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

MB-Lite+

Example Designs Manual

Version 12.1.2

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

MB-Lite+ *Example Designs Manual*

Table of Contents

This page intentionally left blank

1 Introduction

This document describes a number of example FPGA designs, based on the MB-Lite+ that has been developed on the Delft University of Technology.
The procedures followed in the designs are strongly driven by the available hardware, tools and licenses provided by our University and my own preferences and experience. In no way I pretend that the strategy proposed here is the ultimate or the most efficient one, but it surely works for me.
In any case, it shows my/our preference for a command line approach.

The resulting bit-files are for Xilinx Spartan 3 and Spartan 6 FPGA's, more specifically situated on an AVNET XC3S2000 Development Kit and on an AVNET Spartan 6 LX9 MicroBoard (both provided with an X-tal clock of 100 MHz).
Simulations have been carried out with Mentor Graphics' Model*Sim* SE-64 v10.0c, while for obtaining the bit-files Synopsys' Synplify Premier F-2011.09-SP1-1 and Xilinx' ISE Design Suite 13.2 have been used. All programs executed on a Windows 7 PC with Cygwin (1.7.9-1) installed.

1.1 Example Designs

The examples to be described are:

1.1.1 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.

1.1.2 SW Test

A more comprehensive test (again tumbl/uart), where the `tumbl` now includes a hardware multiplier and a hardware 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.

1.1.3 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.

1.1.4 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.

1.2 General Setup of an MB-Lite+ SoC and design

Referring to the MB-Lite+ User Guide, Figure 1.1 gives an impression of a circuit with a `tumbl_JTAG_FSL_M_S` as the basic building block, connected to several interfaces to peripheral devices. In the figure, `tumbl_SoC` indicates the synthesizable part, that can be simulated (with different parameters if needed) using a `tb_tumbl_SoC` testbench.

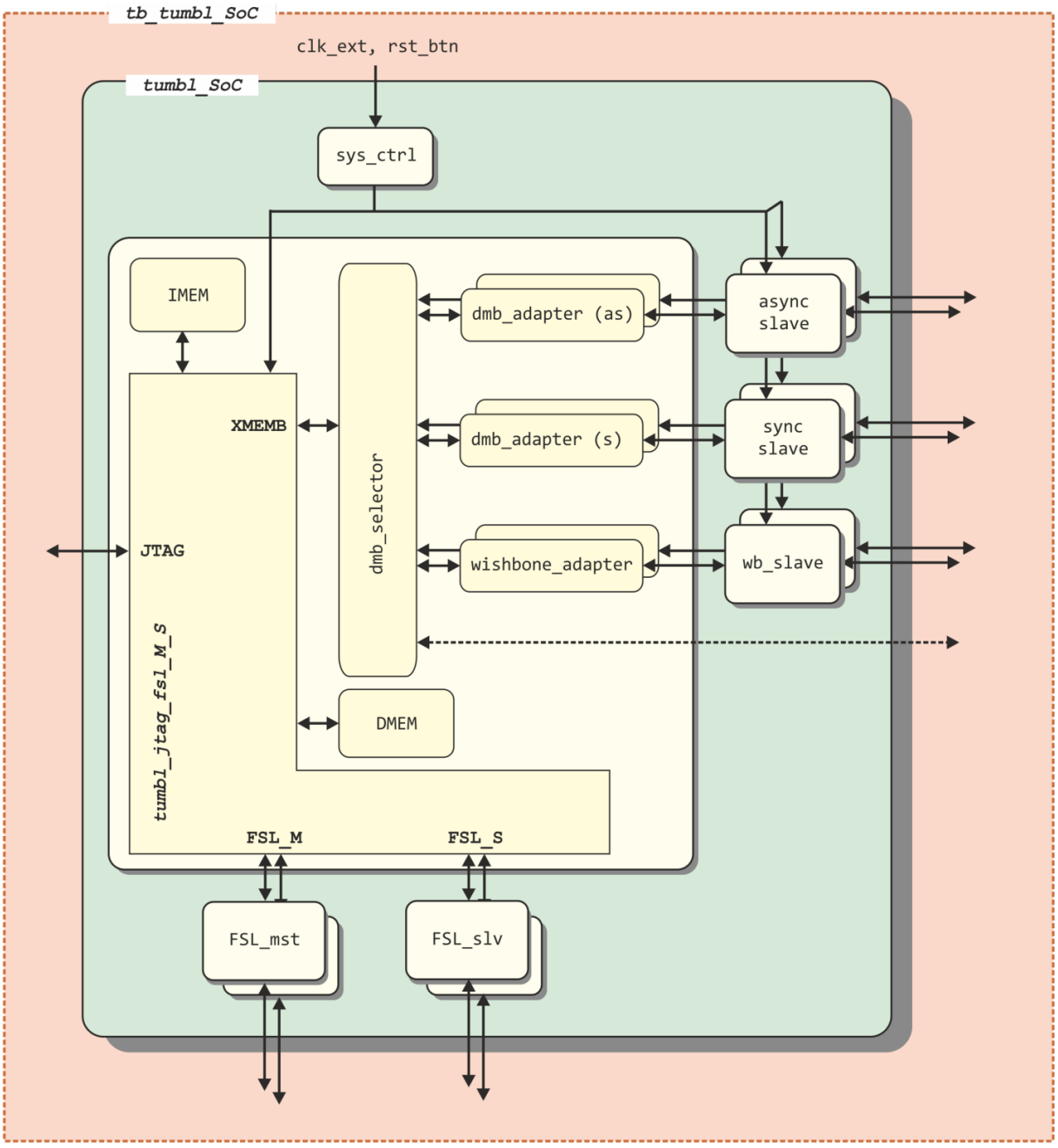


Figure 1.1 *Example scheme of a SoC with an MB-Lite+ with JTAG i/o and connections to synchronous and asynchronous slave interfaces, wishbone slaves, as well as FSL_master- and FSL_slave-interfaces.*

1.3 Implementation Tree

In the MB-Lite+ release package the `designs/`-directory is subdivided in a number of subdirectories:

```
designs
    |
    ├── hello                       vhdl descriptions for the hello 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
    |               ├── rev_1       work directory for Synplify and ISE (XC3S2000 here)
    |               └── rev_2       work directory for Synplify and ISE (6LX9 here)
    |
    ├── sw_test                     vhdl descriptions for the sw_test example
    |       ├── sw                 dedicated software for 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
```

Figure 1.2 ***designs/*** *directory tree.*

This is also illustrated in Figure 1.3, together with the files expected to be present or to be created in the several subdirectories.

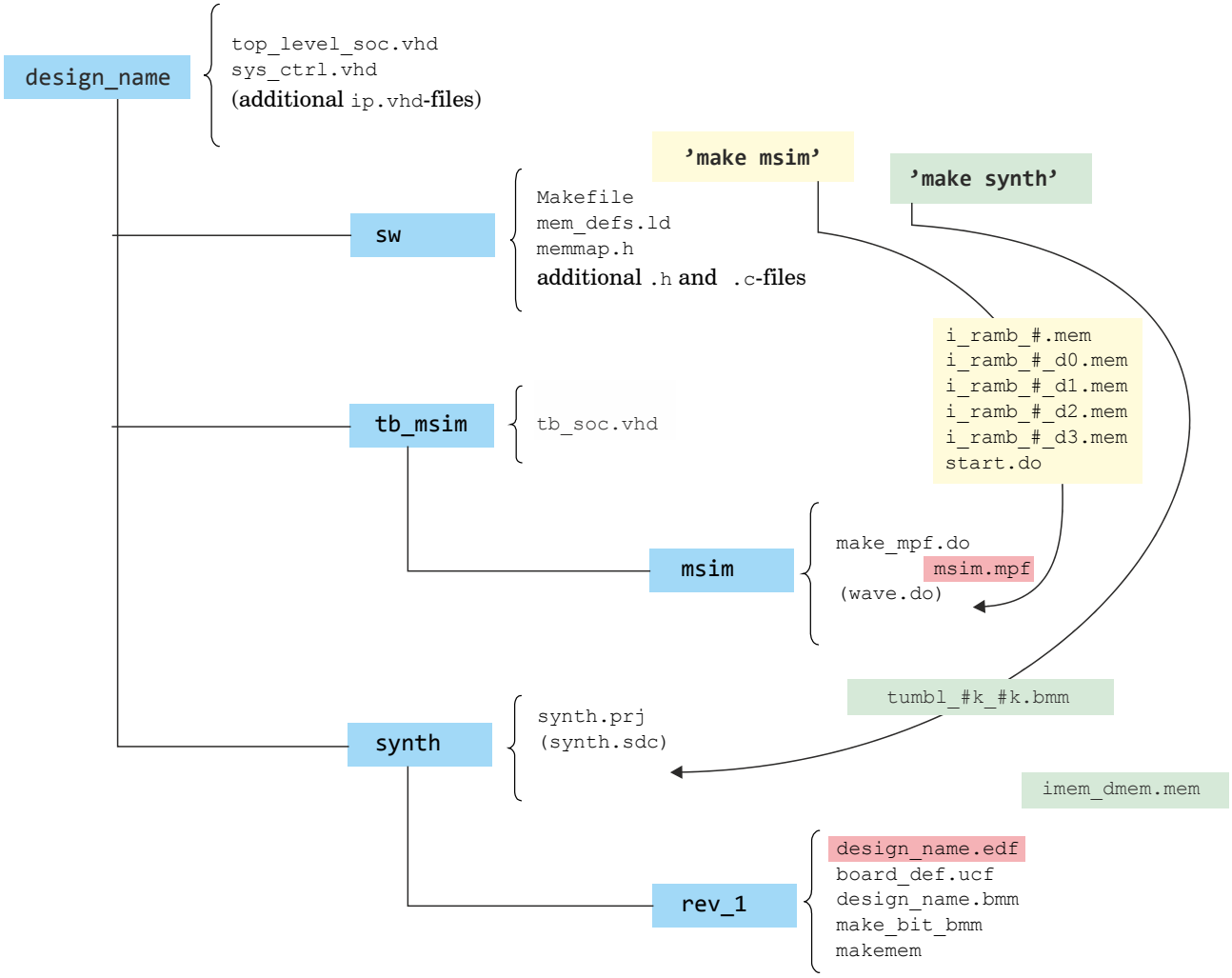


Figure 1.3 *Directory structure and basic files setup.*

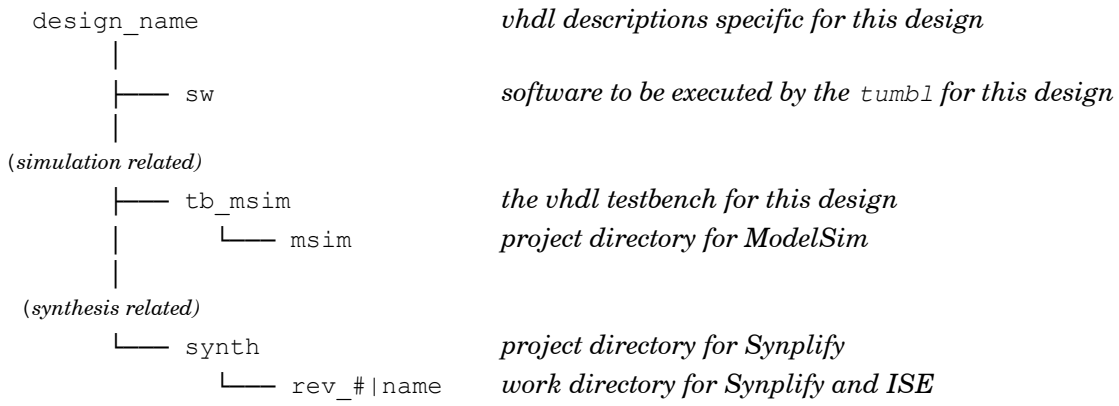
1.4 Initial Setup

To ease the design process, a number of scripts and supporting programs are provided.

