Application Note:
Cortex™-M0 Implementation in the Nexys2 FPGA Board – A Step by Step Guide.

Pedro Martos (pmartos@fi.uba.ar / pimartos@gmail.com)
& Fabricio Baglivo (baglivofabricio@gmail.com)

Buenos Aires University – School of Engineering
www.fi.uba.ar

Embedded Systems Laboratory
laboratorios.fi.uba.ar/lse

Contents

Revision History:

Date	Version	Change
24/May/2011	1.00	Initial Release.

For questions, comments, or improvements to this application note, please contact the authors at the emails shown above. Thanks.

Introduction

This application note is a step by step guide for a basic implementation of the Cortex-M0 DesignStart (or “Cortex-M0_DS” for short) processor in an FPGA board. It’s intended as a starting point to build a system around the Cortex-M0_DS processor. The system described in this application note has a Cortex-M0_DS processor, reset, preloaded memory with a program that fetches constants from memory at regular intervals, and a pattern detector attached to the data bus. When a specific pattern appears on the data bus, an LED turns on; when another specific pattern appears on the bus, the LED turns off.

The Cortex-M0_DS deliverables from ARM include only the processor and a non sinthesizable testbench, so we will implement these other parts around the processor to build a synthesizable system: the software executable image, a reset synchronizer, a memory holding the program, the system clock, and the detector module that will command the LED.

This system was developed using Microsoft Windows XP SP3 32-bit edition as a host OS, Xilinx ISE 12.2 for WinXP32 as the FPGA development tool, and the ARM/Keil MDK 4.14 Evaluation Version for the software development tools. The hardware is based on a Xilinx Spartan3E-500 FPGA (Nexys2 board from Digilent). Other combinations of development tools, operating systems, and FPGA boards should work with minor changes.

Prerequisites

Hardware & Software

a) Nexys2 board
   http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,789&Prod=NEXYS2
b) Xilinx ISE12.2 development tools
   http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm
c) Digilent Adept software
   http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,66,828&Prod=ADEPT2
d) Digilent Plugin for Xilinx tools
   http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,66,768&Prod=DIGILENT-PLUGIN
e) Cortex-M0 DesignStart deliverables
   http://www.arm.com/support/university/index.php

f) ARM/Keil MDK evaluation version
   http://www.keil.com/arm/mdk.asp
g) BIN to COE conversion utility
   http://www.sourceforge.net/projects/bin2coe

Recommended level of knowledge:

a) Digital systems design and VHDL language: intermediate
b) Processor architectures: intermediate
c) Verilog language: basic
d) “C” programming language: intermediate
e) Embedded systems programming using “C”: intermediate
f) Cortex-M0 assembler language: basic
g) AMBA-LITE™ bus: basic
h) ARM/Keil MDK: basic
i) Xilinx ISE: intermediate; up to the level of the Xilinx FPGA Design Flow Workshop available at the Xilinx University Program: www.xilinx.com/university/workshops/fpga-design-flow/index.htm

Workflow

A) Software Development and Simulation: Using the ARM/Keil MDK, we will develop and simulate a simple software program to verify memory fetches of predefined constants.

B) Basic System Implementation: Using the Xilinx ISE, we will implement a basic synthesizable system that can execute the code developed in (A)

C) Functional Simulation: Using the ISIM tool, we will simulate the system generated in (B) and verify that the predefined constants appear on the processor’s data bus.

D) Hardware Verification: A ChipScope Pro module will be added to the system developed in (B), so that we can see the internal signals. The system will be synthesized and downloaded to the board, and we will verify the memory fetches and the LEDs flashing on and off.

Software Development and Simulation

In this step we will create a project in the ARM/Keil MDK IDE with a program that fetches two constants from memory at regular intervals.

To do that we create a new project with ARM Cortex-M0 as the chosen CPU:

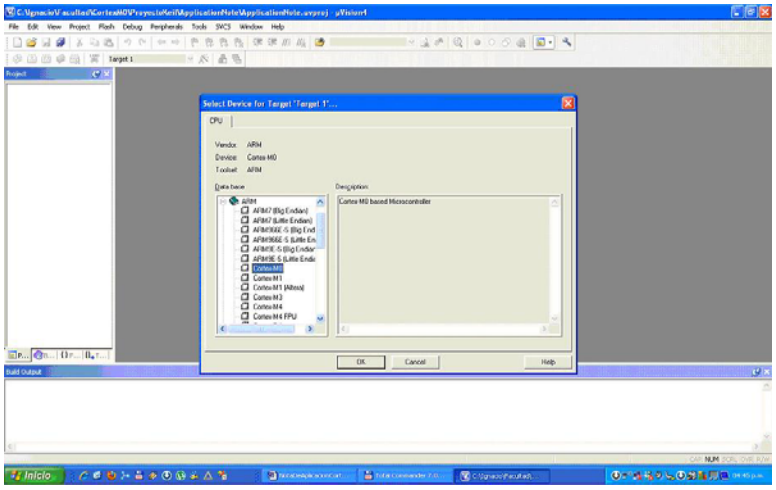


Fig.1

Next, add the sources “main.c” and “vectors.c” (listings in the “Source Code” section). If desireable, change the name “Target 1” to “CM0_DS” and “Source Group 1” to “BlinkingLed”, or choose other names too.

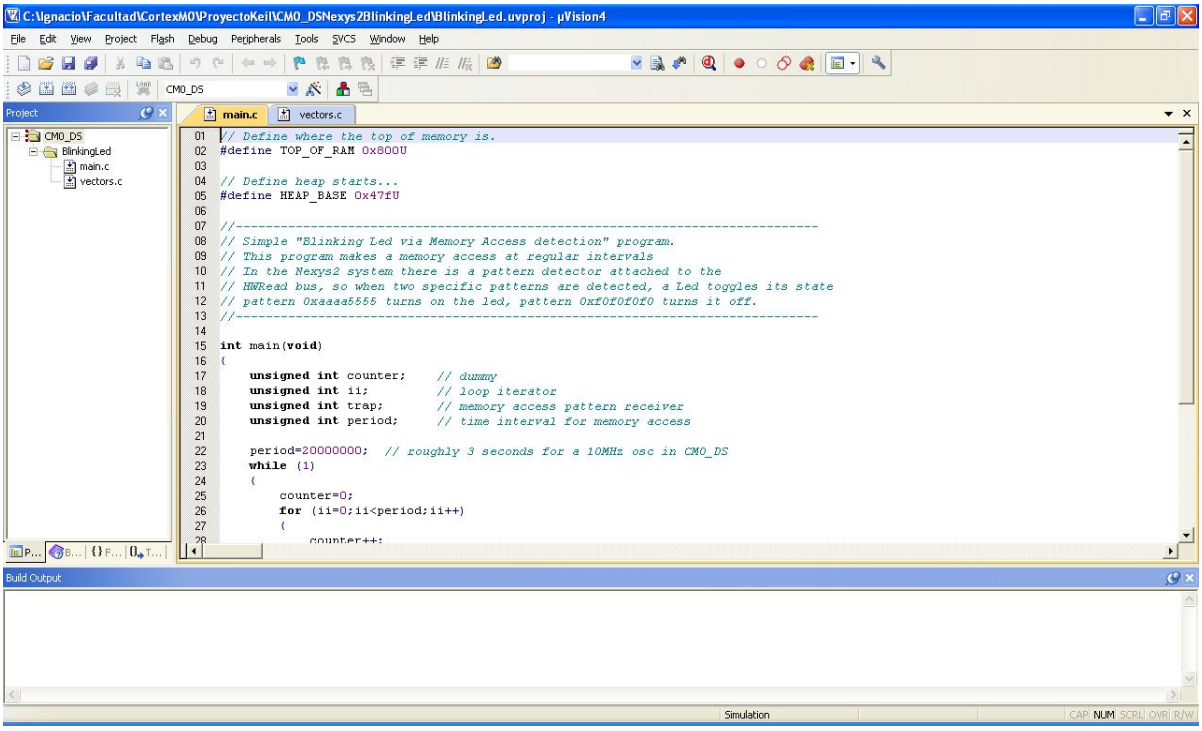


Fig.2

The main.c source has the executable code, and vectors.c source has the base address for the stack and ARM exception vectors. The only exception vector used in this application is the Reset Handler. Remember to use the correct value for “period”, depending if you are simulating or if you are building an image for FPGA implementation. For simulation purposes, use 200 as “period” value. For implementation in FPGA, use 20000000 instead.

Next, set the Options for Target CM0_DS in tabs “Target”, “Output”, “ASM”, “Linker”, & “Debug”: In the “Target” tab, choose a working frequency, for example 10 MHz (this frequency should be the same as the system clock frequency in the FPGA project in step (B)), and also implement a ROM memory with size 1024 bytes, and a RAM memory with size 1024 bytes. This provides a total memory size of 2048 bytes.

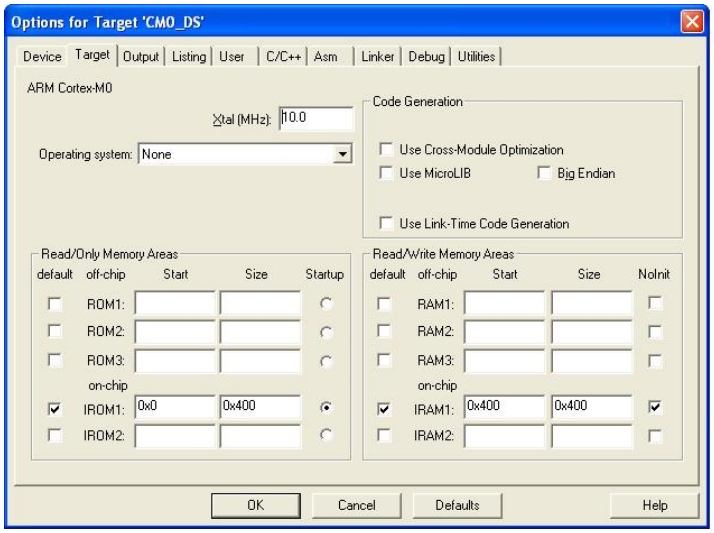


Fig.3

In the “Output” tab, choose a folder for the objects generated by the toolchain, the name of the executable image, and the option to generate a .HEX file with the code.

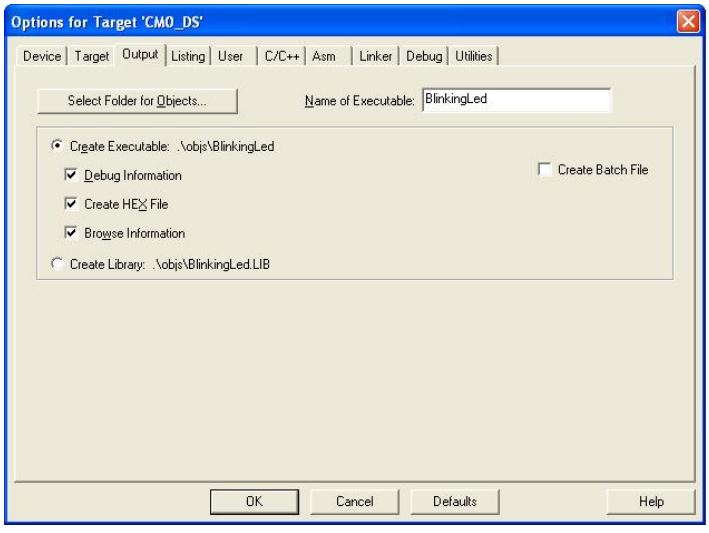


Fig.4

In the “Asm” tab, choose Thumb Mode.

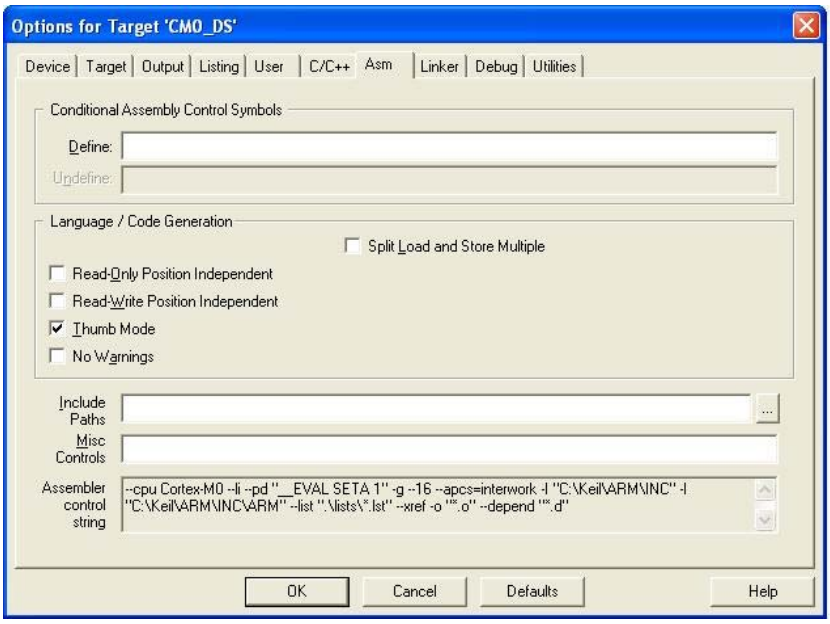


Fig.5

In the “linker” tab, because we use the evaluation version of the toolchain, we need to set the base addresses by hand and instruct the linker how to deal with the exception vector table. Set the “R/O” (ROM) base address to 0x00000000 and “R/W” (RAM) base address to 0x00000400, then add “--entry 0x15 --first=vectors.o(__Vectors)”, to the “Misc. Controls” to locate the vector table defined by vectors.c.

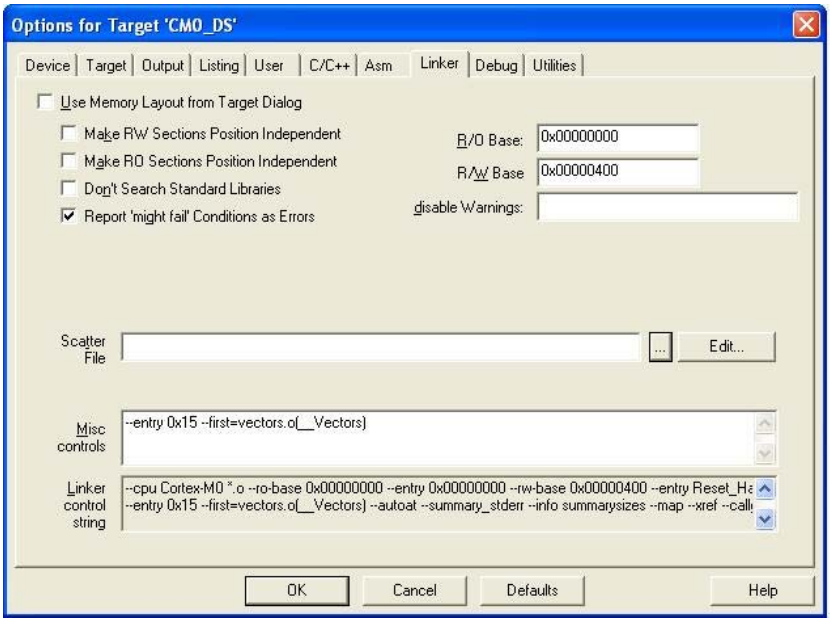


Fig.6

In “Debug” tab, ensure that “Use Simulator” tick is selected for debugging purposes.

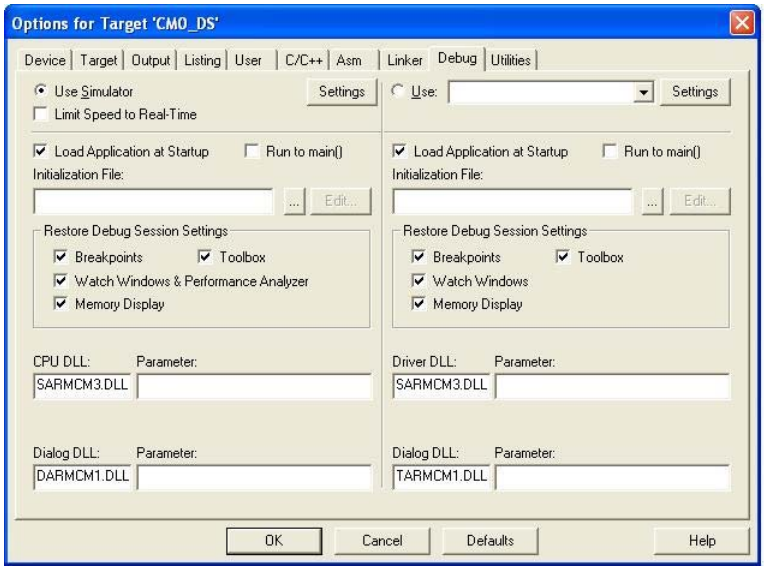


Fig.7

Now, build the executable image from the sources using the “build” button or the project menu “build target”.

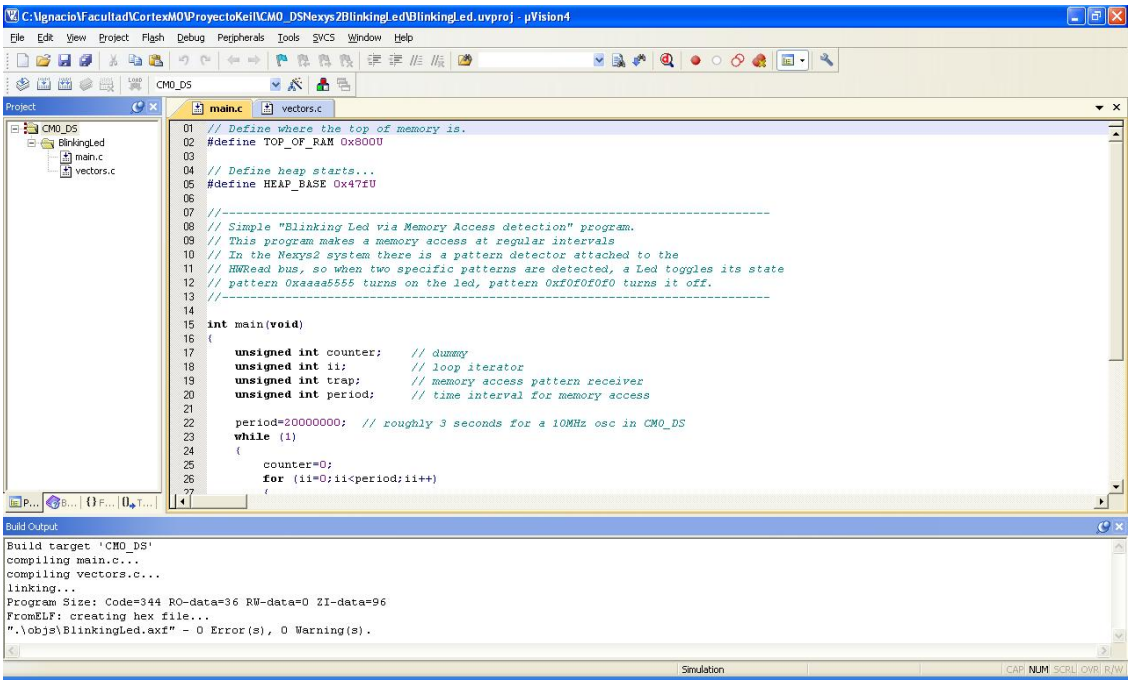


Fig.8

If everything went fine, you should have the executable image in elf format “BlinkingLed.axf”. Simulate the program with the “start/stop debug session” button.

