execute In this stage, the ALU is performing the operation and data is read from the data memory.

Stage 3: Writeback Data is written back to the register file and written to the data memory. Possible sources for this data are the data read from the memory in the last clock cycle or the contents of register b.

3.4 Registers

The *LT16x32* has 16 registers, 14 of which can be used as general purpose registers. The other four have special functions, VHDL defines are given in Listing 3.3 and can be found in the `lt16x32_internal` package.

3.4.1 General Purpose Registers

Registers `r0` to `r11` can be used without special consideration as general purpose registers. Registers `r12` (stack pointer) and `r13` (link register) can also be used as general purpose register but are additionally altered with special instructions.

Listing 3.3: Constant Defines for the Special Purpose Registers

```
-- register number for stack pointer
constant sp_num : reg_number := to_reg_number(12);
-- register number for link register
constant lr_num : reg_number := to_reg_number(13);
-- register number for status register
constant sr_num : reg_number := to_reg_number(14);
-- register number for pc register
constant pc_num : reg_number := to_reg_number(15);
```

3.4.2 Stack Pointer

The stack pointer (`r12`) is used when jumping into or out of interrupt handlers (i.e. external interrupts or `trap` instructions and `reti`). It must be set to the correct address of the stack, before any interrupt is allowed to occur. In the default configuration, stack grows downwards. The stack pointer should be word-aligned at all times.

The interpretation of the stack pointer is always 'The current stack pointer points to the topmost element on the stack'. When interfacing the stack pointer manually, always adhere to the following rules:

- When writing to the stack (push), always decrement the stack before writing your element.
- When reading from the stack (pop), read your element, then increment the stack pointer.

Only when adhering to these rules the integrity of the stack can be ensured.

The initial stack pointer position does not need to be a memory address, but the word address below it needs to be valid.

3.4.3 Link Register

The link register (`r13`) is set by the `call` instruction. This allows for flexible function returns using the pseudo instruction `ret` (a branch to the link register).

3.4.4 Status Register r14

The status register is a special 32bit register. The register contents are displayed in table 3.1, reserved bits always read zero and should be written to as zeroes for future compatibility.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved								priority						T	O

Table 3.1: Status Register Contents

3.4.4.1 Runtime Priority - SR[7-2]

Bits seven to two define the current runtime priority, larger numbers mean higher priority. The priority is left aligned. With a runtime priority width configured to four bits, the priority is stored in bits seven to four and the other bits read zero. Writes to not used priority bits are discarded.

3.4.4.2 Truthflag - SR[1]

Bit one is the truth flag, which can be set/reset by compare instructions and are used for conditional branches and calls. Also, the truth flag can be set manually by writing to the status register.

3.4.4.3 Overflowflag - SR[0]

Bit zero is the overflow flag, which is set by arithmetic instructions, such as addition and subtraction.

In additions an overflow has occured, if both operands are the same sign and the result's sign is different (i.e. $5 + 6 < 0$ or $(-5) + (-7) > 0$, listing 3.4). In subtractions an overflow occured, if both operands are from opposing sign and the minuend's sign differ from the result's (i.e. $(-8) - 5 > 0$ or $5 - (-4) < 0$, listing 3.5). This follows normal rules for signed calculations, which are as well used in other processor designs.

Listing 3.4: Overflow Generation for Addition

```
c_out <= (in_a(reg_width-1)      AND in_b(reg_width-1)
                                 AND not result(reg_width-1))

      OR (not in_a(reg_width-1)  AND not in_b(reg_width-1)
                                 AND result(reg_width-1));
```

Listing 3.5: Overflow Generation for Subtraction

```
c_out <= (in_a(reg_width-1)      AND not in_b(reg_width-1)
                                 AND not result(reg_width-1))

      OR (not in_a(reg_width-1) AND in_b(reg_width-1)
                                 AND result(reg_width-1));
```

3.4.5 Program Counter

Register `r15` is reserved for the program counter and is read only from the user interface. Write accesses are ignored. It may be altered with special instructions such as branch and call and in interrupt request situations by the processor itself.

