IL2206 EMBEDDED SYSTEMS

Laboratory 1: Embedded Computing Platform Design

Version 1.2.2

1 Objectives

This laboratory will help you to

- familiarise with a typical design flow from vendors, e.g. Intel/Altera[1];
- develop an embedded computing platform from specifications; and
- understand the architectural features of your embedded system and how to make use of them in your software applications, e.g.:
  - use the Application Binary Interface (ABI) to embed Assembler functions within C programs;
  - handle I/O with polling and interrupts;
  - access embedded peripherals via C-Macros and the Hardware Abstraction Layer (HAL) System Library;
  - distinguish between different types of memories; and
  - explore application's performance and memory footprint trade-offs.

This laboratory shall give you the necessary skills required to work with the DE2 FPGA board, the Nios II processors, Altera IP blocks, and the related software. This laboratory is split into four main parts which gradually guide the reader into building an own custom embedded hardware platform, write software for it and finally test it. This laboratory is mainly organized as a home laboratory and each student is individually responsible for acquiring the necessary knowledge. The laboratory will be conducted by groups of two students in private, i.e. outside scheduled lab sessions. During the lab session the teaching assistants will mainly assess the knowledge acquired *by each student* in forms of individual discussions, quiz questions and surprise tasks. Both the knowledge and the artifacts generated from this lab will be used in *Lab 2: Introduction to Real-Time Operating Systems*.

The students are strongly encouraged to interact with other students through the course's online platform (Canvas) in case technical issues are encountered. The discussion forum will be monitored by the course assistants, who will take actions in

[1] Altera and IntelFPGA, or Intel in short, will be used interchangeably throughout this lab manual and its references.

case of larger problems. It is very important that students are well-prepared for the labs. Students who are not properly prepared shall not get help from the assistants during the lab sessions!

NOTE: *The preparation tasks shall be documented and shown to the teaching assistants at the beginning of the lab session! Students can work together with their lab partners. However, both students in a group need to demonstrate individual work and proficiency in all tasks. All program code shall be well-structured and well-documented. The language used for documentation is English.*

2 Preliminary Work

Read this laboratory manual in detail and complete the preparation tasks before your lab session in order to be allowed to start the laboratory exercises. The first part of the lab is concerned with setting up the lab environment and learning how to use the many Altera tools. In order to complete this part you need both a working lab virtual machine (VM) installed on your personal computer and a good overview of the reference literature.

2.1 Setting up the Lab Environment

In order to perform the lab, a virtual machine (VM) with all the needed software is provided. On the course home page you will find instructions on how to install and use the VM, as how to test if the lab environment is set up properly. Mind that downloading and installing the VM will take very long time, so you might want to start going through Section 2.2 in the meantime.

lab set up ◯

2.2 Literature

A lot of information that helps to solve the laboratory tasks can be found in the Altera documentation on Nios II [HBb, HBc, UGc], which is available on the course homepage. The amount of information in these documents is huge. Since one goal of this lab is to familiarise you with typical design flows from tool vendors, you are expected to cope with industrial documentation, learn to navigate through the relevant parts of the problem you are solving and make use of it. Often the given references provide example code, for instance how to program the timer to generate “alarm” and how to program the input keys to generate interrupts. The main focus of the reading should be on the following:

- Introduction to the Altera Nios II Soft Processor [TTh].
  Read this thoroughly, it is a very good introduction. A shorter overview is found here [HBb, Chapter 1].
- Nios II “Processor Architecture” and “Programming Model” [HBb, Chapters 2 and 3].
- “My First Nios II Software Tutorial” [TTb] along with “Getting Started with the Graphical User Interface” [HBc, Chapter 2]; or alternatively “Nios II Command-Line Tools” [HBa, Chapter 4].
- Nios II “Application Binary Interface” and “Instruction Set Reference” [HBb, Chapters 7 and 8].
  Especially important are parameter passing and size of datatypes.
  Chapter 8 provides detailed information about all instructions.
- Hardware Abstraction Layer and HAL system library [HBc, Chapters 5, 6, 8, and 14].

Here you can find out which macros to use for reading and writing registers in I/O-circuits: PIO and Timer. You can also find information on how to use HAL-functions to initialise and use interrupts.