In the simulation, use the F11 key to step through the assembly code, noting its relationship with the C source code (the assembly code may differ depending on the MDK version used).

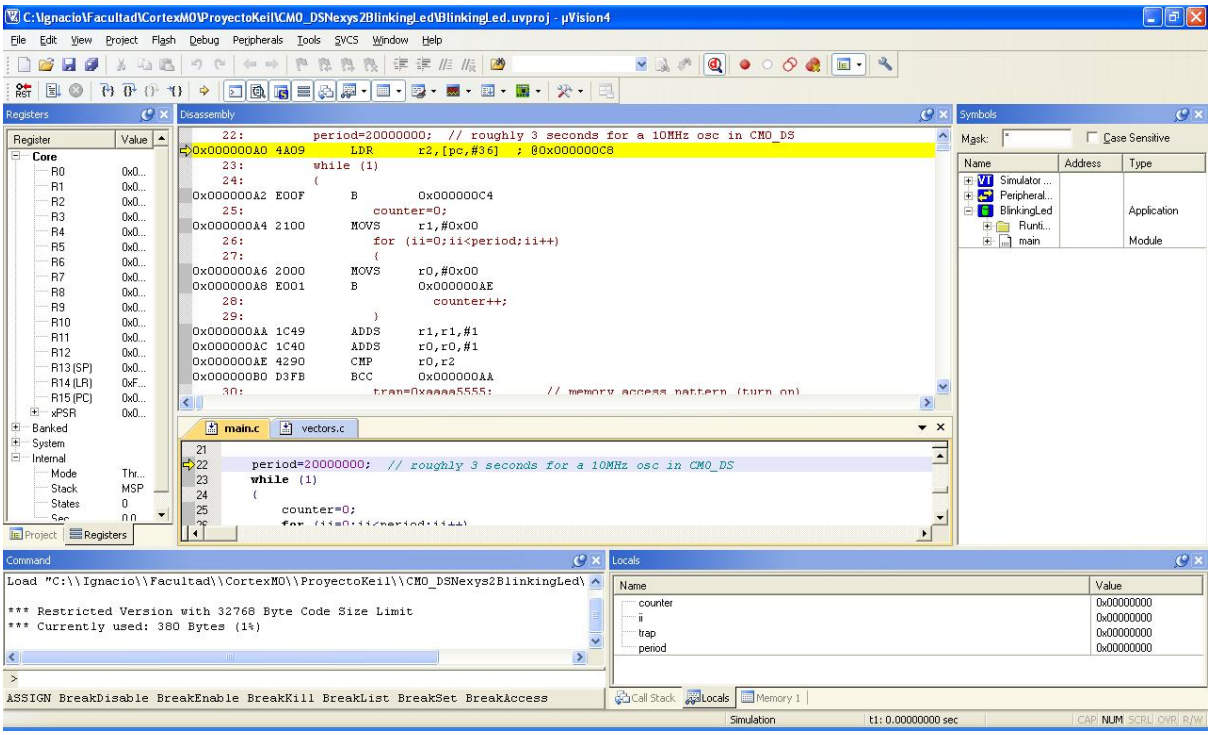


Fig.9

When the C assignment line executes with constant 0xaaaa5555 (place a breakpoint on the instruction to stop the processor before the instruction executes), note that the assembler generates an ARM PC-relative load instruction (LDR) to load that constant from memory into a register, so there is a memory access using the HRDATA bus (the HRDATA bus is the processor's read data bus. Data coming into the processor uses this bus). In the FPGA, a detector module connected to the HRDATA bus will search for this constant or the constant 0xf0f0f0f0 to turn on and off an on-board LED.

Now reset the processor and place a breakpoint in line 40 where the first memory access that loads the constant 0xf0f0f0f0 (LedOff) is done. Run the program (with F5) until the breakpoint is reached and take note the simulation time 242.20 uS, which is the execution time starting from the "main" function (it doesn't take into account the processor setup time). This time will be compared to the hardware functional simulation time.

In this section:

- A software project was created in ARM/Keil MDK configured for the Cortex-M0_DS processor.
- C sources “main.c” and “vectors.c” were added to the project which included a ready to execute a program with the exception vectors and stack configured.
- The source code was simulated and debugged.

Basic system implementation

This section will describe how to generate a project in ISE using the Cortex-M0_DS processor. Necessary modules will also be added to implement a basic system capable of running programs.

To do this, create a new project in ISE called “CM0_DSSystem” using the Spartan3E-500 speed grade 4 device, with preferred language “VHDL”.

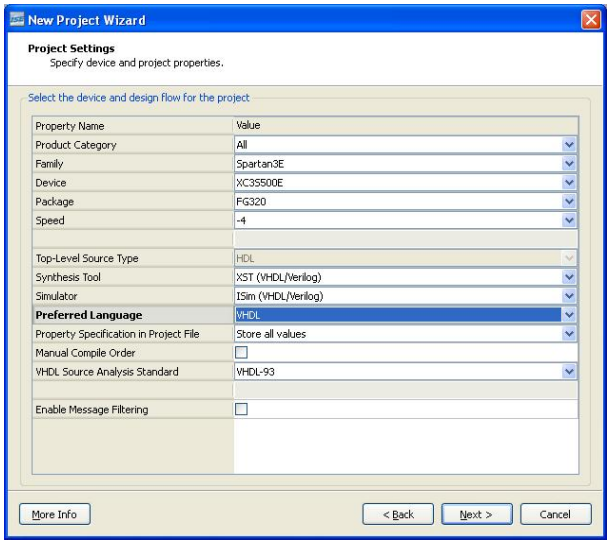


Fig.10

For this Project, make a mixed implementation using Verilog and VHDL. The processor is described in Verilog and the additional modules in VHDL. The information to make a mixed language project is in the XST User’s Guide. The architecture of the project and the implementation files are shown here (the names in parenthesis are submodules of each module). The .vhd sources are in the “Source Code” section.

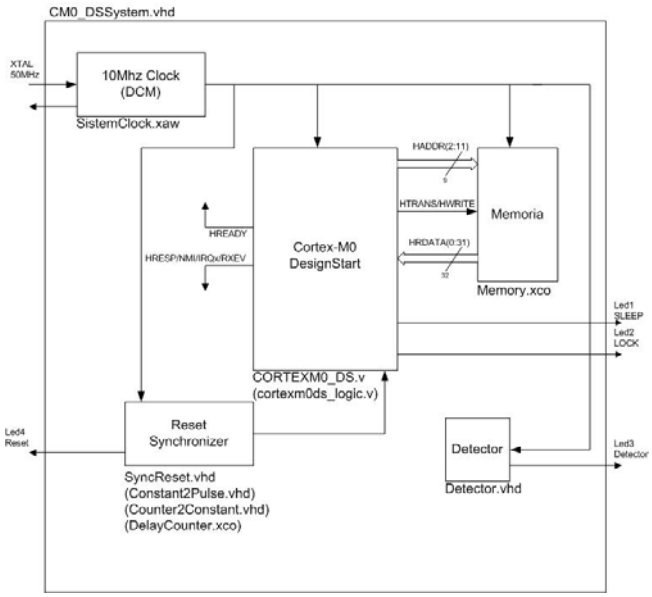


Fig.11

These .vhd and .v files will be added to the project. The next sections will explain how to generate/add each one.

CM0_DSSystem:

This is the top module of the implementation. It has the connections between the submodules and the interface to the LEDs and Crystal Oscillator on the board. The module description is in the file “CM0_DSSystem.vhd”, so add it to the project using the “Project - >Add source…” or “Project - > Add copy of source...” menu in ISE. The submodules (marked with a “?”) will be added in the next subsections.

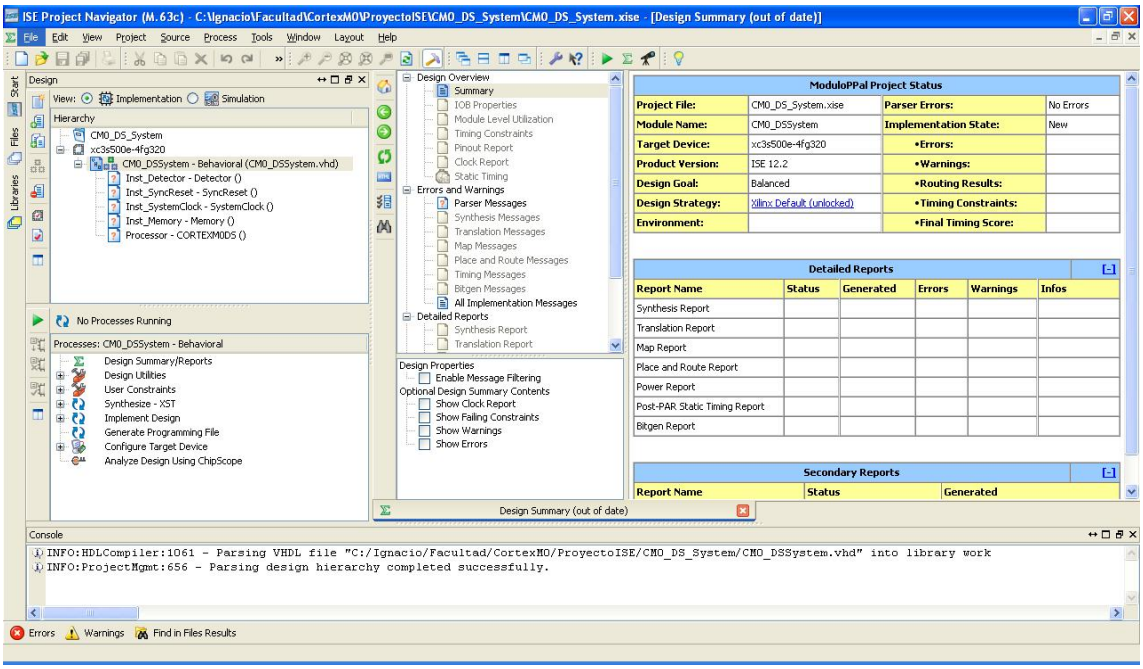


Fig.12

Cortex-M0 DesignStart:

This is the processor. Its implementation is in “cortexm0ds_logic.v” (obfuscated implementation) and “CORTEXM0DS.v” (interfaces and processor registers and signals). They are part of the deliverables from ARM. There is more information about these files in the processor release notes. You need to add those files to the project using the “Project - >Add source…” or “Project - > Add copy of source...” menu in ISE.

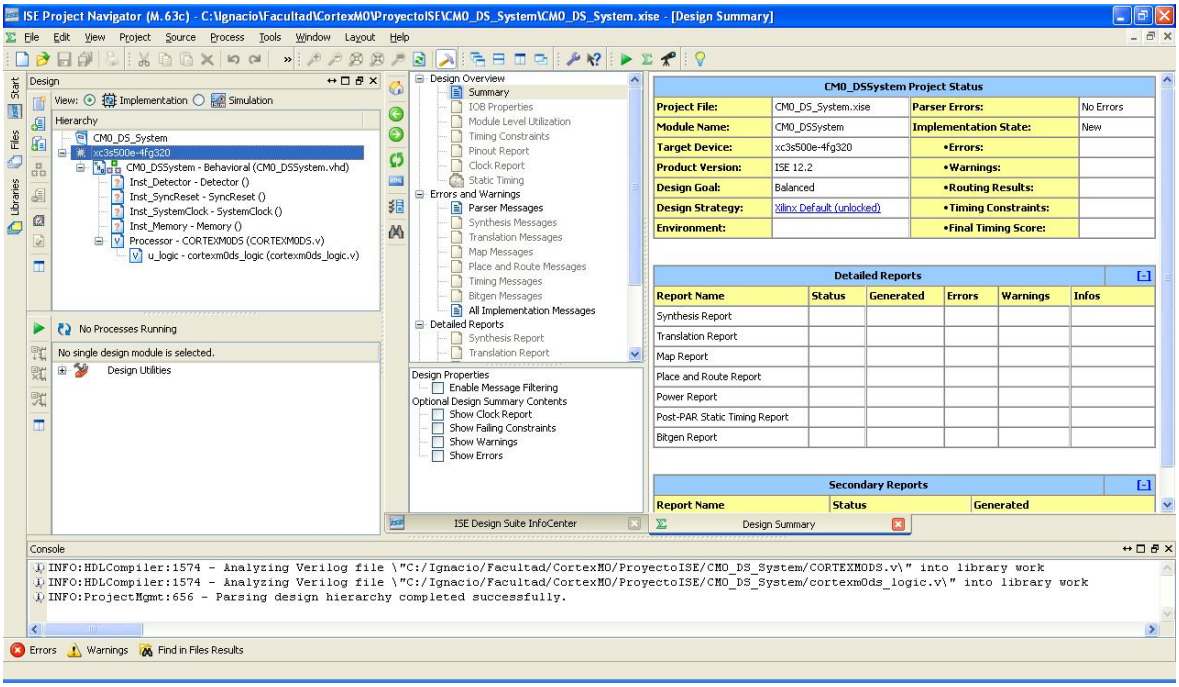


Fig.13

10 MHz Clock:

This module generates the system clock at 10 MHz from the 50 MHz board's external oscillator. To do this in ISE, create a new ipcore called "SystemClock" using a DCM. In "Project", select "new source" from type "ip (coregenerator)" and type "Single DCM SP", and name it "SystemClock", and with description in VHDL:

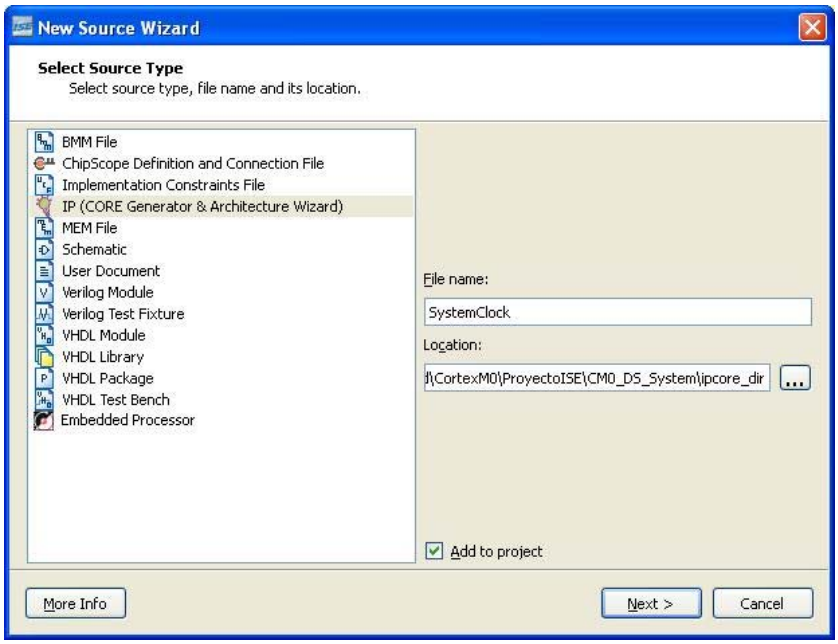


Fig.14

Select the “View by Function” tab and choose “Single DCM_SP” from the “FPGA Features and Design - > Clocking - > Spartan-3E, Spartan-3A" subtree.

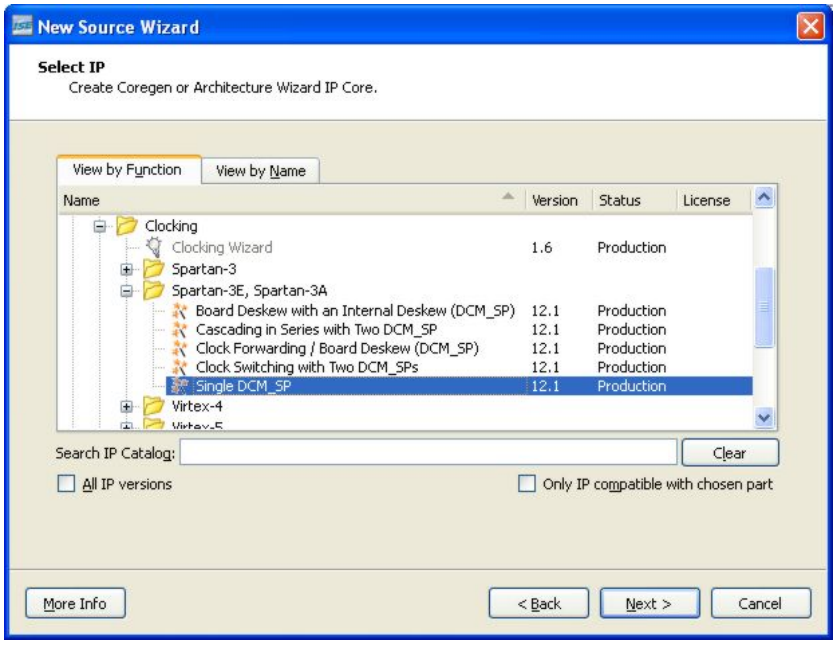


Fig.15

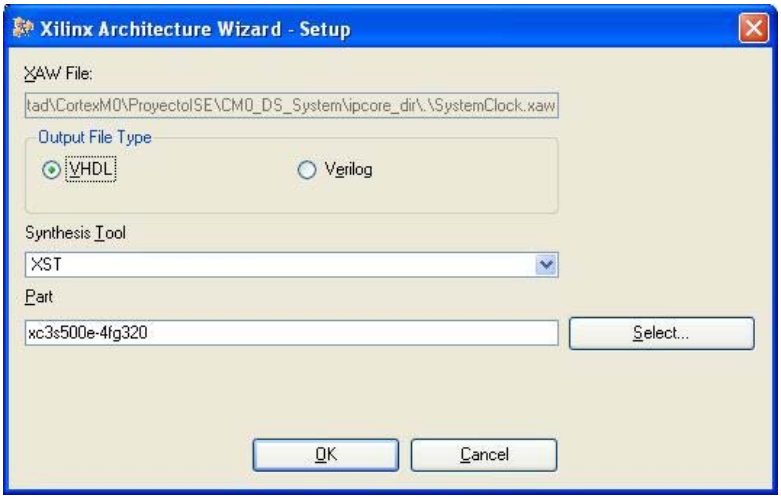


Fig.16

Generate the ipcore with the following configuration:

In this window select the input frequency to 50MHz (This is the oscillator frequency of the board).