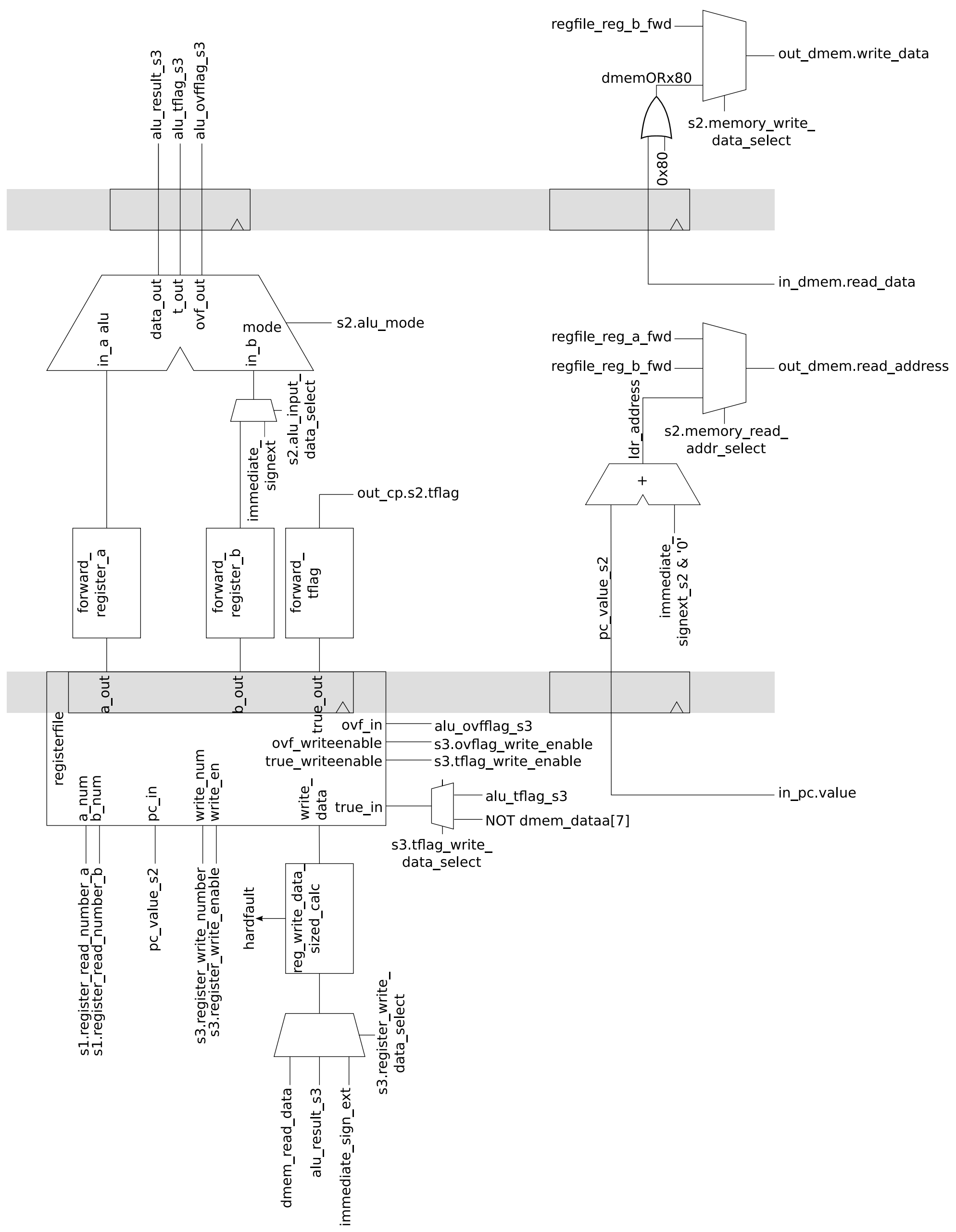

Figure 3.2: Block diagram of the datapath
Pipeline registers in gray, for reasons of clarity, not all signals are included.

Chapter 4

Assembler and Instructions

4.1 Processor Instructions

The *LT16x32* features a RISC instruction set with 16bit instructions and is prepared to be extended with 32bit and allows multicycle instructions. In 16bit instructions, the first nibble (4bits) of the instruction code is the opcode determining the operation. The opcode 1111 is reserved for 32bit instructions which are always are word aligned.

4.1.1 Instruction Format

Several different instruction formats are used in the *LT16x32*. As at the moment, no 32bit instruction is implemented, this section describes instruction formats for 16bit instructions only. Each halfword is split up into four nibbles (each of a size of 4bits), whose meaning can be seen in Table 4.1.

Table 4.1: 16bit Instruction Formats

Format	15-12	11-8	7-4	3-0	Example Uses
Three Registers	opcode	Rd	Ra	Rb	Calculations
4bit Immediate	opcode	Rd	Ra	Imm(3-0)	Shift
8bit Immediate	opcode	Rd	Imm(7-4)	Imm(3-0)	Add Immediate, Load PC-Relative
Mode with Register	opcode	Mode	Ra	Rb	Compare, Load/Store
Mode with Immediate	opcode	Mode	Imm(7-4)	Imm(3-0)	Branch/Call to Immediate

4.1.2 List of available Instructions

The following instructions are supported:

Arithmetic Operations

- Signed Addition
- Signed Subtraction
- Signed Addition with Immediate

Bitwise/Logic Operations

- Bitwise AND
- Bitwise OR
- Bitwise XOR
- Logic Shift Left
- Logic Shift Right
- Compare

Memory Operations

- Load PC Relative
- Load Data from Pointer
- Store Data to Pointer

Branch/Call/Trap Operations

- Branch to Offset
- Branch to Register
- Call to Offset
- Call to Register
- Trap
- Return from Interrupt
- Branch to Table

Miscellaneous Operations

- Test and Set

4.1.2.1 Signed Addition

Opcode: `0011 dddd aaaa bbbb`

Assembler: `add rd, ra, rb`

Operation: Registers ra and rb are treated as two signed numbers and are added. The result is stored in rd.

C-Equivalent: `rd = ra + rb;`

Status Register: The overflow flag is updated.

4.1.2.2 Signed Subtraction

Opcode: `0001 dddd aaaa bbbb`

Assembler: `add rd, ra, rb`

Operation: Registers ra and rb are treated as two signed numbers and are subtracted. The result is stored in rd.