- Nios II "JTAG UART Core", "PIO Core", "Interval Timer Core", "System ID Core" and "Performance Counter Core" [UGc, Chapters 6, 10, 28, 33 and 34] Especially I/O-register organization and use in PIO and Timer.
- A couple of system architecture design tutorials such as [TTa], [TTf], [TTe] [HBb, Chapter 4], [TTc], [TTd], [TTg]
  Only [TTa]and [TTf], alternatively [TTe] will be followed during Section 4. The rest can be used as reference.

Details of the Cyclone II or Cyclone IV FPGA boards used in this laboratory are provided in the DE2 Development and Education Board User Manuals [UGb] or [UGa].

Valuable information can also be found in the lecture notes. literature studied ◯

2.3 Mastering the Lab Environment

You are free to choose the working style you feel comfortable with, meaning that within the provided virtual machine (VM) you can install the tools or editors you like in order to deliver the lab tasks. This lab manual gives instructions on *what* is expected of students, as for the *how* it is up to you to figure out in a "learning by doing" style. You might want to try out a few online tutorials to get used to the development flow. The document attached to the provided lab source files contains more instructions on how to set up and use your lab workspace.

At this point you should take some time to try out and get comfortable with the work style chosen. Understand the tools associated with compilation and deployment, and go through some Nios II "Hello World" examples starting from C code down to their implementation on the DE2 board[2]. Use the resource files provided for the lab whenever necessary. "Hello World" app run ◯

3 Input/Output and Interrupts in Nios II Systems

The second part of this lab is a gentle introduction to using the main features of a Nios II processor, with focus on communication with peripherals. The students are provided with a pre-built hardware architecture and the code solutions for all tasks. Depending on your previous experience with the Nios II processor and on how confident you feel in using the Intel/Altera tools you might dedicate as much time as you need for acquiring/consolidating your skills required for this section. Nevertheless we do recommend trying to solve the tasks on your own and only verify your solutions against the provided ones, so that you really gain new *personal* experience.

3.1 C and Assembler

In this lab you will develop a program that will display a time value with different output devices and read from input devices for manipulation. You will start with simply printing out the value into a terminal window.

Under the folder `app/lab1-io` you fill find the source files. Note that you might have to answer questions about the source code during the lab session. You need to be able to explain what every function does. Either set up a new project with those files or use the existing folder hierarchy, and use the provided Nios II hardware architecture as target. Here is a description of the provided source files:

[2] e.g. [TTb] or the example in the lab repository at `il2206-lab/app/hello_world`

puttime.c contains the C-function `void puttime (int* timeloc)` that reads the time value stored at the memory address given by the pointer parameter `timeloc` and prints it to a terminal window. The printout is in the format 42:33.

tick.c contains the C-function `void tick (int* timeloc)` that increments the time value stored at memory address given by the pointer parameter `timeloc` by one.

delay_asm.s contains a Nios-II assembly subroutine which delays program execution by the number of milliseconds given by a parameter.

hexasc_asm.s contains a stub for a Nios-II assembler subroutine which you need to implement, as described below. This subroutine is called from the C-code within the function `puttime`.

lab0.c contains the main loop of the program.

Take a look at the file delay_asm.s. The subroutine `delay` has one input parameter in r4 that determines how many milliseconds the subroutine will wait before returning to its caller. Draw a flow diagram that visualizes how the subroutine `delay` works.

In order to achieve a 1 millisecond delay in the inner loop, you need to adjust the parameter `delaycount`, which is defined at the top of the file and currently set to 0. To calculate the delay-time, use the fact that the processor runs at 50 MHz and check out the cycles per instruction in the altera documentation for NiosII economy class processors. This will give a reasonable starting-value for the delay routine parameter.

NOTE: *For the following part of Section 3.1, it is not mandatory that the students are able to write the assembler code on their own. Instead they can view the solution and gain an understanding on how C and Assembler interact using the Nios II Application Binary Interface [HBc].*

Draw a flow-chart for a subroutine to convert a 4-bit hexadecimal value to the corresponding 7-bit ASCII-code. See the full specification for `hexasc` below.

***Examples*:**
binary 0010 (hexadecimal digit 2) is converted to 011 0010 (ASCII-code for '2')
binary 1011 (hexadecimal digit B) is converted to 100 0010 (ASCII-code for 'B')

Implement the subroutine starting from the skeleton provided within the file `hexasc_asm.s`. Make sure that your subroutine is commented so that the teacher can follow your line of thought and understand your code.

Specification

Name: The subroutine must be called `hexasc`.