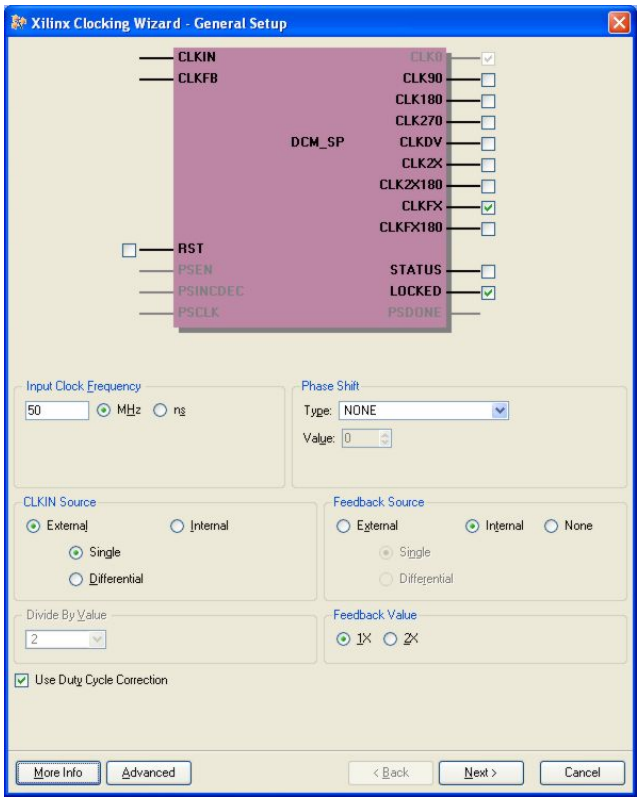

Xilinx Clocking Wizard - General Setup
CLKIN
CLKFB
RST
PSEN
PSINCDEC
PSCLK
DCM_SP
CLK0
CLK90
CLK180
CLK270
CLKDV
CLK2X
CLK2X180
CLKFX
CLKFX180
STATUS
LOCKED
PSDONE
Input Clock Frequency
50
MHz
ns
Phase Shift
Type:
NONE
Value:
0
CLKIN Source
External
Internal
Single
Differential
Feedback Source
External
Internal
None
Single
Differential
Divide By Value
2
Feedback Value
1X
2X
Use Duty Cycle Correction
More Info
Advanced
< Back
Next >
Cancel

Fig.17

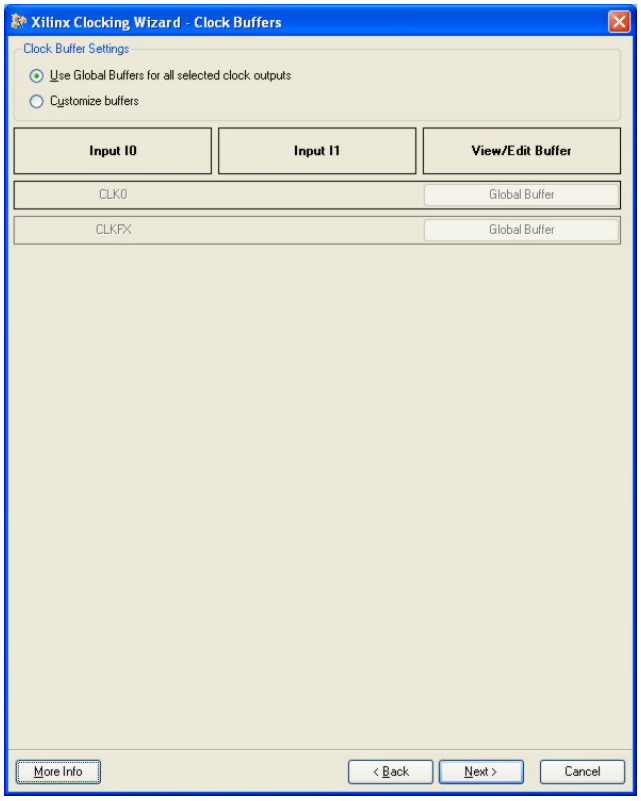

Xilinx Clocking Wizard - Clock Buffers
Clock Buffer Settings
Use Global Buffers for all selected clock outputs
Customize buffers
Input I0
Input I1
View/Edit Buffer
CLK0
Global Buffer
CLKFX
Global Buffer
More Info
< Back
Next >
Cancel

Fig.18

In this window, set the output frequency to 10 MHz (This is the frequency set up in the ARM/Keil MDK project) and press “Calculate” button.

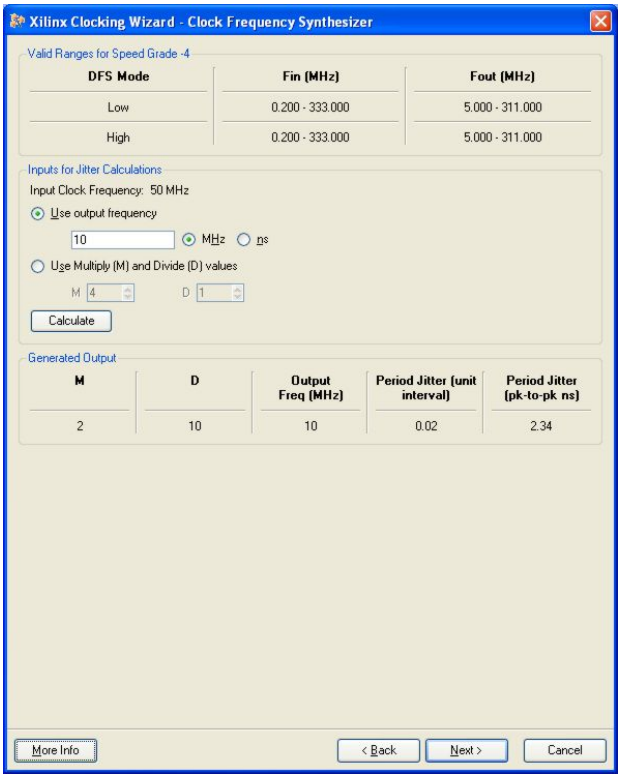


Fig.19

This should be the configuration:

Fig.20

Detector:

This module will be attached to the processor's HRDATA bus and will be responsible for reading the data on that bus. When the value 0xaaaa5555 is on the bus, LED3 turns on. When 0xf0f0f0f0 is on the bus, LED3 turns off. The module description is in the file "Detector.vhd". You need to add it to the project using the "Project - >Add source…" or "Project - > Add copy of source..." menu in ISE.

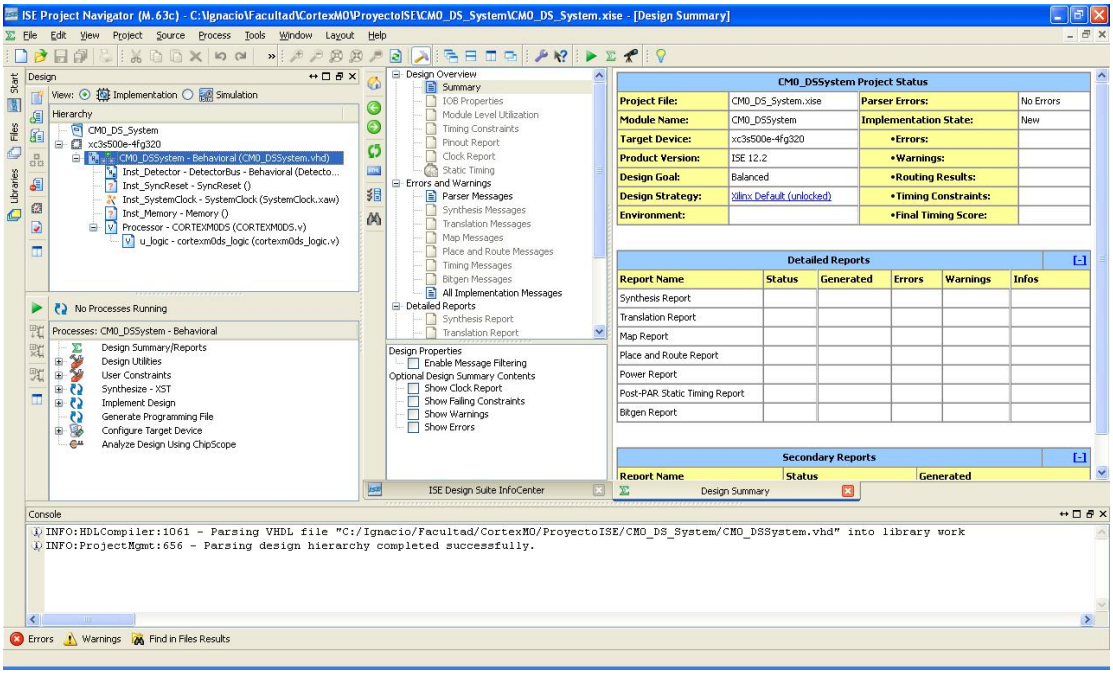


Fig.21

Reset synchronizer:

This module generates a delay using a counter and then it generates an active-low reset pulse lasting five clock cycles. This is synchronized with the rising edge of the system clock. The synchronizer can be implemented in a single module, but for clarity here, it is implemented using one module (SyncReset.vhd) and 3 submodules (DelayCounter.xco; Counter2Constant.vhd y Constant2Pulse.vhd). You need to add the .vhd sources using the "Project - >Add source…" or "Project - > Add copy of source..." menu in ISE.

The module works as such: a counter generates a pulse that overflows, generating a delayed pulse synchronized with the system clock. This solution has some disadvantages: a) the pulse lasts for only one clock cycle (we need at least two clock

cycles to reset the processor) and b) the pulse is periodic (its period is the counter overflow), so the processor will be reset periodically.

To overcome this situation, add the modules Counter2Constant, Constant2Pulse, and SyncReset to the project. The module Counter2Constant generates a ‘0’ to ‘1’ transition in the first overflow of the counter, but the output is held as a ‘1’ in the next counter overflows. The module Constant2Pulse generates a one clock cycle pulse synchronized with the ‘0’ to ‘1’ transition in the module Counter2Constant. Finally, the module SyncReset receives the output of the module Constant2Pulse, negates it, and generates a low pulse lasting five clock cycles. This pulse is used to reset the processor.

Here are the timing diagrams:

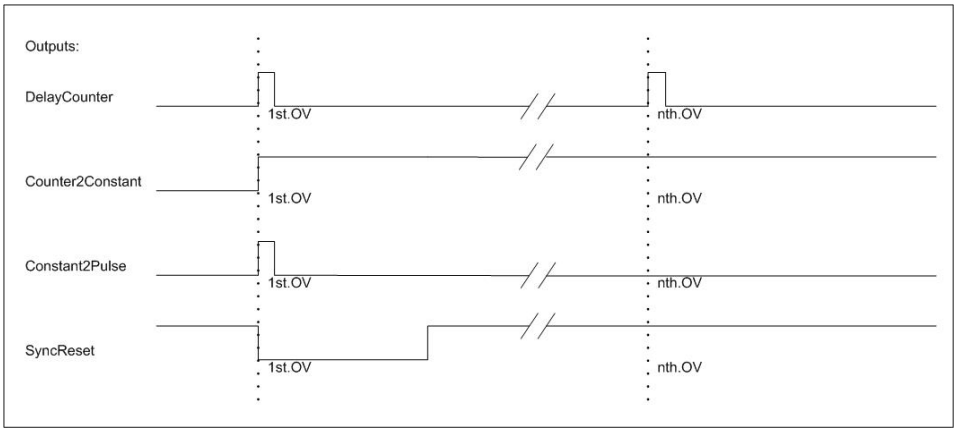


Fig.22

The logical diagram (using gates) for each module are listed below. We used FFs to be able to set initial values at ‘0’. In the next section you will create the Delay Counter module.

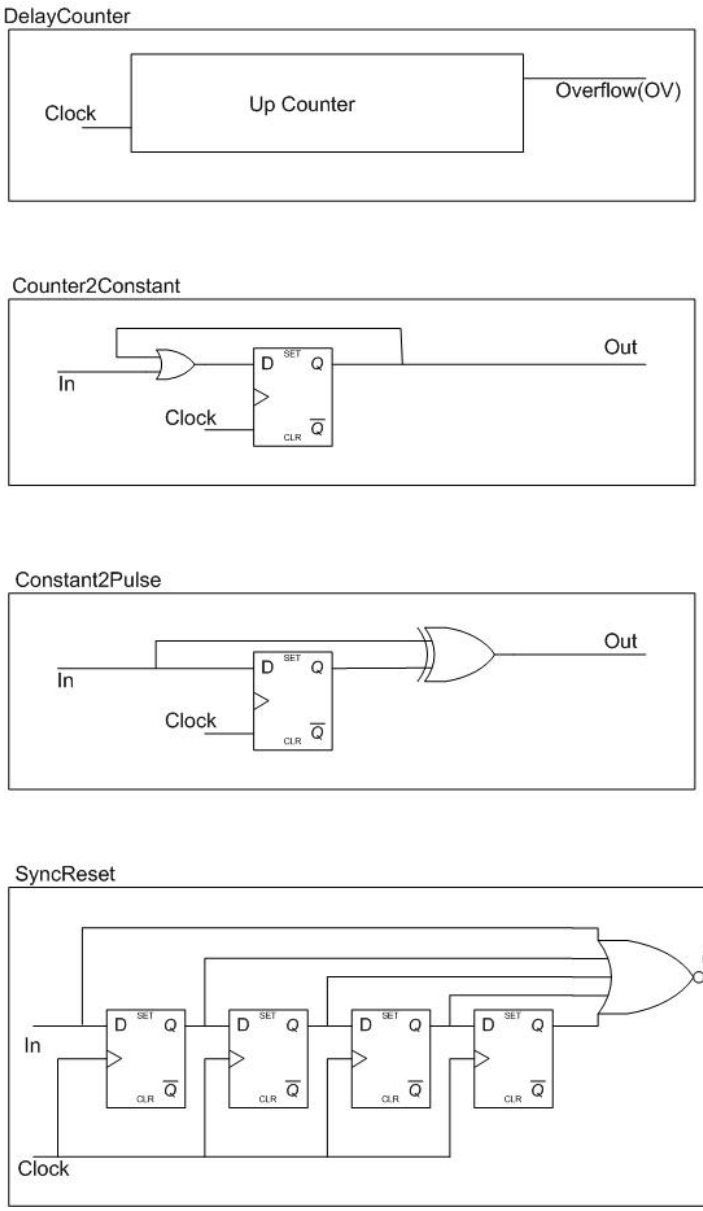


Fig.23

Binary counter configuration:

To generate the Reset Synchronizer counter in ISE, create a new ipcore called "DelayCounter" using a binary counter. In "project", select "new source" from type "ip (coregenerator)" and type "Binary Counter", name it "DelayCounter", and with description in VHDL:

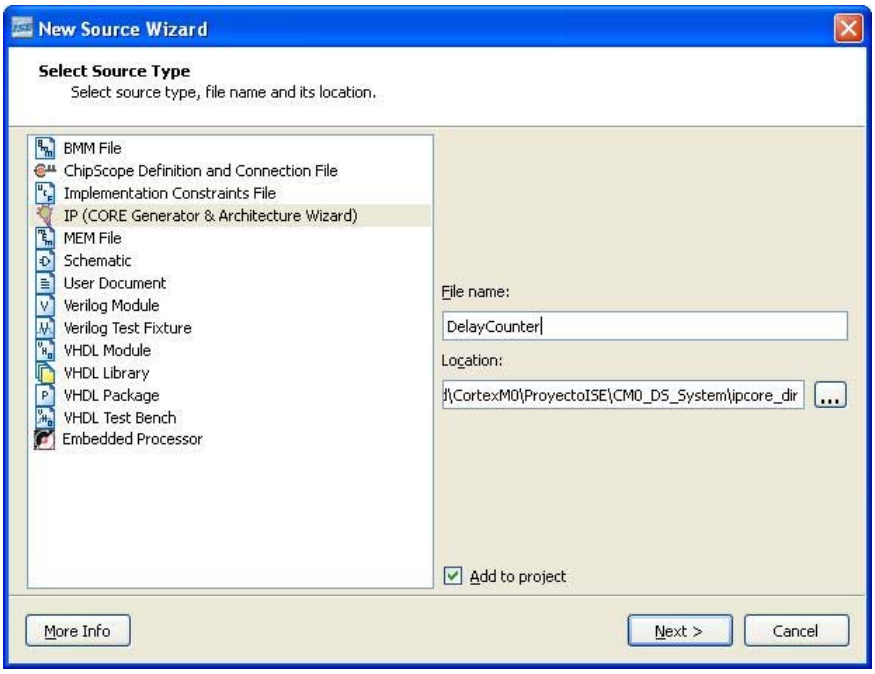


Fig.24

Select the “View by Function” tab and choose “Binary Counter” from the “Basic Elements - > Counters" subtree.

