Delft University of Technology
Faculty of Electrical Engineering,
Mathematics and Computer Science
Department Microelectronics and Computer Engineering
Circuits & Systems Group

MB-Lite+

User Guide

Version 12.1.2

H.J. Lincklaen Arriëns, BSc.
April, 2012.

MB-Lite+ *User Guide*

Table of Contents

1 Preface

This document describes the implementation (VHDL code and more) of a 32-bit, Xilinx MicroBlaze derived soft-core processor. The kick-off for this implementation was in fact given by Tamar Kranenburg's MB-Lite design obtainable from the OpenCores site [MB-Lite]. Except from bug fixes, it also supports Fast Simplex Link I/O ports and the possibility to be JTAG programmable (and readable).
Several designs have already proved its usefulness.
The most recent tests have been performed on a Windows 7 Ultimate PC with Cygwin (1.7.9-1), Mentor Graphics' Model*Sim* SE-64 v10.0c, Synopsys' Synplify Premier F-2011.09-SP1-1 and Xilinx' ISE Design Suite 13.2.

All code and files described in this document are available as a `.zip`-file from our site.

This User Guide is organized as follows.
First, an overview is given of the processor's setup, instruction set, data handling and memory space. Next, more detail is provided about the hardware and the software for this particular implementation, after which some debugging possibilities are mentioned. Finally, in the Appendix a detailed description can be found of all individual files in the release package.

2 Introduction

One of the very popular 32-bit microprocessors nowadays is the MicroBlaze: a 32-bit RISC processor, for use in FPGA designs [MicroBlaze]. The MicroBlaze has been designed by Xilinx., Inc. and is distributed as part of their Embedded Development Kit (EDK in DesignSuite and WebPack).

2.1 RISC Processors

RISC, or Reduced Instruction Set Computer, is a term that is conventionally used to describe a type of microprocessor architecture that employs a small but highly-optimized set of instructions, rather than the large set of more specialized instructions often found in other types of architectures. This other type of processor is traditionally referred to as CISC, or Complex Instruction Set Computer.
Early RISC processors emerged in the late 1970s and early 1980s, and the basic design architecture of all RISC processors has generally followed the characteristics that came from those early research projects and which can be summarized as follows:

- One instruction per clock cycle execution time: RISC processors have a CPI (clock per instruction) of one cycle, due to the optimization of each instruction on the CPU. To allow for high clock frequencies, pipelining is used. This technique allows each instruction to be processed in a set number of stages that are processed in parallel. This in turn allows for the simultaneous execution of a number of different instructions, each instruction being at a different stage in the pipeline.
- Load/Store machine with a large number of internal registers: the RISC design philosophy typically uses a relatively large number (often 32) of internal registers. Most instructions operate on these registers, with access to memory made using a very limited set of Load and Store instructions. This reduces the need for continuous access to usually slower memory for loading and storing intermediate data.
- Separate Data Memory and Instruction Memory access paths: different stages of the pipeline perform simultaneous accesses to memory.

2.2 From MicroBlaze to MB-Lite+

The MicroBlaze is a 32-bit RISC machine that follows the classic RISC architecture described above. It is a load/store machine with 32 general purpose registers. All instructions are 32-bits wide and most of them execute in a single clock cycle.
However, the processor is designed specifically for Xilinx FPGAs and is consequently highly optimized for their FPGA circuits. The MicroBlaze is distributed with the Xilinx Embedded Development Kit (EDK) as a parametric netlist, and although the HDL source code can be obtained from Xilinx at additional costs, it is not to be distributed freely.
Several Microblaze inspired processors are available as open source projects, like e.g. the aeMB and the Openfire, but neither of them did exactly what we were looking for.
Therefore, one of our MSc students, Tamar Kranenburg, recently developed a vhdl version with only the features that we really need to start with. It has been named the MB-Lite and the code can be obtained freely from the OpenCores site [MB-Lite].

Here, we present a revised and extended version of the MB-Lite, called the MB-Lite+ (with internal codename –and often referred to from now on as– '`tumbl`').

Except from repairing the bugs that were present in the MB-Lite, the MB-Lite+ features

- a slightly different approach for connecting I/O,
- the possibility to connect and address Fast Simplex Link (FSL) Masters and/or FSL Slaves,
- the possibility to be programmed by means of JTAG ports,
- separate code and data to be stored in Instruction and Data Memory,
- C, IE and FSL flags implemented in the MSR register.

Internally, nearly all VHDL code has in fact been redesigned such that all control and registered signals are contained in a separate entity/architecture.

2.3 Architecture

Like the original MicroBlaze, the MB-Lite+ uses a pipelined architecture. Most of the instructions take only 1 clock cycle, except for the branch- and return-from-subroutine instructions. These have to flush the pipeline to start fresh from a new instruction address. Also, trying to process data that isn't available, since not having been read yet by a previous instruction, causes the processor to stall for one or more cycles. Next to that, I/O devices that need more cycles before responding may stall the processor too.

For connecting to the outside world, memory mapped I/O or special FSL ports can be used.

2.4 Memory-mapped I/O

Since the MicroBlaze is a 32-bit processor, reserving ranges of memory address space for I/O is generally no real problem, as the memory address space is usually much larger than the required space for all memory and I/O devices together.
There are two major advantages of using memory-mapped I/O instead of dedicated ports for I/O. One of them is that the CPU requires less internal logic and thus will be cheaper, faster, easier to build, less power hungry and physically smaller, which is according to the basic RISC philosophy.
The other advantage is that, because regular memory instructions are used to address devices, all of the CPU's addressing modes are available for the I/O as well as the memory, and instructions that perform an operation directly on a memory operand –loading an operand from a memory location, storing the result to a memory location, or both- can be used with I/O device registers as well.

In fact, all that is needed is an interface to facilitate communication and data transport between the processor's memory bus and the peripheral device.
Clearly, there are several I/O kinds of connection possible. Here, we have chosen for easy connecting to devices with either asynchronous or synchronous interfaces (i.e. data can be read in the same cycle, or is just available in the next one), and to devices using the popular Wishbone interface architecture. Note that, if the intention is to handle devices that may take several processor clock cycles for reading and/or writing data, the need to be able to stall the processor for one or more clock cycles becomes obvious.

2.4.1 Adapters for Asynchronous and Synchronous I/O

The MB-Lite+ can communicate with asynchronous, as well as with synchronous devices with the aid of interconnection adapters. Asynchronous devices are defined here as circuits that, when read, have their output data available in the same clock cycle as in which the read address is applied. For synchronous devices, their data is just available after the next rising edge of the clock (so, in the next

clock cycle).
With the adapter in the distribution package, it is possible to control the amount of cycles that the MB-Lite+ outputs remain unchanged (the processor itself is stalled) to cope with the setup time of a slower device, while with the aid of a pulse extension circuit the processors data can be latched for a number of cycles for coping with the hold time specified for the device.

2.4.2 Wishbone Interconnection Architecture and Wishbone adapter

The Wishbone bus is a simple scalable bus specification to connect IP blocks [WBSpec]. The main objective is to use a flexible, robust, easy to understand and technology-independent communication interface. This bus was initially specified by the Silicore company and is now being further developed by OpenCores, so the specification is public domain.
As a consequence, many IP blocks have been developed using this type of interface and many are available. All Wishbone bus data transfers can execute in one clock cycle. It can be configured as an 8, 16 or 32 bit wide bus. All bus cycles use a handshaking protocol between the master and the slave IP block(s). The architecture of the bus is not defined; it is up to the user/designer to choose one.

From the processor side, no distinction has to exist between 'real' memory and a Wishbone I/O device. Seen from the other side of the bus, everything has to behave like a fully compliant Wishbone master. This can be accomplished with an appropriate adapter circuit that is responsible for the correct transfer of data, address values and control signals between the MB-Lite+ and the Wishbone compliant peripherals (slaves) using the specified Wishbone control signals.

2.4.3 Multiple Slaves

If in a design more than one memory mapped slaves are involved, every one of them needs its own adapter and a private section in memory space.
For Wishbone slaves, that communicate with the processor using a handshake signal called ACK, no a priori knowledge has to exist about the speed of the slaves. The simple Asynchronous or Synchronous slaves mentioned above usually don't have active handshake/feedback signals available, so for such slaves figures for setup and hold delays should be known beforehand and be translated into integer numbers of clock cycles.

2.5 Fast Simplex Link (FSL) I/O

Next or instead of memory mapped i/o, the MB-Lite+ can also be equipped with 32 bits wide Fast Simplex Link (FSL) interfaces ([XAPP529], [DS449]).
These interfaces are divided in FSL-Master and FSL-Slave ports, depending on the direction of the Data and Control flow: master ports are intended for writing data with the MB-Lite+, slave ports for reading. It is possible to implement upto 16 FSL_M ports, and also upto 16 FSL_S ports. Since the FSL channels are dedicated uni-directional point-to-point data streaming interfaces, there is no connection between FSL_M and FSL_S ports, while there is no requirement that they should be combined.
Dedicated instructions are provided to directly transfer 32-bit words to or from the internal General Purpose Registers. Xilinx defined the FSL interfaces to contain a separate bit to indicate whether the sent/received word is of a control or data type, therby differentiating between blocking data, non-blocking data, blocking control, and non-blocking control.
For detailed information on the FSL interface, see [DS449] and [XAPP529].

2.6 The Distribution Package

In the MBLite_Plus_v12.1(.#) software package (see the Appendix for all details) that can be downloaded from our web-site http://ens.ewi.tudelft.nl/~huib/vhdl/ all VHDL entity and architecture descriptions, package files, software utilities, design examples, etc. that are needed to implement a System-on-Chip are available. Some script-files to ease the generation of (parts of) the design are also provided.

The design examples are extensively treated in a separate Example Designs Manual, also obtainable from the same web-site.

In the following sections, more information about the MBLite_Plus_v12.1(.#) software and its use are presented.

3 Hardware Architecture

In this chapter an overview will be given of the MB-Lite+ and, after a short summary of the I/O possibilities, how to connect it to peripheral circuitry.

3.1 MB-Lite+ Instruction Set

Being a “lite” version of the regularly improved and expanded MicroBlaze, only a subset of the MicroBlaze’s instruction set can be executed. Table I list the available mnemonic opcodes. See the Reference Guide [MicroBlaze] for a detailed explanation of each instruction.

Table 1

arithmetic functions:	ADD, ADDC, ADDK, ADDKC, ADDI, ADDIC, ADDIK, ADDIKC, BS, BSI, [1]) MUL, MULI, [2]) RSUB, RSUBC, RSUBK, RSUBKC, RSUBI, RSUBIC, RSUBIK, RSUBIKC
logical functions:	AND, ANDI, OR, ORI, XOR, XORI, ANDN, ANDNI
compare functions:	CMP, CMPU
extend instructions:	IMM SEXT8, SEXT16
shift right:	SRA, SRC, SRL
unconditional branch instructions:	BR, BRD, BRLD, BRA, BRAD, BRALD, BRI, BRID, BRLID, BRAI, BRAID
conditional branch instructions:	BEQ, BNE, BLT, BLE, BGT, BGE, BEQD, BNED, BLTD, BLED, BGTD, BGED, BEQI, BNEI, BLTI, BLEI, BGTI, BGEI, BEQID, BNEID, BLTID, BLEID, BGTID, BGEID
load and store instructions:	LBU, LHU, LW, LBUI, LHUI, LWI, SB, SH, SW, SBI, SHI, SWI
return from interrupt, subroutine	RTID, RTSD
special purpose:	MFS, MTS (MSR Register only)
FSL instructions:	GET, GETD, PUT, PUTD

[1]) Barrel Shift instructions either executed by a hardware barrel shifter, or by means of software emulation (selectable with the `USE_BARREL_g` generic and compiler switches).

[2]) Multiplier instructions either executed by hardware multiplier(s), or by means of software emulation (selectable with the `USE_HW_MUL_g` generic and compiler switches).

3.1.1 Memory architecture

The MB-Lite+ is based on a Harvard architecture and thus features separate address- and data-buses for instruction memory (`imem`) and data memory (`dmem`). Both instruction memory and data memory start at address 0x00000000, while each address refers to a byte-wide memory location. Given the 32-bit address widths, both memories have a maximum size of 4 GBytes.
Thus, although being a 32-bit machine which addresses and processes 32-bit data units, memory sizes and addresses are specified in bytes, i.e. the 32-bit instructions are found on addresses on a 4-byte boundary only, so the program counter's lsb-value will always be 0, 4, 8, or c (hex).
The same holds true for data memory accesses when addressing 32-bit data.

The actual sizes for both memories can be specified individually in the VHDL descriptions with the aid of generic variables, with

`IMEM_ABITS_g` for the number of address bit-lines for the instruction memory, and
`DMEM_ABITS_g` for the number of address bit-lines for the data memory.