Input parameters: Only one, in register r4. The 4 least significant bits in register r4 specify a number, from 0 through 15. The values of all other bits in the input must be ignored.

Return value: Only one, returned in register r2. The 7 least significant bits in register r2 must be an ASCII code as described below. All other bits in the output must be zero.

Required action: Input values 0 through 9 must be converted to the ASCII codes for the digits '0' through '9', respectively. Input values 10 through 15 must be converted to the ASCII codes for the letters 'A' through 'F', respectively.

Side effects: The values in registers r2 through r15 may be changed. All other registers must have unchanged values when the subroutine returns.

You MUST follow the specification, which is based on the application binary interface of the Nios II [HBb].

Helpful hints

Use registers r8 through r15 for any temporary values within your subroutine. You can find the ASCII chart on many websites, e.g., on Wikipedia.

Edit the main program such that the time value is incremented and printed out once every second. Tune your `delay` subroutine to an accuracy better than 10% (no more than 6 seconds of error in one minute of execution). Verify that the time is printed once per second.

3.1 completed ◯

3.2 Microcontroller architecture

By now you should have generated at least one board support package (BSP) for the provided architecture. Study it, as it contains (among others) information about the platform we provided. You should be able to answer *in a few words* these questions:

1. What is BSP and what does it contain?
2. What is a soft-core processor? What type of processor implementation is present? Can you describe its features (i.e. clock frequency, data/instruction cache, pipeline, branch prediction, etc...)?
3. How does the processor access peripherals? What is the main mechanism, and what main piece of hardware stands between the processor and a peripheral?
4. What peripherals/IP cores are present in the provided architecture? Identify their symbolic names and base addresses.
5. What types of memory are present in the provided architecture? Identify their symbolic names, base addresses and sizes. What is the typical access time for each of these memories?

3.2 prepared ◯

3.3 Parallel I/O

3.3.1 PIO macros

In order to access the hardware, such as I/O-ports on the DE2-board, by software, Altera provides hardware abstraction layer (HAL) library functions (macros) that "hide" the memory-mapped low-level interface to the device. Information about these macros can be found in *Software Developer's Handbook* ([HBc]), Chapter 7: "Developing Device Drivers for the Hardware Abstraction Layer"

1. What macro is used in order to write to a parallel I/O port? How about to read from one?
2. What command will you use to turn on all red LEDs, i.e. to write '1's to all 18 red LEDs?
3. What command will you use to read the current status of all push-buttons?
4. Which header file you need to include in your program to access the symbolic names? What about the read/write macros?

Helpful hint:

Use the symbolic names identified in Section 3.2 to access each resource.

3.3.1 prepared ◯

A seven-segment display on the DE2 board has four digits. The leftmost digit is controlled by the seven most significant bits (`HEX3` = bits 27 through 21); the left-middle digit is controlled by the next seven bits (`HEX2` = bits 20 through 14); then follows the right-middle digit (`HEX1` = bits 13 through 7), and the rightmost digit by the least significant bits (`HEX0` = bits 6 through 0).

Update the main loop in your program and include a macro that will cause the time value to be written in binary form on the 16 red LEDs `LEDR15` through `LEDR0`. Use the natural BCD[3] encoding with four groups of four bits each.

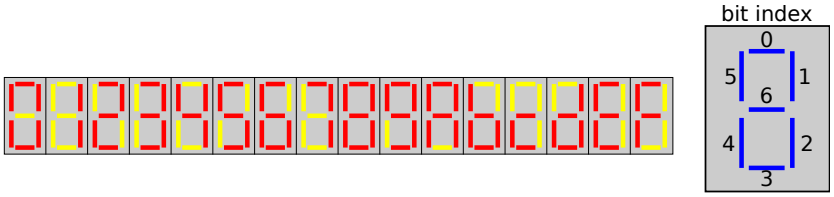
3.3.1 completed ◯

3.3.2 The bcd2sevenSeg function

Write a C function bcd2sevenSeg with the following prototype:

```
int bcd2seven(int inval)
```

The purpose of the function is to convert 4-bit binary code to 7-bit "seven-segment-code". The four least significant bits of the input value are to be converted into a 7-bit value, such that this value can be sent directly to a seven-segment display to produce the appropriate digit. Recommended digit patterns are shown in the figure below. To light up a segment of a seven-segment display, the corresponding bit should be zero (0). A one (1) will shut the segment off.