Especially the software `make`-process relies heavily on such utilities. They are provided as `c`-source files in the `sw_utils/src`-directory, and should be compiled for the OS to be used beforehand (see the MB-Lite+ User Guide).

The path to where the resulting executables are located has to be made known in the `mbl_settings.def`-file (see the `SUPATH` variable), which in turn will be read by the `Makefile` in the `sw`-directory later on (only a single `mbl_settings.def`-file will be necessary on a certain machine, and in these examples it is located in the `MB-Lite_Plus_v12.1`-directory.
The `mbl_settings.def` file also has to contain information about the location of the `mb-gcc` executables (`MBPATH`).

From Figures 1.2 and 1.3, it can be seen that each design is fit in a directory structure like



It is supposed here that such a tree has been set up beforehand.
Next to that, it can be good practice to also copy

- `Makefile_template`, `memmap.h_template` and `mem_defs.ld_template` from the `misc_sw/`-directory into the `sw/`-directory,
- `make_mpf_template` from the `scripts/`-directory into the `tb_msim/msim/`-directory,
- `makebit_bmm` and `makemem` from the `scripts/`-directory into the `synth/rev/`-directory, and
- the `.ucf`-file to be involved from the `scripts/`-directory into the `synth/rev/`-directory.

2 Hello

2.1 Top Level HDL Description

This example describes a basic tumbl/uart setup to check serial communication (19200 bps).
The clock signals for both the tumbl and the uart are chosen to be 50 MHz and are derived from the 100 MHz X-tal clocks on the development boards.
The uart used here originates from the AVR8 opencore release by Ruslan Lepetenok. It can be controlled by means of 4 consecutive 8-bit registers, viz.

- a Baud Rate Register (UBRR, memory address UART_BASEADDR + 0x03, see uart_AVR.h),
- a Control Register (UCR, addressable at UART_BASEADDR + 0x07),
- a Status Register (USR, addressable at UART_BASEADDR + 0x0b), and
- the I/O Data Register (UDR, addressable at UART_BASEADDR + 0x0f).

Since the uart is the only 'external' device, no dmb_selector has been used, and ART_BASEADDR can be every address on a 32-bit word boundary above the data memory (arbitrary set to 0x80000000 in memmap.h, see also the Software Description section).

The following files should be present in the **hello/**-directory (Figure 1.2):

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)*

sys_ctrl.vhd is an edited version of sys_ctrl.vhd_template from the misc_hdl/ directory and consists only of a clock divider to obtain the clock for the tumbl and the usrt, and a debouncer circuit for the reset buttons on the pcbs.

In tumbl_uart_soc.vhd the generics MST_DIV_FACTOR_g and MST_PERIODS_HIGH_g set the division factor and duty-cycle for the clock signal. MST_PERIODS_HIGH_g controls the integration time-constant for the reset signal, expressed in number of clock cycles of the X-tal clock (1 ms here).

IMEM_ABITS_g and DMEM_ABITS_g set the sizes for respectively imem and dmem both to 14 bits, i.e. 16 kBytes or 4 kWords of 32-bits each.

Since the uart is the only device to be addressed in the external memory space, the single XMEMB_sel_o signal can be used as the selector for the uart, so avoiding the need for a dmb_selector.

The uart is prevented to stall or interrupt the tumbl by means of hard wired connections to V_{CC} and GND respectively.

2.2 Software Description

If the hardware setup is assumed to be correct, the software to be run by the tumbl can be developed. The sw/-directory is meant for this purpose, and for this example will contain:

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

Except for the main c-file (hello.c here), all files have been copied from the misc_sw/ directory, while some of them needed to be tailored to this design.

The purpose of memmap.h is to define the base-adresses of all devices to be visible in the memory map. Since now the uart is the only device, while always being selected when an address outside dmem's range is seen, the UART_BASEADDR given here has in fact no real meaning.

uart_AVR8.h defines the addresses of the uart's registers (all 8-bits wide), and the functions of the specific bits in these registers. Since this uart has originally been designed for an Atmel AVR soft core, the reader is referred to the AVR's manual for further information.

Also in uart_AVR8.h, the relationship between uart-clock and Baudrate is given. Notice again that the uart has been designed for an 8-bit device and a clock of around 4 MHz, so is equipped with an also 8-bits programmable divider register. This means that without any changes, the lowest supported Baudrate when using a clockspeed of 50 MHz will be 19200 bps. If lower Baudrates would be really needed, the easiest solution would be to increase the fixed internal divide-by-16 counter to e.g. divide-by-128 and to change the macro in uartAVR8.h accordingly.

Some things that always should be accounted for:

- All necessary c-files have to be mentioned in the Makefile,
- The variables IMEM_ABITS and DMEM_ABITS in the Makefile should be equal to respectively the generics IMEM_ABITS_g and DMEM_ABITS_g in the top level vhdl file, and should correspond with the LENGTH's definitions (size in Bytes) of imem and dmem in mem_defs.ld,
- If different or additional directory names (e.g. several revisions) appear in the tree, reflect this in the TBPATH and SYNPATH variables (Makefile).

At this time –provided the tree has been created first- it is possible to create the data files needed for simulation with

```
make msim
```

in a Cygwin environment, and/or all files to be used for synthesis with

```
make synth
```

As a result of the make msim command, a number of files containing memory data will be copied to the designs/hello/tb_msim/msim/-directory, the number of which being dependent on the memory type and sizes involved. Instruction memory data will be recognizable as i_ramb_#.mem, data memory contents will be named i_ramb_#_d0.mem .... i_ramb_#_d3.mem.

Next to that, a file start.do will be created in the afore mentioned directory containing the commands to load these memory files into the simulator.

As a reminder about the memory sizes in the current design, a file IMEM_ABITS_#__DMEM_ABITS_# will be created in the tb_msim/-directory.

2.3 Simulation

The designs/hello/tb_msim/ directory is intended to contain the top level testbench file, tb_soc.vhd.

Being the top level file, all generics defined in the testbench entity will overrule all others. In some case this can be profitable to speed up simulations that tend to be time consuming when used with realistic values needed for synthesis, while leaving the synthesis generics unaltered.

The subdirectory msim/ is intended to contain all 'lower level' and command files that are needed for the simulation, while all data created by the simulator itself is written in still lower subdirectories.

In the designs/hello/tb_msim/msim/-directory a script-file make_mpf.do can be found to ease creation of the project file for Model*Sim*. The make_mpf.do here, contains the information specific for this example, but can be the basis for other designs.

It can be invoked by a

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

from the command line, and will create the msim.mpf project file (derived from Model*Sim*'s current modelsim.ini file).

Special care has to be taken that the memories to be loaded with start.do are all visible and recognizable after compilation by the names given in there, instead of probably being optimized and renamed. Moreover, it is the intention of this example to also show many of the signals of the tumbl, its architecture and peripherals, and so optimization is not a good idea here.

To completely avoid optimization, edit the project file and change the default value for the VoptFlow directive from a 1 into a 0 [1]). VoptFlow can be found in the [vsim]-section.

A prefabricated version of an msim.mpf project file will already be available in the tb_msim/msim/-directory.

Note that –by using Xilinx Block RAMs in this example- the UNISIM library should be precompiled and accessible. Of course, pathnames in the project file given are only valid for the system it has been tested on.

Also available in the tb_msim/msim/-directory will be a file called wave.do. This file can be executed during a simulation and sets and controls the layout of the Wave window.

The procedure to be followed now, can be as follows.

- start Model*Sim* and open the existing msim.mpf project file,
- create a work-subdirectory by entering

  ```
  vlib work
  ```

  in the Transcript window,
- compile all files in the correct order (Auto Generate available),
- simulate tb_soc (Library tab, work library, 'without Optimization' if default setting hasn't been changed before)
- if wanted, check under the Memory List tab that all memories are visible (path/mem)
- in the Transcript window, enter

  ```
  do wave.do
  ```

  and

  ```
  do start.do
  ```

  (second Return needed, or clock OK)

- now the simulation can be Run ….

[1]) If totally disabling optimization is not an option, selective visibility of the memories can be accomplished by starting vsim with -voptargs="+acc=v+/path_to_the_memory/mem commands. See the Model*Sim* documentation.

Note:

Displaying waveforms for large memory blocks in Model*Sim* may severely slow down working with the waveform viewer. In the before mentioned `wave.do` file the lower level parts of memory blocks are omitted. On the other hand, (only when simulating, not synthesized) 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).