Next to these numbers of bits, provided that not all available data memory space is occupied by the `dmem`, the subdivision of this space needs to be specified. This also needs to be done using a generic, viz. with `MEMORY_MAP_g`.
This `MEMORY_MAP_g` should be a one-dimensional array of 32-bit addresses, giving the base addresses of all external (i.e. external to the core, above `dmem`) devices, e.g.

```
MEMORY_MAP_g : memory_map_type := (X"A0000000", X"FFFFFF00");
```

Should we e.g. have to make a subdivision like:
`dmem` starting at zero, 16k Byte size (i.e. 14 address bits),
a 1st extended memory part starting at `0x80000000`, a 2nd memory part starting at `0xfffffe00` and a 3rd part with its base address at `0xffffffc0`, where
this last part is reserved for an 8-bit slave (data bits connected to the least significant bits of the 32-bit data-bus),
then the VHDL entity at the highest level, i.e. in the testbench will have to look like:

```
GENERIC (
      |
    MEMORY_MAP_g : MEMORY_MAP_Type := (X"80000000", X"FFFFFE00", X"FFFFFFC0");
      |
```

resulting in the memory map given in Table 2.

Be aware that when only one base address has to be specified, positional association as listed above is not allowed in VHDL. In that case the use of named association will be asked for, i.e.

```
MEMORY_MAP_g : MEMORY_MAP_Type := (0 => X"FFFFFE00");
```

Next to that, again for the case that only one external device has to be connected, the possibility exists to dedicate the whole external address range to that device (the device's addresses are replicated many times) by using a dedicated `XMEMB_sel_o` selection signal from a `tumbl` configuration (see Section 4 and the "hello"-example's description).

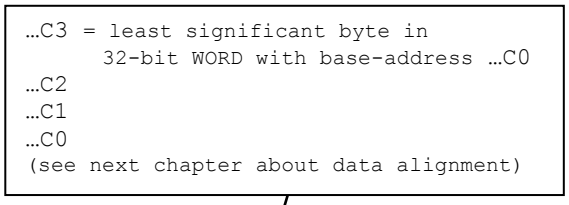


Table 2

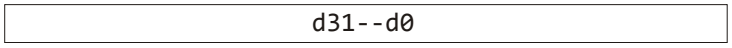
Byte Addresses	32-bit Boundaries	Word Addresses	
FFFF_FFFF		3FFF_FFFF	third claimed memory block, size 64 Bytes, (i.e. maximally 16 32-bit registers) n=3
	FFFF_FFFC	3FFF_FFFE	
	FFFF_FFC4	3FFF_FFF1	
FFFF_FFC0	FFFF_FFC0	3FFF_FFF0	
FFFF_FFBF		3FFF_FFEF	second claimed memory block, size (512 – 64) = 448 Bytes n=2
	FFFF_FFBC	3FFF_FFEE	
	FFFF_FFB8	3FFF_FFED	
	FFFF_FE04	3FFF_FF81	
FFFF_FE00	FFFF_FE00	3FFF_FF80	
FFFF_FDFF		3FFF_FF7F	first claimed memory block, size (2048 M – 512) Bytes n=1
	FFFF_FDFC	3FFF_FF7E	
	FFFF_FDF8	3FFF_FF7D	
	8000_0004	2000_0001	
8000_0000	8000_0000	2000_0000	
7FFF_FFFF		1FFF_FFFF	reserved for internal data memory total free space 2048 MByte, of which only 16384 Bytes occupied (**Note**: STACK and HEAP defined in a Makefile)
	7FFF_FFFC	1FFF_FFFE	
	0000_0010	0000_0004	
	0000_000C	0000_0003	
	0000_0008	0000_0002	
	0000_0004	0000_0001	
0000_0000	0000_0000	0000_0000	

3.1.2 Data Alignment

As mentioned before, the MB-Lite+ is a 32-bit cpu working with 32-bit data (WORDs), but also being capable of handling 16-bit (HALFWORDs) and 8-bit data (BYTEs) units.
Nevertheless, in all memory accesses to `c`-data types, pointers, and also the program counter, are addressing bytes.
Regarding data handling, it is important to know that the MicroBlaze –at least in its early appearances- uses the **Big Endian** data format, which means that the most significant byte of an operand or data unit is stored at the lowest address in memory.
To enable the access of HALFWORDs and BYTEs, a 4-bit '`sel`' signal can be used to select the appropriate part in a 32-bit unit. The relationship between addresses, data types and the `sel`-signal is shown below.

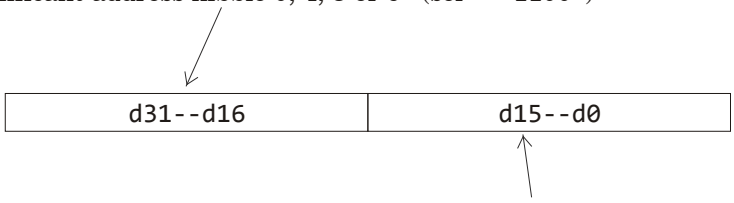
WORD alignment on 4-byte boundaries only

least significant address nibble 0, 4, 8 or c (sel = "0000")



HALFWORD alignment in 32-bit data field (2-byte boundaries only)

least significant address nibble 0, 4, 8 or c (sel = "1100")

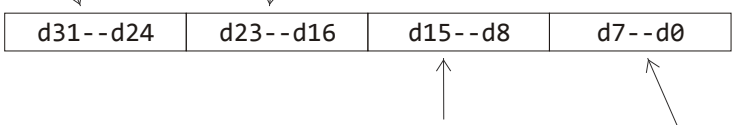


least significant address nibble 2, 6, a or e (sel = "0011")

BYTE alignment in 32-bit data field

least significant address nibble 0, 4, 8 or c (sel = "1000")

least significant address nibble 1, 5, 9 or d (sel = "0100")



least significant address nibble 2, 6, a or e (sel = "0010")

least significant address nibble 3, 7, b or f (sel = "0001")

4 Hardware Implementation

In this chapter an overview will be given of the MB-Lite+ and the interfaces/adapters for connecting it to peripheral circuitry.

The codename for the VHDL-entity of the core of the MB-Lite+ is decided to be ‘`tumbl`’ for the most basic configuration, i.e. only core with instruction and data memory. By adding predefined entities, this `tumbl` can be extended such that totally 8 different configurations can be distinguished.

4.1 Core Configurations

4.1.1 tumbl

In its most basis configuration, `tumbl`, the processor is built using the units (Figure 4.1) indicated as

- fetch — *fetches the correct instruction from instruction memory,*
- decode — *interprets the instruction,*
- exeq — *executes the instruction,*
- mem — *handles the data to be read from or written to (the) data memory (bus)* ,
- gprf — *General Purpose Register File containing the 32 bit wide registers r0—r31,*
- core_ctrl — *all data flow control and additional registers (also MSR),*
- imem and dmem — *instruction and data memories.*

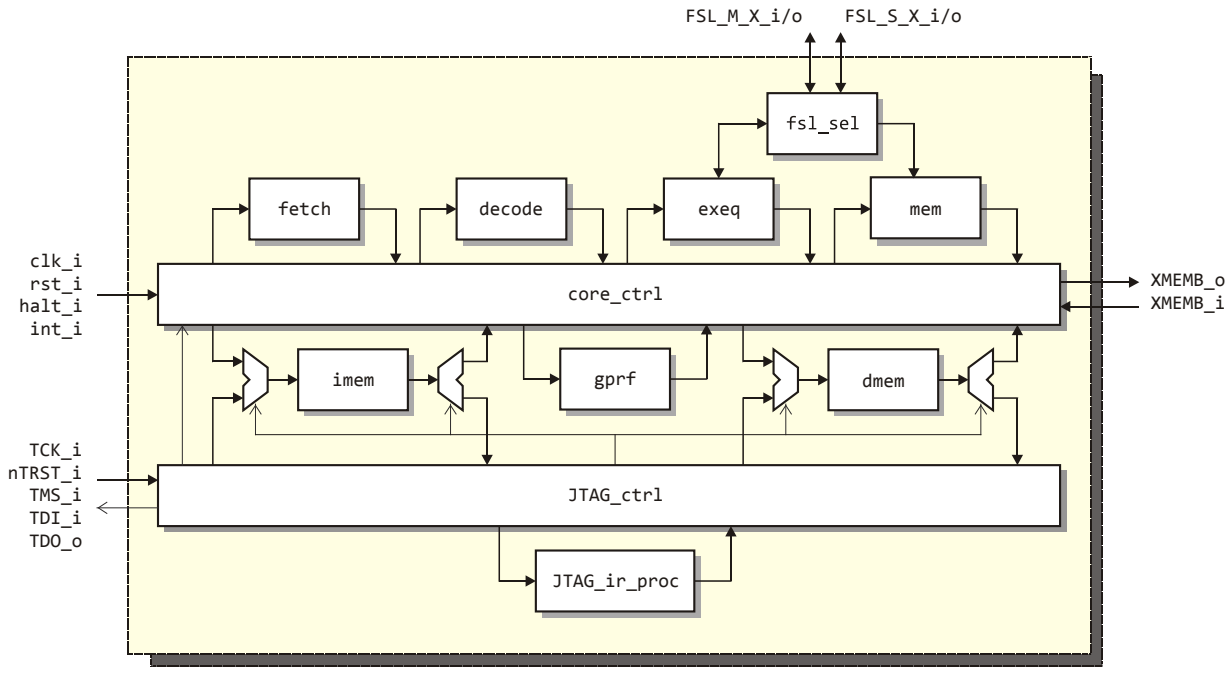


Figure 4.1 *Block scheme of the a featured tumbl_JTAG_FSL_M_S configuration.*

4.1.2 tumbl_FSL

This tumbl can be extended with an 'fsl_sel' block, which offers the possibility to implement a `tumbl_FSL_M`, a `tumbl_FSL_S` or a `tumbl_FSL_M_S`, which respectively contain one or more FSL_Master ports, one or more FSL_Slave ports or one or more FSL_Master and one or more FSL_Slave ports.

4.1.3 tumbl_JTAG

By adding the JTAG_ctrl and JTAG_ir_proc (instruction processor) to the basic `tumbl`, the `tumbl_JTAG` can be created that opens the possibility to (re)program or read the instruction and/or data memory from a JTAG interconnection port.

4.1.4 tumbl_JTAG_FSL

Each `tumbl_FSL` configuration can also be combined with the JTAG circuits.

4.1.5 VHDL entity/architecture/ component

The configurations mentioned above are described in a number of component definitions, so the one needed for a specific design can be instantiated. In Figure 4.2, all available port connections and signal names are shown, highlighted parts being optional.
VHDL generics are used for communicating top level choices to lower parts in the hierarchy, while strongly related signals –usually for bus communication- are combined in VHDL records.

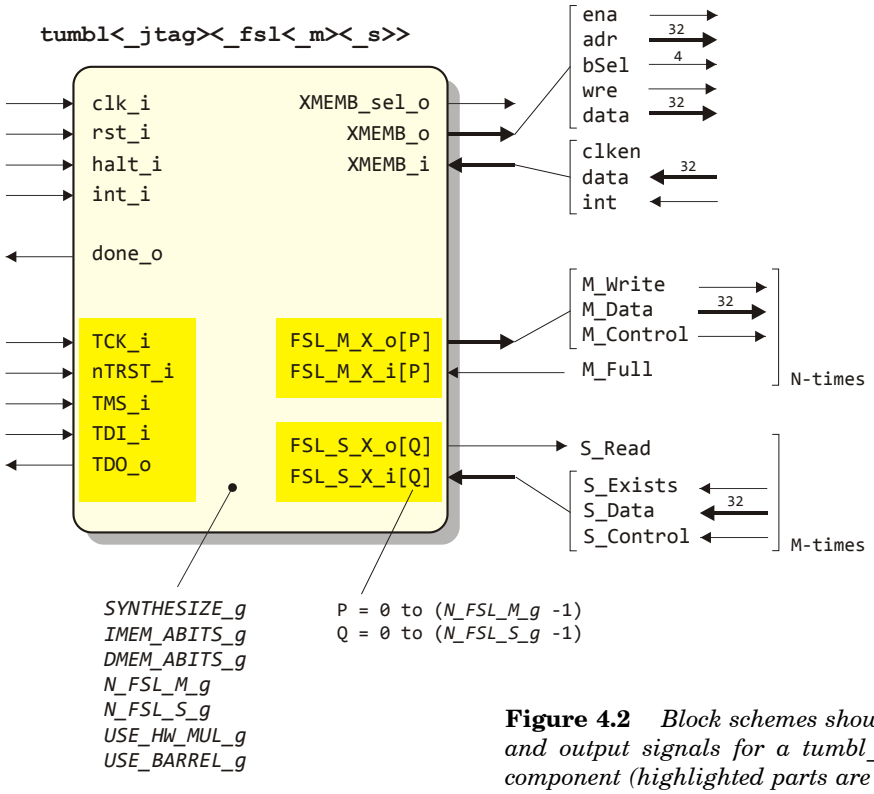


Figure 4.2 *Block schemes showing ports and in- and output signals for a tumbl_JTAG_FSL_M_S component (highlighted parts are optional, and are here added to the most basic configuration).*

4.2 'internal' Instruction and Data Memory

As seen before, the processor core needs 'internal' asynchronous instruction and data memory blocks, the size of which being selectable with generics `IMEM_ABITS_g` and `DMEM_ABITS_g` (number of address bits) respectively. For configurations to be programmable via JTAG ports, a writable IMEM version will be needed (see Figure 4.3b).
The descriptions of these blocks depend on the implementation platform to be used, viz. FPGA, ASIC, etc. At the moment of writing this document, code is available for implementations as Xilinx BRAM, as Faraday ASIC memory, or as automatically inferred memory.