3.3.2 completed ◯

3.3.3 The puthex function

Write a C function puthex with the following prototype:

```
void puthex (int inval)
```

The purpose of the function is to display the current time on the seven-segment digits `HEX3` through `HEX0`. The puthex function should call `bcd2sevenSeg`. Add a call to the `puthex` function in your C-coded main loop.

3.3.3 completed ◯

3.4 Polling

Recall which macro to use when you want to read values from the buttons `KEY3` to `KEY0`.

Create a new project, called `lab0_poll`, and copy all files from the previous project to this new project.

Write a C function `pollkey` with the following prototype:

```
void pollkey()
```

The function shall poll the keys and affect the behaviour of the time presentation. Each time you push a button the behavior must change.

As a simplification you may assume that a button is always released before another (or the same) button is pressed and that two or more buttons are never pressed at the same time. (It is out of the scope of this lab to take care of all possibilities but you are of course allowed to do it).

KEY0 Each time you push the button `KEY0` the time shall *start* counting.

KEY1 Each time you push the button `KEY1` the time shall *stop* counting.

KEY2 Each time you push the button `KEY2` the time value presented shall be incremented by one second.

[3] BCD stands for *binary coded decimal*: four bits are used for one digit, e.g., 87 is represented as 1000 0111

KEY3 Each time you push the button KEY3 the time value shall be reset to 00:00.

Draw a flow chart to describe the behavior of the pollkey function.
Add a call to a pollkey function in your C-coded main loop.

Helpful hints

Write the C-code of the function pollkey in the same file as the timeloc variable and the main program. This makes it possible for the pollkey function to access the timeloc variable.
You can use a variable RUN to indicate if the counting is ON or OFF and use this variable to decide if tick is going to be called or not.

3.4 completed ◯

3.5 Response Time

If you run the program above you will find that the response time when you push a button might be disturbingly long. Probably also some button pushes are lost because the pollkey function is called only once per second.

Modify your program so that the pollkey function is called once per millisecond. Make sure that the time variable (timeloc) still will be incremented only once per second.

3.5 completed ◯

3.6 Interrupt using HAL-functions

Refresh your memory concerning how to use interrupts in software and hardware, especially in the Nios II CPU. Find out which HAL-functions to use to initiate and register interrupts in Nios II.

Information on HAL-functions can be found in:
Software Developer's Handbook ([HBc]), chapter 8: "Exception Handling".

In the previous program most of the CPU-time is spent polling the keys to detect changes. Most of the (CPU-) time no changes occur and valuable CPU-time is wasted. It might be smarter to let the CPU work with something useful instead of waiting for buttons to change.

The buttons have hardware support for interrupts. Information on this can be found in the *Embedded Peripherals IP User Guide* ([UGc]), chapter 9: "PIO Core".

1. What is the IRQ level for buttons?
2. Which other peripherals support interrupts? What are their IRQ levels?

3.6 prepared ◯

Create a new project called lab0_int and copy all files from the previous project to the new project.

Modify the program in your new project to use interrupts from the keys instead of polling. This means that you must write an interrupt service routine for the button interrupt (name the function key_InterruptHandler), and that you must initialize the system so that interrupts from the button PIO will cause the key_InterruptHandler function to be executed. You must also enable interrupts from the button PIO by writing appropriate value(s) to the appropriate register(s) in the button PIO using the appropriate macro(s).

The behavior caused by the buttons shall be changed compared to the previous exercise:

KEY0 The button KEY0 shall toggle between *start* and *stop* counting the time.

KEY1 Each time you push the button KEY1 the time value shall be incremented by one.

KEY2 Each time you push the button KEY2 the the time value shall be set to 00:00.

KEY3 Each time you push the button KEY3 the the time value shall be set to 59:57.

3.6 completed ◯

3.7 Using a Hardware Timer

Most processors offer possibilities to use hardware timer circuits. As the timer circuits are clocked by a crystal oscillator the accuracy is in the order of 1 per million or even better. In the Nios-II processor several timers are available.

Update your program to use a timer instead of using a programmed delay. Information on timers with the Nios-II can be found in the *Software Developer's Handbook* ([HBc]), Section "Using Timer Devices" of Chapter 6: "Developing Programs Using the Hardware Abstraction Layer".

Create a new project called `lab0_timer` and copy all files from the previous project to the new project.

Modify the new project to use HAL-functions to order "alarm" once per second to print the current time value on terminal and hex-display. Keep the functionality of the buttons from the previous exercise.