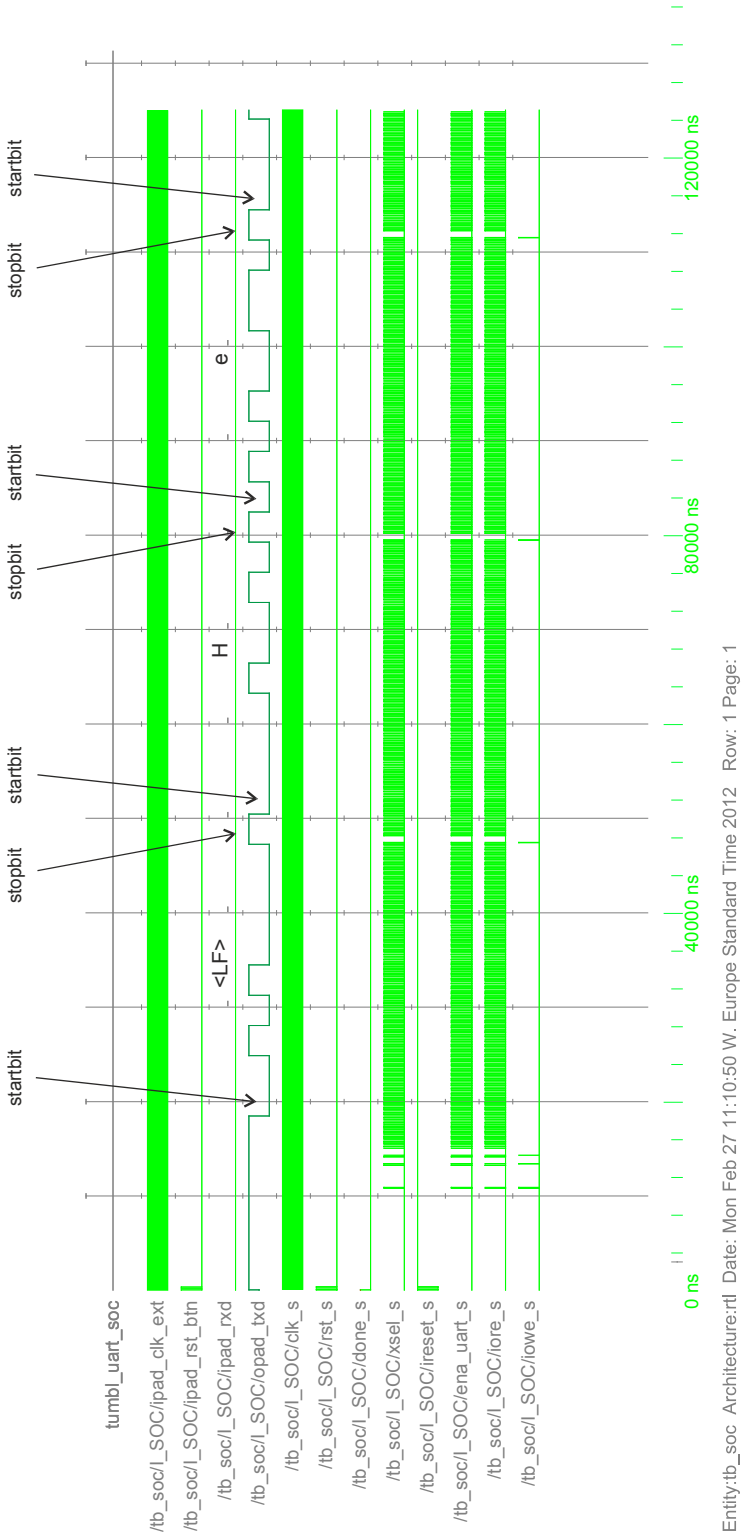


Figure 2.1 *ModelSim wave output showing the first characters transmitted by the uart (by lowering the division factor in uartAVR8.h to 10, simulated baudrate is about 300 kbit/s).*

2.4 Synthesis and Implementation

In the `designs/hello/synth/`-directory, 2 files will already be present, viz.

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

The project file can be created by building the circuit in the Synplify GUI, but it may be far easier to simply edit a (template) project file. The `.prj`-file given here can be a good starting point.

When previously a `make synth` has been issued from the `sw`-directory, a file called `tumbl_#kB_#kB.bmm` can be found here, where `#` indicates the sizes in Bytes of program and data memory respectively.
This file is needed by the Xilinx tools, and –for being recognized needs to be copied to the revision directory and to be renamed to `"the_name_of_the_edf_file".bmm` (`tumbl_uart_soc.bmm` for this example).
Its contents are used to indicate how exactly the `imem_dmem.mem`-file data that –also by the `make synth` command copied into the revision directory- should be located into the FPGA's BRAMs (see also Figure 1.3).

It is supposed here that synthesis with Synplify, using the project file given, has been accomplished and that, let's say `rev_1` (the directory for the XC3S2000 implementation) has been created and has been written to by the synthesizer.
This `rev_1` will also the working directory for the Xilinx tools.
To run these tools, next to the `.bmm`-file mentioned before, 3 more files will be needed, viz.

- `AVNET_DK_xc3s2000.ucf`, which lists the pin definitions for the AVNET Development Kit,
- `makebit_bmm`, which is a script for generating a bit file using the Xilinx ISE tools, and
- `makemem`, which is a script for writing or updating the BRAMs reserved for `imem` and `dmem` in the previously created or current bit-file.

The commands for executing the scripts for this example will be:

```
./makebit tumbl_uart_soc AVNET_DK_xc3s2000
```

If successful, to be followed by

```
./makemem tumbl_uart_soc imem_dmem.mem
```

The result will be a completely programmed `.bit`-file, by default named `program.bit`, that can be uploaded into the fpga with Xilinx's iMPACT.

In `designs/hello/synth/rev_1/` already a prefabricated, reference `hello.bit` will be present.

Notice, that once a correct `bit`-file exists (i.e. the implemented hardware is believed to function correctly), and when only changes in the software have been made, it suffices to only run `make synth` and `makemem` as mentioned above.

The designs/hello/synth/rev_2/-directory already contains files for the LX9 MicroBoard:

- AVNET_DK_xc3s2000.ucf, which lists the pin definitions for the MicroBoard,
- the makebit_bmm and makemem scripts mentioned before, and
- the hello.bit reference for this LX9 MicroBoard.

Notice that the description of the Xilinx memories refers to RAMB16_S36, RAMB16_S9 and RAMB16_S36_S36 Block RAM Library primitives. These are completely valid for the Spartan 3 device on the AVNET Development Kit. The Spartan 6 on the MicroBoard in fact prefers to use the more elaborate RAMB16BWER and RAMB8BWER primitives.

Fortunately, the Xilinx ISE tools know how to translate the RAMB16_S# primitives into the correct RAMB16BWERs, at the penalty of a warning issued for each translation needed (so expect to see 19 warnings for a design with 16 kB imem and 16 kB dmem when makebit_bmm is executing).

2.5 Final Test

Xilinx's iMPACT can be used for transferring the bit-file to an FPGA.

Figure 2.2 shows a screenshot of a TeraTerm window after programming a MicroBoard with the hello.bit file.

In the Terminal Setup, a LF was assigned to be the newline character when receiving, and a CR for transmitting.

The serial port has been configured for 19200 Baud, 8 data bits, no parity and 1 stop bit.

The com-port number itself depends on the FPGA board used and whether or not special drivers are needed.

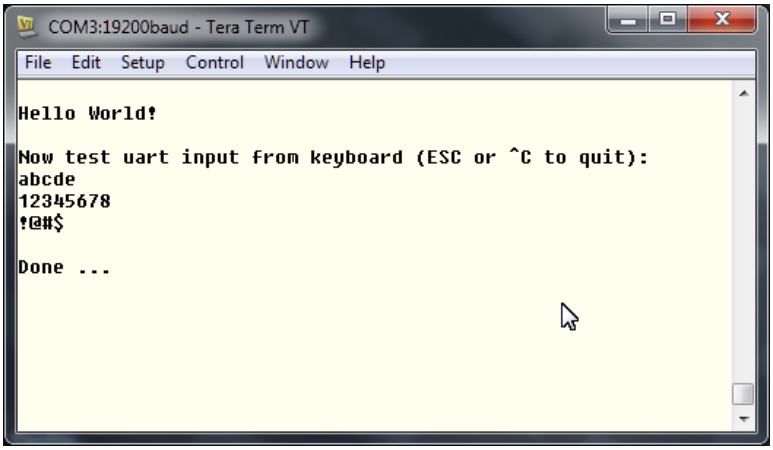


Figure 2.2 *Screenshot of a TeraTerm window after programming the LX9 MicroBoard with* hello.bit

3 SW Test

This example is a somewhat extended version of the `sw_testbench`, described by Tamar Kranenburg in his Master of Science Thesis "Design of a Portable and Customizable Microprocessor for Rapid System Prototyping", *CAS-MS-2009-13,* available from http://opencores.org/project,mblite,overview

The procedure that has been followed is equivalent with the one described in the previous example.

3.1 HDL Setup

First the HDL description is completed. Now, although for this software test again only one external uart will be used that can be selected by the `XMEMB_sel_o` signal, here the selection is done with the aid of a `dmb_selector` component.

Also, the `tumbl` is extended with a hardware multiplier and a hardware barrel-shift unit by setting the generics `USE_HW_MUL_g` and `USE_BARREL_g` both to `TRUE`.

Thus, in the **`designs/sw_test/`**-directory can be found:

`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)*

Since it is expected that the software testbench will require larger memory sizes, `IMEM_ABITS_g` and `DMEM_ABITS_g` are increased to both 15 bits, i.e. 32 kBytes or 8 kWords of 32-bits each, which will nicely fit in the XC3S2000.

However, the intention is to also run this test on the LX9 MicroBoard, which supports a significantly smaller FPGA device. Therefore, software and synthesis (there is no intention to simulate the complete design until the end) are split in two tracks, one for the complete design on the XC3S2000 and one with a design that lacks the memory consuming Dhrystone test, to run on the LX9 (`IMEM_ABITS_g` and `DMEM_ABITS_g` remain 14 bits in that case).

3.2 Software Setup

For clarity, we will distinguish two separate software directories:
`sw/` will contain the files for the complete test, `sw_6LX9/` for the smaller one.

Since the `tumbl` now is equipped with a hardware multiplier and a hardware barrel shifter, the flags in the `Makefile`(s) are adapted to reflect this in the compiled code by setting the directives

```
HWMUL  = no-xl-soft-mul
BARREL = xl-barrel-shift
```

Then, 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*

and 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 (19200 Bd)*
`uart_AVR8.c`	*low level functions for serial communication*
`uart_AVR8.h`	*description of the uart's registers and BaudRate*

As can be expected, a closer inspection of the files will show only small differences.

3.3 Simulation

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:

`msim.mpf`	*(template) project file for ModelSim*
`wave.do`	*waveform layout definition for this design*

3.4 Synthesis and Implementation

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

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

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

- they are created during the software creation process, or since
- their purpose shall be clear from the previous example.

3.5 Test Results

Figure 3.1 shows a screenshot of a TeraTerm window after programming the LX9 MicroBoard with the software testbench, while Figure 3.2 shows the output of the Dhrystone part of the complete test performed on an XC3S2000.

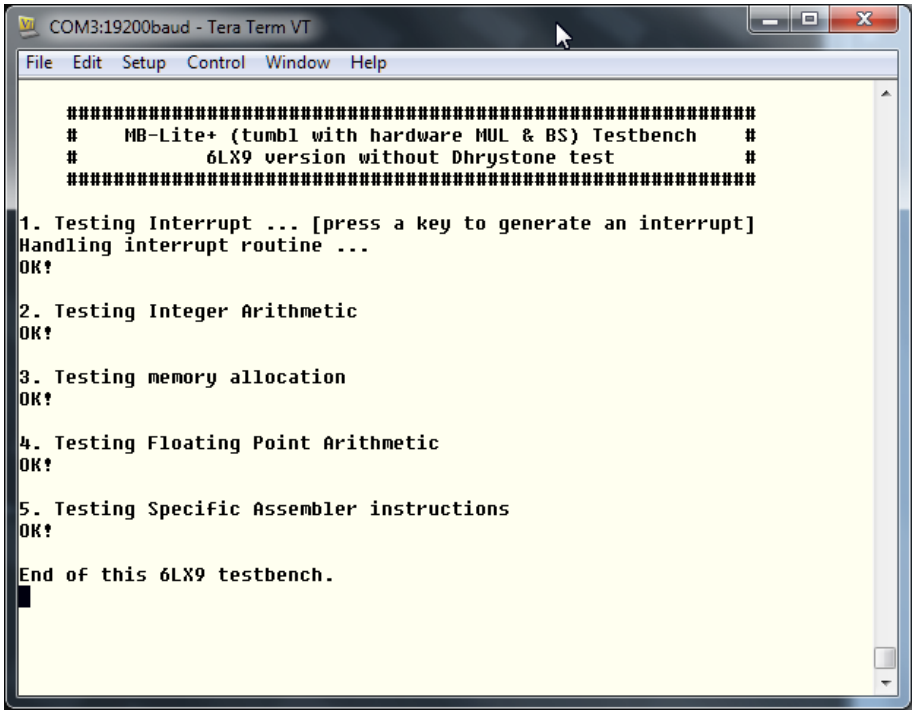


Figure 3.1 *Screenshot of a TeraTerm window after programming the LX9 MicroBoard with* `sw_test_6LX9.bit`