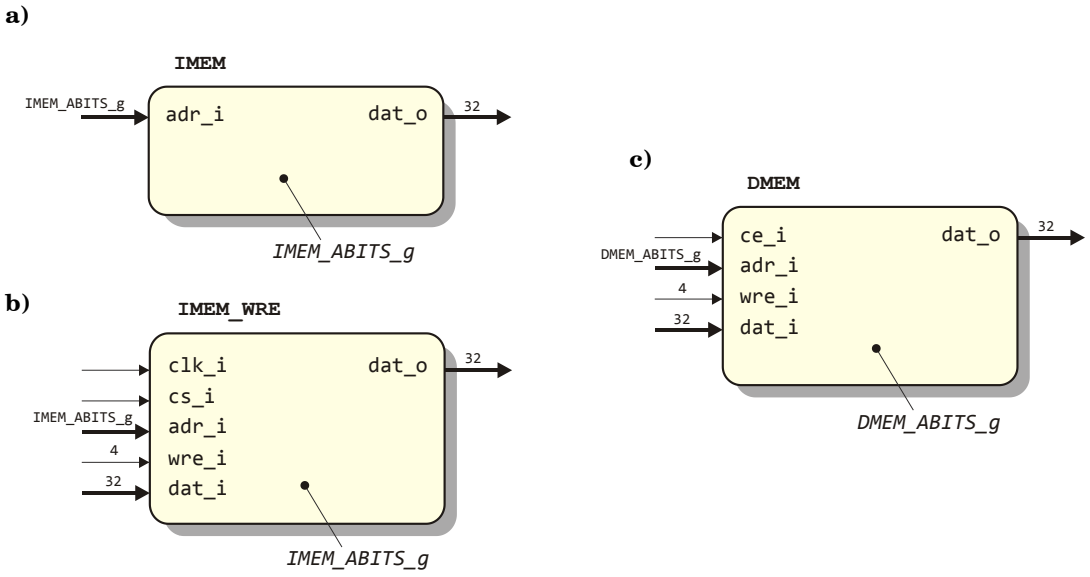


Figure 4.3 *Block schemes showing memory blocks, viz.* **a)** *simple read only Instruction Memory (IMEM),* **b)** *readable and writable Instruction Memory (IMEM_WRE), and* **c)** *Data Memory (DMEM).*

4.3 Memory I/O Extensions

As mentioned before, memory space above the internal dmem data memory space, can be subdivided and assigned to memory i/o extensions.

The electrical connections XMEMB_o and XMEMB_i (Figure 4.2) are based on two VHDL data types, viz. the record definitions given by CORE2DMEMB_Type and DMEMB2CORE_Type in the mbl_Pkg package:

```
TYPE CORE2DMEMB_Type IS RECORD
    ena   : STD_LOGIC;
    addr  : STD_LOGIC_VECTOR (31 DOWNTO 0);
    bSel  : STD_LOGIC_VECTOR ( 3 DOWNTO 0);
    wre   : STD_LOGIC;
    data  : STD_LOGIC_VECTOR (31 DOWNTO 0);
END RECORD;

TYPE DMEMB2CORE_Type IS RECORD
    clken : STD_LOGIC;
    data  : STD_LOGIC_VECTOR (31 DOWNTO 0);
    int   : STD_LOGIC;
END RECORD;
```

The MBLite signals an active memory cycle by raising the ena signal.
If the clken feedback input from the device is high, the processor continues its activities. If this input is taken low, the processor is halted until the next positive going clk edge after clken becomes high again. This enables the use of devices that can't handle the processors speed directly.
The output signal XMEMB_sel_o will be '0' when the internal dmem is addressed, and '1' otherwise. This will be sufficient in case only one slave has to be incorporated. When more i/o devices are needed, a memory map selector is needed to control the memory map subdivision and slave addressing.

4.3.1 Timing Relations

In Figures 4.3 and 4.4, timing diagrams are shown with respect to the XMEMB_o and XMEMB_i ports for a number of different situations.

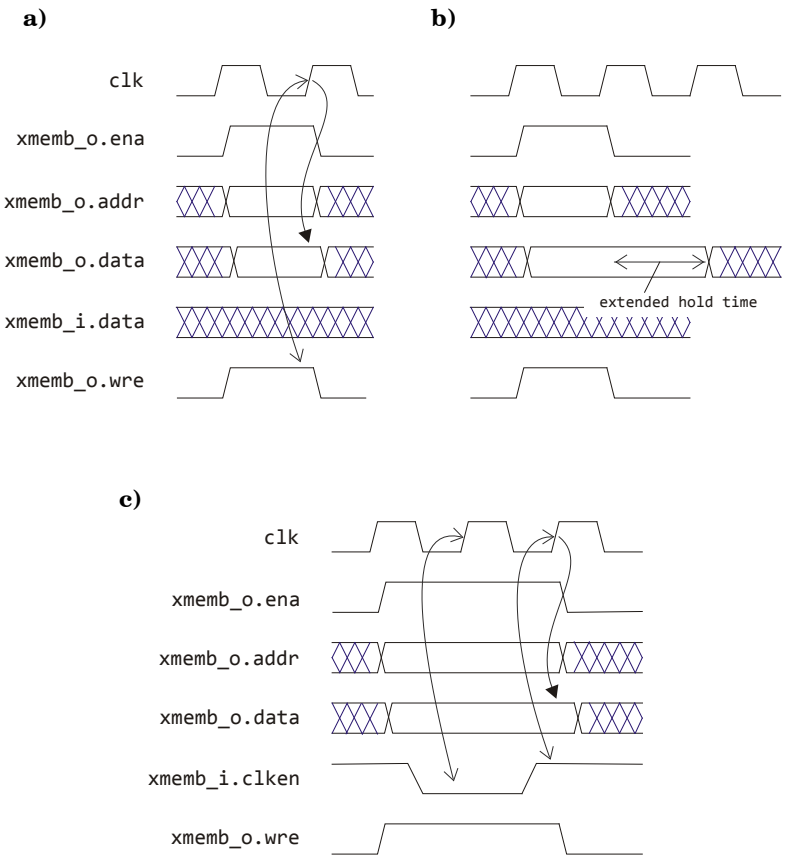


Figure 4.4 *Waveform diagrams for writing data* **a)**, *and* **b)** *with extended data hold time for relatively slow peripheral devices (*`xmemb_i.clken` *high in both cases). In* **c)**, *the effect of stalling (one clock cycle here) the processor during the write process by lowering* `xmemb_i.clken` *is shown.*

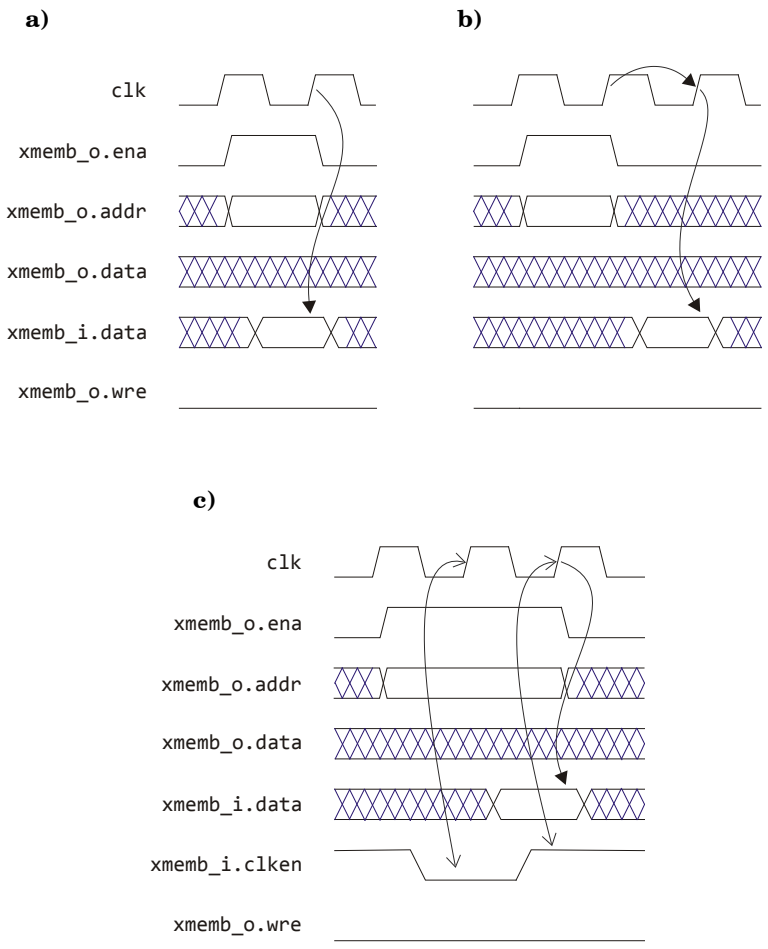


Figure 4.5 *Waveform diagrams for asynchronously reading data* **a)**, *and* **b)** *reading synchronous data (*`xmemb_i.clken` *high in both cases). In* **c)**, *the effect of stalling (one clock cycle here) the processor during an async read by lowering* `xmemb_i.clken` *is shown.*

4.3.2 Memory Map Selector

When more than one i/o devices are needed, the single output signal `XMEMB_sel_o` can't address all slaves, so the need for a memory map selector becomes evident. Figure 4.5 shows the block diagram of the `dmb_selector` that is present in the distribution package: this component translates the `XMEMB_o` and `XMEMB_i` signals into `DMBA_o` and `DMBA_i` signals that are specific for a particular slave and which are derived from the generic `MEMORY_MAP_g`, given at the top level.

`DMBA_o` and `DMBA_i` here are array versions (see `dmb_ext_Pkg.vhd` package file), defined as

```
TYPE CORE2XMEMB_ARRAY_Type IS ARRAY(NATURAL RANGE <>) OF CORE2DMEMB_Type;
TYPE XMEMB2CORE_ARRAY_Type IS ARRAY(NATURAL RANGE <>) OF DMEMB2CORE_Type;
```

of the memory extension type mentioned before.

Note that the clock is not part of the record `CORE2DMEMB_Type`:
for avoiding additional (zero) 'delays' in simulations, that may alter the succession of evaluations of signals, `clk_i` should be connected directly to the appropriate highest level clock.

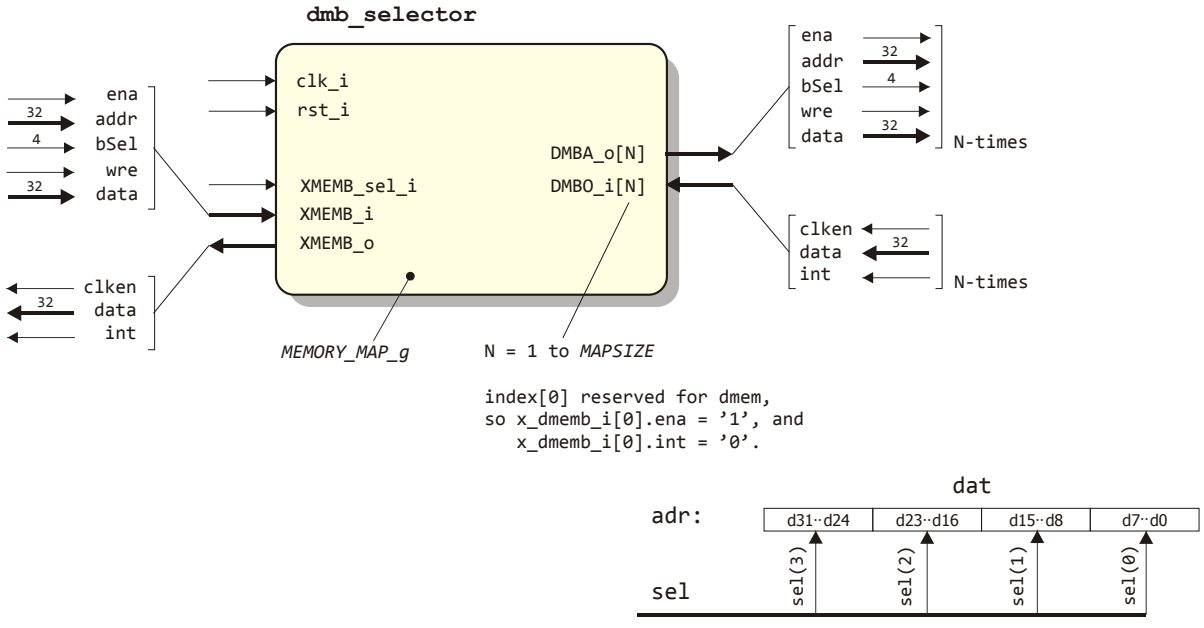


Figure 4.6 *Block diagram of the memory map selector.*

4.3.3 Async/Sync Adapter

The purpose of this block is to synchronize data exchange between `tumbl` and slave in such a way that all reading and writing occurs correctly.
By means of the `ASYNC_SLAVE_g` generic (`TRUE` or `FALSE`) the type of slave can be selected, while `SETUP_TICKS_g` can be used to stall the processor a number of clock cycles in order to cope with a slow slave (`SETUP_TICKS_g` of 1 will cause no delays, larger numbers will). The same value will be used for writing as well as for reading.