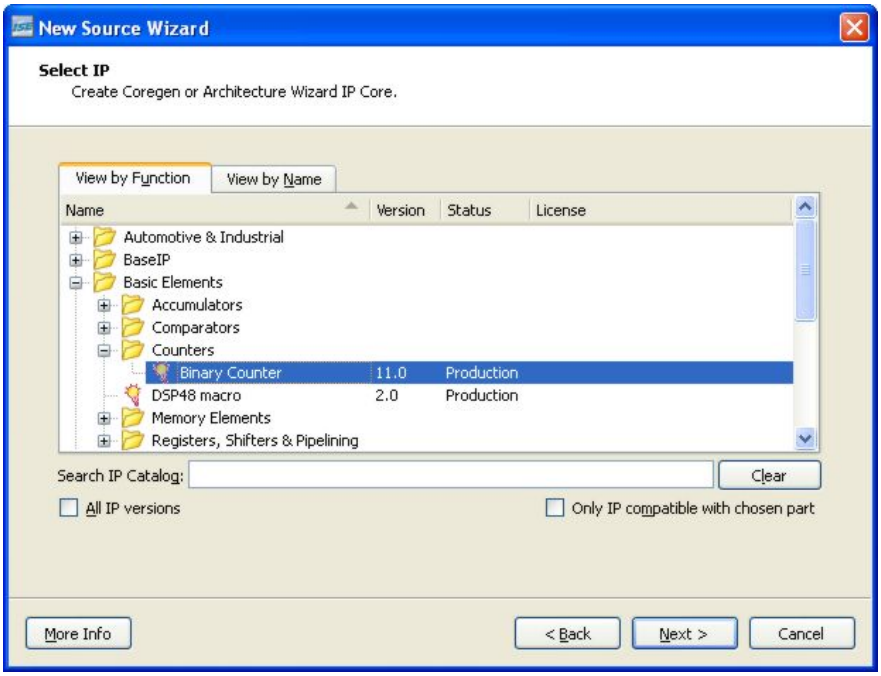


Fig.25

The binary counter should have the following configuration:

The output width is set for 20 bits (2^20 counts), with 1-bit increment.

The final count is 250 (0xFA) and the overflow value is 249 (0xF9). We intentionally used a small counter value to speed up the simulation. In the hardware implementation you may use values according to the counter's size to control the processor's startup delay time.

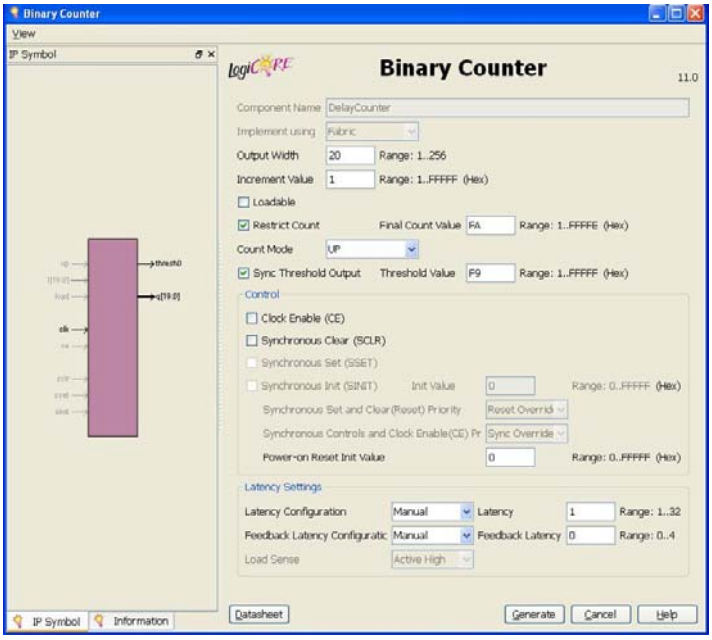


Fig.26

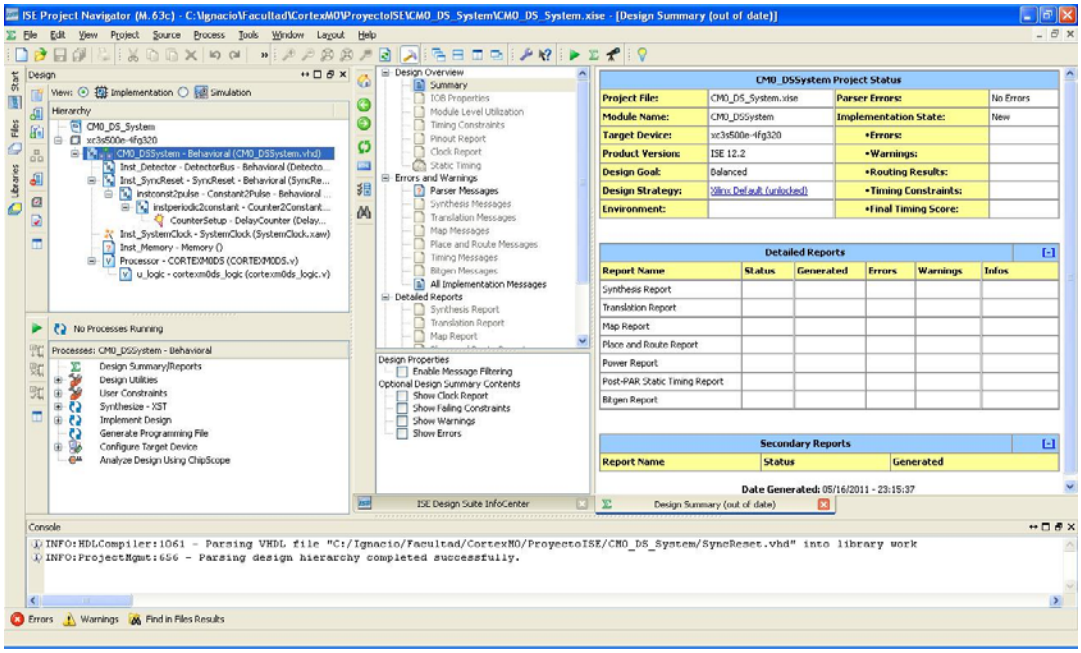


Fig.27

Memory:

This is a basic system, so the RAM and ROM will be implemented as a single memory component with the ROM part containing the program. A 2KB system was defined in the software program, but the processor always accesses 32 bits (4 bytes) per memory access, so it's necessary to implement a memory with 512 words of 32-bit length each.

To initialize the memory, a .coe file is needed with the memory contents. To get this file, we need a .bin (binary image) file first. This can be obtained from the .axf file generated in ARM/Keil MDK. To get "BlinkingLed.bin" file from "BlinkingLed.axf" file, use the "fromelf" utility (this tool is bundled with ARM/Keil MDK). This is a command line utility, so it can be invoked as such:

"**fromelf** - -bin - -o BlinkingLed.bin BlinkingLed.axf"

Use the utility "Bin2Coe" to generate a COE formatted ASCII file from "BlinkingLED.bin" that will be used to initialize the memory. This is also a command line utility, so it can be invoked as such:

"**bin2coe** BlinkingLed.bin BlinkingLed.coe 512"

To generate the memory, we need to create a core called "Memory" using the block memory available in the FPGA, Again, in "project", select "new source" from type "ip (coregenerator)", type "Block Memory Generator", name it "Memory", and with description in VHDL:

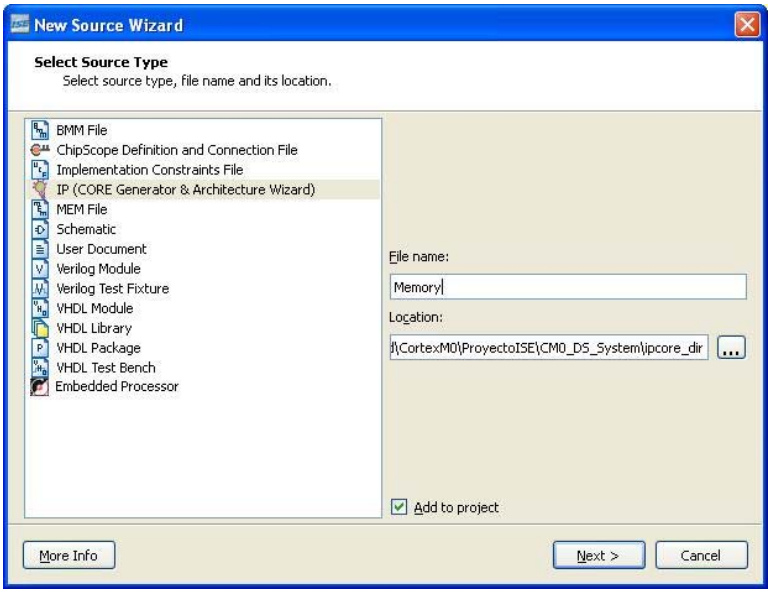


Fig.28

Select the “View by Function” tab and choose “Block Memory Generator” from the “Memories and Storage Elements - > RAMs & ROMs" subtree.

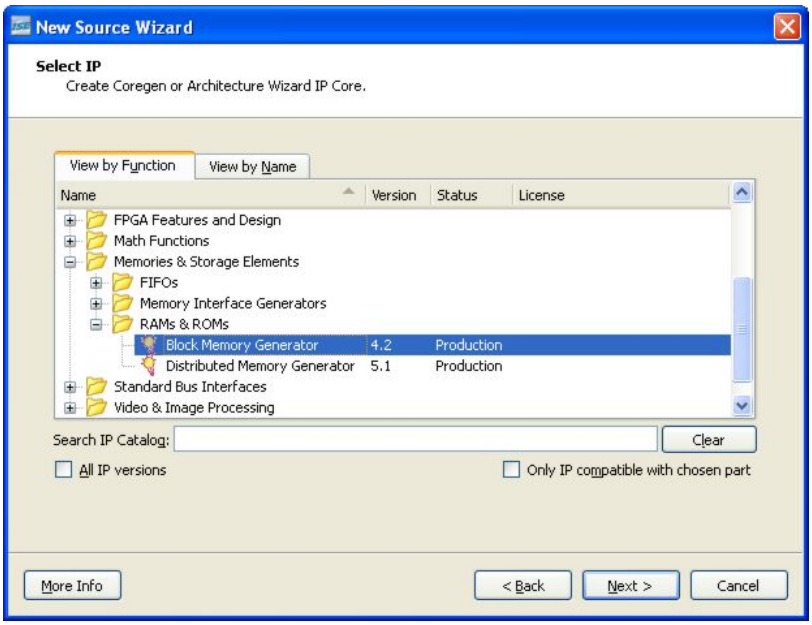


Fig.29

The memory must be generated with the following configuration:

Here we use the default values.

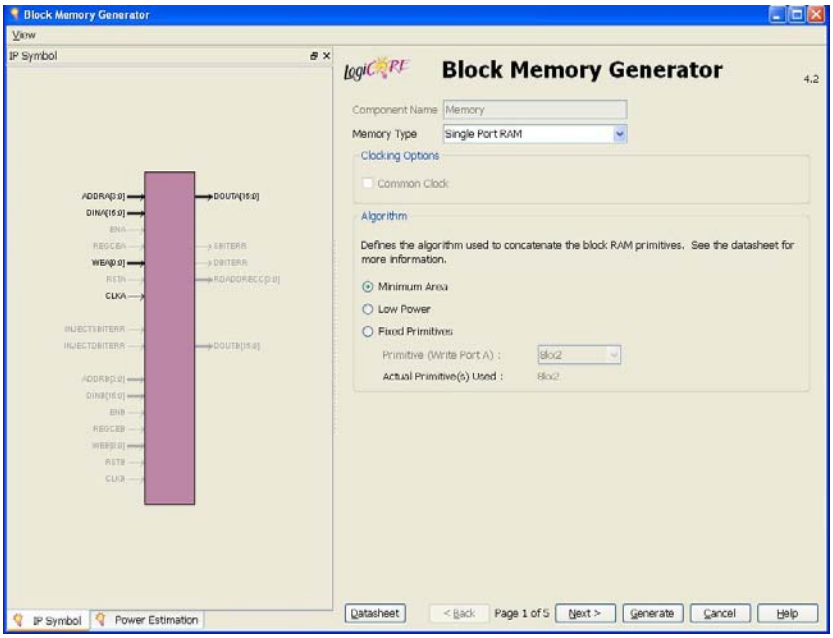


Fig.30

Here we configure the memory to be one of 512 words (depth) of 32-bit each (width), and add an “enable” signal (check “Use ENA Pin” box).

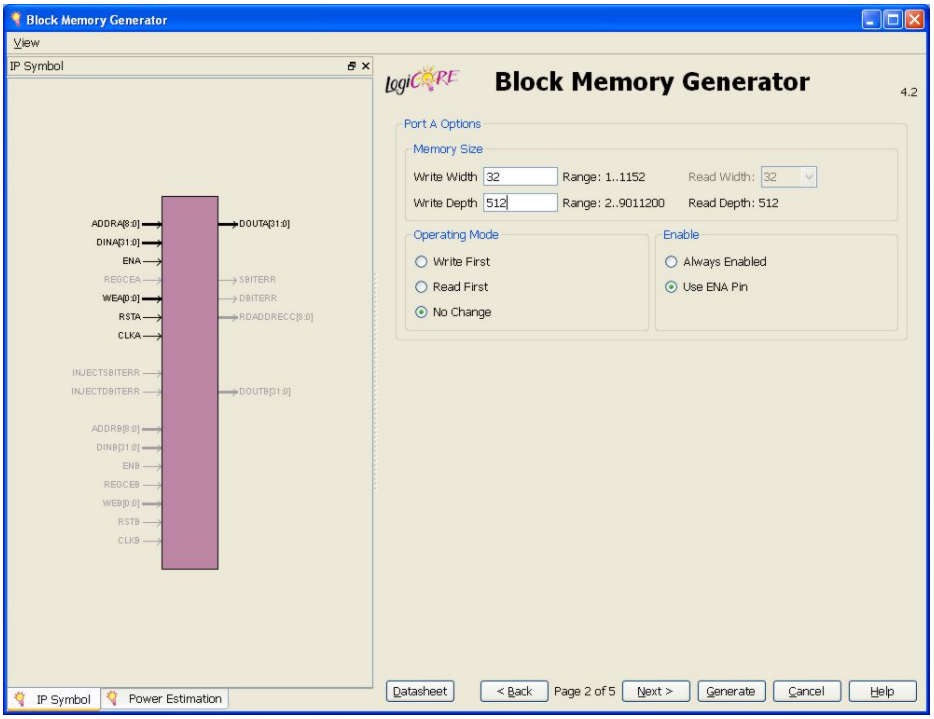


Fig.31

Here tell the memory generator where is the .coe file with the program (it will be copied to the ..\ipcore directory).

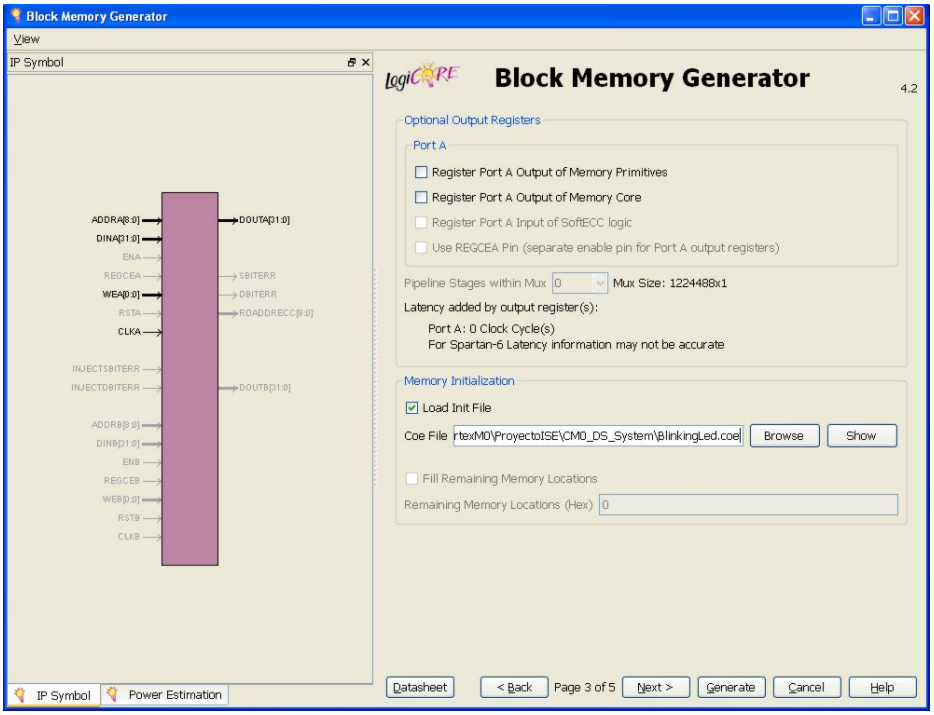


Fig.32

Then add a reset input for the memory.