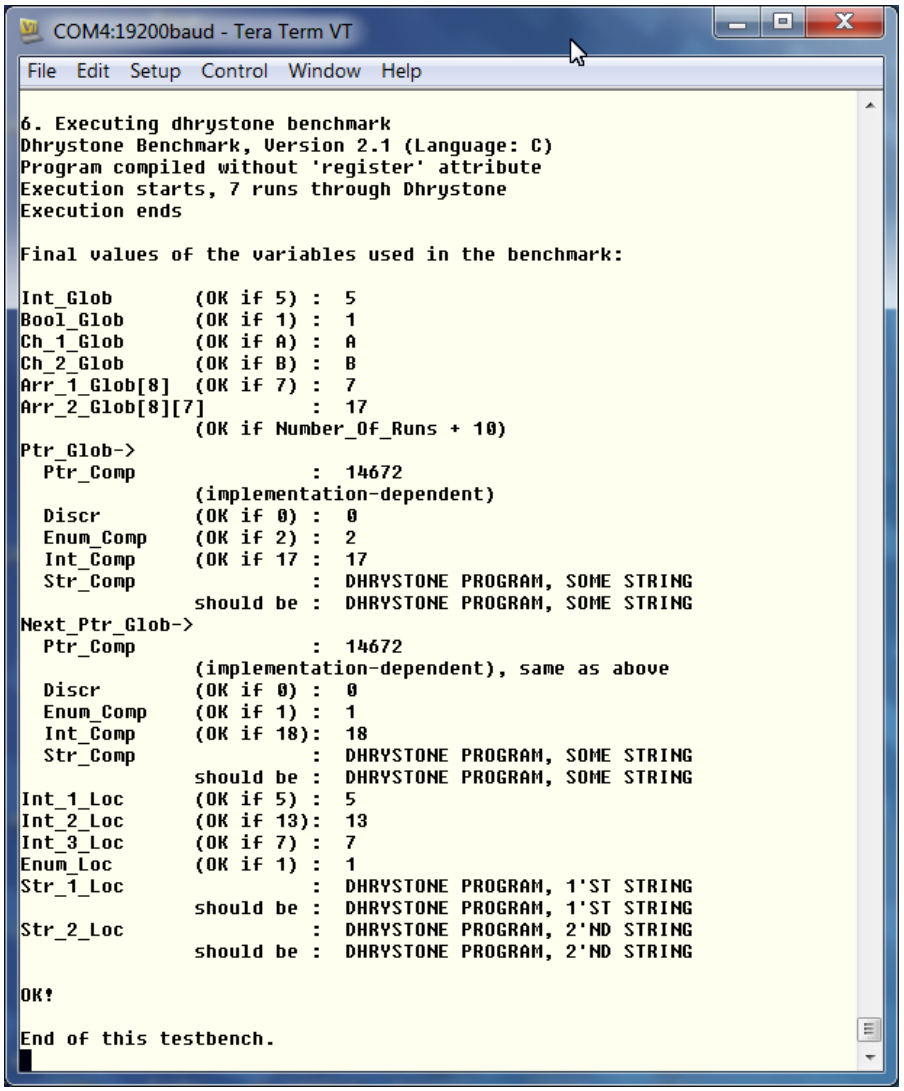


Figure 3.2 *Screenshot of the output of the Dhrystone part of the complete test performed on an XC3S2000*

4 Integer-DCT with FSL

This example describes the interconnection of a `tumbl_fsl` with 1 FSL_M and 1 FS_S port, and a dedicated 8x8 integer-DCT IP block that also communicates by means of FSL ports.
The `tumbl_fsl` writes batches of 8 data values to the iDCT block that performs the DCT transformation, and that returns a batch of 8 results to be read by the `tumbl_fsl`.
Between the `tumbl_fsl`'s FSL ports and those of the iDCT block, two uni-directional FSL bus interface are to be connected, usually consisting of a number of FIFOs. Xilinx provides dedicated FSL_V20 LogiCore elements for this purpose, with depths selectable from 1 to 8k.
For this example, a single, synchronously clocked FIFO has been used, the VHDL code of which is given in `fsl_bb1.vhd`.

4.1 Some Math

The main computational task in the iDCT module is the multiplication of two 8-by-8 integer matrices, e.g. series of multiplications and summations of vector elements.
One of these matrices is hard coded in the VHDL hardware, while the other data matrix will be defined in software.
To give this example a somewhat scientific flavor, the hardware matrix is chosen to be a (scaled version of a) Discrete Cosine Transform (DCT) kernel, defined with

$$C_N^{II} = \sqrt{\frac{\rho_k}{N}} \cos\left[\frac{\pi}{2N}\, k\,(2n+1)\right]$$

$$with \quad k,n = 0,1,\cdots,N-1, \quad N = 8,$$

$$\text{and} \quad \rho_k = \begin{cases} 1, & k = 0 \\ 2, & k > 0 \end{cases}$$

With a scale factor $\alpha = 2^{15}\sqrt{2}$, and rounding to the nearest integer, we define a 2-dimensional array

$$\texttt{COEFF\_MAT\_c} = \alpha A = \alpha C_8^{II}$$

in `fsl_idct.vhd`.
In order to obtain a result from the computations that will be quickly recognizable as being correct or not, we let the software data array to be a (differently scaled) transposed version of the same kernel,

$$\texttt{data\_to\_idct[]} = \beta A^T, \text{ where } \beta = 2^{15}.$$

Since the DCT kernel is orthonormal by definition, such that $A^T = A^{-1}$, and since

$$\mathbf{A}\, A^{-1} = I, \text{ we expect to obtain}$$

$$\alpha\mathbf{A}\, \beta A^T = \alpha\beta\, I,$$

e.g. a scaled version of an 8x8 Identity Matrix.

In plain text, if we correlate functions that have identical shapes with each other, we expect maximum correlation. If we also know that the several functions are orthogonal to each other, we know that all cross-correlations will be zero. The result from our matrix multiplication then will be a matrix with (relatively large) positive values on the main diagonal and zeros elsewhere (I').

However, since we are working with rounded, integer data, the resulting output won't be ideal and we may expect to find (small) values differing from zeros off-diagonal and the values on the diagonal to also show small differences. In the example, the output values are again scaled, such that the smallest values differing from zero are + or −1.

The final result can be described with

$$\frac{I'}{\gamma} \quad with\ \gamma = 2^{12}$$

Since all input data values and all transform data values are less than 16-bit wide, a single MUL18x18S multiplier element in case that the XC3S2000 is used or a single DSP48A1 in case of the Spartan 6, will be inferred.

While all intermediate results (additions) will be calculated using a 32-bit (or even 48-bit in case of the DSP48A1) data bus (`fsl_idct.vhd`), overflow errors can't occur. Given the scaling factors mentioned, the final output returned to the `tumbl_fsl` will need 20 significant bits at most.

4.2 HDL Setup

In the **`designs/fsl_idct/`**-directory, the already familiar `sys_ctrl.vhd` can be recognized, together with the dedicated top level `fsl_idct_uart_soc.vhd` (50 MHz clock, both generics `N_FSL_M_g` and `N_FSL_S_g` set to 1).

Next to these, three more IP files are to be found, viz.

`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*

Block diagrams of the `fsl_bb1` and `fsl_idct` components showing their i/o connections are given in Figure 4.1, while Figure 4.2 shows their interfacing to the `tumbl_fsl`.

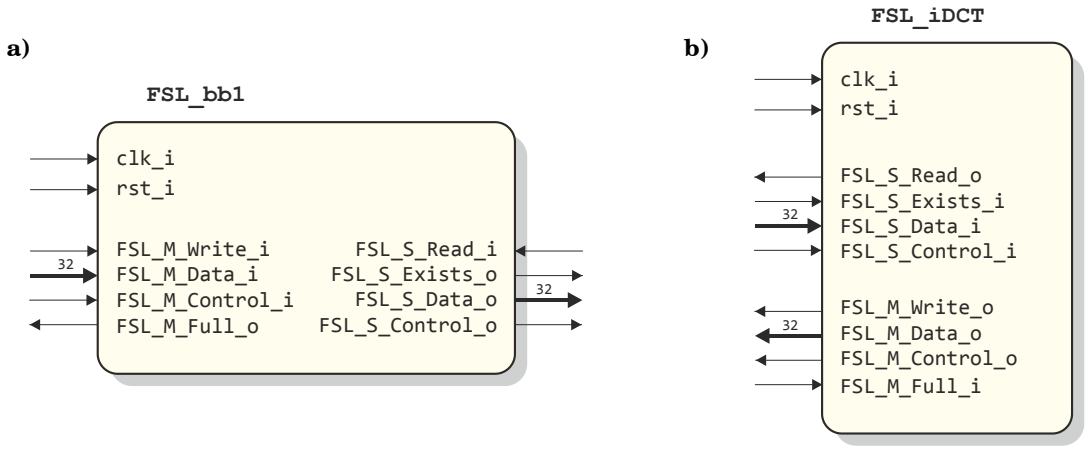


Figure 4.1 *Block diagrams of* **a)** *the* `fsl_bb1`*, and* **b)** *the* `fsl_idct` *components.*

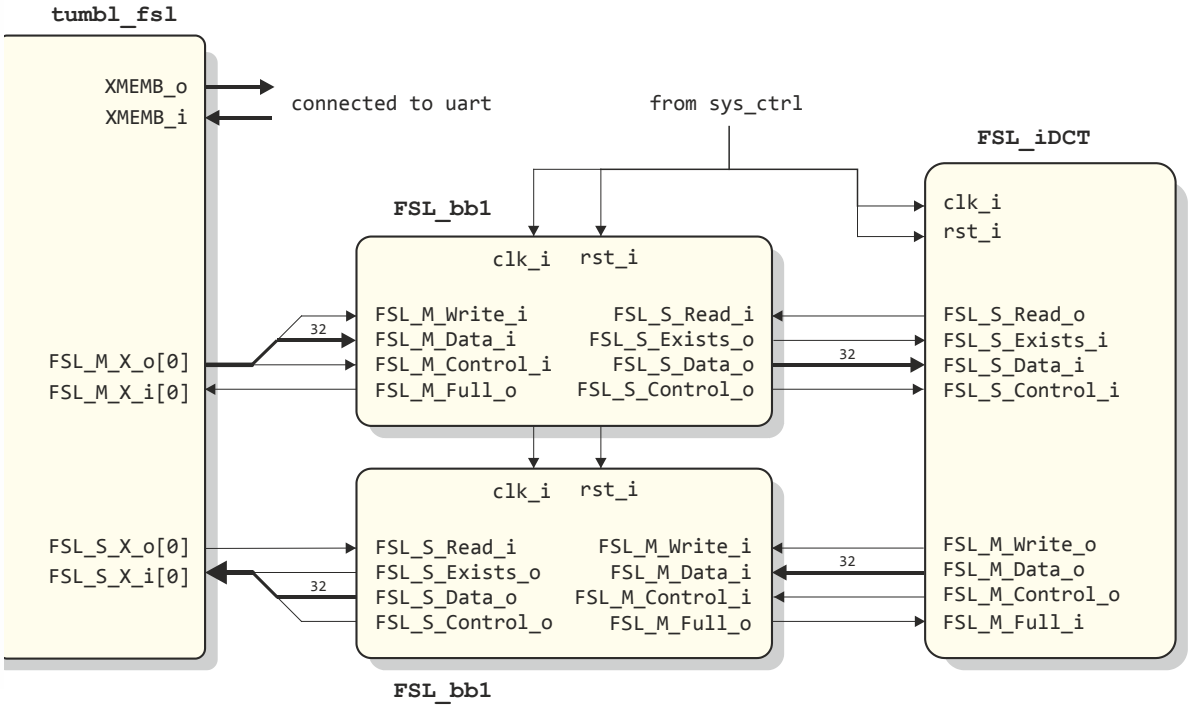


Figure 4.2 *Setup of the* `tumbl_fsl` *with the* `idct` *module and two bus interfaces.*

4.3 Software Setup

In the `designs/fsl_idct/sw/`-directory, the already familiar files can be found:

`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*

together with two `c`-files:

`fsl_idct.c`, which is the source file used for synthesis including uart *(19200 Bd)* output, and `fsl_idct_msim.c`, which is a special version without printout for faster simulation and testing.