C-Equivalent: `rd = ra - rb;`

Status Register: The overflow flag is updated.

4.1.2.3 Bitwise AND

Opcode: `0010 dddd aaaa bbbb`

Assembler: `and rd, ra, rb`

Operation: Registers ra and rb are treated as bit masks and are anded with each other. The result is stored in rd.

C-Equivalent: `rd = ra & rb;`

Status Register: No flag is updated.

4.1.2.4 Bitwise OR

Opcode: `0000 dddd aaaa bbbb`

Assembler: `or rd, ra, rb`

Operation: Registers ra and rb are treated as bit masks and are ored with each other. The result is stored in rd.

C-Equivalent: `rd = ra | rb;`

Status Register: No flag is updated.

4.1.2.5 Bitwise XOR

Opcode: `0100 dddd aaaa bbbb`

Assembler: `xor rd, ra, rb`

Operation: Registers ra and rb are treated as bit masks and are xored with each other. The result is stored in rd.

C-Equivalent: `rd = ra ^ rb;`
Status Register: No flag is updated.

4.1.2.6 Logic Shift Left

Opcode: `0101 dddd aaaa bbbb`
Assembler: `lsh rd, ra, imm`
Operation: Register ra is treated as bit masks and shifted to the left by imm bits, ra is filled with zeroes. imm is treated as unsigned number. The result is stored in rd. Internally, imm is incremented by one, which is compensated for in the assembler.
C-Equivalent: `rd = ra << imm;`
Status Register: No flag is updated.

4.1.2.7 Logic Shift Right

Opcode: `0110 dddd aaaa bbbb`
Assembler: `rsh rd, ra, imm`
Operation: Register ra is treated as bit masks and shifted to the right by imm bits, ra is filled with zeroes. imm is treated as unsigned number. The result is stored in rd. Internally, imm is incremented by one, which is compensated for in the assembler.
C-Equivalent: `rd = ra >> imm;`
Status Register: No flag is updated.

4.1.2.8 Signed Addition with Immediate

Opcode: `0111 dddd iiii iiii`
Assembler: `addi rd, imm`
Operation: Register rd and imm are treated as signed numbers and are added. The result is stored in rd.
C-Equivalent: `rd = rd + imm;`
Status Register: The overflow flag is updated.

4.1.2.9 Compare

Opcode: `1000 mmmm aaaa bbbb`
Assembler: `cmp mode ra, rb`
Operation: Registers ra and rb are treated as signed numbers and are compared. If the condition, given by mode is true, the truth-flag is set, otherwise reset. Allowed modes are

0000, eq equal
1000, neq not equal

0010, gg greater
0001, ge greater or equal
1001, ll less
1010, le less or equal

C-Equivalent: `(ra == rb) ? (T=1) : (T=0);`
`// where == is interchangable by mode`

Status Register: The truth flag is updated.

4.1.2.10 Load PC Relative

Opcode: `1010 dddd iiii iiii`

Assembler: `ldr rd, imm`

Operation: Register rd is loaded with memory data from address (PC+(imm<<1)), where imm is treated as signed number and is left shifted by one bit.

C-Equivalent: `rd = *(PC + (imm<<1)`

Status Register: No flag is updated.

4.1.2.11 Load Data from Pointer

Opcode: `1011 0mmm aaaa bbbb`

Assembler: `ld08 ra, rb`
`ld16 ra, rb`
`ld32 ra, rb`

Operation: Loads data from pointer. Register rb's content is used as absolute address, the loaded data is written to register ra. Following modes are supported:

000 Load byte
001 Load halfword
010 Load word

C-Equivalent: `ra = *rb`

Status Register: No flag is updated

4.1.2.12 Store Data to Pointer

Opcode: `1011 1mmm aaaa bbbb`

Assembler: `st08 ra, rb`
`st16 ra, rb`
`st32 ra, rb`

Operation: Stores data to pointer. Register ra's content is used as absolute address to store the content of register rb. Following modes are supported:

000 Store byte

001 Store halfword
010 Store word

As the data is valid only after two clock cycles, the user must ensure that the same address is not used in read transactions in the following instruction (such as ld, ldr, tst).

C-Equivalent: `*ra = rb`

Status Register: No flag is updated

4.1.2.13 Branch to Offset

Opcode: `1100 010c iiii iiii`

Assembler: `br imm`
`br always/true imm`

Operation: Sets the program counter to PC+imm, where imm is treated as signed number. For details about the branch delay slot see section 3.2. If c==1, the branch is conditional and performed only, if the truth flag is set.

C-Equivalent: `if ((c==0) || (T==1)) goto (PC+imm<<1);`

Status Register: No flag is updated

4.1.2.14 Branch to Register

Opcode: `1100 011c aaaa xxxx`

Assembler: `br ra`
`br always/true ra`

Operation: Sets the program counter to register a. For details about the branch delay slot see section 3.2. If c==1, the branch is conditional and performed only, if the truth flag is set.

C-Equivalent: `if ((c==0) || (T==1)) goto (ra<<1);`

Status Register: No flag is updated

4.1.2.15 Call to Offset

Opcode: `1100 100c iiii iiii`

Assembler: `call imm`
`call always/true imm`

Operation: Sets the program counter to PC+imm, where imm is treated as signed number, and stores the current program counter in the link register. If c==1, the call is conditional and performed only, if the truth flag is set. This is a multicycle operation which consumes two clock cycles.