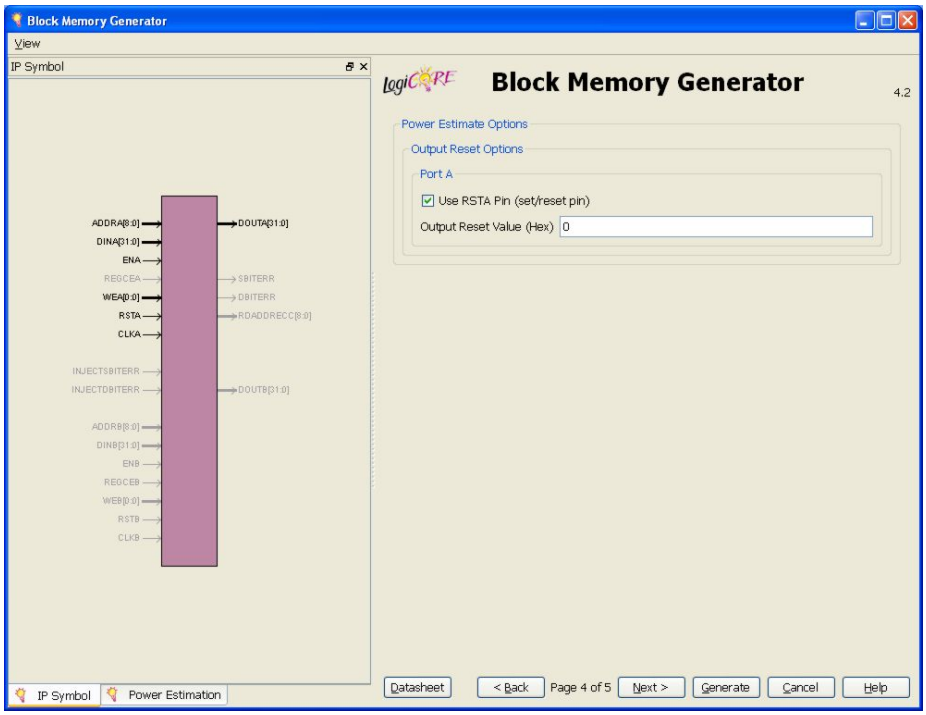


Fig.33

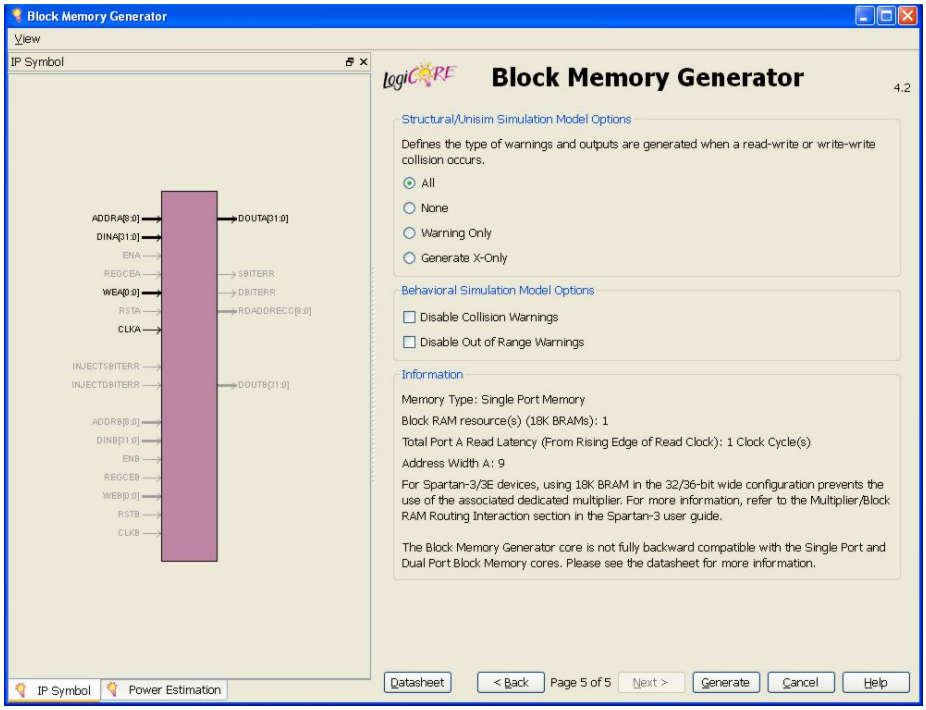


Fig.34

When the memory is added to the project, you will see in the top module (CM0_DSSystem.vhd), in the memory instantiation section (Inst_Memory : Memory at line 174), that the there is a “misaligment” between the memory address bus and the processor address bus (see “addra” signal at line 180):

Memory address Bus[0] < - - > Processor Address Bus[2]
Memory address Bus[1] < - - > Processor Address Bus[3]…

and so on. Also, the processor’s address bus bits [1] and [0] are discarded.
This is because on the Cortex-M0, all data accesses need to be aligned on the appropriate memory addresses, otherwise the processor does an unaligned data access and takes data from the wrong place. So, as the processor always performs a 32-bit data fetch on a memory access in hardware, if the data needed by the software is less than a 32 bits, it's trimmed from the 32-bit word fetched. So, the memory layout should always be 32 bits in length. As an example, suppose this data:

mem address 0x00: 0xFEDCBA98
mem address 0x01: 0x76543210

(In Xilinx ipcore memories configured for 32-bit word length, each memory address holds 32 bits/4Bytes, so memory address 0x01 stands for the 5th byte).

So, if the software asks for the byte at address 0x02, the processor puts 0x00000002 in HADDR, and takes the 3rd quarter of the 32-bit word fetched on the HRDATA bus at memory address 0x00 (0xDC in this case). If the software asks for the half word (16-bit) at address 0x02, the processor puts 0x00000002 on HADDR, and takes the 2nd half of the 32-bit word fetched in HRDATA bus at memory address 0x00 (asking for the halfword at address 0x01 generates an exception), and so on. If the software asks for the 32-bit word at address 0x04, the processor puts 0x00000004 on HADDR and takes the full 32-bit word fetched in HRDATA bus from memory address 0x01. Further, asking for a word at address 0x01, 0x02, 0x03, 0x05, 0x06, or 0x07 generates an exception because word access must be 4-byte aligned and halfword access must be 2-byte aligned.

So, the misalignment between the processor address bus and the memory address bus not only avoids an unaligned access in hardware, (if the software does an unaligned access because of bad pointer arithmetics, the exception still happens), it also simplifies the memory device to use. Only a simple 32-bit width memory is needed and the routing problems associated with accesses to sub-word data in hardware can be dismissed.

Once all of the modules are correctly generated and added to the project, the system is ready for a functional simulation:

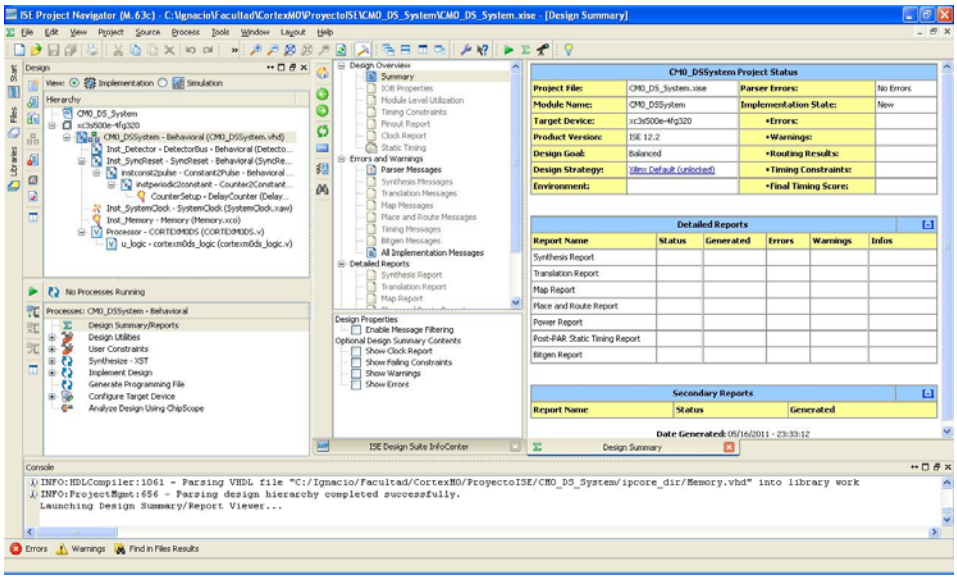


Fig.35

In this section:

- A new Xilinx ISE project was created and a basic system using the Cortex-M0_DS processor and other modules were created, configured, and added to the project. All of these components are needed to run a precompiled program.

Functional Simulation

Functional simulation of the system can verify that the signals in the data read bus (HRDATA) are the expected ones and that the detector module works as expected. The ISIM tool (ISE Integrated Simulator) can be used to perform the simulation. Select the simulation option in the Design View ("Design -> View: Simulation").

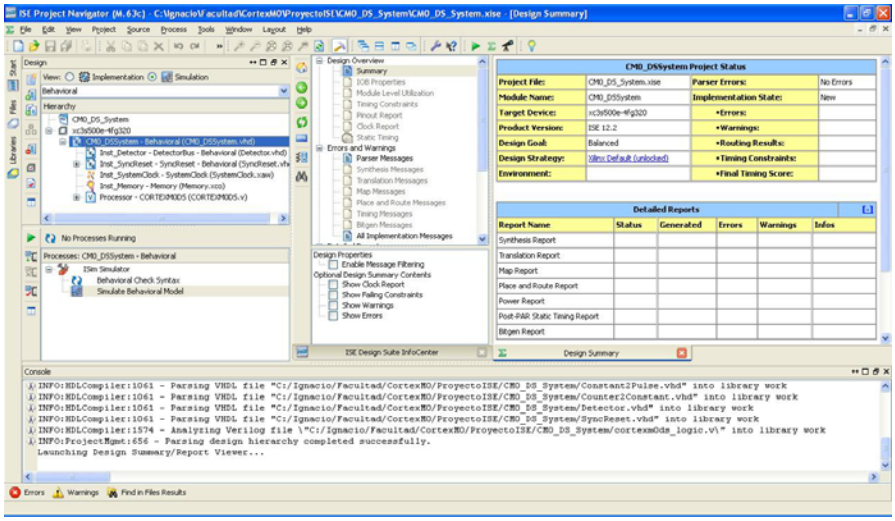


Fig.36

Highlight the top module (CM0_DSSystem) and run the "Simulate Behavioral Model" process. Upon completion, the ISIM application starts.

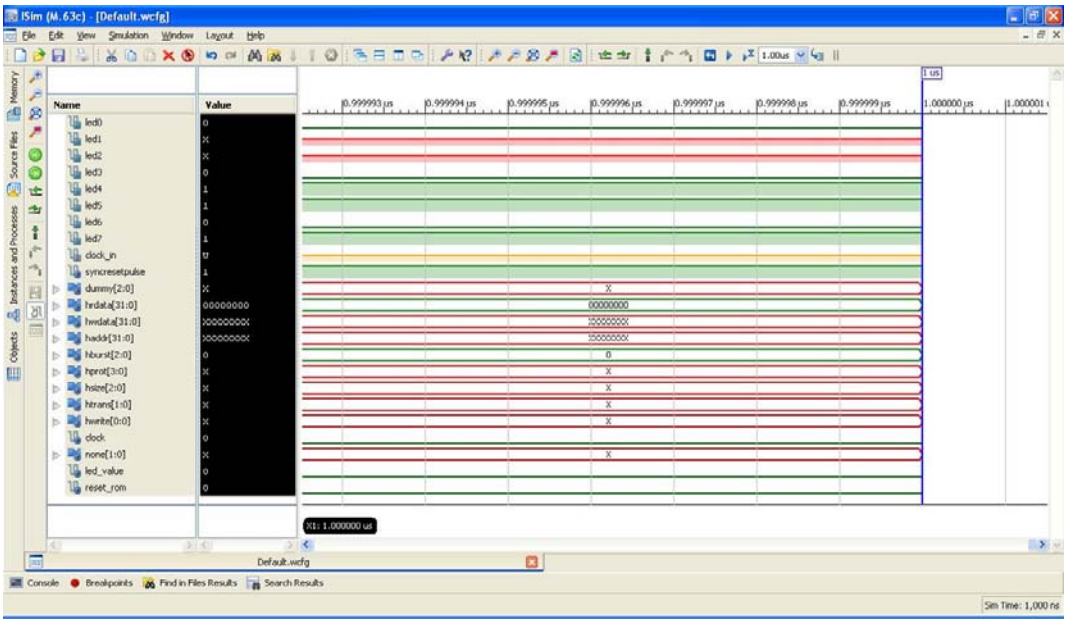


Fig.37

By default the simulator runs for 1 uS, but the clock signal is not yet defined, so it's necessary to restart the simulator with the "Restart" button and force the clock signal that comes from the board's oscillator. To do this, select the "clock_in" signal and click with the right mouse button, then select the option "force clock" in the contextual menu. The forced signal parameters are shown in this figure:

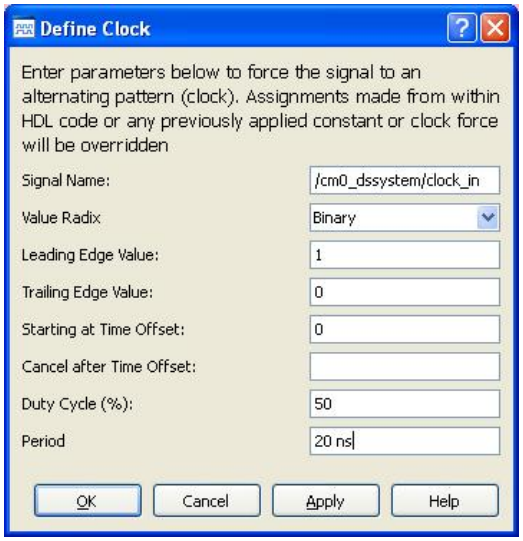


Fig.38

The period of 20 nS is for the 50 MHz oscillator in the Nexys2 board. Press the "OK" button and set 300 uS as simulation time. Then, simulate it by pressing the "Run for the time specified in the toolbar" button.

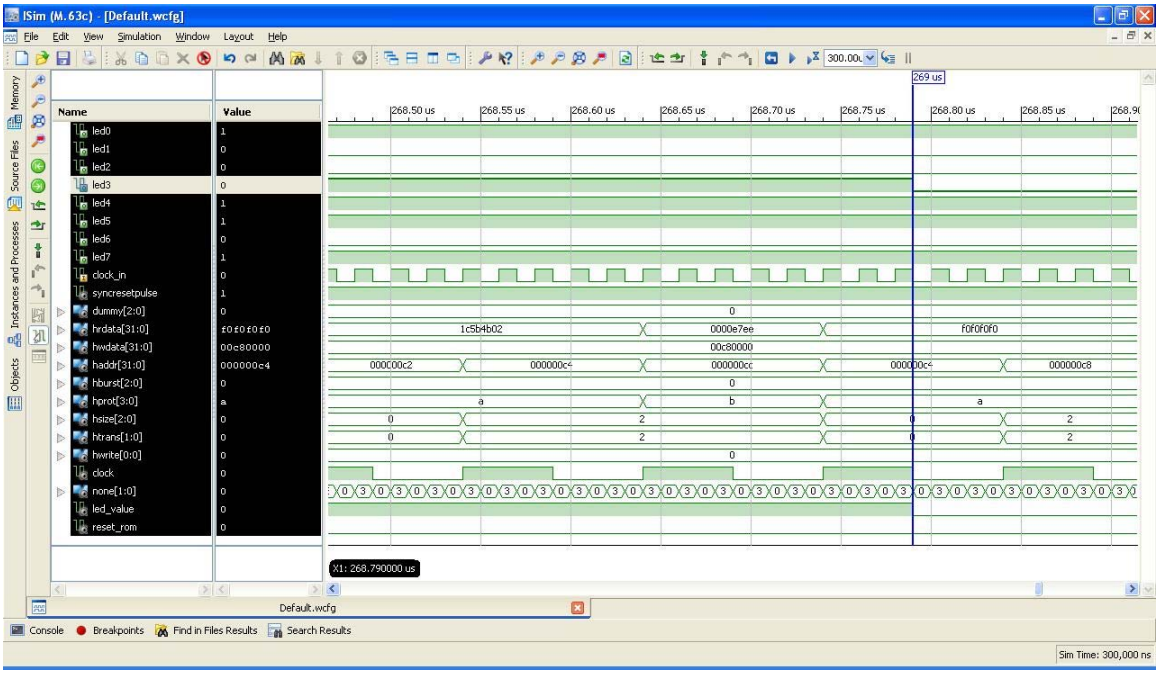


Fig.39

When the simulation completes, note that at simulation time 268.79 uS there's a 0xf0f0f0f0 pattern on the read data bus (HRDATA), and the detector module output (LED03) changes state from '1' to '0'. Compare this value with the one from software simulation (242.20 uS). The difference is because the software simulator timer starts at the "main" function (assembler instruction MOVS r2,#0xC8). This instruction is executed at simulation time 26.49 uS, so this simulation shows the system working as expected from the software simulation.

Note that if the software uses a "period" value less than 511 (0x1FF), as in this case (remember that "period" is 200 for simulation), you will see that the LED3 turns off for a very short period of time (a "glitch") instead of equal periods for '1' and '0'. This is due to the nature of the processor pipeline. When the "period" is less than 0x1FF, the memory image generated is such that when the software turns off the led, at execution of the "while(1)" branch, the processor is also fetching the next memory position (which has the 0xaaaa5555 value).

You can see the memory map in the debugger/simulator in ARM/Keil MDK that with a "period" less than 511, the 0xaaaa5555 constant is right next the "while(1)" assembler branch instruction. So, it's fetched at branch execution and the value appears in the HRDATA bus activating the Detector module output and changing the LED3 value back to '1'. When the "period" is greater than or equal to 511, the assembler generates a constant in memory to represent the "period" value, and this constant is placed right next the "while(1)" branch. So, LED3 has equal periods of '1' and '0' instead the of the "glitch" observed with the "period" values less than 511. You can see this in the memory map if you simulate the software with a "period" value of 512 or greater.

Other important simulation times and signals:

- At 25.14 uS the reset signal goes from '1' to '0', so the processor signals "Reset" and "Lock" start to have a defined value.
- At 25.64 uS the processor leaves the reset state. You can also see that the reset state lasts for five "clock" signal periods (remember that "clock" signal is the system clock).
- At 26.94 uS there is the first appearance of the 0xaaaa5555 constant on the HRDATA bus, so LED3 signal has a defined state ('1').

- At 269.04 uS there is a transition from ‘0’ to ‘1’ in the detector signal (LED3), so the glitch (because a “period” value lesser than 512) lasts for three “clock” periods.
- LED0 is the DCM “lock” signal, indicating that the DCM is working.
- LED1 is the processor “sleep” signal, and it should always be ‘0’.
- LED2 is the processor “lock” signal, and it should always be ‘0’.
- LED3 is the detector output. It has a ‘1’ to ‘0’ transition with the constant 0xaaaa555 on the HRDATA bus and has a ‘0’ to ‘1’ transition with the constant 0xf0f0f0f0 one the HRDATA bus
- LED4 is the processor “reset” signal.
- LED5, LED6, and LED7 have the value ‘101’ hardcoded.

In this Section:

- The correctness of the system was verified using a functional simulation with the ISIM tool. A correspondence between the software simulation time and the functional simulation time was also verified.

Hardware Verification

In this Section, implement the system in real hardware and check it using the ChipScope Pro tool. To do this, add information about the hardware outside the FPGA (the external oscillator, LEDs, etc.). Then add an ILAC (Integrated Logic Analyzer Core) module to see the internal signals of the system.

To add information about the external hardware, generate a restrictions file (Unified Constraints File – UCF). It's called a "Constraints File" because this file also contains the timing and placement constraints of the system as needed. In this simple system there are no placement contraints; in a more complex system it could be useful to add a placement constraint (using the FloorPlanner tool) for the CM0_DS processor, so it won't be resynthesized every time there is a change in the design. So, in "project", select "new source" from type "Implementation Constraints File" and call it "CM0_DSSystem", as the top module.