4.4 Simulation

The testbench `tb_soc.vhd` can be found in the `designs/fsl_idct/tb_msim/`-directory, as expected, with again the `make_mpf.do`, `msim.mpf` and `wave.do` for this design in the `designs/fsl_idct/tb_msim/msim/`-directory.

4.5 Synthesis and Implementation

The `synth.prj` project file referring to the VHDL files needed for this example is again written in the `designs/fsl_idct/synth/`-directory, with the prefabricated `fsl_idct.bit` files, respectively for the XC3S200 Development Kit in `designs/fsl_idct/synth/XC3S2000/`, and for the LX9 MicroBoard in `designs/fsl_idct/synth/6LX9/`.

4.6 Test and Verification

In the `designs/fsl_idct/matlab/`-directory, two files will be present:

`check_idct.m`, which is a Matlab script file to calculate the results that have to be expected, and

`check_idct.out`, which is a text file created with the above mentioned `m`-file showing final and intermediate results to compare with simulation results (see Figure 4.4).

Figures 4.3 to 4.5 respectively show a screenshot of the last part of the output text, simulation output (compared with reference values) and a listing of the complete text output.

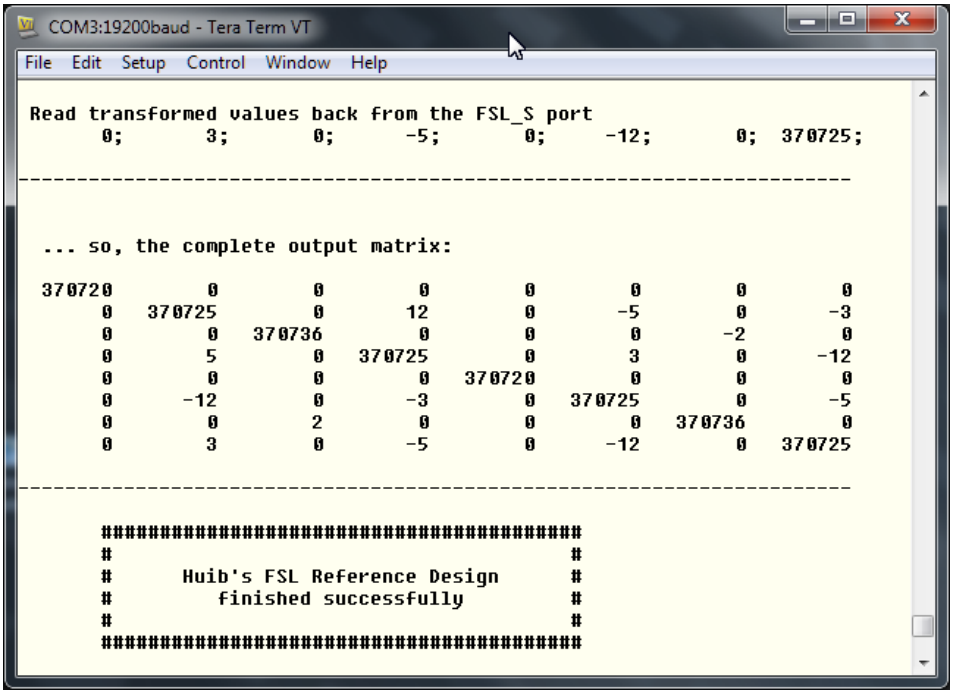


Figure 4.3 *Screenshot of a TeraTerm window showing the last part of the output stream.*

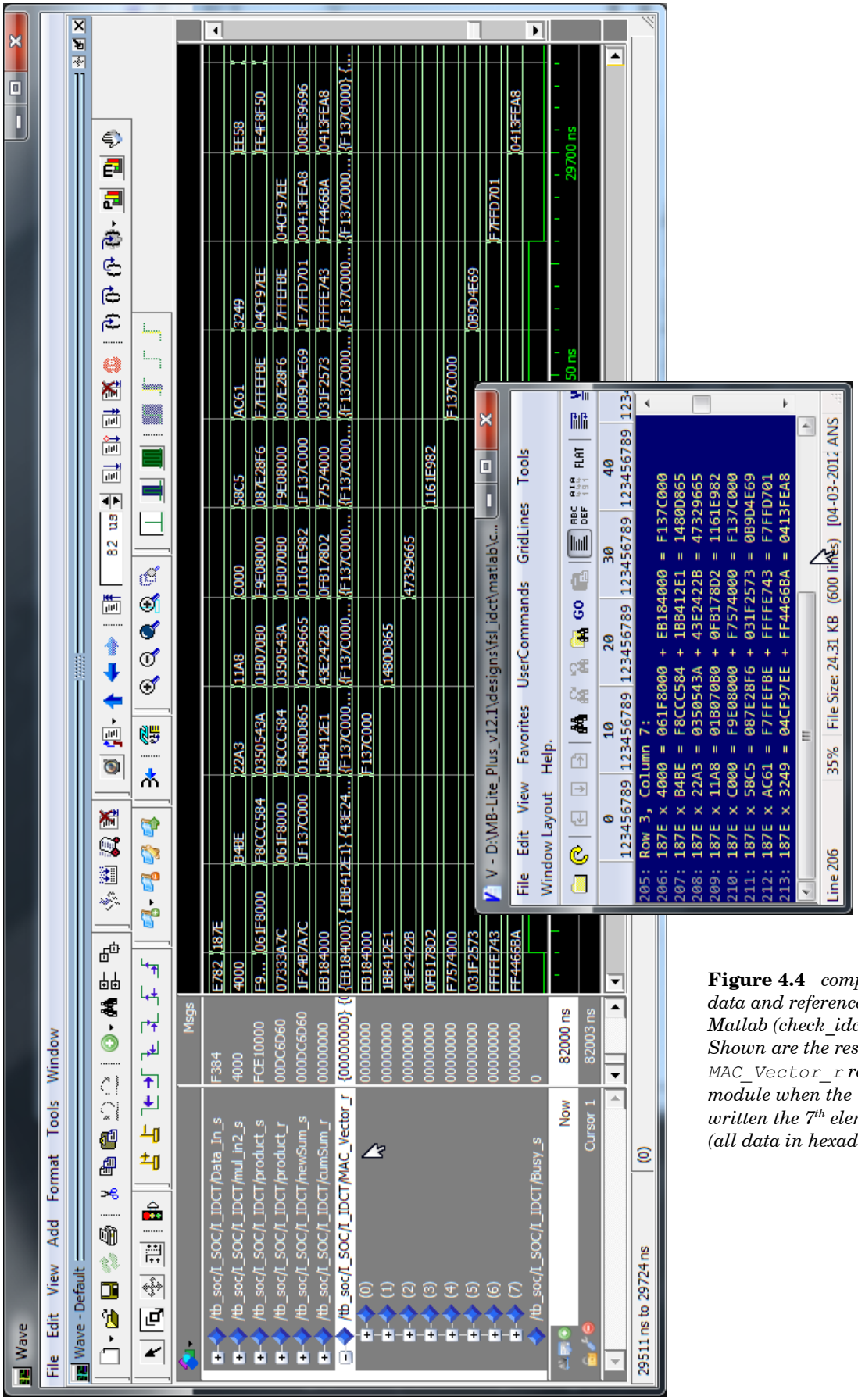


Figure 4.4 *comparison of simulation data and reference data obtained with Matlab (check_idct.out). Shown are the results of the* `MAC_Vector_r` *register in the iDCT module when the* `tumbl_fsl` *has just written the 7th element of the 3rd row (all data in hexadecimal format).*

```
       ##########################################
       #                                        #
       #       Huib's FSL Reference Design      #
       #    with "tumbl_fsl_M_S" and "fsl_idct"  #
       #                                        #
       ##########################################


 Compute 1st datablock out of 8

 Write input values to the FSL_M port
 11585; 11585; 11585; 11585; 11585; 11585; 11585; 11585;

 Read transformed values back from the FSL_S port
  370720;       0;       0;       0;       0;       0;       0;       0;

------------------------------------------------------------------------

 Compute 2nd datablock out of 8

 Write input values to the FSL_M port
 16069; 13623; 9102; 3196; -3196; -9102; -13623; -16069;

 Read transformed values back from the FSL_S port
       0;  370725;       0;      12;       0;      -5;       0;      -3;

------------------------------------------------------------------------

 Compute 3rd datablock out of 8

 Write input values to the FSL_M port
 15137; 6270; -6270; -15137; -15137; -6270; 6270; 15137;

 Read transformed values back from the FSL_S port
       0;       0;  370736;       0;       0;       0;      -2;       0;

------------------------------------------------------------------------

 Compute 4th datablock out of 8

 Write input values to the FSL_M port
 13623; -3196; -16069; -9102; 9102; 16069; 3196; -13623;

 Read transformed values back from the FSL_S port
       0;       5;       0;  370725;       0;       3;       0;     -12;

------------------------------------------------------------------------

 Compute 5th datablock out of 8

 Write input values to the FSL_M port
 11585; -11585; -11585; 11585; 11585; -11585; -11585; 11585;

 Read transformed values back from the FSL_S port
       0;       0;       0;       0;  370720;       0;       0;       0;

------------------------------------------------------------------------

 Compute 6th datablock out of 8

 Write input values to the FSL_M port
 9102; -16069; 3196; 13623; -13623; -3196; 16069; -9102;

 Read transformed values back from the FSL_S port
       0;     -12;       0;      -3;       0;  370725;       0;      -5;

------------------------------------------------------------------------
```

Figure 4.5 *listing of the output of the* `fsl_idct` *example*

continuing **Figure 4.5**

```
 Compute 7th datablock out of 8

 Write input values to the FSL_M port
 6270; -15137; 15137; -6270; -6270; 15137; -15137; 6270;

 Read transformed values back from the FSL_S port
       0;       0;       2;       0;       0;       0;  370736;       0;

-------------------------------------------------------------------------

 Compute 8th datablock out of 8

 Write input values to the FSL_M port
 3196; -9102; 13623; -16069; 16069; -13623; 9102; -3196;

 Read transformed values back from the FSL_S port
       0;       3;       0;      -5;       0;     -12;       0;  370725;

-------------------------------------------------------------------------


  ... so, the complete output matrix:

  370720        0        0        0        0        0        0        0
       0   370725        0       12        0       -5        0       -3
       0        0   370736        0        0        0       -2        0
       0        5        0   370725        0        3        0      -12
       0        0        0        0   370720        0        0        0
       0      -12        0       -3        0   370725        0       -5
       0        0        2        0        0        0   370736        0
       0        3        0       -5        0      -12        0   370725

-------------------------------------------------------------------------

       ########################################
       #                                      #
       #      Huib's FSL Reference Design     #
       #         finished successfully        #
       #                                      #
       ########################################
```

5 Memory Mapped Slaves and Slave Emulators

Here, a tumbl will be connected to a number of modules that emulate slave devices using memory mapped registers for data communication (tumbl_slaves_ex_SoC). Each of these special slaves is intended to emulate an operation that takes an adjustable number of clock cycles.
Also connected are the uart and a memory mapped register to enable software control of LEDs when present on a pcb.