C-Equivalent: `if ((c==0) || (T==1)) {LR=PC; goto (PC+imm);}`

Status Register: No flag is updated

4.1.2.16 Call to Register

Opcode: `1100 101c aaaa xxxx`

Assembler: `call ra`
`call always/true ra`

Operation: Sets the program counter to register a and stores the current program counter in the link register. If c==1, the call is conditional and performed only, if the truth flag is set. This is a multicycle operation which consumes two clock cycles.

C-Equivalent: `if ((c==0) || (T==1)) {LR=PC; goto (ra<<1);}`

Status Register: No flag is updated

4.1.2.17 Trap

Opcode: `1100 11xx iiii iiii`

Assembler: `trap imm`

Operation: Request an interrupt of number imm. Number of bits for imm can be set in processor configuration, the number must be right aligned.

C-Equivalent: `no equivalent`

Status Register: No flag is updated

4.1.2.18 Return from Interrupt

Opcode: `1100 000x xxxx xxxx`

Assembler: `reti`

Operation: Returns from Interrupt, by popping the link and status register from the stack and branching to the current link register contents. This is a multicycle operation which consumes five clock cycles.

C-Equivalent: `no equivalent`

Status Register: No flag is updated.

4.1.2.19 Branch to Table

Opcode: `1100 001c aaaa xxxx`

Assembler: `brt ra`

Operation: Increments the program counter by register a. This allows for easy creation of branch tables. For details about the branch delay slot see section 3.2. If c==1, the branch is conditional and only performed if the truth flag is set.

C-Equivalent: `PC = PC + (ra<<1)`

Status Register: No flag is updated.

4.1.2.20 Test and Set

Opcode: `1001 dddd aaaa xxxx`
Assembler: `tst rd, ra`
Operation: Test and set can be used to implement mutexes. The lower seven bits of the byte can be used to implement semaphores.
C-Equivalent: `temp=`$_{\texttt{byte}}$`*ra; rd = temp;`
`T = (temp & 0x80==0); temp=temp | 0x80; *ra=`$_{\texttt{byte}}$`temp;`
Status Register: The T-flag is set if the mutex is free and reset if the mutex is already reserved.

4.2 Assembler

The supplied assembler is a two-pass assembler (see thesis, Section 2.6) based on `flex` and `bison` which are used to parse the input file.

4.2.1 HowTo: Assemble Input Files

A prepared assembler file `input.prog` (see examples in `/source/programs/`) can easily assembled into a formatted file `output.ram` supported by the *LT16x32* by calling the assembler `asm`:

```
asm input.prog -o output.ram
```

4.2.2 Allowed Input

The input to the assembler is case sensitive and all mnemonics have to be lower case. A typical line follows Listing 4.1, where all parts are optional (a mnemonic is needed if parameters are supplied though).

Listing 4.1: General Inputline

```
label: mnemonic mode par1, par2, par3 // comment
```

Parameters can either be registers names r0 to r15, SP, LR, SR or PC, or can be immediate values in various number formats (decimal, binary or hexadecimal numbers), for prefixes see Table 4.2. Binary and hexadecimal numbers are not sign extended and always treated as unsigned values. Also, absolute or relative references to labels are allowed as parameters, see Section 4.2.5 for details.

Number and type of parameters vary for each instruction, allowed combinations are listed in Section 4.1.2.

Numberformat	Prefix	Example
Decimal	None	-45
Binary	0b	0b11011
Hexadecimal	0x	0xFE

Table 4.2: Number Format Prefixes

4.2.3 Pseudo Instructions

Several instructions are supported which are not directly processed by the *LT16x32* but which are mapped to other instructions. For details on the instructions used, see Section 4.1.2.

No Operation (nop) No operation is performed, wait one clock cycle. Mapped to `or r0, r0, r0`.

Move Register (mov rd, ra) The content of `ra` is copied to `rd`. Mapped to `or rd, ra, ra`.

Return (ret) Returns to callee after a call instruction. Mapped to `br always lr`.

Clear Register (clr rd) Resets `rd` to zero. Mapped to `xor rd, rd, rd`.

4.2.4 Directives

The assembler supports several directives which can be included in the input code, see Listing 4.2.

.word stores the following parameter directly into the memory.

.address inserts nops until the address given as parameter is reached.

.align if needed, a 16bit nop is inserted to align the following instruction

4.2.5 Labels

Labels can be set in the code directly, as seen in Listing 4.1. They can be reffered to in absolute or relative manner (see Listing 4.3). Label names can include upper and lower case letters, numbers and underscores. While absolute referencing puts the whole address into the output, relative referencing adds only the difference to the address of instruction which is referencing to the label.

Listing 4.2: Example for the use of directives

```
// load variables from constant pointer
           ldr     r0,     >variableA
           ldr     r1,     >variableB

// store variableA
           .align       // fix eventual non-word-alignment
variableA: .word 0x1234

// store variableB
           .address 0x100 // store variableB at address 0x100
variableB: .word    0xABCD
```

Listing 4.3: Label References

```
label:  br   >label // relative reference
       .word =label // absolute reference
```

4.2.6 Assembler Options

Several command line options are available:

-o filename Determines the output file.