You must write the alarm-handler function and use the HAL-function to initiate the system to use it.

3.7 completed ◯

3.8 Introducing Valuable Foreground Work

You need to write a function `nextPrime` with the prototype below, which calculates and returns the smallest prime number greater than the input parameter `inval`. Calling `nextPrime` with a value of 17 will for example return the value 19, since this is the next prime. Calling it with the value 26 will return the value 29, and so on[4].

```
int nextPrime(int inval)
```

Update your program from the previous task so that prime numbers are calculated by the main program loop and printed on the console window at the "same time" as time is shown on the HEX-displays (and LEDRs). Make sure that the behavior can still be manipulated by the buttons, as specified above.

3.8 completed ◯

Table 1: Specification of the Nios II System

Component	Name	Details
System ID	*groupID*	each systems should be uniquely named using *groupID*
Nios II processor	nios2_*groupID*	economy version, debug level 1, no caches, by default programs on `sram` and data on `sdram`
Clock	clk	50 MHz
JTAG UART	jtag_uart_0	IRQ 5
On-chip RAM	onchip_memory	size: 32 KB
SRAM controller	sram	
SDRAM controller	sdram	
Interval timer 1	timer_0	period: 1ms, IRQ 7
Interval timer 2	timer_1	period: 2ms, IRQ 9
Performance counter	p_counter	7 sections
PIO - switches	de2_pio_toggles18	18 switches, IRQ 6
PIO - buttons	de2_pio_keys4	4 buttons, IRQ 8
PIO - red LEDs	de2_pio_redled18	18 LEDs
PIO - green LEDs	de2_pio_greenled9	9 LEDs
PIO - Hex display 1	de2_pio_hex_high28	highest 4 digits (left side)
PIO - Hex display 2	de2_pio_hex_low28	lowest 4 digits (right side)
LCD display	de2_lcd	optional

[4] An example is found at `app/lab1-io-sol/lab1_timer/next_prime.c`. You can use that one.

4 Building Your Own Nios II System

The goal of this section is to guide you through building your own hardware platform which you will be using for the rest of this lab course. You will need to make heavy use of tutorials and rely on the Intel/Altera documentation in order to produce the architecture that meets the following specification in Table 1. The design process is split into three iterative steps throughout this section, but you can already start off by having the specifications in Table 1 in mind.

The new system should be called `nios2_`*`groupID`*, where *`groupID`* is a unique identifier for each group ID is formed by concatenating the semester name (`ht18`) with the (first) surnames of each student in a lab group:

```
ht18_<surname1>_<surname2>
```

4.1 The Core System

Follow the tutorial in [TTa] which goes through the main steps in building a subset of the system from Table 1.

Helpful hints:

- it is enough to follow the document until (and including) Section 6: "Compiling the Quartus Prime Project".
- for convenience you can start a new Quartus project somewhere underneath the `il2206-lab/hardware` folder.
- be very careful when naming the project, as the name conventions are quite strict in the Altera design process (e.g. the project name and the top module of a Quartus II project). Please use the names suggested in Table 1 rather than the ones in the tutorial.
- have in mind that the tutorial is meant for another board and another FPGA family than yours. You will need to adapt to your situation.
- you don't need to assign all pinouts by yourself. Use the the pinout (`.qsf`) files for the DE2 or DE2-115 boards (below), and load them to your Quartus II project.
  `il2206-lab/hardware/DE2-pre-built/DE2.qsf`; or
  `il2206-lab/hardware/DE2-115-pre-built/DE2_115.qsf`;
- use the top-level VHDL files for the provided pre-built architecture (below), to get inspiration on how to name and assign the ports to pins, as compared to the examples in the tutorials.
  `il2206-lab/hardware/DE2-pre-built/IL2206_DE2_Nios2.vhd`; or
  `il2206-lab/hardware/DE2-115-pre-built/IL2206_DE2_115_Nios2.vhd`;

Test your newly-synthesized system using the "Hello World" example provided in the lab repository[5] at `il2206/app/hello_world`. Once the greeting message is properly printed out, you can proceed to the next subsection. 4.1 completed ◯

4.2 Adding peripherals

Once the core system is functioning correctly, start adding peripherals to your Qsys project. Ignore the different types of memory for now, and focus on the timers and PIO.

[5] you can either create a new project or just a new build script with properly set up variables.