5.1 Some thoughts about slaves

We will distinguish three kinds of peripheral slaves, viz.

- slaves that are completely located on the same FPGA or ASIC as the tumbl, with or without external connections off-chip,
- slaves with their bus interface on the same chip as the tumbl, and another part located in a dedicated component or electronics off-chip, e.g. an Ethernet controller, an LCD driver, etc., and
- slaves completely off-chip with a bus and bus drivers between the tumbl's SoC and the slave.

Here, we will shortly discuss the second type mentioned, while the example is focussed on types of the first kind.

Figure 5.1 shows block diagrams of resp. an Xmb_slave_interface (with the X representing either an 'a' for an asynchronous or an 's' for a synchronous interface) and a wishbone slave interface.
The amb_ and smb_slave interfaces are defined with the same records and port names as used with the dmb_adapter, the wishbone interface is shown with the port names defined in the Wishbone Specs document.
The _slave_o output and _slave_i input represent records connecting to pads, specific for interfacing the off-chip component.

Communication with the tumbl usually takes place by reading or writing to registers in the slave: data registers, as well as control and status registers. These registers may either be direct copies or registered versions of the off-chip components records, or be adjusted or combined ones, e.g. to obtain a certain data bus width. In some cases, straight through connections to the off-chip component will be possible.
Usually these registers are consecutively mapped in the slaves address space. Since the tumbl is a 32-bit processor, slaves with a different data bus width need some kind of address mapping as explained below.

The data bus width will generally be determined by the slave's properties, and is usually but not necessarily a multiple of 8-bits (given in MW_DBITS_g or WB_DBITS_g). Referring to the section about data alignment in the MB-Lite+ User Guide, and noticing that the software recognizes WORDS, HALFWORDS and BYTES, it seems reasonable to align a 16-bit slave with data lines d15--d0, and an 8-bit slave with data lines d7--d0, i.e. the least significant bit of the slave's data will always be connected to d0.
Data busses that are not multiples of 8, 16 or 32 bits will have to be –for a processor read- extended to resp. BYTES, HALFWORDS and WORDS by prepending zero bits.

This implies that the data registers of an 8-bit (or less) slave, seen from the tumbl, will then be accessible at addresses with their least significant nibbles equal to 3, 7, b or f (hex).

Data of 16-bit slaves will be located at addresses with least significant nibbles 2, 6, a or e (hex), and of 32-bit at addresses with least significant nibbles 0, 4, 8 or c (hex).
This is accomplished by the address mapping in the `dmb_` and `wb_`adapters given in the release package. This way, the hardware is very simple at the expense of unusable addresses.
Should this be an issue, then the use of 32-to-8|16 and 8|16-to-32 multiplexers have to be considered.

Other choices would make the results of e.g. simulator output, especially when working with a (signed or unsigned) decimal data format, more difficult to understand.
If needed, the `bSel` lines can be used to select individual bytes or combinations of bytes.

It should be clear that always the number of address bits (given in `MW_ABITS_g` or `WB_ABITS_g`) should be enough to handle the number of registers to be accessed.

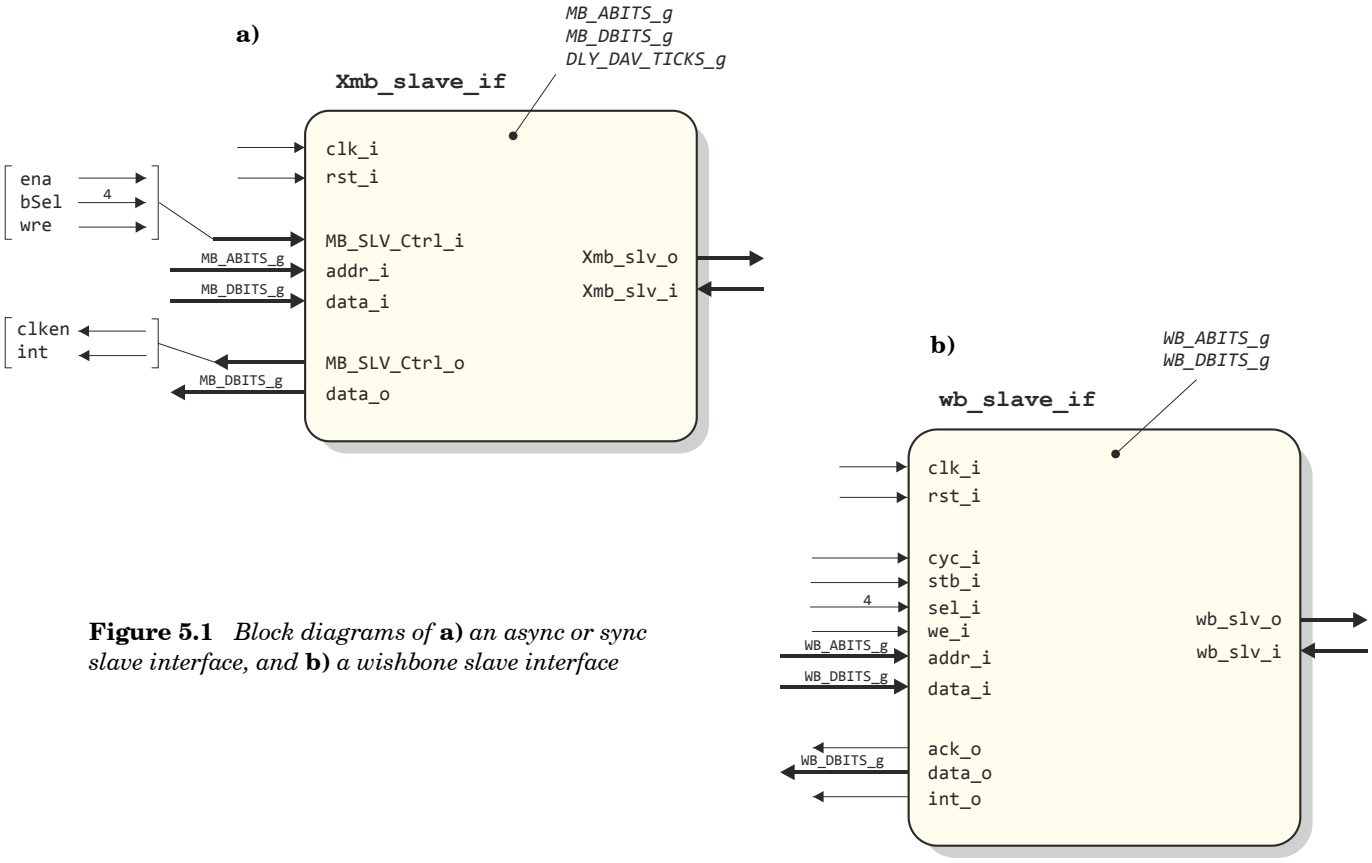


Figure 5.1 *Block diagrams of* **a)** *an async or sync slave interface, and* **b)** *a wishbone slave interface*

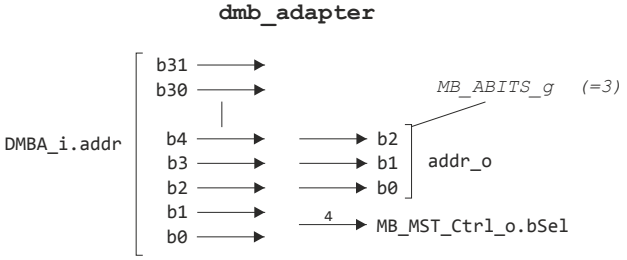


Figure 5.2 *Address mapping between* `tumbl` *and memory mapped slaves in the* `dmb_` *and* `wb_adapter`*s.*

If the interface's data registers are located on the same chip as the `tumbl`, there is no reason why e.g. setup and hold times would differs from those as in the `tumbl` hardware itself, so reading and writing won't take more than one clock cycle.
Off-chip components usually demand different (longer times are assumed) values for data read/write operations or for the component's data becoming valid. This can be dealt with using the generics `SET_UP_TICKS_g`, `HOLD_TICKS_g` and `DLY_DAV_TICKS_g` that can lengthen these times, expressed in multiples of clock cycles (see Figure 5.3).
Note that `SET_UP_TICKS_g` and `DLY_DAV_TICKS_g` will stall the processor for a number of cycles, while `HOLD_TICKS_g` will need the instantiation of a `pulse_extender` component.

Since a Wishbone slave communicates using a handshake signal (`ACK`) to signal that it is ready to accept data or to signal that it outputs valid data on the data bus, such a slave doesn't need a priori knowledge about setup and data-valid times. An extended hold time may still be needed.