The `DMBA_i` and `DMBA_o` ports here are of single `CORE2DMEMB_Type` and `DMEMB2CORE_Type` and are connected to one of the `dmb_selector`'s array output combinations (Figure 4.7).
Here also, `clk_i` should be connected directly to the appropriate highest level clock.

The effective width of the slave's address and data busses have to be passed by means of respectively the `MB_ABITS_g` and `MB_DBITS_g` generics. The `data_i` port will internally be padded with zeros to obtain a 32-bit value if necessary.

The component declaration for the `dmb_adapter` can again be found in the `dmb_ext_Pkg.vhd` file, as well of the type definitions for the `MB_MST_Ctrl_i` and `MB_MST_Ctrl_o` records:

```
TYPE MBL2XMB_Ctrl_Type IS RECORD
    ena    : STD_LOGIC;
    bSel   : STD_LOGIC_VECTOR ( 3 DOWNTO 0);
    wre    : STD_LOGIC;
END RECORD;

TYPE XMB2MBL_Ctrl_Type IS RECORD
    clken : STD_LOGIC;
    int   : STD_LOGIC;
END RECORD;
```

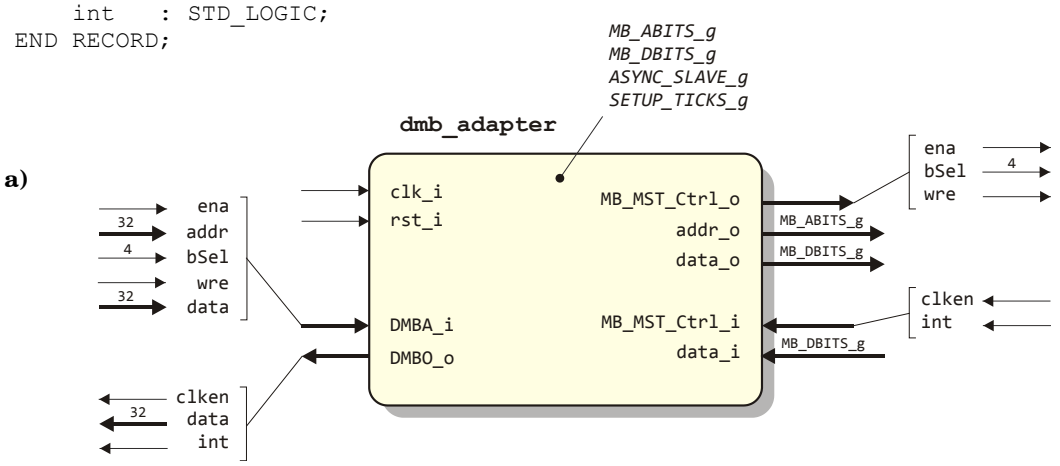


Figure 4.7 **a)** *Block diagram of the Asynchronous/ Synchronous Data Memory Bus Adapter, and* **b)** *address mapping inside the adapter for an* `MB_ABITS_g` *= 3.*

4.3.4 Master-Wishbone Adapter

To enable communication between the `tumbl` with a `dmb-selector` and a Wishbone slave, only a very simple `wishbone_adapter` entity/architecture which will act as a Wishbone Master will be needed (Figure 4.9).

An interconnection scheme with the `record_name.signal_name` convention used is shown in Figure 4.8, together with waveform diagrams representing read and write actions

The effective width of the slave's address and data busses have to be passed by means of respectively the `WB_ABITS_g` and `WB_DBITS_g` generics. The `data_i` port will internally be padded with zeros to obtain a 32-bit value if necessary.
No information will be needed about the slaves response behavior, since this will be controlled by means of the slave's `ACK_o` signal.

The component declaration for the `wishbone_adapter` can again be found in the `dmb_ext_Pkg.vhd` file, as well as the type definitions of the records it refers to.

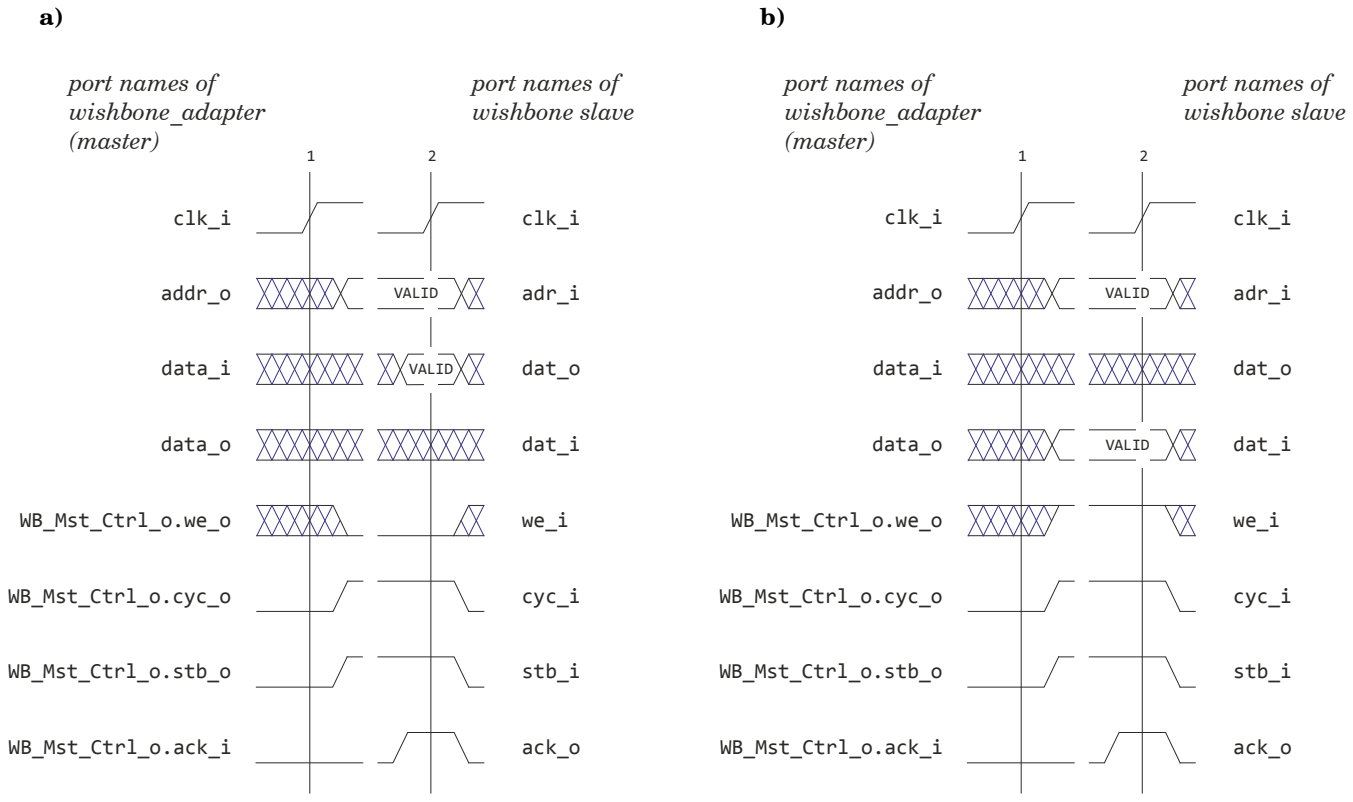


Figure 4.8 *Wishbone* **a)** *SINGLE READ and* **b)** *SINGLE WRITE waveform signals.*

The `WB_MST_Ctrl_o.bSel` signal will be needed in case slaves are involved that allow selecting particular bytes within a larger word (not shown in Figure 4.8).

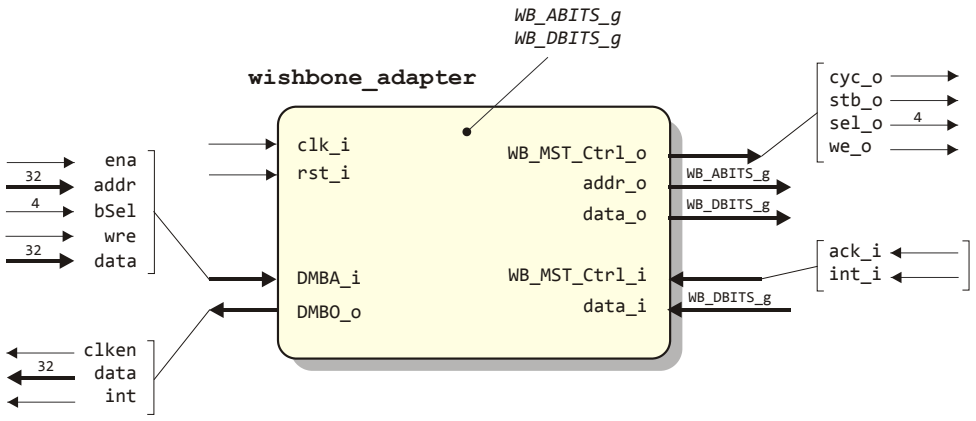


Figure 4.9 *Block diagram of a Wishbone adapter. See Figure 4.7b for internal address mapping between* `DMBA_i.addr` *and* `addr_o`.

4.3.5 Pulse Extender

For coping with slave devices that need a certain data hold time (see Figure 4.4b) before it's data bus is switched into a 3-state mode, a simple `pulse_extender` component is available that can be used to control the amount of hold time, expressed in number of clock cycles, by means of the generic `HOLD_TICKS_g`. It's `pulse_i` input is expected to be connected to the `wre_o` signal controlling this slave, while it's `pulse_o` output controls the data bus mode.

4.4 FSL ports and signals

The number of FSL Master and Slave Ports can be determined with the generics `N_FSL_M_g` and `N_FSL_S_g` respectively. Both can be individually chosen from 1 to 16, since there is no necessity to connect both a Master as well as a Slave port to a certain FSL device. Devices with only a Master connection or only a Slave connection are allowed, without the burden of having unconnected ports at the `tumbl`'s side. It will be only a matter of correctly administrating the several ports and connections to prevent problems.
Of course, a user may decide to select pairs of Master-Slave FSL ports.

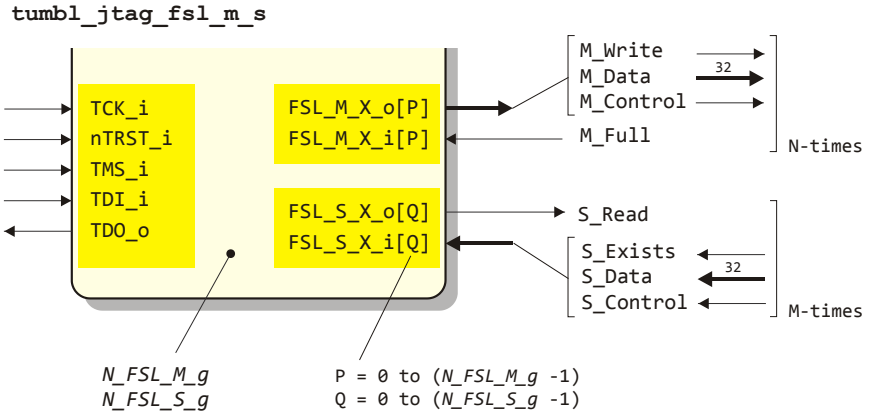


Figure 4.9. *FSL and JTAG port connections.*

Record types have been defined (in the `mbl_Pkg.vhd` package file) that separate and combine output and input signals, viz.

```
TYPE CORE2FSL_M_Type IS RECORD
    -- connect M_Clk directly to highest level clock
    M_Write   : STD_LOGIC;
    M_Data    : STD_LOGIC_VECTOR (31 DOWNTO 0);
    M_Control : STD_LOGIC;
END RECORD;

TYPE FSL_M2CORE_Type IS RECORD
    M_Full    : STD_LOGIC;
END RECORD;

TYPE CORE2FSL_S_Type IS RECORD
    -- connect S_Clk directly to highest level clock
    S_Read    : STD_LOGIC;
END RECORD;

TYPE FSL_S2CORE_Type IS RECORD
    S_Exists  : STD_LOGIC;
    S_Data    : STD_LOGIC_VECTOR (31 DOWNTO 0);
    S_Control : STD_LOGIC;
END RECORD;
```

The FSL_M_X_i/o and FSL_S_X_i/o ports are array types according to

```
TYPE CORE2FSL_M_ARRAY_Type IS ARRAY(NATURAL RANGE <>) OF CORE2FSL_M_Type;
TYPE FSL_M2CORE_ARRAY_Type IS ARRAY(NATURAL RANGE <>) OF FSL_M2CORE_Type;
TYPE CORE2FSL_S_ARRAY_Type IS ARRAY(NATURAL RANGE <>) OF CORE2FSL_S_Type;
TYPE FSL_S2CORE_ARRAY_Type IS ARRAY(NATURAL RANGE <>) OF FSL_S2CORE_Type;
```

4.5 JTAG

It is supposed here, that the reader is familiar with the JTAG protocol.
See Figure 4.9 for the pin names and signal flow directions.