-m filename The assembler outputs a map file, containing all labels and their addresses.

-v or --verbose Outputs all intermediate information.

--autoalign Automatically aligns 32bit instruction and immediate values

--fillbds Fills branch delay slots with nop instruction (i.e. inserts nop after all branches)

--continue-on-error Continues parsing even with errors.

4.2.7 Output

The standard output format is a one-word-per-line bitstring representation of the instruction memory, which can directly be read by the testbench provided and used as input for memory synthesis.

4.3 HowTo: Add more instructions

4.3.1 Processor Side

1. Add opcode constant define to `lt16x32_internal` package.
2. Add case to 16bit decoder and set control signals that differ from default settings.

4.3.2 Assembler Side

1. Add enumerate item to op_t in global.h
2. Add pattern to flex code in asm.l
3. Add handling to handle_xy.c according to wished syntax style.
4. Recompile assembler by running the make script.

4.3.3 32bit Extension

If 32bit instruction should be implemented, follow the structure of the 16bit decoder to implement the 32bit decoder in `decoder_32bit.vhd`.

4.3.4 Multicycle Instructions

To implement multicycle operations, changes need to be implemented in `decoder_fsm.vhd`. Use the implementation of *reti* as an example.

Chapter 5

Interrupts

5.1 Interrupt Controller

The interrupt controller features a configurable number of interrupt lines with a configurable amount of priorities. The configuration follows that of the *LT16x32* (see Section 3.2). These intterupts lines need to be asserted for one clock cycle to trigger an interrupt request to the processor core. Additionally, trap requests from the core are handled (as described in 5.3.2).

5.2 Priority, NMI

The processor features multiple levels of runtime priority (configurable, see Section 3.2) where a greater number means higher priority. Additionally, non maskable interrupts (NMI) are supported which are always executed regardless the current processor runtime priority.

The processor starts with the highest possible runtime priority by default to disable interrupts at startup (with the exception of NMI).

This is needed as the stack pointer needs to be set to a valid address before any interrupt (including NMI) can be executed.

5.3 Interface and Timing Diagram

Both ports `in_irq` and `out_irq` (for type definitions see Listing 5.1) should be connected to a fitting interrupt controller (for example the implementation described in Section 5.1).

5.3.1 Interrupt Request

The external world (i.e. the interrupt controller) can request an interrupt by writing the interrupt number, its priority to `in_irq` and setting the `nmi` bit accordingly. After these are

Listing 5.1: Type definitions for Interrupt Controller Ports

```
-- collection of all signals from the
-- interrupt controller to the core
type irq_core is record
   -- interrupt number of requested interrupt
   num      : unsigned(irq_num_width - 1 downto 0);
   -- priority of requested interrupt
   -- (higher number means higher priority)
   priority : unsigned(irq_prio_width - 1 downto 0);
   -- request signal, active high
   req      : std_logic;
   -- non maskable interrupt flag, active high
   nmi      : std_logic;
end record;

-- collection of all signals from the core
-- to the interrupt controller
type core_irq is record
   -- interrupt acknowledge
   -- high if requested interrupt is processed
   ack      : std_logic;
   -- number of interrupt requested by
   -- internal trap instruction
   trap_num : unsigned(irq_num_width - 1 downto 0);
   -- request signal for internal trap, active high
   trap_req : std_logic;
end record;
```

set, `req` can be asserted and must be held high until the core acknowledges the interrupt request by asserting `ack` for one clock cycle (see Figure 5.1).

5.3.2 Trap Implementation

When executing a trap instruction, an interrupt line to the interrupt controller is asserted for one clock cycle with the interrupt number asserted as well, see Figure 5.2. Note, that an arbitrary time (and number of instructions) may take place between a trap and the interrupt handler execution.

5.3.3 Processor Interrupt Behavior

When an interrupt is accepted by the processor, it saves the current PC and status register on the stack. Then it replaces the runtime priority in the status register with interrupt's priority and replaces the PC with the interupt vector number.

The *reti* instruction reverses this and resturn to the previous executed program. Note

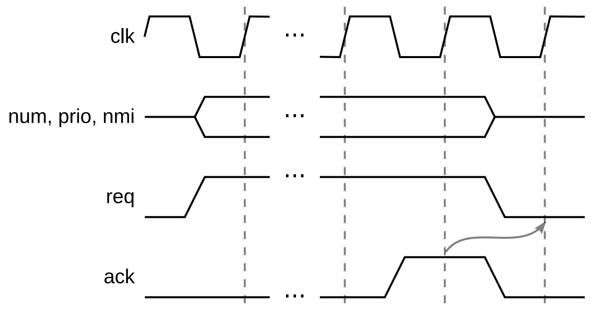

Figure 5.1: Signal Pattern for Interrupt Requests

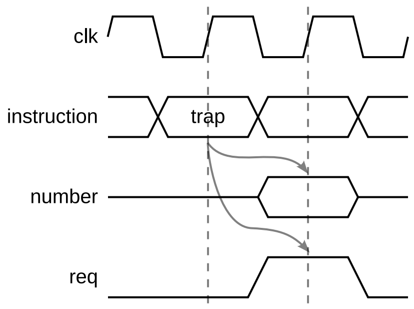

Figure 5.2: Signal Pattern for Trap Instruction

that the processor by itself does not save any registers apart from status register and program counter. Any other register that needs to be preserved needs to be saved manually.