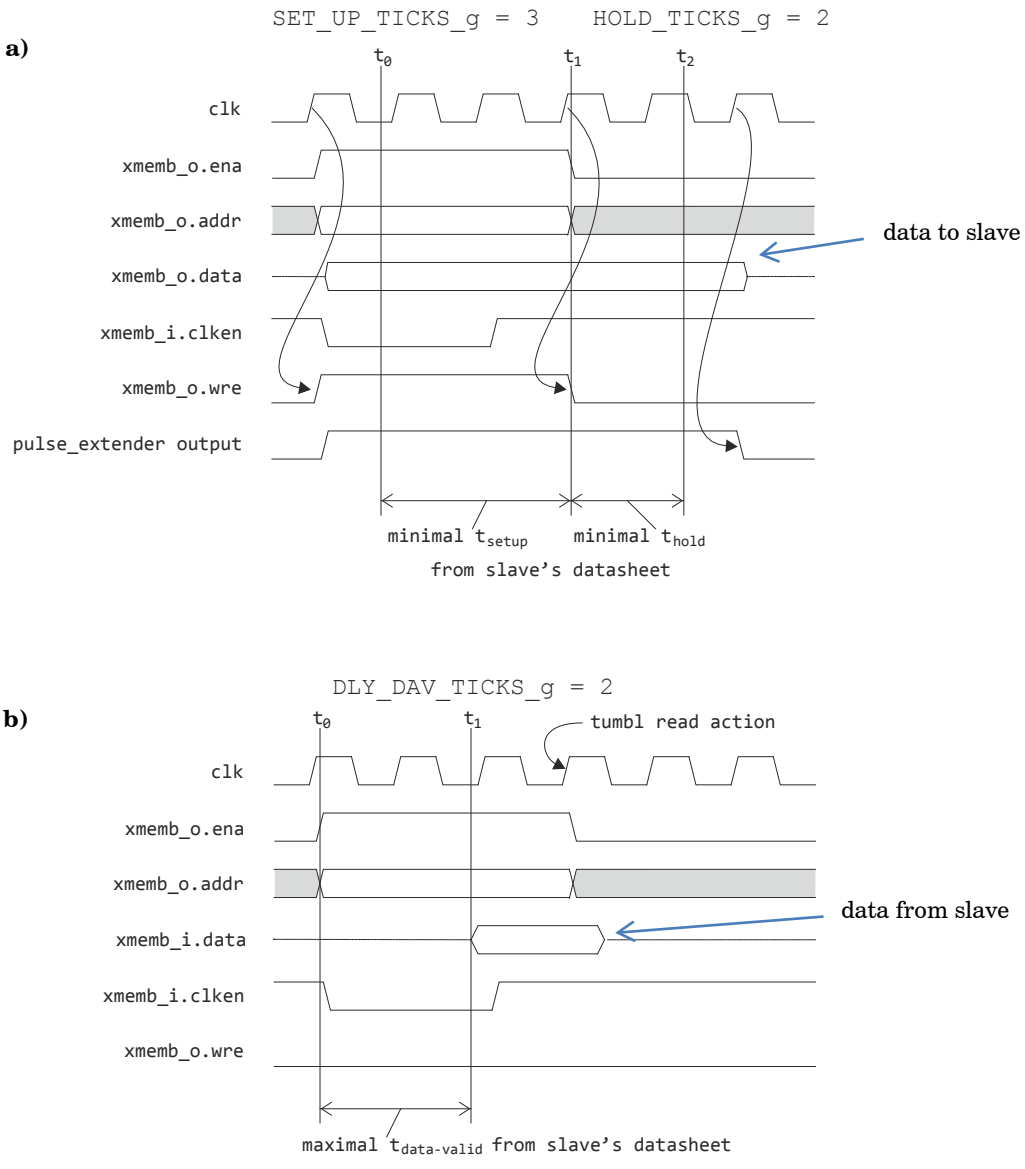


Figure 5.3 *Examples of specific timing needed for* **a)** *writing to, and* **b)** *for reading an off-chip component.*

5.2 Setup of the tumbl_slaves_ex_SoC

In this design example a number of slave emulators will be instantiated that are completely located on the same FPGA as the `tumbl`, viz. an asynchronous slave, a synchronous slaves and 3 wishbone slaves. Non of these contains external connections off-chip.
For each slave all parameters are individually adjustable to investigate all thinkable situations. Next to that, each slave can be programmed to emulate some kind of “time consuming” operation.

To complete the SoC, the already familiar uart is added, together with a special memory mapped write-only register intended as a software controllable LED driver.

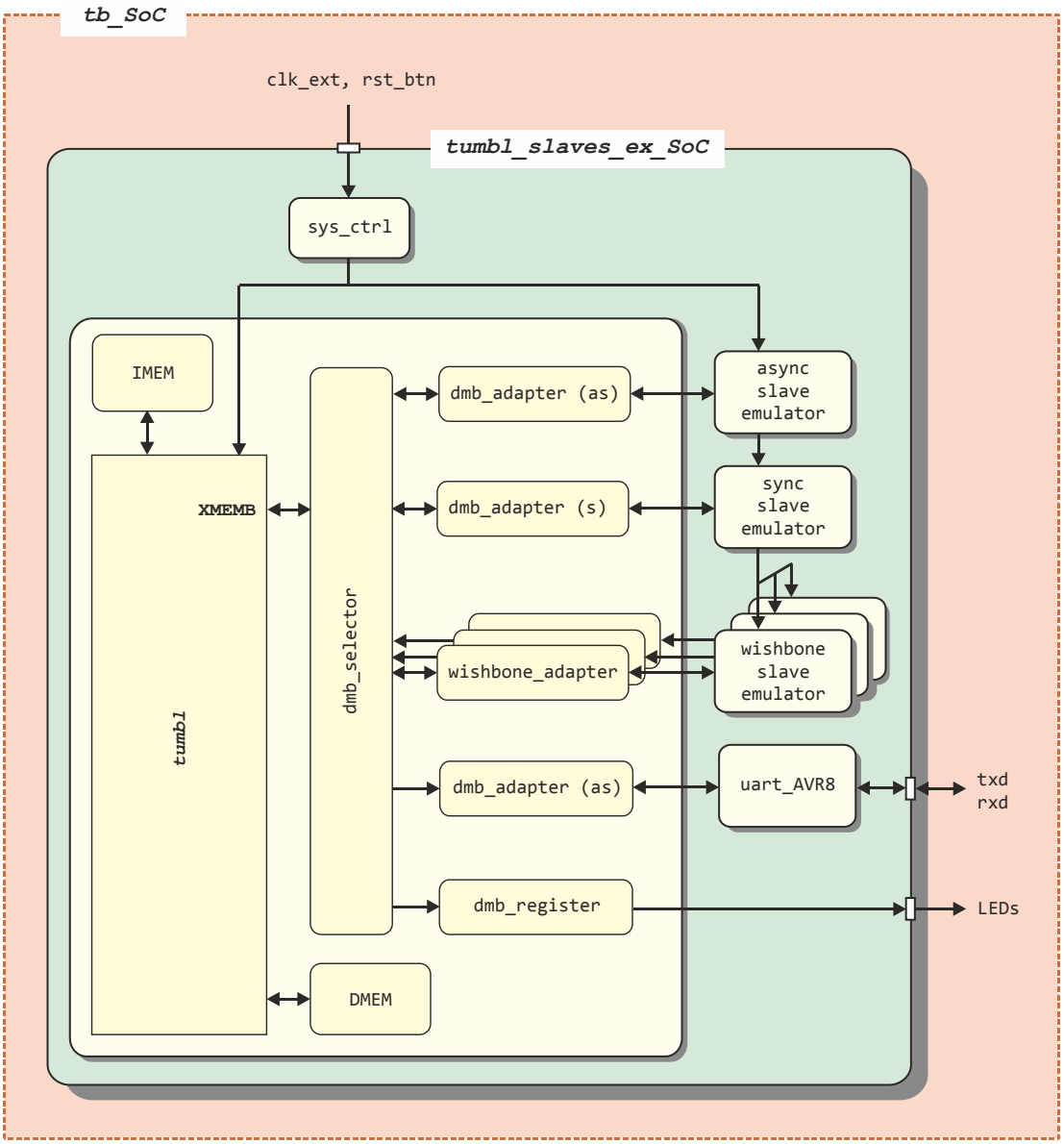


Figure 5.4 *Block diagrams of the* `tumbl_slaves_ex_SoC` *and its testbench.*

5.3 HDL Setup

In the **designs/slaves_ex/**-directory, the already familiar `sys_ctrl.vhd` can be recognized, together with the dedicated top level `tumbl_slaves_ex_SoC.vhd` (50 MHz clock).

Next to these, the slave emulator files are to be found with a helper package, viz.

`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*

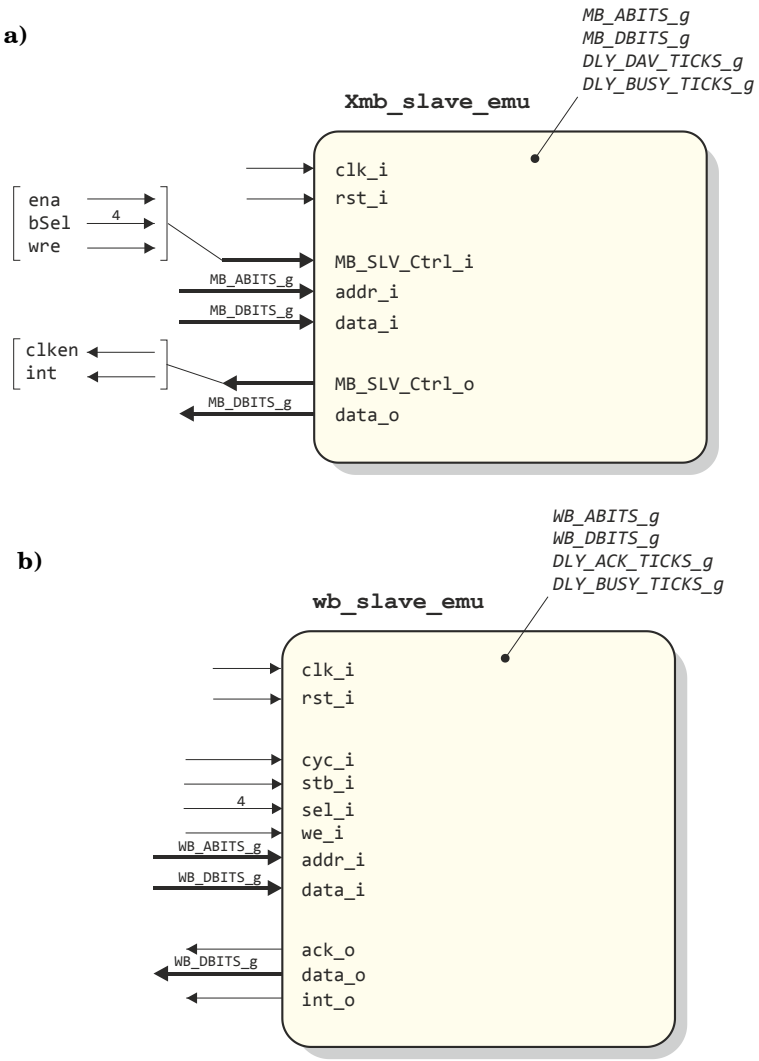


Figure 5.5 *Block diagrams of* **a)** *the* `Xmb_slave_emu` *where* `X` *can be* `a` *or* `s`*, and* **b)** *the* `wb_slave_emu` *components.*

In contrast to the `wb_slave_if` in Figure 5.2, the `wb_slave_emu` here needs an additional generic `WB_DLY_ACK_TICKS_g` to indicate its read/write reaction timing properties.

Each of the slave emulators contains a number of read/write registers, together with a control (`Ctrl`) and a status (`Stat`) register (see the comment header in the VHDL files and the definitions in the software `.h`-files for the details).

As mentioned before, the slaves can simulate being busy with performing an operation that takes a number of clock cycles, given with `DLY_BUSY_TICKS_g`.

This can be accomplished by setting a particular Start-bit in `Ctrl` (see Figure 5.6, writing `01` to address `00` raises `Start_s`), after which a counter that has been preset to the value of `DLY_BUSY_TICKS_g` (17 decimal here) is decremented until it reaches `0`.