The JTAG Controller used here is 32-bits oriented, e.g. all data and addresses are treated to be 32-bit quantities. JTAG Instructions are 4-bits wide (LSB first) and are listed in Table 3.

Table 3

Instruction	4-bit code	description
JTAG_ON	0000	switch to JTAG Program Mode
JTAG_OFF	0001	switch to RUN Mode
TELL_IDCODE	0010	read back this JTAG's ID-code (1190AF37 hex) [3])
START_ADDR	0011	set 32-bits start address for reading/writing imem/dmem
READ_IMEM	0100	read 32-bit data from imem at address, and auto-increment address
READ_DMEM	0101	read 32-bit data from dmem at address, and auto-increment address
WRITE_IMEM	0110	write 32-bit data to imem at address, and auto-increment address
WRITE_DMEM	0111	write 32-bit data to dmem at address, and auto-increment address
CLEAR_DMEM	1000	clear dmem pointed to by address, and auto-increment address
BYPASS	1001	pass data unaltered

- JTAG_ON, JTAG_OFF and BYPASS are single instructions, not followed by special data.
- The START_ADDR instruction has to be followed by a 32-bit address value (often 0x00000000).
- WRITE_IMEM and also WRITE_DMEM are issued once, and should be followed by all data to be written (LSB first, starting from lowest address = start address). Writing stops when a new instruction is detected by toggling of the TMS_i line.
- Zeroing data memory can be accomplished by issuing a sequence of CLEAR_DMEM instructions.
- Reading from this JTAG implementation itself can be done with TELL_IDCODE, and should result in a 32-bit (hard coded) ID-code.

[3]) from left to right: 4 bits Version Number (1), 16 bits Part Number (190A hex), 11 bits Manufacturer ID (79B hex), with the last bit always 1.

- For reading from the MB-Lite+ internal memory, `READ_IMEM` and/or `READ_DMEM` are available. Both are expected to be preceded by the `START_ADDR` instruction, while reading stops whenever `TMS_i` indicates the start of a new instruction.
- With the `JTAG_OFF` instruction, control is switched back to the MB-Lite+, which will perform a fresh restart of code execution from `IMEM`-address `0x00000000`.

Note that the use of 32-bit data here, deviates from the approach to define memory sizes and addresses in Bytes as has been used throughout this User Guide.

In case JTAG devices are daisy chained, the TDO output of a particular device should be connected to the TDI input of the next one in the chain, except for the first and the last devices in the chain which are both connected to a JTAG Programmer. If only one device is involved, both its TDI and TDO pins should be connected to the Programmer.

Notice however, that usually the **TDI** pin of a Programmer is defined to be an **Output**, while its **TDO** pin is an **Input**.

4.6 A System Controller

The purpose of a System Control module is to provide all signals that are needed by the units mentioned before, viz.

- a continuous clock derived from e.g. a 100 MHz crystal controlled system oscillator. The division factor and 'low versus high times' ratio (duty cycle) of this clock generator are controllable with generics. Default values result in a divide by 4 with 50% duty cycle in order to obtain a symmetrical 25 MHz clock for both the processor and its peripheral devices.
- optional clock signals for slaves (also adjustable with generics) depending on the demands of the slave(s) used.
- a 'clean' reset signal derived from e.g. a push-button that has been assigned to perform the reset function. Debouncing is accomplished by integrating (with an up-down counter) the signal from such a switch. The output changes state when a certain threshold level is reached. The 'time constant' of this integrator is also depending on a generic. Higher values result in a better suppression of unwanted signals at the penalty of a longer delay before the output reset pulse will appear.

Note: it seems reasonable, and is also advised, to use a small value when only simulating with already 'clean' signals (see the comments in the source file(s)). Since the testbench is the top-level entity then, the value of the generic set here will overrule all other values.

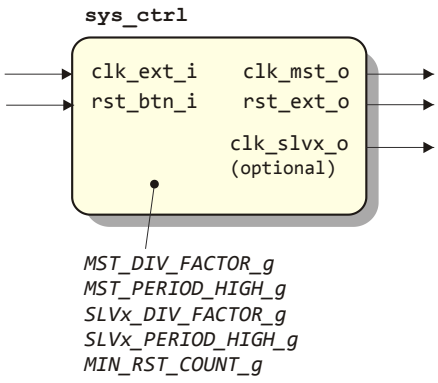


Figure 4.10. *Block scheme of the system controller.*

5 SoC Setup

In Figure 5.1, an impression of a full featured setup for a `tumbl_SoC` is shown (here also embedded in a testbench top level `tb_tumbl_SoC`).

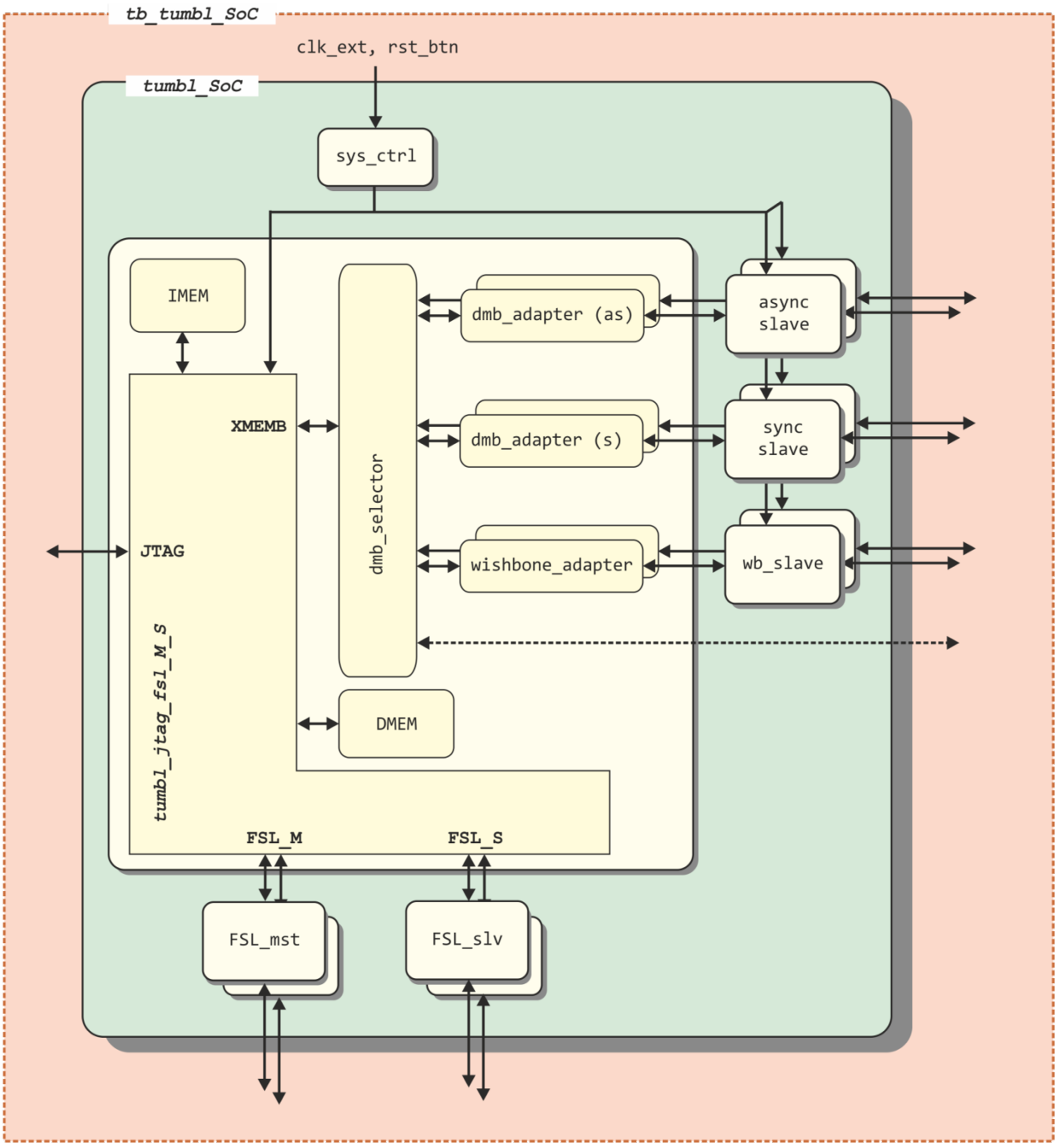


Figure 5.1. *Example scheme of a SoC with an MB-Lite+ with JTAG i/o and connections to synchronous and asynchronous slave interfaces, wishbone slaves, as well as FSL_master- and FSL_slave-interfaces.*

6 Programming the MB-Lite+

Code for the instruction memory can be developed in the usual way by writing one or more `.c-f`iles and accompanying header file(s).
With the aid of the open source mb-gcc compiler, first two binary files can be created:

`imem.bin,`	containing all instruction code, and
`dmem.bin`	that will be needed to initialize data memory.

Sizes for both instruction and data memories have to be defined in a file, called `mem_defs.ld` .

The binary files are next translated into the proper formats for further processing by simulator or synthesizers. All commands for realizing the above are combined in a `Makefile` to be used as input for the Linux or Cygwin `make` utility.

Special care has to be taken that the definitions for `imem` and `dmem` sizes and the memory map definitions as used in the vhdl 'hardware' descriptions, are reflected correctly in the `.h` and `.c`-files, as well as in the `Makefile` and the `mem_defs.ld` file.
Details can be found in the Appendix and by studying the example designs.

Note: Although an interrupt mechanism is implemented in the hardware, we don't supply the low level library code for implementing interrupt service routines, since the necessary Xilinx code has not been released in the public domain.
Those who do have a valid Xilinx license can find the necessary files in the EDK tree, viz. in .....`/EDK/sw/lib/bsp/standalone_v#_0#_a /src/microblaze/`

6.1 Simple Disassembler

The release package contains the `c`-code to create a very basic disassembler. See the Appendix for details.

7 Basis SystemC Model

At the moment, the package includes a basic SystemC model description of the `tumbl` consisting of only the core with instruction and data memory. In fact, this is a stripped down version of the complete description (no FSL yet, to be released) that mimics (cycle accurate, bit accurate) the previously described VHDL architectures.
This basic version is not aware of (external) memory above `dmem` itself. Should this memory be addressed, then writing will have no effect, while reading an 'invalid' address will return `0xdeadbeef` as a result. FSL ports and/or instruction are also not supported. This version's main purpose is the use as instruction set simulator: the resulting executable after compilation –`tumbl_iss` or `tumbl_iss.exe`– (by default) reads the same `imem.bin` and `dmem.bin` files as used for simulation and/or programming the hardware implementation.

Command line options are available, as illustrated below:

```
Usage: tumbl_iss <option(s)>
 simulate MBLite behavior
 Options are:
  -c   Specify clock-period in ns (default 10 ns)
  -t   Specify simulation-time in ns (default 10000 ns)
  -r   Specify rst_start and optionally rst_width (default 100 and 150 ns)
  -i   Specify irq_start and optionally irq_width (default no irq, and if any: width
150 ns)
  -p   Specify path to imem.bin and dmem.bin (if omitted, search in current directory)
  -P   same as a single -p or -p .
  -s   Specify path/filename of single binary file
  -S   Read imem.bin in current directory (single binary)
  -h   Display this information

Note: use a comma as separator between start and width values
```

The only possible (and adjustable) inputs are thus a clock, a reset and an interrupt signal.
Examples:

```
tumbl_iss
tumbl_iss -c40 -r100,300 -t5000000
tumbl_iss -r10,150 -c10 -t2000 -p ../sw  > test.iss
```

Memory sizes (in Bytes) for `imem` and `dmem` can be set in the `main.h` file, and defaults to

```
#define IMEMSIZE_g  32768
#define DMEMSIZE_g  32768
```