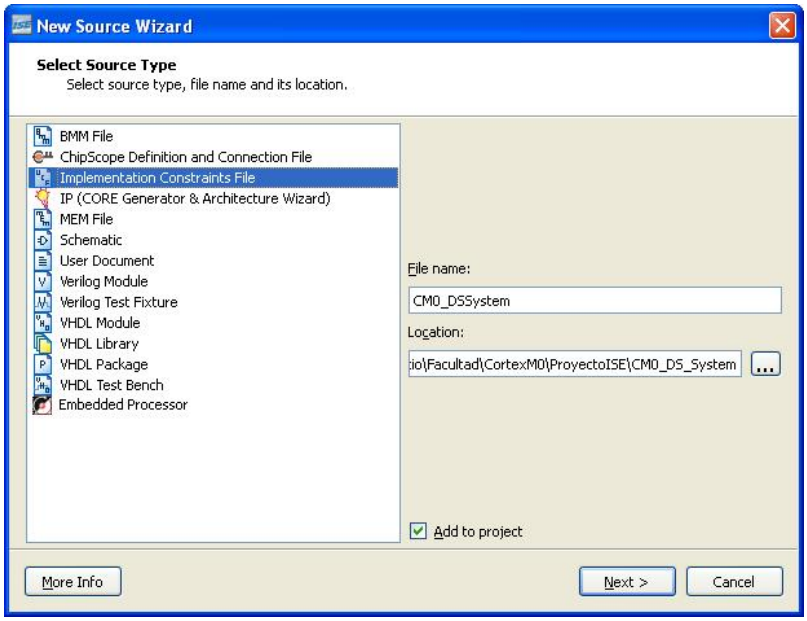


Fig.40

This adds to the project an empty file called "CM0_DSSystem.ucf". Edit it to add the information about the board. The information to add for the Nexys2 board is in the "Source Code" section and in Digilent's web site. Other boards should have their own .UCF files. The restrictions added include the 50 MHz external oscillator input and the LED positions.

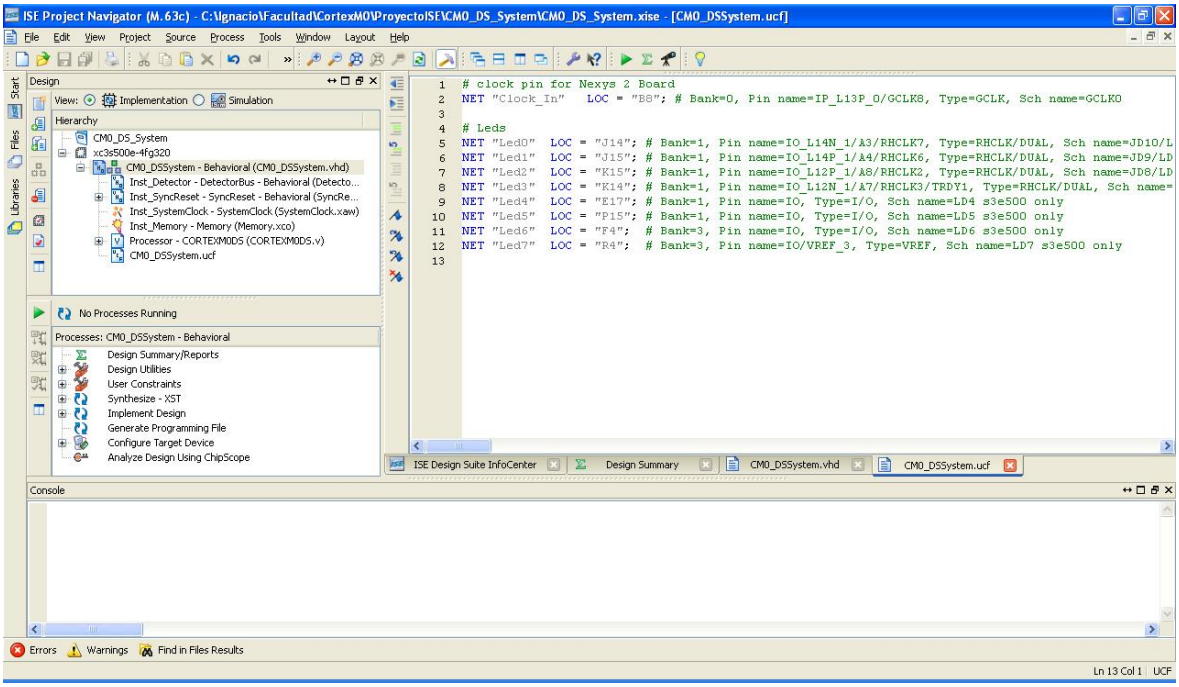


Fig.41

Now it's time to add the ILAC module. Add the ChipScope Pro definition and connection file. This file will have the information about the signal connections that will be monitored and other configuration parameters of the ILAC module. So, in "project", select "new source" from type "ChipScope Definition and Connection File" and call it "LogicAnalyzer".

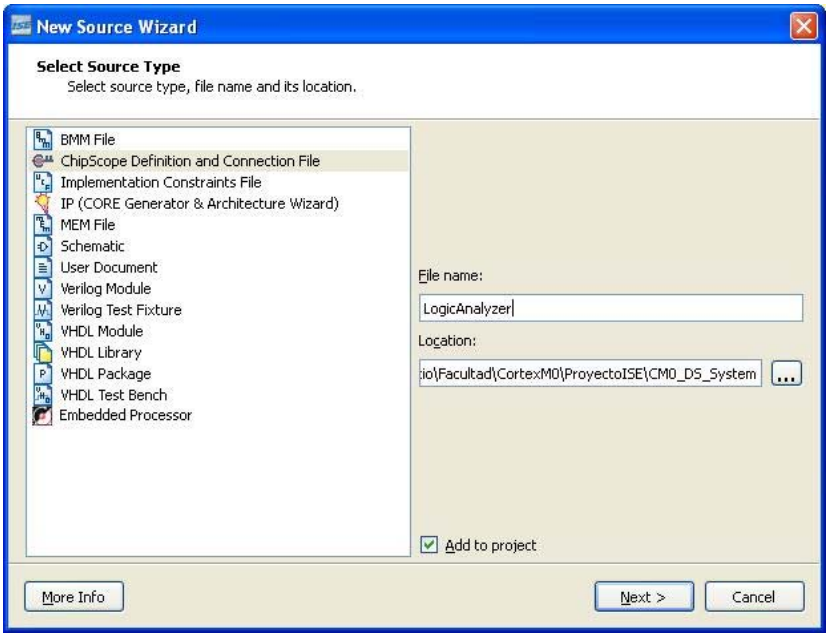
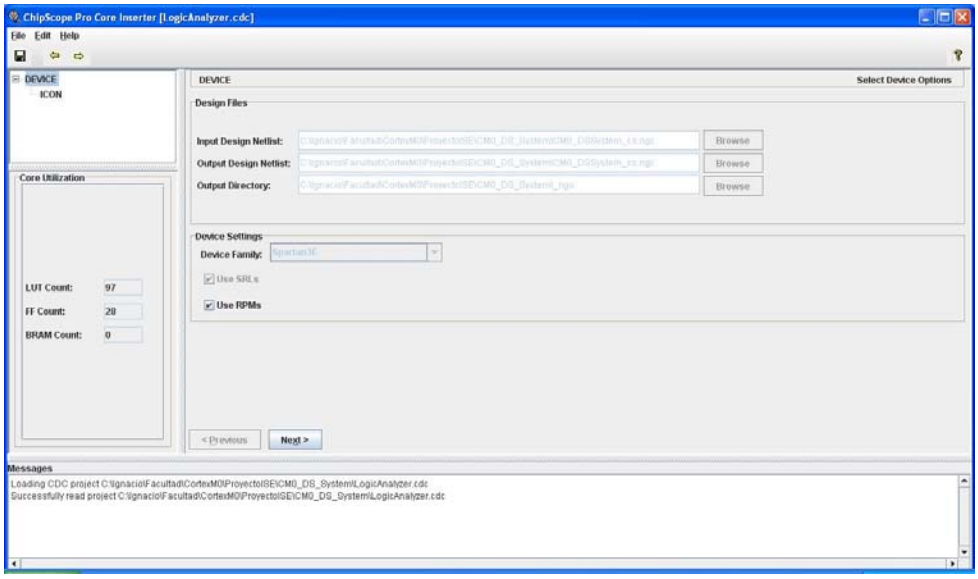


Fig.42

A new file will be added to the project called “LogicAnalyzer.cdc”. Double click on it, the system will be synthesized and you will be able to add an ILA module (Integrated Logic Analyzer). After the “Synthesis” process ends, a new window will pop-up. This is the configuration window for the ChipScope Pro tool. Press the “Next” button.



Create the new ILA module using the button “new ILA unit” at the bottom right of the window.

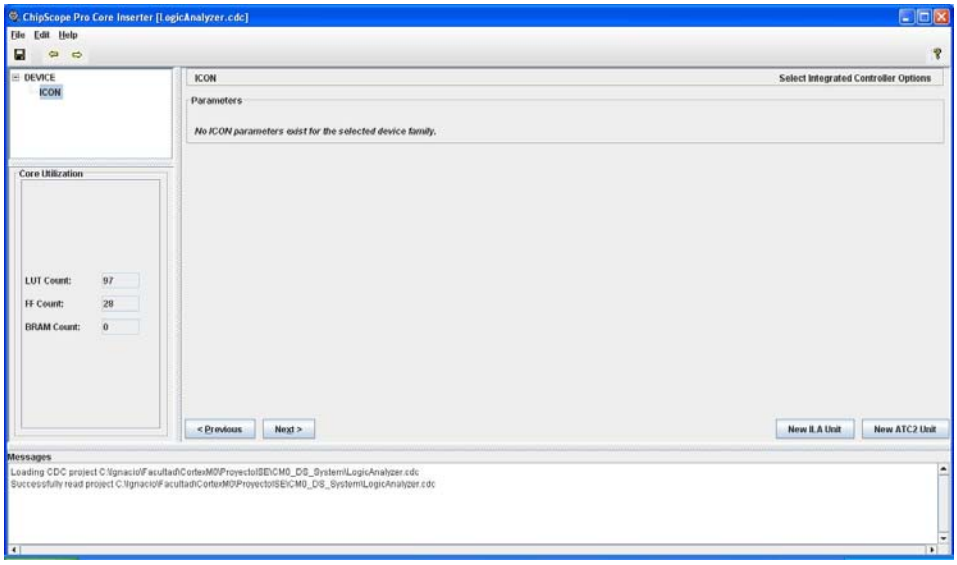


Fig.44

A new ILA module appears at the top left of the window.

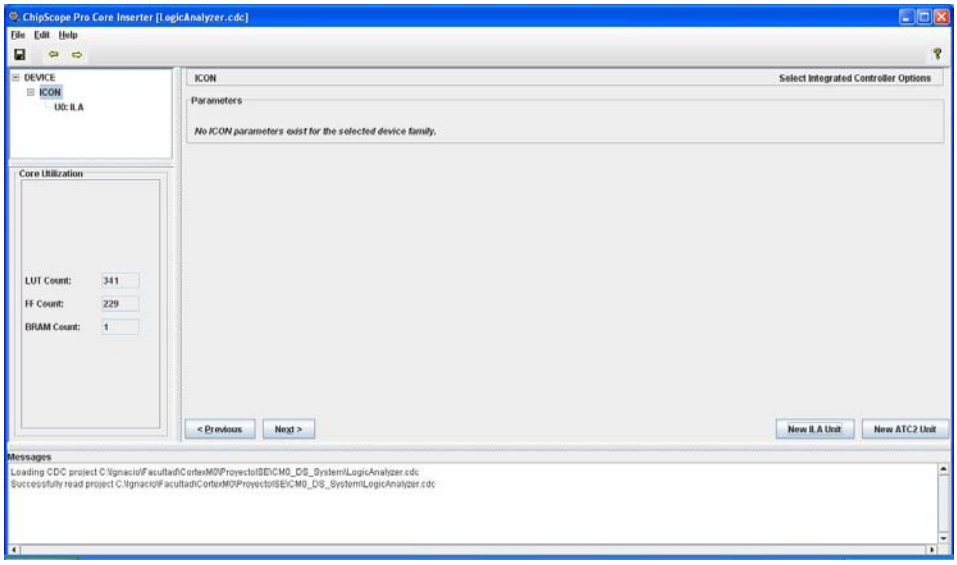


Fig.45

Select the ILA module, and its configuration window appears

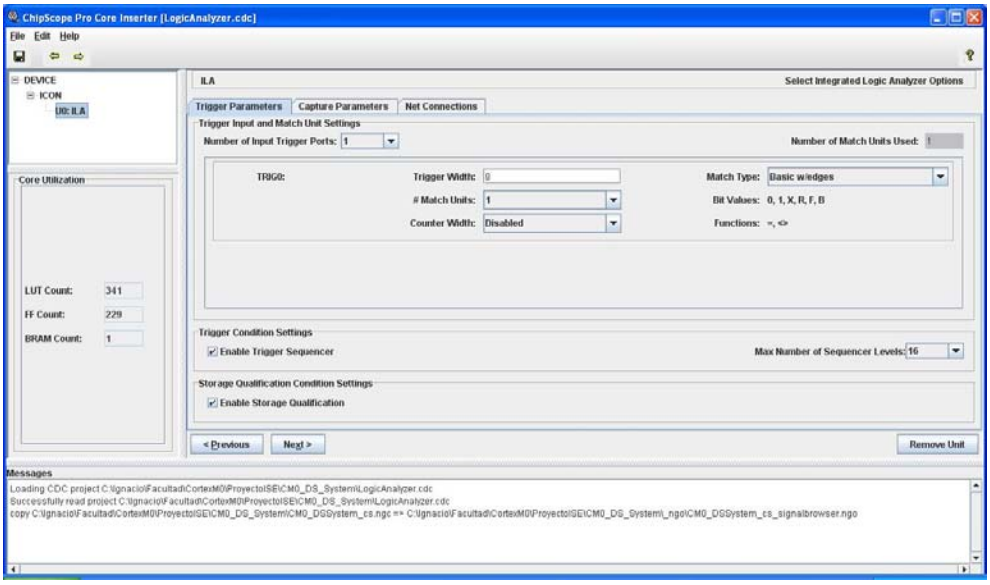


Fig.46

Edit the trigger parameters as in Fig. 47. Change only the Trigger Width from ‘8’ to ‘32’. This instructs the monitoring hardware on what conditions it should start taking samples of the signals. In this case the trigger condition is the value of the HRDATA bus (hence the change to ‘32’). There will be only one trigger condition and it will based on a

comparision against a constant value (the specific value will be set in “Analyze Design using ChipScope” process ahead).

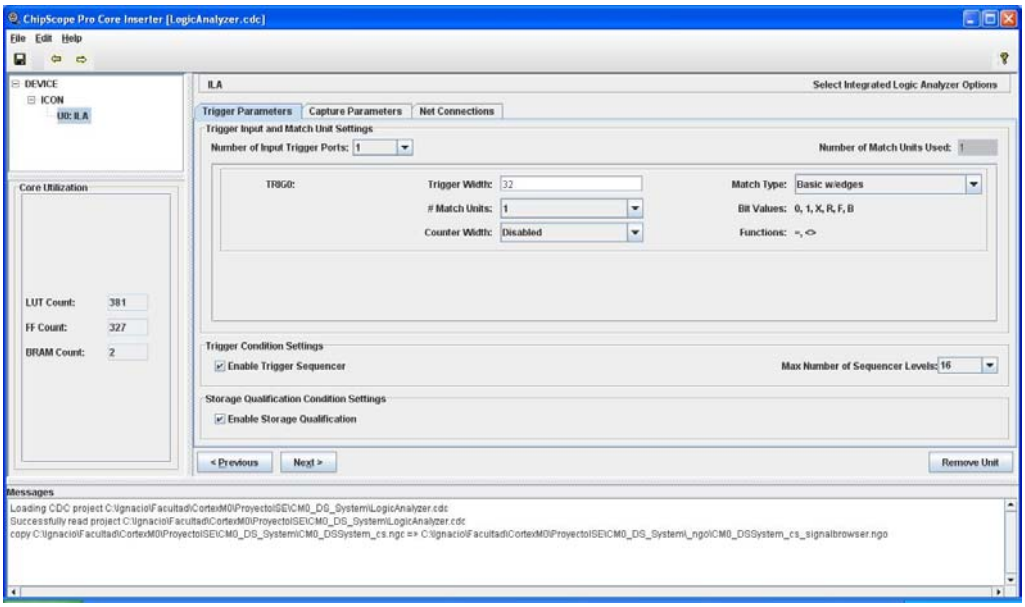


Fig.47

Press the “Next” button. Now configure the capture parameters. The signals of interest are HRDATA bus (to see the constant value we need the full 32 bits), the HADDR bus from [2] to [10] (remember that HADDR[0] and [1] were discarded and the memory has 512 words, so we need 9 bits for the memory address), and LED3 (Detector module output). So the data width is 42 bits. Untick the “Data Same as Trigger” checkbox and set the Data Width to 42. The Data Depth default value of 512 samples is enough, and each sample will be taken at the rising edge of the ILA clock signal (it will configured next). After those changes, press the “Next” button.

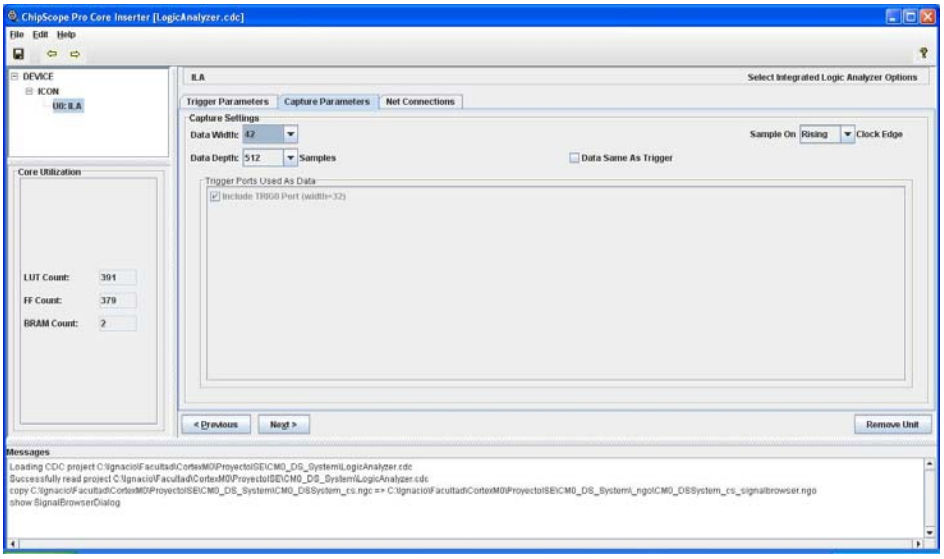


Fig.48

Here assign the different signals from the system to the trigger, data, and clock ports of the ILA module.