This situation can be detected either by polling `Stat` (address `01`, data read `00` when the slave is still busy, `01` when ready) or by acknowledging the `int_o` interrupt signal.

More slaves can be busy at the same time, while all register remain accessible.

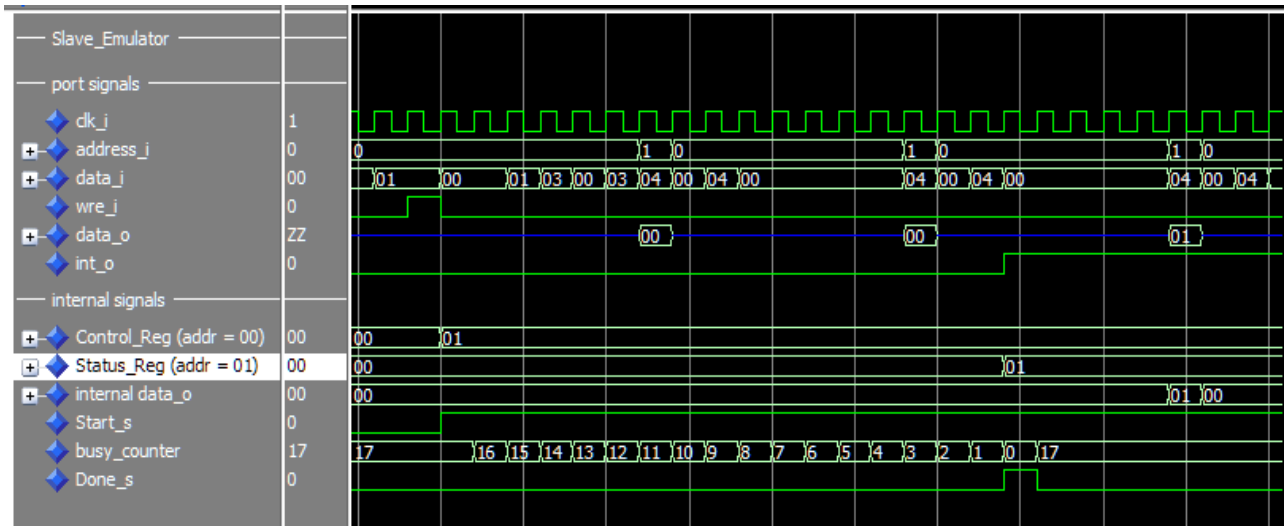


Figure 5.6 *Showing the emulation of a "busy" slave for a `DLY BUSY TICK g` value of 17.*

The `dmb_reg` (`dmb_reg.vhd`) that will be used for controlling the LEDs can be found in the **`hdl/dmb_ext/`**-directory.

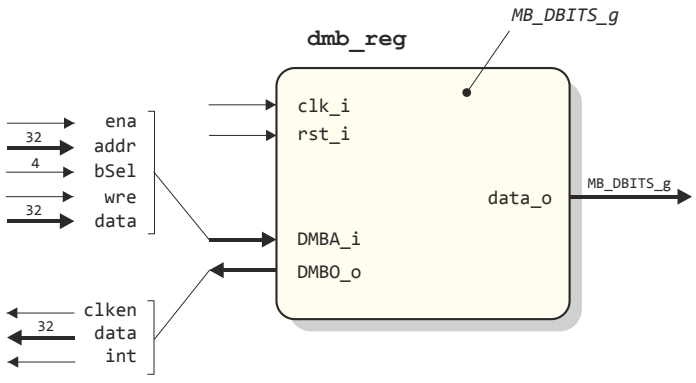


Figure 5.7 *Block diagrams the `dmb_reg` component.*

In the top level tumbl_slaves_ex_SoC.vhd, all generics needed for specifying the circuitry are listed and given values to be used for synthesis, so it will be very easy to experiment with different settings.

In the example, the data busses for the slaves are deliberately given different values, even not being multiples of 8-bit widths, to check the truncations and prepending of zeros when transferring data between the slaves.

The VHDL code describing the slave emulators, can be perfectly used as the starting point for a real slave interface by deleting and/or rewriting the DLY_ACK_TICKS_g and DLY_BUSY_TICKS_g dependable parts.

5.4 Software Setup

In the designs/slaves_ex/sw/-directory, the familiar files can be found:

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*

together with two c-files:

slaves_ex.c, which is the source file used for synthesis including uart *(19200 Bd)* output, and
slaves_ex.c, which is a special version without printout for faster simulation and testing.

Next to these, there is a header file for each of the remaining slaves with information about the addressing of that particular slave and all other specific information needed, viz.

amb_slv1.h, smb_slv2.h, wb_slv3.h, wb_slv4.h, wb_slv5.h and dmb_reg.h

In slaves_ex.c and slaves_ex.c rather arbitrary data values are written to slave registers, register contents are read back and written to other slaves.
Slaves 2 and 3 are instructed to emulate some action (shown with LEDs, see Figure 5.9).
Please, look closely at the comments in these files to see what has to be expected.

5.5 Simulation

The testbench tb_SoC.vhd can be found in the designs/slaves_ex/tb_msim/-directory, as expected, for generating the clock and the reset stimuli signals, clk_ext_tb and rst_btn_tb respectively. Furthermore, the values given to the generics here overrule those in tumbl_SoC for synthesis, and can be used to quickly inspect the effects of different choices.
In designs/slaves_ex/tb_msim/msim/ again, the make_mpf.do, msim.mpf and wave.do (no optimization to show all signals) for this design can be found.

5.6 Synthesis and Implementation

The synth.prj project file referring to the VHDL files needed for this example is again written in the designs/slaves_ex/synth/-directory.

In this particular case, only the `slaves_ex.bit` file for the LX9 MicroBoard is given due to the lack of an XC3S200 Development Kit for testing at the moment of writing this part of the manual.
(see `designs/slaves_ex/synth/6LX9/`).

5.7 Test and Verification

It is advised to have a close look at the simulator output and investigated the effects of different parameter values.
Shown below are finally a screenshot of the terminal output after programming the FPGA, and the indications given by the LEDs on the LX9 MicroBoard.

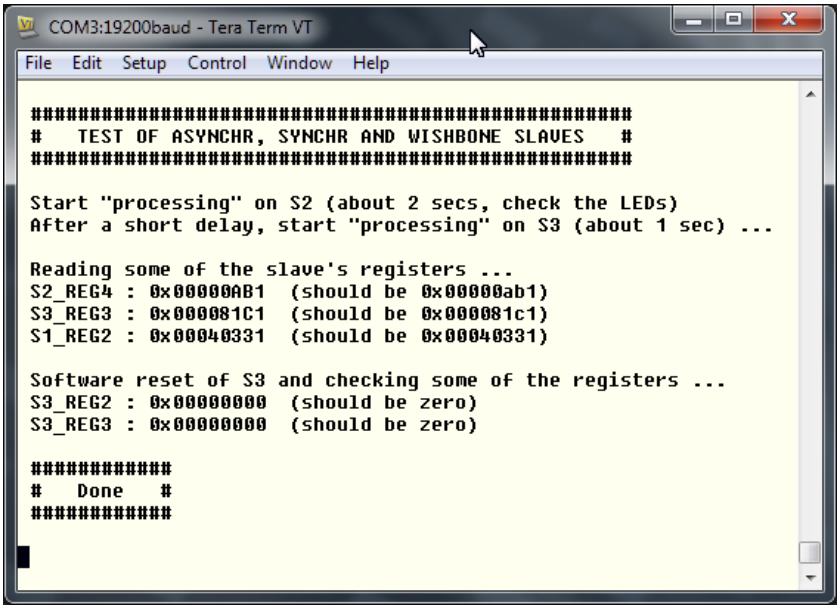


Figure 5.8 *Screenshot of a TeraTerm window showing the output of* `slaves_ex.bit`.

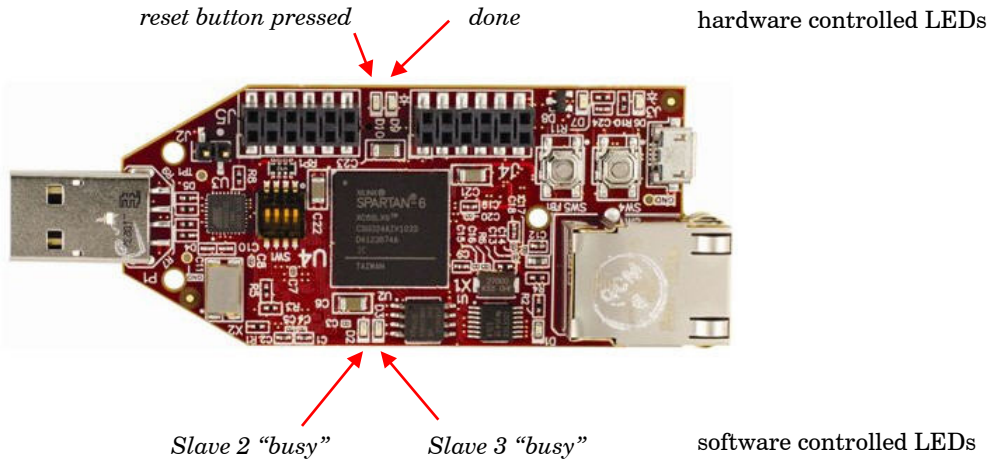


Figure 5.9 *LED assignments on LX9 MicroBoard.*

6 Conclusion

This document described four designs based on an MB-Lite+ processor soft-core.

Example 3 shows the possibility to use the FSL ports for data streaming applications, while Example 4 shows the straight forward way to connect slaves for performing parallel processing tasks.

Except for the last one, all examples have been tested on a Spartan 3 and on a Spartan 6 FPGA, and the implementations worked as expected.

Regarding the examples shown, next to several designs that have been implemented at the Delft University of Technology in the past years, it may be concluded that this particular MB-Lite+ version is a versatile and reliable one.

Appendix

For completeness, a block diagram showing the ports of the `uart_AVR8`, that appears in each of the four design examples, is given below. See the VHDL soc-descriptions for a translation of the port signals to those of the MB-Lite+ and the `uart_AVR8.h` header file for the definition of the registers and the specific bit assignments in these registers.

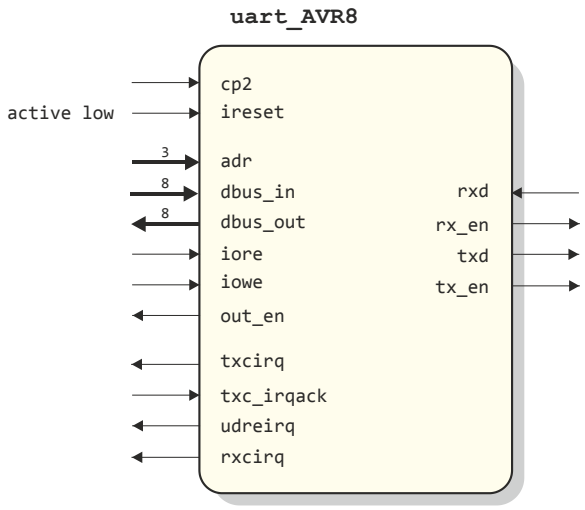


Figure A.1 *Block diagram of the* `uart_AVR8`.

End of Document