Note:
The `-s` and `-S` options are in fact outdated, being intended to be backwards compatible with the first version of the MB-Lite [MB-Lite]. This old version was programmed from a single file that contained all data for both instruction and data memory.

```
     |
     |
  840 ns - 0158: 20c60004  addi r6, r6, 0x4          r6 := 0x6b4 (0x6b0 + 0x4), MSR_C := 0
  850 ns - 015c: 06463800  rsub r18, r6, r7         r18 := 0x0 (0x6b4 - 0x6b4), MSR_C := 1
  860 ns - 0160: bc92fff4  bgti r18, 0xfff4         r18 = 0x0
  870 ns - 0164: b9f4020c  brild r15, 0x20c         r15 := 0x164
  880 ns - 0168: 80000000  or r0, r0, r0            nop
  900 ns - 0370: b60f0008  rtsd r15, 0x8            back to 0x16c (0x164 + 0x8)
  910 ns - 0374: 80000000  or r0, r0, r0            nop
  930 ns - 016c: b9f403c4  brild r15, 0x3c4         r15 := 0x16c
  940 ns - 0170: 80000000  or r0, r0, r0            nop
  960 ns - 0530: 3021fff8  addik r1, r1, 0xfff8     r1 := 0x2e8c (0x2e94 + 0xfffffff8)
  970 ns - 0534: d9e00800  sw r15, r0, r1           dmem[0xba3] <= 0x0000016c
  980 ns - 0538: b9f4fb94  brild r15, 0xfb94        r15 := 0x538
  990 ns - 053c: 80000000  or r0, r0, r0            nop
 1010 ns - 00cc: b0000000  imm 0x0000
 1020 ns - 00d0: 30600000  addik r3, r0, 0x0        r3 := 0x0 (0x0 + 0x0)
 1030 ns - 00d4: 3021ffe4  addik r1, r1, 0xffe4     r1 := 0x2e70 (0x2e8c + 0xffffffe4)
 1040 ns - 00d8: f9e10000  swi r15, r1, 0x0         dmem[0xb9c] <= 0x00000538
 1050 ns - 00dc: 30a0068c  addik r5, r0, 0x68c      r5 := 0x68c (0x0 + 0x68c)
 1060 ns - 00e0: 30c0069c  addik r6, r0, 0x69c      r6 := 0x69c (0x0 + 0x69c)
 1070 ns - 00e4: bc03000c  beqi r3, 0xc             r3 = 0x0
 1100 ns - 00f0: e8600690  lwi r3, r0, 0x690        r3 := dmem[0x1a4] = 0x00000000
 1110 ns - 00f4: b0000000  imm 0x0000
 1120 ns - 00f8: 30800000  addik r4, r0, 0x0        r4 := 0x0 (0x0 + 0x0)
 1130 ns - 00fc: bc030014  beqi r3, 0x14            r3 = 0x0
 1160 ns - 0110: e9e10000  lwi r15, r1, 0x0         r15 := dmem[0xb9c] = 0x00000538
 1180 ns - 0114: b60f0008  rtsd r15, 0x8            back to 0x540 (0x538 + 0x8)
 1190 ns - 0118: 3021001c  addik r1, r1, 0x1c       r1 := 0x2e8c (0x2e70 + 0x1c)
 1210 ns - 0540: b9f4ffb0  brild r15, 0xffb0        r15 := 0x540
 1220 ns - 0544: 80000000  or r0, r0, r0            nop
 1240 ns - 04f0: e8600570  lwi r3, r0, 0x570        r3 := dmem[0x15c] = 0xffffffff
 1250 ns - 04f4: 3021ffe0  addik r1, r1, 0xffe0     r1 := 0x2e6c (0x2e8c + 0xffffffe0)
 1260 ns - 04f8: fa61001c  swi r19, r1, 0x1c        dmem[0xba2] <= 0x00000000
 1270 ns - 04fc: f9e10000  swi r15, r1, 0x0         dmem[0xb9b] <= 0x00000540
 1280 ns - 0500: 32600570  addik r19, r0, 0x570     r19 := 0x570 (0x0 + 0x570)
 1290 ns - 0504: aa43ffff  xori r18, r3, 0xffff     r18 := 0x0 (0xffffffff ^ 0xffffffff)
 1300 ns - 0508: bc120018  beqi r18, 0x18           r18 = 0x0
 1330 ns - 0520: e9e10000  lwi r15, r1, 0x0         r15 := dmem[0xb9b] = 0x00000540
 1340 ns - 0524: ea61001c  lwi r19, r1, 0x1c        r19 := dmem[0xba2] = 0x00000000
 1350 ns - 0528: b60f0008  rtsd r15, 0x8            back to 0x548 (0x540 + 0x8)
 1360 ns - 052c: 30210020  addik r1, r1, 0x20       r1 := 0x2e8c (0x2e6c + 0x20)
 1380 ns - 0548: c9e00800  lw r15, r0, r1           r15 := dmem[0xba3] = 0x0000016c
 1400 ns - 054c: b60f0008  rtsd r15, 0x8            back to 0x174 (0x16c + 0x8)
 1410 ns - 0550: 30210008  addik r1, r1, 0x8        r1 := 0x2e94 (0x2e8c + 0x8)
 1430 ns - 0174: 20c00000  addi r6, r0, 0x0         r6 := 0x0 (0x0 + 0x0), MSR_C := 0
 1440 ns - 0178: 20e00000  addi r7, r0, 0x0         r7 := 0x0 (0x0 + 0x0), MSR_C := 0
 1450 ns - 017c: b9f4002c  brild r15, 0x2c          r15 := 0x17c
 1460 ns - 0180: 20a00000  addi r5, r0, 0x0         r5 := 0x0 (0x0 + 0x0), MSR_C := 0
 1480 ns - 01a8: 3021fff0  addik r1, r1, 0xfff0     r1 := 0x2e84 (0x2e94 + 0xfffffff0)
 1490 ns - 01ac: fa61000c  swi r19, r1, 0xc         dmem[0xba4] <= 0x00000000
     |
     |
```

Figure 6.1. *Snippet of text output from the tumbl_iss, when compiled for assembly code output instead of --or next to- waveform trace output. Shown are sequential code execution steps for a clock input of 100 MHz.*

8 The MB-Lite+ Package

The aforementioned SoC architecture is described in a number of VHDL files. Some of these files can be used without any alterations as they are independent of the rest of the design, while others need to be tailored to the exact wishes of the designer.

8.1 Hierarchy

As also shown in Figure 5.1, a top level file called e.g. `tumbl_soc.vhd` is used to describe the synthesizable implementation. In this description, VHDL 'generics' define parameter values that are passed to the lower level architectures, so all parameters can be adjusted from a central place.

Also shown in Figure 5.1 is, that the top level file for synthesis can be overridden by a top level simulation/testbench file, given here as `tb_tumbl_soc.vhd` (or just `tb_soc.vhd`).
This testbench's architecture instantiates the `tumbl_soc` using parameters for simulation (also given in 'generics') that may overrule those in use for synthesis.
Generics for simulation usually only differ from those for synthesis in order to obtain either more realistic or more bearable simulation times (e.g. to shorten delays in 'slow' slaves, lower counter thresholds, etc.) without affecting e.g. the values to be used for synthesis later on.
Next to that, the testbench defines stimuli signals and possibly reads data from and/or writes data to disk files.

8.1.1 Naming conventions used in the vhdl-files

All entities can be found in files with the same name as the entity with the `.vhd` extension appended.

In the VHDL files, signal groups that connect the several entities, are combined in VHDL records. The definition of the signal types can be found in the `_Pkg.vhd` package descriptions, viz. `mbl_Pkg.vhd`, `dmb_ext_Pkg.vhd`, `JTAG_Pkg.vhd`, etc.

Input ports for each entity are indicated with the postfix `_i`, and output ports consequently with `_o`.

The types of the signals between the instantiated entities or components indicate the direction of the signal flow, e.g.

> from core to data memory bus `CORE2DMEMB_Type`, signal name e.g. `c2dmemb_s`
> from data memory bus to core `DMEMB2CORE_Type`, signal name e.g. `dmemb2c_s`

9 Example Designs

In the release package, 3 example designs can be found. Here, only a short summary will be given of their purposes. Detailed descriptions can be found in a separate "Example Designs Manual", also available from my website.

For each example, all that is needed to perform simulation, synthesis, place-and-route and bit-file generation is available, either in a directly to be used format, in a template form that has to be adapted first or as a file that can be generated by means of a utility.
Resulting `.bit`-files are given that can be programmed directly onto an AVNET XC3S2000 Development Kit, as well as `.bit`-files for an AVNET Spartan-6 LX9 MicroBoard.

9.1 Hello

This example describes a basic `tumbl/uart` setup to check serial communication (19200 Bd). Since the uart is the only 'external' device, no `dmb_selector` has been used.

9.2 SW Test

A more comprehensive test (again `tumbl/uart`), where the `tumbl` now includes a hardware multiplier and a barrel shifter. The software checks the behavior of these modules, as well as the interrupt mechanism (interrupt generated by the uart when a key is pressed), and several other low level software/assembler instructions.
Although again the uart is the only 'external' device, a `dmb_selector` has been used here.

9.3 Integer-DCT with FSL

In this example, which has been inspired by the (deprecated) XAPP529 Application Note from Xilinx, a `tumbl_FSL_M_S` is connected to an FSL component that performs an Integer Discrete-Cosine-Transform on an 8x8 data matrix. The FSL Channels (from the `tumbl_FSL_M_S`'s Master output to the iDCT module's Slave input, and back from the iDCT's M-output to the `tumbl_FSL_M_S`'s S-input are both made up with a custom single delay deep FIFO element.

9.4 Memory Mapped Slaves and Slave Emulators

Here, a `tumbl` is connected to a number of modules that emulate slave devices using memory mapped registers for data communication and that each can emulate a (relatively) time consuming operation. Also connected are the uart and a memory mapped register to enable software control of LEDs present on a pcb.

10 What's next?

A small next step would be to not only generate the `tumbl` configurations, but to also easily generate the VHDL descriptions of more complete SoCs, so including memories, a dmb_selector, dmb_adapters, etc., either from a configuration-file or using a GUI.

A bigger project would be to complete the SystemC model of the `tumbl` with FSL and JTAG ports. In fact to create a high level SystemC model describing a complete SoC, that behaves exactly as the VHDL does now. [4])

[4]) An older, but more elaborate SystemC model of a so-called MBL1C processor (one cycle, no pipeline) that can connect to a number of slaves by means of a Wishbone bus has been part of the ET4351 course as an MSc project. It can be downloaded from http://ens.ewi.tudelft.nl/Education/courses/et4351/ (search for the SystemC Simulation Package and for the User Guide).

11 References

[MB-Lite] Design of a Portable and Customizable Microprocessor for Rapid System Prototyping, *CAS-MS-2009-13*, Master of Science Thesis, Tamar Kranenburg B.Sc.
Source code available through http://opencores.org/project,mblite,overview

[MicroBlaze] MicroBlaze Processor Reference Guide UG081 (v10.2), *Xilinx Embedded Development Kit EDK 11.3*

[DS449] LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11c), *Xilinx Embedded Development Kit EDK 11.3*

[XAPP529] Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL) Channel, *Xilinx Application Note*

[WBSpec] Specifications for the WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IO Cores, Revision B.3, September 7, 2002
http://cdn.opencores.org/downloads/wbspec_b3.pdf
or
Prerelease Rev. B4, 06/22/2010:
http://cdn.opencores.com/downloads/wbspec_b4.pdf

[Modeltech] Model*Sim* User Guide and Reference Guide
http://model.com/content/modelsim-se-downloads-support or
http://model.com/content/modelsim-pe-student-edition-hdl-simulation
under tab DownLoads

[Cygwin] Cygwin is a Linux-like environment for Windows ….
http://www.cygwin.com/

Appendix

A.1 Installation and software requirements

The VHDL files are Operating System independent, and should work on any system.
Everything has been tested both on Windows with Cygwin, and on Linux machines.
In our setups, we used Mentor Graphics' Model*Sim*-SE for simulation, Synopsys' Synplify Pro or Premier for synthesis, and the Xilinx ISE programs for place-and-route and for loading the memories.
Xilinx's iMPACT has been the choice for programming the `.bit`-files into an FPGA, or –when programming in a JTAG configuration- an Amontec JTAGkey2 and a custom executable.

For creating the initialized memory files `imem.bin` and `dmem.bin`, the `mb-gcc` compiler will be needed.

The most recent MB-Lite+ release can be downloaded from http://ens.ewi.tudelft.nl/~huib/vhdl
Preserve full path names when unzipping.
The example designs are extensively described in a separate manual (follow same link as above).

In the `MB-Lite_Plus_v12.1` top level directory, the following sub-directories should then be present:

```
MBLite_Plus_v12.1
       |
       ├── boards
       ├── hdl
       |     ├─── all_tumbl_cfgs
       |     ├─── dmb_ext
       |     └─── memories
       |              ├─── Faraday
       |              ├─── inferred
       |              └─── Xilinx_BRAM
       ├── misc_hdl
       ├── misc_sw
       ├── scripts
       ├── sw_utils
       |         ├─── mb-dasm
       |         └─── src
       ├── sysC
       |
       └── designs
             ├─── hello       (with their own subdirectories)
             ├─── sw_test     (with their own subdirectories)
             ├─── fsl_idct    (with their own subdirectories)
             └─── slaves_ex   (with their own subdirectories)
```

In the following sections, the several VHDL-files, packages, utilities, etc. will be discussed in detail.

A.2 Contents of the release package

Note Files with an extension `_template` are either not complete or presumed to be not usable as is: they should be adapted to the SoC that is to be designed!