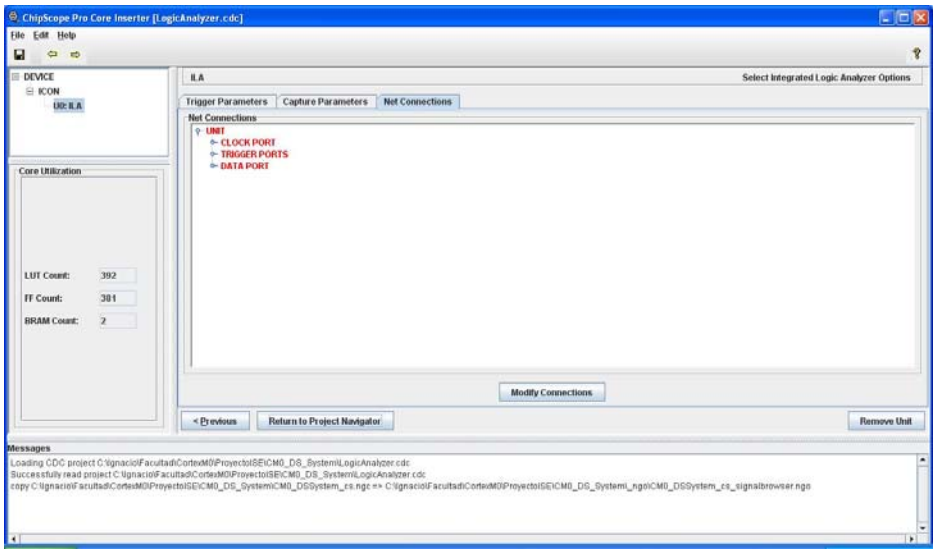


Fig.49

Press the “Modify Connections” button and a new window will pop-up with the trigger signals in red.

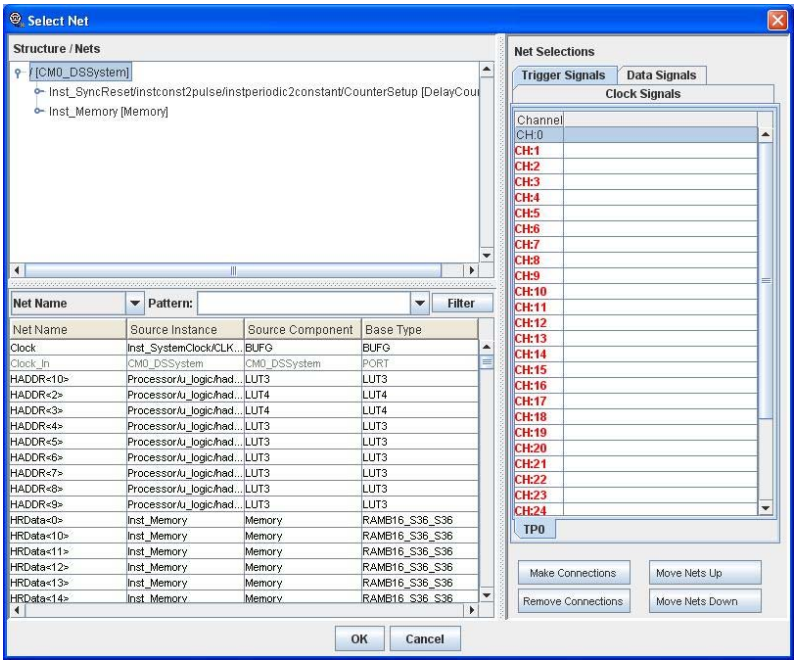


Fig.50

Associate each trigger signal with the corresponding HRDATA value (CH:0 with HRDATA<0> and so on) with the “Make Connections” button (it is possible to select a group of signals and associate them all at once).

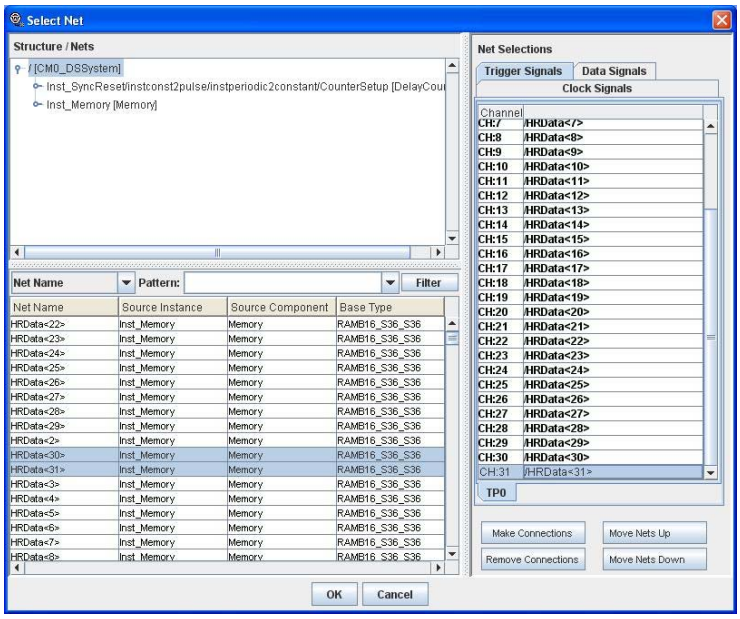


Fig.51

Now select the “Data Signals” tab (top rigth of the window) and associate the data:

- CH:0 to CH:31 with HRDATA[0]..[31]
- CH:32 to CH:40 with HADDR[2]..[10]
- CH:41 to LED3_OBUF

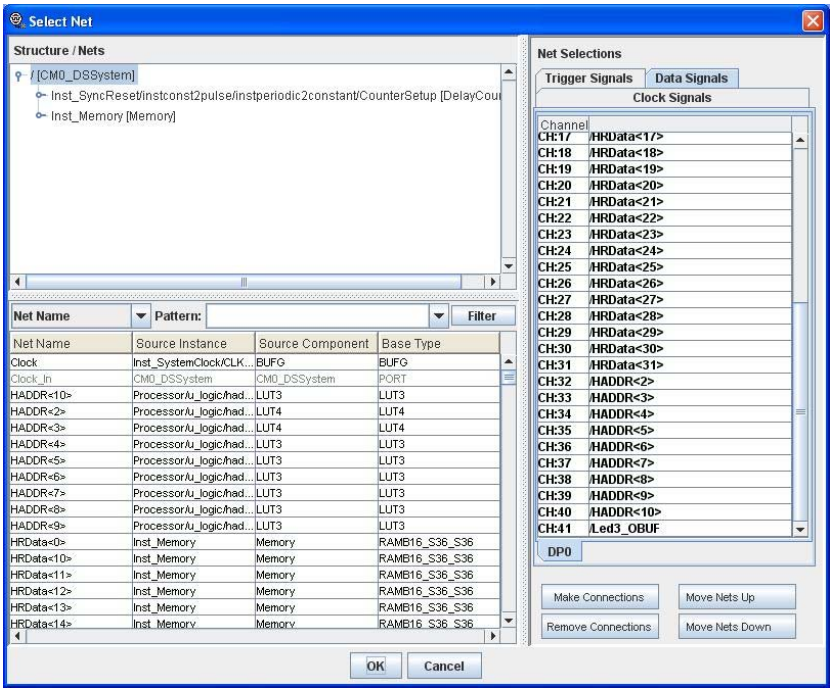


Fig.52

Finally, select the “Clock Signals” tab and associate CH:0 with “Clock” signal. Press “OK” button.

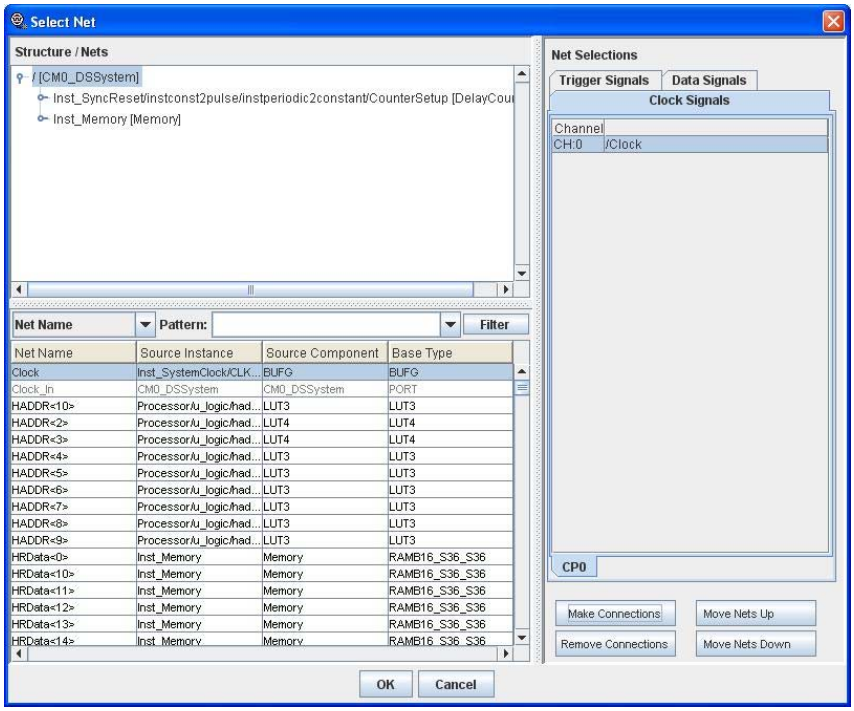


Fig.53

The Clock, Trigger, and Data Ports are now colored black, indicating that all connections were made. Press the “Return to Project Navigator” button and a “Save Project” pop-up window will appear. Select “Yes”. You will return to the Project main window.

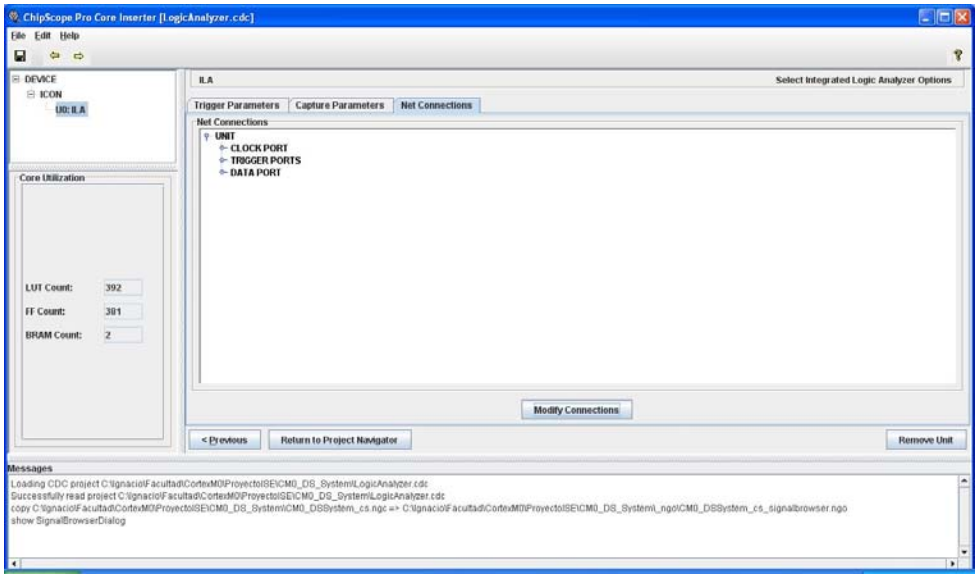


Fig.54

Now highlight the top module and implement the project to generate the "CM0_DSSystem.bit" bitstream file that will be downloaded to the FPGA. To do that, run the "Generate Programming File" process. When all the processes finish, run the "Analyze Design using ChipScope" process. The ChipScope Pro main window will popup.

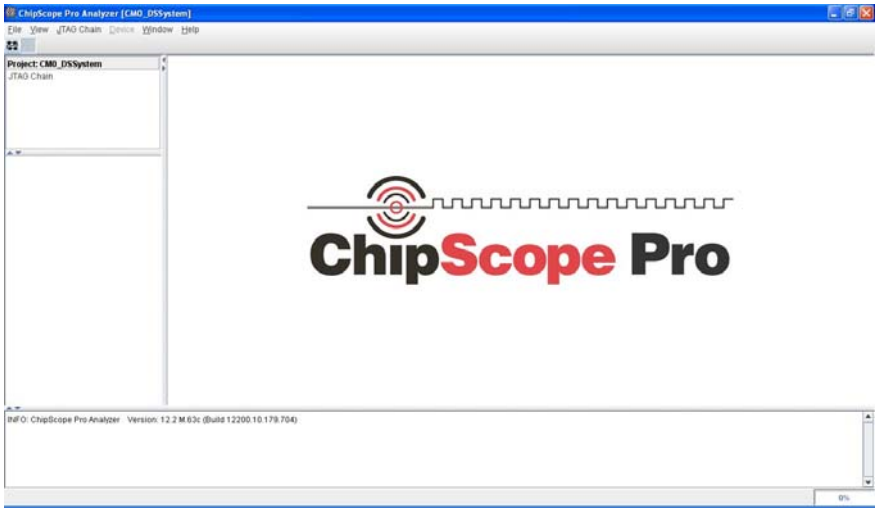


Fig.55

Now the bitstream file generated in "Generate Program File" process above will be downloaded to the FPGA board using the JTAG chain. Normally this is done with a Xilinx programming cable like the Platform USB II, but thanks to Digilent's "Plugin for Xilinx Tools" software and some extra logic in the Nexys2 board, this can be done using a USB connection. The details to configure Adept to be recognized as a programming cable are bundled with the plugin. So, in the ChipScope Pro screen, press the "OpenCable/Search JTAG Chain" button at the top left of the screen and a popup window will appear with two devices (the FPGA and the PROM). Press the "OK" button. Those devices will show up in the top left of the ChipScope Pro main window.

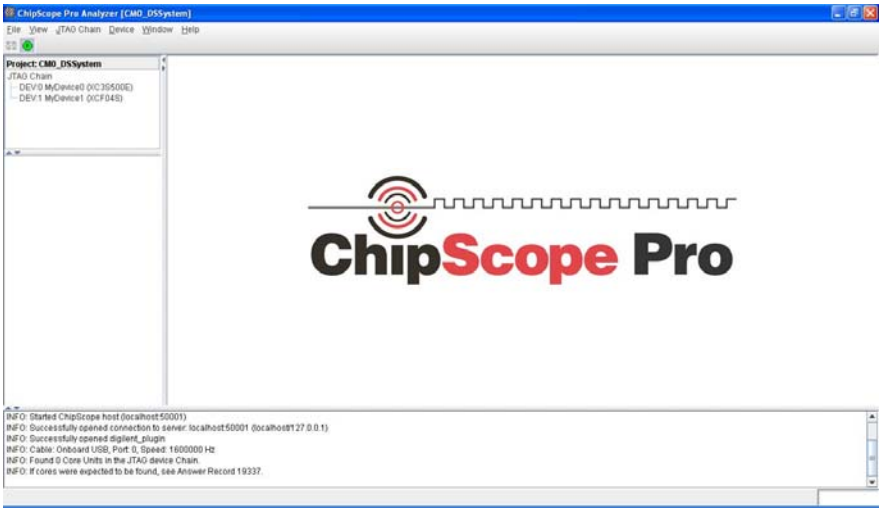


Fig.56

Go to the “Device” menu and select “DEV:0” device (the FPGA), a sub menu will appear, select the “Configure…” option.

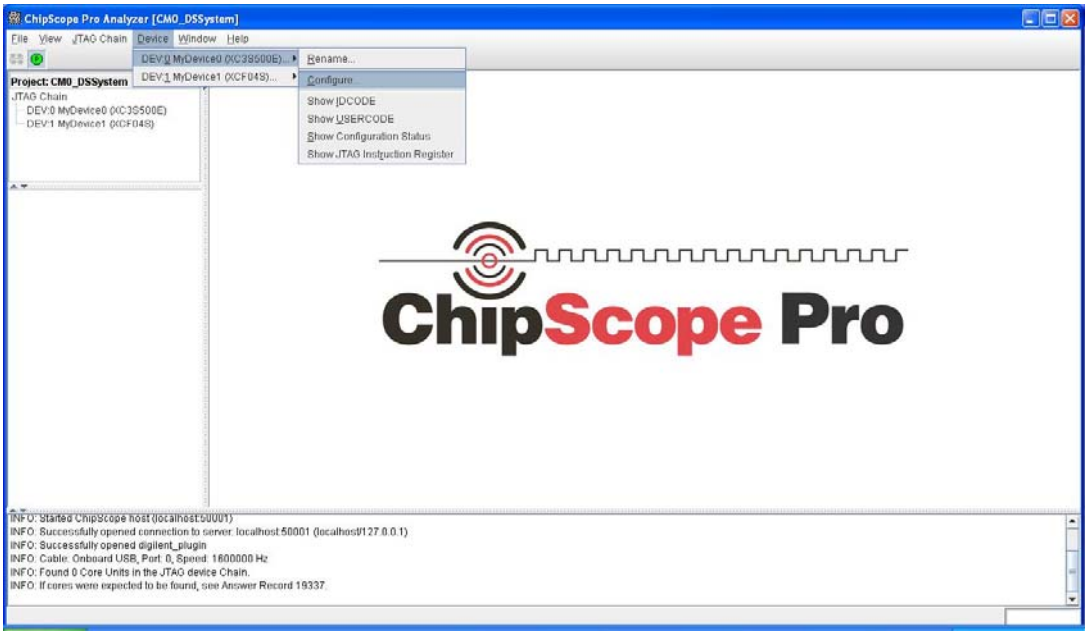


Fig.57

A popup window will appear. Leave the defaults and press the “OK” button. The FPGA will be programmed with the bitstream file generated above and the ILA core will appear in the top left of the screen.