Helpful hint:

Do not forget to assign the right amount of resources as suggested in Table 1, and to enable hardware interrupts when relevant. Also like in Section 4.1, consult the available VHDL files to figure out how to assign the I/O ports.

Test your newly-built system with the applications developed in Section 3. The functionality of the application has to be absolutely the same. If it is not, revise your hardware implementation.

4.2 completed ◯

(Optional:) if you want to play with the LCD display, you can include it in the Qsys project as well. You need to write your own software function to test it. Check the [UGc, Chapter 9] documentation for instructions on how to use the LCD IP core.

4.3 Adding memories

The only components missing to achieve the specification in Table 1 are the different types of memories:

- using an SDRAM controller is quite tricky, and you need to follow [TTf] if you have a DE2 board, respectively [TTe] if you have a DE2-115 board.
- for an SRAM controller there is no tutorial available, but the instantiation procedure is similar to what you should know by now. Use the SRAM controller IP block from the University Program section, since that is pre-configured for your DE2-boards.

The system should be synthesized by Quartus II without errors. Do not worry about testing the functionality of the memories because the next section is entirely dedicated for that purpose.

4.3 completed ◯

5 Time Measurement

From the lab repository, study the code used to measure time in the file `functions.c` from the project `il2206/app/lab1-measure`. Look at [HBc, Chapters 4 and 6] for more information. You shall also take into account the overhead that is generated by the measurement itself! You must understand how it works and explain under which conditions it gives the correct compensation for the timer overhead. Suggest possible improvements.

Helpful hint:

Make sure that `timestamp_timer` is pointing to a timer device in the BSP settings, and it is different than the one pointed by `sys_clk_timer`. Otherwise you will get the message "`No timestamp device available!`".

Duplicate the functions in the main routine of the previous program, but this time use the performance counter IP instead of the timestamp timer. Answer the following questions:

1. How are the timestamp timer and the performance counter functioning? Describe the main steps.
2. What are the advantages and disadvantages of one over the other?

5 prepared ◯

Loop the program several times (at least 50). Fill in Table 2, where *time* is the average measured time for the main computing function, and *error* is the mean absolute error (MAE) between all measurements.
From now on you choose your favourite mean to measure time and stick with it until the end of this lab.

5 completed ◯

Table 2: Time measurements

	Time (ms)	Error (ms)
timestamp timer		
performance counter		

5.1 Compiler Optimization

The Nios II tool suite is based on the GNU compiler `gcc`. The compiler has a huge amount of optimization options. Find out, how the following optimization switches affect the optimization of the code: `-O0`, `-O1`, `-O2`, `-O3`, `-Os`. You can search for this information on the web and in the lecture notes. 5.1 prepared ◯

Run the previous program again, but using different optimization flags. Fill in Table 2 with the average times measured for the main computing function, using one measuring method of your choosing.

Table 3: Compiler optimizations

Flag	Time (ms)	Size (KB)
`-O0`		
`-O2`		
`-Os`		

Helpful hint:

You can use different scripts or different target rules for the same project to be able to show easily the different executions during the lab session without changing the code.

5.1 completed ◯

References

[HBa] Altera. *Embedded Design Handbook*. Version 2.9.

[HBb] Altera. *Nios II Processor Reference Handbook*. Version NII5V1-13.1.

[HBc] Altera. *Nios II Software Developer's Handbook*. Version NII5V2-13.1.

[TTa] Altera. *Introduction to the Altera Qsys System Integration Tool*. for Quartus II 13.1.

[TTb] Altera. *My First Nios II Software Tutorial*.

[TTc] Altera. *Nios II Hardware Development*. Version TU-N2HWDV-4.0.

[TTd] Altera. *Nios II System Architect Design*. Version TU-01004-2.0.

[TTe] Altera. *Using the SDRAM on Altera's DE2-115 Board with VHDL Designs*. for Quartus II 13.1.

[TTf] Altera. *Using the SDRAM on Altera's DE2 Board with VHDL Designs*. for Quartus II 13.0.

[TTg] Altera. *Using Tightly Coupled Memory with the Nios II Processor*. Version TU-N2060305-2.0.

[TTh] *Introduction to the Altera Nios II Soft Processor*.

[UGa] Altera. *DE2-115 User Manual*. Version 1.02.

[UGb] Altera. *DE2 Development and Education Board User Manual*. Version 1.4.

[UGc] Altera. *Embedded Peripherals IP User Guide*. Version UG-01085-11.0.