Chapter 6

Memory

The *LT16x32* is connected to the memory through a harvard architecture style interface. While the interface to the instruction memory is read only, the data memory interface must handle read and write transactions simultaneously. Externally, both instruction and data memory can be mapped to a Von-Neumann architecture. The implemented Harvard architecture is more general and allows for two caches to be implemented.

6.1 Endianess and Memory Width

The memory width is defined in the `lt16x32_internal` package. Currently, memory widths of 32bits are supported only. Of course, this memory width can be mapped to any other memory width by a memory controller. The memory organizes data in big endian format. This can be seen (and changed) in the memory controller, Listing 6.1. The same is valid for halfword-accesses.

Listing 6.1: Byte-Order in Memory

```
case byteaddress is
when "00" =>
   dmem_data(7 downto 0) <= word(7 downto 0);
when "01" =>
   dmem_data(7 downto 0) <= word(15 downto 8);
when "10" =>
   dmem_data(7 downto 0) <= word(23 downto 16);
when "11" =>
   dmem_data(7 downto 0) <= word(31 downto 24);
```

6.2 Interface and Timing Description

6.2.1 Instruction Memory Interface

The *LT16x32* is connected to the instruction memory through two ports, as seen in Listing 6.2. Their datatype definitions can be found in the `lt16x32_global` package.

Listing 6.2: Instruction Memory Interface

```
entity core is
port(
   [...]
   -- signals from instruction memory
   in_imem    : in  imem_core;
   -- signals to instruction memory
   out_imem   : out core_imem;
   [...]
);
end entity core;
```

6.2.1.1 Read Access

As all clocked signals in this design, the address and data signals to the instruction memory must be read on each rising clock edge. Data must be provided if enable (`en`) is asserted and the memory content is read on the next rising clock edge. This single data rate scheme allows for double data rate memory access to allow for pseudo-simultaneous data and instruction memory access. A standard signal pattern can found in Figure 6.1.

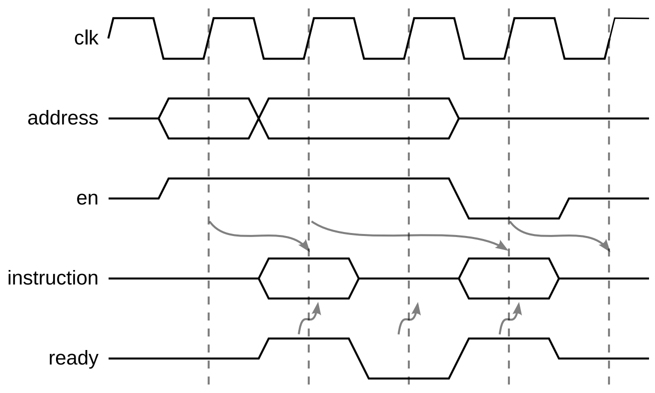

Figure 6.1: Signal Pattern for the Instruction Memory Interface

6.2.2 Data Memory Interface

The *LT16x32* is connected to the data memory through two ports, as seen in Listing 6.3, their datatype definitions can be found in the `lt16x32_global` package. The data memory interface must be able to handle simultaneous read and write accesses, as these are performed in different pipeline stages and can overlap. This is good for memory transaction performance but introduces a high memory load.

Listing 6.3: Data Memory Interface

```
entity core is
port(
   [...]
   -- signals from data memory
   in_dmem    : in   dmem_core;
   -- signals to data memory
   out_dmem   : out  core_dmem;
   [...]
);
end entity core;
```

6.2.2.1 Read Access

The read access to the data memory is similar to the instruction memory interface and a standard signal pattern is shown in Figure 6.1.

6.2.2.2 Write Access

In a write access, data and address are supplied at the same time and should be copied to the memory at the rising clock edge if enable is active, see Figure 6.2.

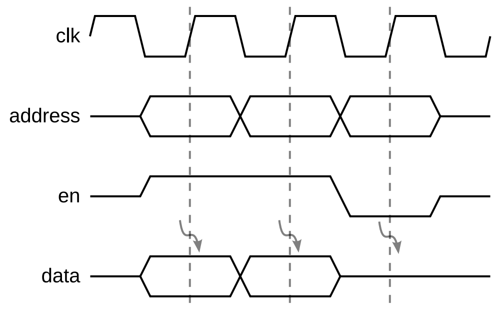

Figure 6.2: Signal Pattern for Data Memory Write

6.2.3 Memory Bandwidth Solutions

As mentioned before, the very liberal memory interface demands high memory bandwidth. In a worst-case scenario three memory accesses are needed per clockcycle (instruction read, data read and data write).