In the `boards/`-directory:

`AVNET_DK_xc3s2000.ucf`	*pin definitions for the AVNET Spartan-3 Development Kit*
`AVNET_6LX9_MicroBoard.ucf`	*pin definitions for the AVNET Spartan-6 LX9 MicroBoard*

In the *`hdl/`-directory:*

`core_ctrl.vhd`	*sequential pipeline-control unit*
`decode.vhd`	*combinatorial decode unit*
`exeq.vhd`	*combinatorial execute unit*
`fetch.vhd`	*combinatorial fetch unit*
`fsl_M_selector.vhd`	*selector controlling the FSL-Master outputs*
`fsl_S_selector.vhd`	*selector controlling the FSL-Slave inputs*
`mbl_Pkg.vhd`	*package with definitions and functions*
`mem.vhd`	*combinatorial mem unit*

In the `hdl/all_tumbl_cfgs/`-directory:

`tumbl.vhd`	*all possible tumbl configurations …*
`tumbl_fsl_M.vhd`	*… created with* `gen_tumbl_vhd.c` (see `sw_utils/src`)
`tumbl_fsl_M_S.vhd`	
`tumbl_fsl_S.vhd`	
`tumbl_jtag.vhd`	
`tumbl_jtag_fsl_M.vhd`	
`tumbl_jtag_fsl_M_S.vhd`	
`tumbl_jtag_fsl_S.vhd`	
`tumbl_comp_Pkg.vhd`	*package with all component declarations*
`tumbl_instants.templ`	*summary of all possible instantiations*

In the `hdl/dmb_ext/`-directory:

`dmb_adapter.vhd`	*interface between data-memory bus and a slave (sync|async)*
`dmb_ext_Pkg.vhd`	*package with additional definitions*
`dmb_selector.vhd`	*memory map controller*
`pulse_extender.vhd`	*extent the length of an active high signal*
`wishbone_adapter.vhd`	*interface between data-memory bus and a Wishbone slave*

In the `hdl/JTAG32/`-directory:

`JTAG_Ctrl.vhd`	*JTAG controller connecting to the outside world*
`JTAG_IR_Proc.vhd`	*JTAG instructions processor*
`JTAG_Pkg.vhd`	*package with additional JTAG definitions*

In the `hdl/memories/Faraday/`-directory:

> Note that the following code is intended only as an example illustrating the use of specific memories. For simulation i.e.. VITAL_Primtives and VITAL_Timing libraries will be needed, as well as layout data in case of synthesis.

`dmem_Faraday.vhd`	*wrapper to instantiate the correct Faraday component*
`Faraday_mem_Pkg.vhd`	*package with additional definitions*
`gprf_abd_Faraday.vhd`	*wrapper to instantiate the correct Faraday components*
`imem_Faraday.vhd`	*wrapper to instantiate the correct Faraday component*
`imem_wre_Faraday.vhd`	*wrapper to instantiate the correct Faraday component*
`SJAA90_32X32X1CM4.vhd`	*Faraday component used for the `gprf`*
`SHAA90_4096X8X4CM4.vhd`	*Faraday component used for `dmem`*
`SHAA90_4096X32X1CM4.vhd`	*Faraday component used for `imem` `imem_wre`*

In the `hdl/memories/inferred/`-directory:

`dpram.vhd`	*Dual Port RAM entity/architecture*
`gprf_abd_inferred.vhd`	*infer the General Purpose Register File*
`dmem_inferred.vhd`	*infer data memory*
`imem_inferred.vhd`	*infer instruction memory ROM*
`imem_wre_inferred.vhd`	*infer R/W instruction memory (needed for JTAG)*

In the `hdl/memories/Xilinx_BRAM/`-directory:

`gprf_abd_RAMB16_S36_S36.vhd`	*instantiates and connects Block RAMs for the `gprf`*
`dmem_RAMB16_S9.vhd`	*instantiates and connects Block RAMs for `dmem`*
`imem_RAMB16_S36.vhd`	*instantiates and connects Block RAMs for `imem`*
`imem_wre_RAMB16_S36.vhd`	*… and for `imem_wre` (R/W for JTAG)*

In the `misc_hdl/`-directory:

`clk_div.vhd`	*clock divider using generics for division factor and duty-cycle*
`debouncer.vhd`	*eliminate bouncing of a (simulated) reset button*
`misc_comp_Pkg.vhd`	*package with additional definitions and functions*
`sys_ctrl.vhd_template`	*connects clock generator(s) and the reset debouncer*
`uart_AVR8.vhd`	*simple UART copied from the AVR8 release by R. Lepetenok*

In the `misc_sw/`-directory:

`Makefile_template`	*starting point for creating the `.bin`-files, etc.*
`mbl_asm.h`	*additional Macros and assembler code*
`mbl_settings_def_template`	*(path) definitions referred to by the Makefile*
`memmap.h_template`	*defines memory base-addresses, for a specific design*
`mem_defs.ld_template`	*memory setup info, to be read by the Makefile*
`uart_AVR8.h`	*defines for the AVR8 uart*
`uart_AVR8.c`	*low level functions for controlling the AVR8 uart*

In the `scripts/`-directory:

`makebit_bmm`	*for generating a bit file using Xilinx ISE executables*
`makemem`	*to be used for updating the memory in an existing bit-file*
`make_mpf_template`	*utility for creating a ModelSim `.mpf` project file*

In the `sw_utils/mb-dasm/`-directory:

`imem.bin_example`	*example of a binay instruction file*
`Makefile`	*simple makefile with commands for the `make`-utility*
`mb-dasm.cpp`	*source for the disassembler*

In the `sw_utils/src/`-directory:

`bin2imem_dmem.c`	*for creating the `imem_dmem.mem` memory file for programming*
`bin2mem_4x8b.c`	*for creating `dmem0.mem` … `dmem3.mem` memory files*
`bin2mem_32b.c`	*for creating the `imem.mem` memory file*
`bin2mem_ramb16_4x8b.c`	*for creating `.mem` data memory files when RAMBs are involved*
`bin2mem_ramb16_32b.c`	*creates `.mem` instruction memory files for RAMBs usage*
`bin2vhd_dmem4.c` [5])	*for creating the initialized `dmem-init.vhd` (inferred memory)*
`bin2vhd_imem.c` [5])	*for creating the initialized `imem-init.vhd` (inferred memory)*
`elf2bins.c`	*custom `.elf`-file translator (replaces binutil's objdump)*
`gen_bmm.c`	*for creating a `.bmm`-file description*
`gen_start_do.c`	*for creating a `start.do` file for ModelSim (based on RAMBs)*
`gen_tumbl_vhd.c`	*for creating one of the possible configurations*
`makeit`	*very simple script for creating the executables (Linux/Cygwin)*
`makeit.bat`	

In the `sysC/`-directory:

`imem.bin_example`	*example file with binary instructions*
`main.cpp`	*top level command line interface*
`main.h`	*top level header file*
`Makefile`	*simple makefile with commands for the `make`-utility*
`mblite_cid_iss.h`	*SystemC model of the Instruction Set Simulator*
`run_cid_iss.h`	*stimuli signal simulator to feed the ISS*
`types.h`	*two more header files …*
`utils.h`	

[5]) In fact needed by previous versions, and now superseded by the use of the `.mem`-files

The `designs/`-directory contains four examples, according to the following setup.
These designs are extensively described in a separate Example Designs Manual.

```
designs
   |
   ├── hello                    vhdl descriptions specific for this design
   |     ├── sw                 software to be executed by the tumbl
   |     ├── tb_msim            the vhdl testbench for this design
   |     |     └── msim         project directory for ModelSim
   |     └── synth              project directory for Synplify
   |           ├── rev_1        work directory for Synplify and ISE (XC3S2000 here)
   |           └── rev_2        work directory for Synplify and ISE (6LX9 here)
   |
   ├── sw_test                  vhdl descriptions specific for this design
   |     ├── sw                 software to be executed by the tumbl (full version)
   |     ├── sw_6LX9            software version adapted for the 6LX9
   |     ├── tb_msim            the vhdl testbench for this design
   |     |     └── msim         project directory for ModelSim
   |     └── synth              project directory for Synplify
   |           ├── XC3S2000     implementation directory for Synplify and ISE
   |           └── 6LX9         implementation directory for Synplify and ISE
   |
   ├── fsl_idct                 vhdl descriptions for the fst_dct example
   |     ├── matlab             Matlab verification data
   |     ├── sw                 dedicated software to be executed by the tumbl
   |     ├── tb_msim            the vhdl testbench for this design
   |     |     └── msim         project directory for ModelSim
   |     └── synth              project directory for Synplify
   |           ├── XC3S2000     implementation directory for Synplify and ISE
   |           └── 6LX9         implementation directory for Synplify and ISE
   |
   └── slaves_ex                vhdl descriptions for the slaves_ex example
         ├── sw                 dedicated software to be executed by the tumbl
         ├── tb_msim            the vhdl testbench for this design
         |     └── msim         project directory for ModelSim
         └── synth              project directory for Synplify
               └── 6LX9         implementation directory for Synplify and ISE
```

The following files should be present.

In the **designs/hello/**-directory:

sys_ctrl.vhd	*the controller (clock divider and reset circuitry) for this design*
tumbl_uart_soc.vhd	*top level circuit description for synthesis (50 MHz tumbl-clock)*

In the designs/hello/sw/-directory:

hello.c	*the actual c–source of the actions to be performed*
Makefile	*input commands for the make utility*
memmap.h	*the memory map base address of the uart*
mem_defs.ld	*definition of imem and dmem sizes*
uart_AVR8.c	*low level functions for serial communication*
uart_AVR8.h	*description of the uart's registers and BaudRate*

In the designs/hello/tb_msim/

tb_soc.vhd	*top level testbench file (generics given here overrule all others)*

In the designs/hello/tb_msim/msim/-directory:

make_mpf.do	*script for creating the project file for ModelSim*
msim.mpf	*(template) project file for ModelSim*
start.do	*memory load and simulation file as created for this design*
wave.do	*waveform layout definition for this design*

In the designs/hello/synth/-directory:

synth.prj	*project file for Synplify*
synth.sdc	*(timing) constraints for Synplify*

In the designs/hello/synth/rev_1/-directory:

AVNET_DK_xc3s2000.ucf	*pin definitions for the AVNET Spartan-3 Development Kit*
hello.bit	*this is the working code to be programmed in the XC3S2000*
makebit_bmm	*script for generating a bit file using Xilinx ISE tools*
makemem	*script for updating imem and dmem in the bit-file*
tumbl_uart_soc.bmm	*info needed by makebit_bmm to assign space to memories*

In the designs/hello/synth/rev_2/-directory:

AVNET_6LX9_MicroBoard.ucf	*pin definitions for the AVNET Spartan-6 LX9 MicroBoard*
hello.bit	*this is the working code to be programmed in the Spartan6 LX9*
makebit_bmm	*script for generating a bit file using Xilinx ISE tools*
makemem	*script for updating imem and dmem in the bit-file*
tumbl_uart_soc.bmm	*info needed by makebit_bmm to assign space to memories*

In the **designs/sw_test/**-directory:

sys_ctrl.vhd	*the controller (clock divider and reset circuitry) for this design*
tumbl_uart_soc.vhd	*top level circuit description for synthesis (25 MHz tumbl-clock)*

In the designs/sw_test/sw/-directory:

dhry.c	*c-source for the Dhrystone benchmark*
dhry.h	*header file for dhry.c*
Makefile	*input commands for the make utility*
mbl_asm.h	*additional Macros and assembler code needed here*
memmap.h	*the memory map base address of the uart*
mem_defs.ld	*definition of imem and dmem sizes*
testbench.c	*the actual c-source of the actions to be performed (full version)*
uart_AVR8.c	*low level functions for serial communication*
uart_AVR8.h	*description of the uart's registers and BaudRate*

In the designs/sw_test/sw_6LX9/-directory:

Makefile	*input commands for the make utility*
mbl_asm.h	*additional Macros and assembler code needed here*
memmap.h	*the memory map base address of the uart*
mem_defs.ld	*definition of imem and dmem sizes*
testbench.c	*the actual c-source of the actions to be performed*
uart_AVR8.c	*low level functions for serial communication*
uart_AVR8.h	*description of the uart's registers and BaudRate (19200 Bd)*

In the designs/sw_test/tb_msim/-directory:

tb_soc.vhd	*top level testbench file (generics given here overrule all others)*

In the designs/sw_test/tb_msim/msim/-directory:

make_mpf.do	*script for creating the project file for ModelSim*
msim.mpf	*(template) project file for ModelSim*
wave.do	*waveform layout definition for this design*

In the designs/sw_test/synth/-directory:

synth.prj	*project file for Synplify*

In the designs/sw_test/synth/XC3S2000/-directory:

sw_test_xc3s2000.bit	*this is the working code to be programmed in the XC3S2000*

In the designs/sw_test/synth/6LX9/-directory:

sw_test_6LX9.bit	*this is the working code to be programmed in the 6LX9*

In the **designs/fsl_idct/**-directory:

fsl_bb1.vhd	*FSL Master-to-Slave interface (one level deep)*
fsl_idct.vhd	*the integer Discrete Cosine Transform block*
fsl_idct_Pkg.vhd	*package file with component declarations*
fsl_idct_uart_soc.vhd	*top level circuit description for synthesis (50 MHz tumbl-clock)*
sys_ctrl.vhd	*the controller (clock divider and reset circuitry) for this design*

In the designs/fsl_idct/matlab/-directory:

check_idct.m	*Matlab file for computing the expected results*
check_idct.out	*text file with (intermediate) results*

In the designs/fsl_idct/sw/-directory:

fsl_idct.c	*the actual c–source of the actions to be performed*
fsl_idct_msim.c	*c–source for simulation without uart actions and print out*
Makefile	*input commands for the make utility*
memmap.h	*the memory map base address of the uart*
mem_defs.ld	*definition of imem and dmem sizes*
uart_AVR8.c	*low level functions for serial communication*
uart_AVR8.h	*description of the uart's registers and BaudRate*

In the designs/fsl_idct/tb_msim/-directory:

tb_soc.vhd	*top level testbench file (generics given here overrule all others)*

In the designs/fsl_idct/tb_msim/msim/-directory:

make_mpf.do	*script for creating the project file for ModelSim*
msim.mpf	*(template) project file for ModelSim*
wave.do	*waveform layout definition for this design*

In the designs/fsl_idct/synth/-directory:

synth.prj	*project file for Synplify*

In the designs/fsl_idct/synth/6LX9/-directory:

fsl_idct.bit	*this is the working code to be programmed in the XC3S2000*

In the designs/fsl_idct/synth/XC3S2000/-directory:

fsl_idct.bit	*this is the working code to be programmed in the 6LX9*

Some files, especially in the project directories, have been left out here, since either

- they are automatically inserted during the software creation process, or since
- their purposes shall be clear from the previous example.

In the **designs/slaves_ex/**-directory:

sys_ctrl.vhd	*the controller (clock divider and reset circuitry) for this design*
tumbl_slaves_ex_SoC.vhd	*top level circuit description for synthesis (50 MHz tumbl-clock)*
amb_slave_emu.vhd	*slave emulator with an asynchronous data interface*
smb_slave_emu.vhd	*slave emulator with a synchronous data interface*
wb_slave_emu.vhd	*slave emulator with a wishbone interface*
slv_Pkg.vhd	*package file with component declarations*

In the designs/slaves_ex/sw/-directory:

Makefile	*input commands for the make utility*
memmap.h	*the memory map base address of the uart*
mem_defs.ld	*definition of imem and dmem sizes*
uart_AVR8.c	*low level functions for serial communication*
uart_AVR8.h	*description of the uart's registers and BaudRate*
slaves_ex.c	*the actual c-source for synthesis*
slaves_ex.c	*the c-source for faster simulation without uart output*
amb_slv1.h, smb_slv2.h, wb_slv3.h, wb_slv4.h, wb_slv5.h dmb_reg.h.	*description and specs of the other slaves*

In the designs/slaves_ex/tb_msim/-directory:

tb_soc.vhd	*top level testbench file (generics given here overrule all others)*

In the designs/slaves_ex/tb_msim/msim/-directory:

make_mpf.do	*script for creating the project file for ModelSim*
msim.mpf	*(template) project file for ModelSim*
wave.do	*waveform layout definition for this design*

In the designs/slaves_ex/synth/-directory:

synth.prj	*project file for Synplify*

In the designs/slaves_ex/synth/6LX9/-directory:

slaves_ex.bit	*this is the working code to be programmed in the 6LX9*

A.3 Simulation and Synthesis setup

In Figure A.3.1, all files needed for a basic design –based on a particular `tumbl` configuration- are listed. Other ip-files can be easily added.

<table>
<tr><th colspan="4">System-on-Chip involving a</th></tr>
<tr><th>tumbl</th><th>tumbl with FSL</th><th>tumbl with JTAG</th><th>tumbl with JTAG and FSL</th></tr>
<tr><td colspan="4">mbl_Pkg.vhd
fetch.vhd
decode.vhd
exeq.vhd
mem.vhd
core_ctrl.vhd
gprf_abd_xxxx.vhd
dmem_xxxx.vhd</td></tr>
<tr><td>imem_xxxx.vhd</td><td>imem_xxxx.vhd</td><td>imem_wre_xxxx.vhd</td><td>imem_wre_xxxx.vhd</td></tr>
<tr><td></td><td>fsl_M_selector.vhd
and/or
fsl_S_selector.vhd</td><td></td><td>fsl_M_selector.vhd
and/or
fsl_S_selector.vhd</td></tr>
<tr><td></td><td></td><td>JTAG_Pkg.vhd
JTAG_Ctrl.vhd
JTAG_IR_Proc.vhd</td><td>JTAG_Pkg.vhd
JTAG_Ctrl.vhd
JTAG_IR_Proc.vhd</td></tr>
<tr><td>tumbl.vhd</td><td>tumbl_fsl_M.vhd
or
tumbl_fsl_S.vhd
or
tumbl_fsl_M_S.vhd</td><td>tumbl_jtag.vhd</td><td>tumbl_jtag_fsl_M.vhd
or
tumbl_jtag_fsl_S.vhd
or
tumbl_jtag_fsl_M_S.vhd</td></tr>
<tr><td colspan="4">misc_comp_Pkg.vhd
clk_div.vhd
debouncer.vhd
sys_ctrl.vhd
uart_AVR8.vhd</td></tr>
<tr><td colspan="4">tumbl_XXXX_soc.vhd</td></tr>
<tr><td colspan="4">tb_soc.vhd</td></tr>
</table>

Figure A.3.1 *Combination of files for a `tumbl` system.*

Figure A.3.2 gives an impression of the design setup and flow that has also been used in the description of the example designs, and that is more or less expected for a smooth operation of the provide Makefile(s).
Directory names are marked blue. Other colors indicate files that are created and –when needed by another program- are moved to the correct directory.

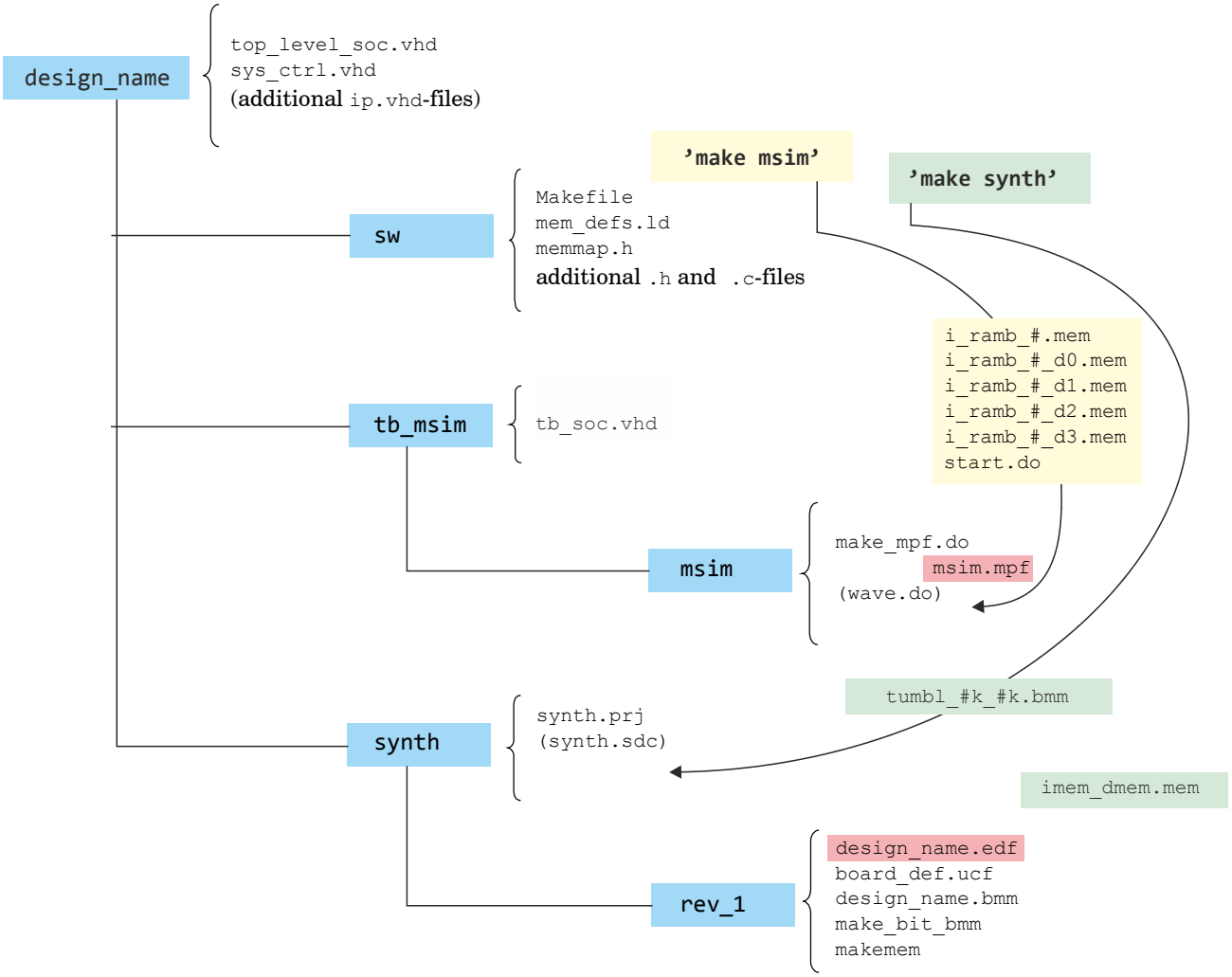


Figure A.3.2 *Directory structure and files.*

Pointwise, assuming that all vhdl-code has been written, some remarks follow about the steps that can be taken to obtain a working `.bit`-file. The description is based on the assumption that Model*Sim*, Synplify Pro or Premier and Xilinx ISE are available.

- First of all the utilities in the `sw_utils/src/`-directory should be compiled, and the path to where the executables are located should be made known in the `mbl_settings.def`-file, which in turn will be read by the `Makefile` in the `sw`-directory.

- Copy the needed files to the `sw`-directory, edit template file(s) and write the code that has to be executed by the `tumbl`.

- If simulation is wanted, a simple

  ```
  make msim
  ```

  will create and copy all files needed for a Model*Sim* simulation session to the `tb_sim` and `tb_sim/msim` directories (these directories should have been created first).

  Note: Ignore warnings about overlapping sections when executing the `Makefile`: Although the `-no-check-sections` linker directive is passed to the linker/loader, this seems to have no effect.

- In `tb_sim/msim` edit and complete the `make_mpf.do` file (see the example designs), i.e. list all vhdl-files to be involved. A project file, `msim.mpf`, will be created by issuing the command

  ```
  vsim -c -do make_mpf.do
  ```

  from the command line.
  Check that all paths and libraries (e.g. `UNISIM`) are listed in this project file, otherwise provide the correct assignments.
  Also, for initializing the memories the way proposed here, it will be necessary that these memories are visible from within Model*Sim*. The same holds true for all signals and variables listed in the `wave.do` files supplied with the example designs.
  The easiest way to obtain this is to change the value of `VoptFlow` im the project file from the default 1 in a 0. For faster simulations, of course only the specific memories need to be available.

  This project file can be opened from within Model*Sim*. However, before trying to compile a project opened this way, first create a `work`–directory by entering

  ```
  vlib work
  ```

  in the Transcript window.
  After starting a simulation (`tb_soc`), a `wave.do` file can be executes if present.
  If the procedure described above has been followed, a

  ```
  do start.do
  ```

  (press Enter two times) will initialize the memories, after which the simulation can be run.

- Note that displaying waveforms for large memory blocks in Model*Sim* may severely slow down working with the waveform viewer. In the before mentioned `wave.do` files the lower level internal parts of memory blocks are omitted.
  On the other hand, (only when simulating) a special array "`ram`" has been added for easily viewing and debugging the contents of the General Purpose Registers File (see the `gprf_abd_xxxx.vhd` files in the `hdl/memories/` directories).

- Synthesis also is expected to be initiated from the `sw`-directory. This time with

  ```
  make synth
  ```

  which will create a template `.bmm`-file with info for the Xilinx tools about the memory partitioning and the `imem_dmem.mem`-file with the contents for these memories.
  The template `.bmm` will be copied to the `synth` directory and the `.mem`-file to the revision directory. The `.bmm`-file should be renamed (same base name as the top level entity) and copied or moved to the revision directory too.
  Here the easiest way to tell Synplify which files to use, is to edit an existing `synth.prj` file, as available in the example designs.

After a successful synthesis run, and after copying the script files `make_bmm` and `makemem` into the revision directory, the command

```
./makebit_bmm <design_filename> <board_specific_user_constaint_filename>  [6])
```

will run the Xilinx ISE tools and result in an initialized `.bit`-file.

Note: When a design is processed for s Xilinx family that isn't equipped with RAMB16 memory blocks, these will usually be replaced with corresponding library elements known to the device in question. For the Spartan 6 LX9 e.g,. RAMB16BWER components will be used. As expected, this results in a fairly large number of warnings .

Next

```
./makemem <design_filename> imem_dmem.mem
```

will initialize the memories, and create the `program.bit`-file that can be used to program the FPGA on the previously selected board.

[6]) without filename extensions

End of Document