6.2.3.1 Instruction Alignment

Fortunately this worst-case scenario can be avoided by clever instruction alignment. As standard instructions are 16bit wide and the instruction memory is organized in 32bit words the instruction memory must be read only once every two clock cycles[1]. If memory access instructions are now aligned in such a way, that read/write actions are performed when the instruction memory is idle, a simple single data rate memory is sufficient. An exemplatory signal pattern is shown in Figure 6.3. Note, that the instruction memory enable must be asserted the whole time, but the memory controller does not need to read the actual memory, as the address is changed only every second clock cycle.

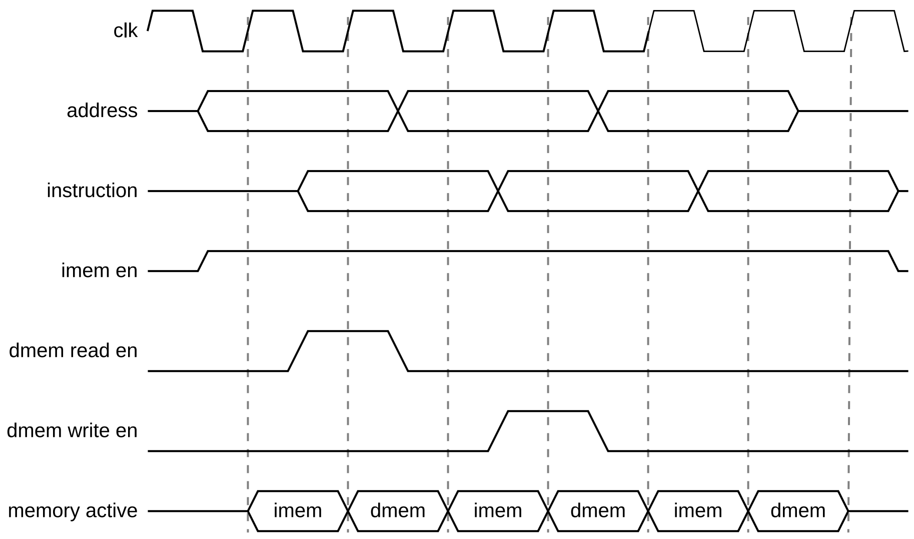

Figure 6.3: Signal Pattern for Memory Interlacing

6.2.3.2 Processor Stalling

If instruction alignment is too cumbersome or the memory is too slow for any other reason, it can deassert the ready signal (`ready`) to the processor (per interface). If `ready` is deasserted, the processor is stalled and waits for `ready` to become asserted again.

[1]Branching may introduce additional read operations.

6.3 Memory Mapped I/O

In the concept of memory mapped input/output (I/O), the processor core does not distinguish between data memory transactions and transactions with other kinds of devices. Writing to special memory addresses (i.e. outside of the memory range) are forwarded to a device, reading from these addresses means accessing the device's registers.

The systems memory is defined as follows:

- The instruction memory starts at address zero, its length should be configured to a word boundary.[2]
- The data memory starts after the instruction memory, its length is also configurable and should also end on a word boundary.
- The address space of the peripheral components starts at address `0x000F0000`

6.4 Memory Controller

A generic memory controller named `dmem` is given. It has a size of 256 bytes, sectioned in four 64 byte blocks with an access size of one byte. The default configuration is that its base address is directly after the instruction memory. It can and should be used to maintain the stack, as frequent accesses to the instruction memory can slow down the performance. The content of the memory at startup is undefined, and should not be used without initialization.

[2]The default component uses is configured using a word count.

Chapter 7

Guides

7.1 How to implement a lookup or branch table

Due to the limitation of the `ldr` instruction, addressing large fields of constants with it is not feasible, especially if those constants are refered to by more than one part of the program.

This can be circumvented by using the assemblers directive `=labelname` to store a pointer to a labels location (See Section 4.2.4). Pointers to data in memory can be stored conviently close to an `ldr` instruction, making the access to the memory content via a pointer based load or store instruction possible.

The most common use of this mechanic is the lookup table. An example is given in Listing 7.1.

Listing 7.1: Example on indirect LDR use

```
.align
lookup_table:
    .word 0x0001
    .word 0x0010
//...
// large amounts of code and/or data
//...

.align
lookuptable_ptr:
    .word =lookup_table
//...
    ldr rX , >lookuptable_ptr
//...
    ld32 rY,rX // load from pointer
    addi rX, 4
    ld32 rZ,rX // load from pointer with offset
//...
```

7.2 How to test a design using the CAN test package

The platform includes a foreign CAN controller IP component. To ease integration and testing, a test package is provided with the platform source code. With its help, input data and behavior of a CAN network can be modeled more easily.

The test package is located in `can_tp.vhd` and a application demonstration is given as a testbench in `can_demo_tb.vhd` in the testbench directory.

In this guide, two approaches on verifying a design are presented.

7.2.1 Simulating a CAN transmission

The first approach focuses on generating CAN transmissions, that can be used as input for the design under test. This is achieved by unsing the procedure `simulate_can_transmission` (Listing 7.2), which is configurable to any data and timing, your CAN interface uses. The user has not to worry about bit stuffing, crc and error detection.

The `id` and `data` parameters are bit vectors (`std_logic_vector`) that specify the CAN ID and DATA bits of the message.

The actual length of the data part is configurable by the `datasize` parameter. It is an integer specifying the amount of Bytes of the `data` parameter, that will be included in the CAN transmission. The rest of `data` will simply be ignored.

The timing is controlled by the `t_bit` parameter, which is of VDHL type time. It stands for the correct overall length of one CAN symbol (this value has to be calculated from the timing configuration of the tested CAN controller).

The `rx` and `tx` signals have to be connected to a simulated CAN bus. See Section 7.2.3 for more information. The signal `tx` is the actual output of the procedure.

The `test_result` parameter is a diagnostic output of an enumeration type that shows whether the transmission was successful, a CAN error occurs, the arbitration is lost or whether the tested CAN controller is not acknowledging the transmission.

Listing 7.2: Interface

```
procedure simulate_can_transmission(
        constant id         : in std_logic_vector(10 downto 0);
        constant data        : in std_logic_vector (0 to 63);
        constant datasize     : in integer;
        constant t_bit        : in time;
        signal rx           : in std_logic;
        signal tx           : inout std_logic;
        signal test_result     : out rx_check_result)
```

7.2.2 Creating a test network

While the first approach is limited to transmitting can messages, this one aims at providing one or more fully functioning can nodes in a test network. Each node is implemented by an instantiation of `can_vhdl_top`, but instead of steering them with a LT16soc design, functions of the can test package will do the job:

If a certain register in a CAN node should be written to, the procedure `can_wb_write_reg` (Listing 7.3) comes in handy. Its parameter signals `wbs_in` and `wbs_out` must be connected to the CAN controller's Wishbone interface. The parameter `addr` is an integer specifying the target register of the write and the parameter data is a bit vector containing the data. And finally a clock signal `clk` is needed as input, in order to write to the CAN controller. (This must be the same clock used for operating the CAN node.)

Listing 7.3: Interface

```
procedure can_wb_write_reg(
        signal wbs_in     : out wb_slv_in_type;
        signal wbs_out     : in wb_slv_out_type;
        constant addr     : integer;
        constant data    : in std_logic_vector(7 downto 0);
      signal clk        : in std_logic)
```

For extracting certain register contents, the procedure `can_wb_read_reg` (Listing 7.4) is provided. It is used with the same parameters like `can_wb_write_reg`, except a data input bit vector. Instead, a data output signal is provided, to access the read register contents.

Listing 7.4: Interface

```
procedure can_wb_read_reg(
        signal wbs_in     : out wb_slv_in_type;
        signal wbs_out     : in wb_slv_out_type;
        constant addr     : integer;
        signal data       : out std_logic_vector(7 downto 0);
        signal clk        : in std_logic)
```

If successive writes are needed, the procedure `write_regs_from_file` (Listing 7.5) is a convenient way to do this. The first Parameter is a path to a text file. Each line in the file stands for a CAN register, that should be written, and consits of a integer for the desired target register number and eight binary digits ('0' or '1') for the data. Both numbers are separated by one SPACE. See 7.8 for an example.

Listing 7.5: Interface

```
    procedure write_regs_from_file(
        constant filename     : in string;
        signal wbs_in        : out wb_slv_in_type;
        signal wbs_out        : in wb_slv_out_type;
        signal       clk     : in std_logic)
```

Successive reading can be done in a similar way with `read_regs_with_fileaddr` (Listing 7.6). Here, the data part in each line of the file is ignored. If the registers written to should be read, the same file can be used. The parameter `out_filename` determines an additional file, that is used to store the read register contents.

Listing 7.6: Interface

```
procedure read_regs_with_fileaddr(
        constant filename        : in string;
        constant out_filename     : in string;
        signal wbs_in           : out wb_slv_in_type;
        signal wbs_out           : in wb_slv_out_type;
        signal clk              : in std_logic)
```

7.2.3 Integrating the approaches in a test bench

In order to use the two presented approaches in a test bench, some details have to be considered:

- Connecting CAN nodes or transmitting custom messages requires the simulation of a CAN network. A VHDL design that does exactly this is provided: `phys_can_sim` (Listing 7.7). The individual tx and rx signals of the connected can nodes are merged into two vectors. Thier size is determind by the generic parameter `peer_num`, which should be equal to the number of clients connected to the CAN network.

- When using the `simulate_can_transmission` procedure, the `tx` signal is only handled while the procedure is working. For all other times the signal has to be assigned manually.

- When simulating the test bench with ISim, the relative file path in `write_regs_from_file` and `read_regs_with_fileaddr` has its root in the project folder. An example file, containing initialization data for a can node is shown in Listing 7.8.

- When instantiating `can_vhdl_top` (a CAN node without a lt16soc), do not forget to initialize its `wbs_in` port signal porperly. This can be done by assigning the constant `wbs_in_default` from the CAN test package to it.
- A demo test bench using all the mechanics described in the two approaches is provided: `can_demo_tb`.

Listing 7.7: Interface

```
entity phys_can_sim
      generic(
         peer_num : integer );
      port(
         rst : in std_logic;
         rx_vector : out std_logic_vector(peer_num - 1 downto 0);
         tx_vector : in std_logic_vector(peer_num - 1 downto 0) );
end entity phys_can_sim;
```

Listing 7.8: default_setup.tdf

```
4 00000000
5 11111111
6 10000000
7 01001000
8 00000010
0 11111110
```

List of Figures

List of Tables