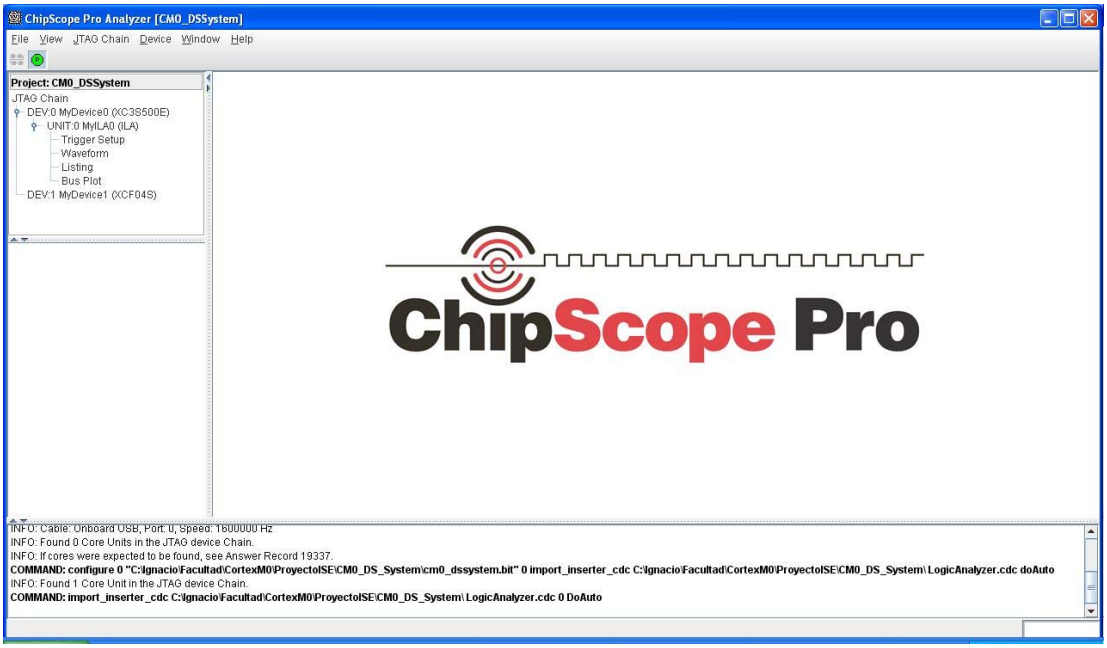


Fig.58

Select the ILA core and its signals (the ones that were configured above) will appear in the empty window under the JTAG Chain window in the left.

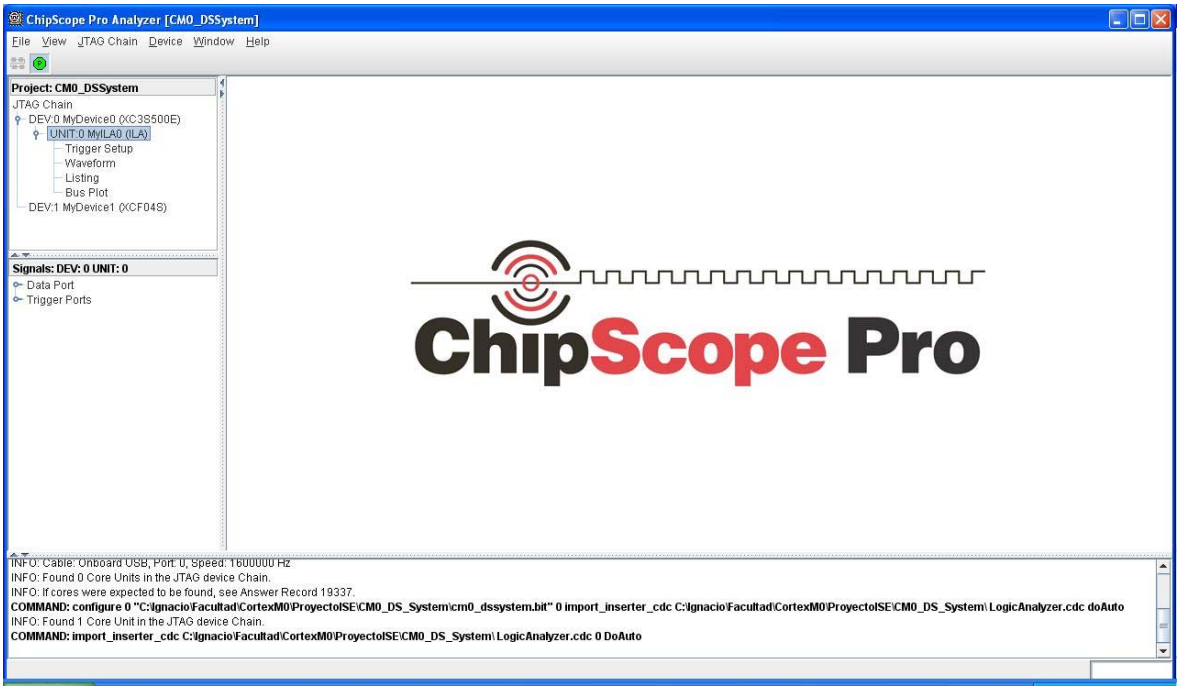


Fig.59

Now double click in the “Trigger Setup” subtree in the “JTAG Chain” window, and the trigger setup window will appear in the main window.

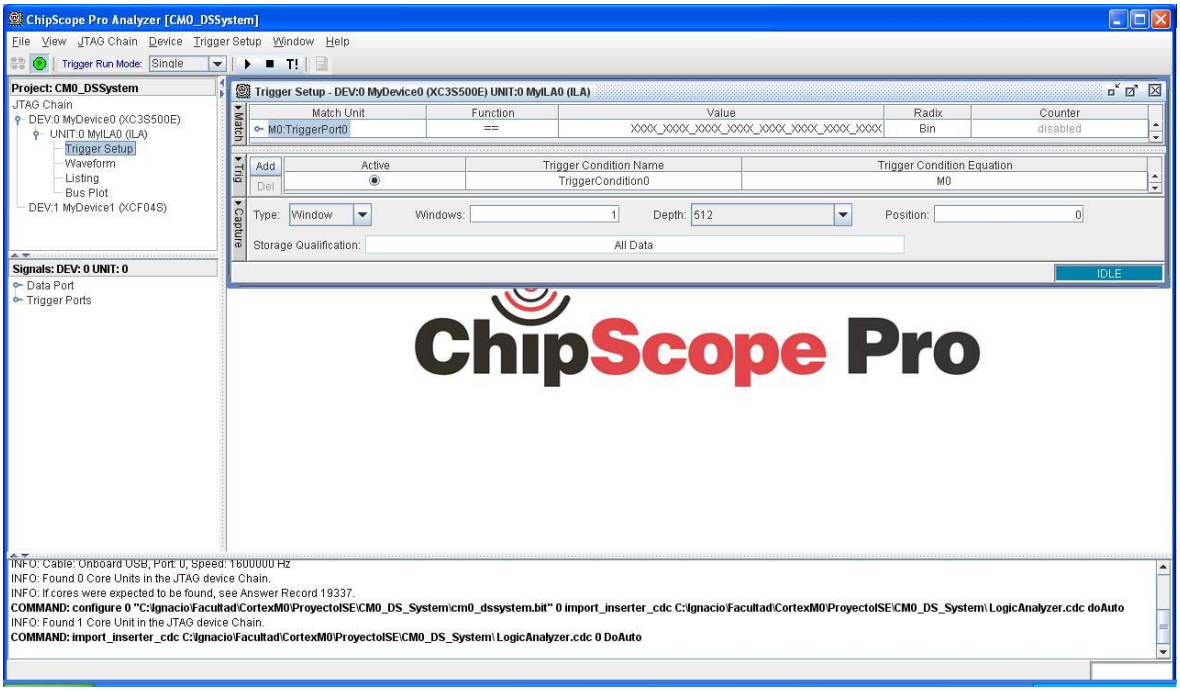


Fig.60

Now program the trigger condition to see the ‘0’ state of the LED3 signal: the 0xf0f0f0f0 pattern in the HRDATA bus makes a ‘1’ to ‘0’ transition, and there is a trigger condition based on a comparision to a constant. So in “Trigger Setup” set “1111_0000_1111_0000_1111_0000_1111_0000” as the trigger value.

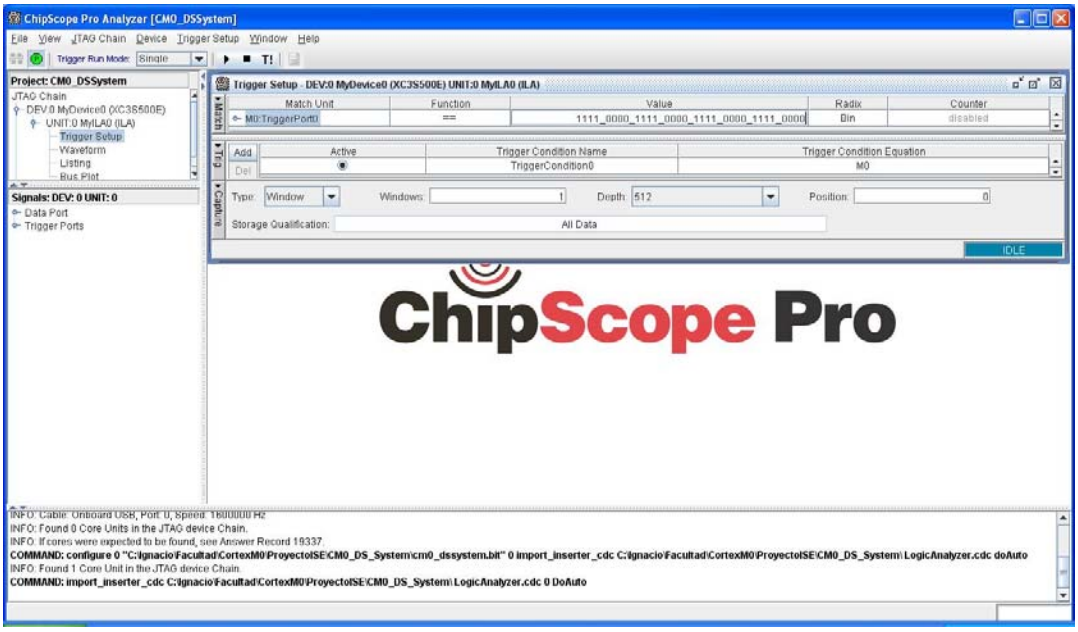


Fig.61

After that, start the acquisition using the “play” button. In few moments the trigger condition is met and the data acquisition is completed (a “Sample Buffer is full” message will appear). Click the “Data Port” subtree in the “Signals” window, right-click, and in the submenu select “Add All to View - > Waveform”.

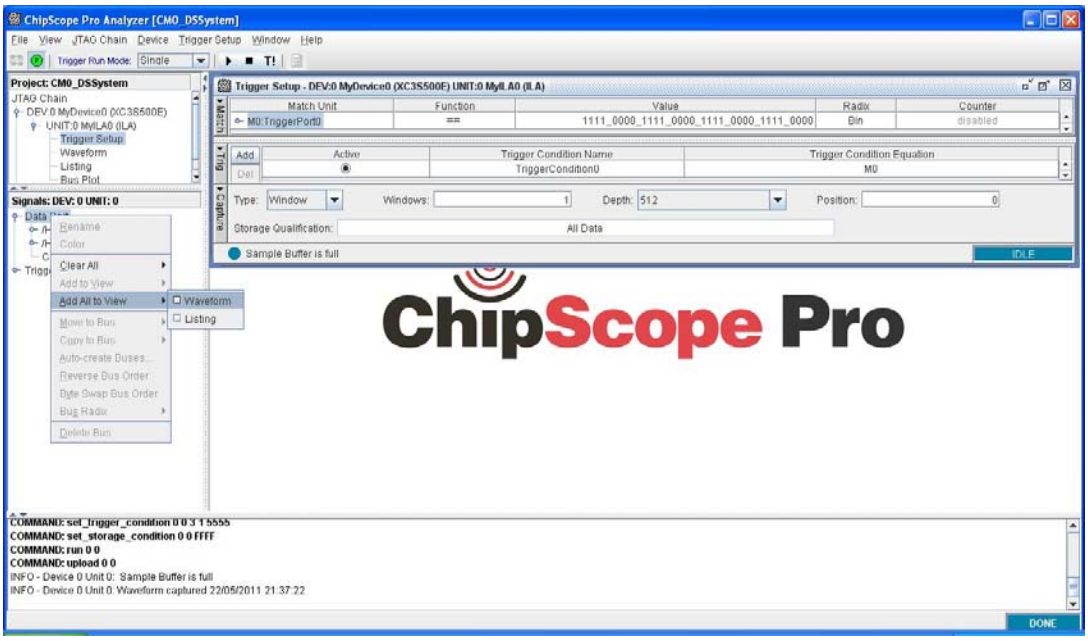


Fig.62

Now all the configured signals are in the main window. You can zoom-in the window to see the values at HADDR and HRDATA when LED3 is '0' and delete the duplicated signals.

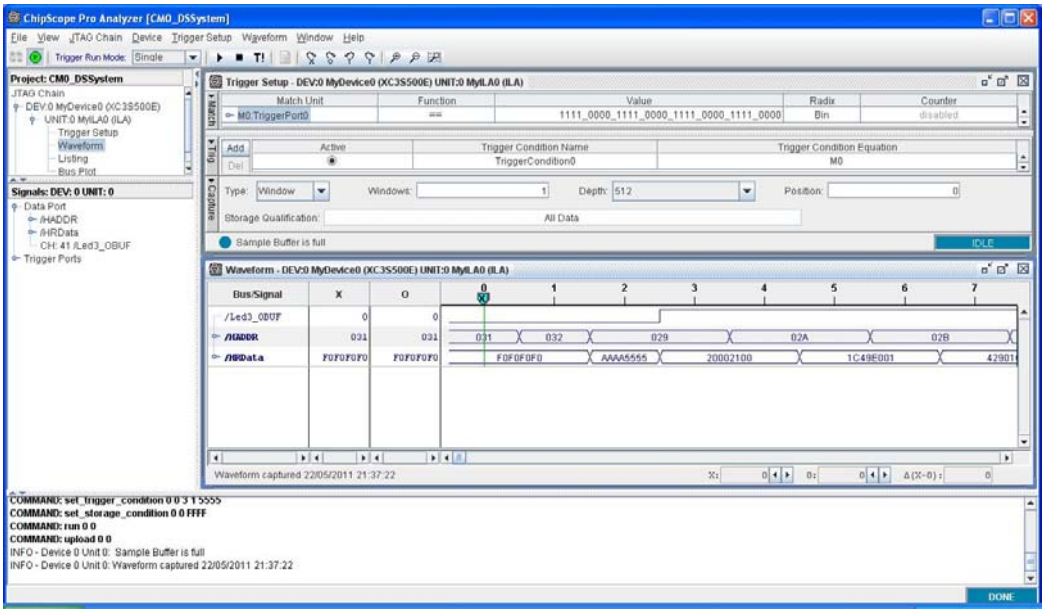


Fig.63

Note that LED3 is '0' for three samples, the same number of rising edges of the system clock signal in the functional simulation (remember that the ILA core was configured to take the samples in the rising edge of the system clock). Also note that the value 0xf0f0f0f0 appears for two samples and the value 0xaaaa5555 appears for one single simple. This is exactly the same number of rising edges of the system clock in the functional simulation. So there is a perfect correspondence between the software simulation, the functional simulation, and the hardware verification.

Now change the "period" value in the software to 20000000, recompile it, regenerate the .COE memory initialization file, and regenerate the memory module changing the .COE initialization file to the one with the new "period" value. Then regenerate the bitstream file and download it to the board (iMPACT tool can be used instead of ChipScope Pro). With this new value, the LED3 will blink with close three seconds in between each blink on the board.

In this Section:

- The patterns 0xf0f0f0f0 and 0xaaaa5555 were verified as appearing on the HRDATA bus inside the FPGA.
- It was checked that the same timing results appeared in the software simulation at ARM/Keil MDK, the functional simulation with the ISIM tool, and the hardware verification with the ChipScope Pro tool.
- It was visually checked that the LED on the board blinked with a period near three seconds and the other LEDs were ‘on’ or ‘off’ according to the functional simulation results.

Conclusions

A step by step implementation of the Cortex-M0_DS processor running a defined software program was performed in a Xilinx FPGA. This includes the software simulation, the system’s functional simulation, and the system’s real hardware implementation using the Xilinx ISE toolchain.

Acknowledgements

To the ARM University Program, including William Hohl and Joe Bungo, as well as the people at the Xilinx University Program (XUP) for their support and cooperation.

Legal Notice

Source Code

Main.c

```
// Define where the top of memory is.
#define TOP_OF_RAM 0x800U

// Define heap starts...
#define         HEAP_BASE 0x47fU

//--------------------------------------------------------------------------
// Simple "Blinking Led via Memory Access detection" program.
// This program makes a memory access at regular intervals
// In the Nexys2 system there is a pattern detector attached to the
// HWRead bus, so when two specific patterns are detected, a Led toggles its state
// pattern 0xaaaa5555 turns on the led, pattern 0xf0f0f0f0 turns it off.
//--------------------------------------------------------------------------

#define LedOn       0xaaaa5555
#define         LedOff          0xf0f0f0f0

int main(void)
{
        unsigned int counter;         // dummy
        unsigned int ii;              // loop iterator
        unsigned int trap;            // memory access pattern receiver
        unsigned int period; // time interval for memory access

        //period=20000000;  // period for FPGA implementation; roughly 3 seconds for a
10MHz osc in CM0_DS
        period=200;  // period for simulations in ARM/Keil MDK and Xilinx ISIM tool

        while (1)
        {
                counter=0;
                for (ii=0;ii<period;ii++)
```

```
                {
                        counter++;
                }
                trap=LedOn; // memory access pattern (turn on)
                for (ii=0;ii<period;ii++)
                {
                        counter++;
                }
                trap=LedOff;   // memory access pattern (turn off)
                trap++;            // dummy
        }
}
```

vectors.c

```
// Define where the top of memory is.
#define TOP_OF_RAM 0x400U

extern int main(void);     // Use C-library initialization function.

__attribute__ ((section("__Vectors")))
static void (* const vector_table[])(void) =
{
  (void (*)(void)) TOP_OF_RAM, // Initial value for stack pointer.
  (void (*)(void)) main,                    // Reset handler is C initialization.
  0,                        // No HardFault handler, just cause lockup.
  0,                        // No NMI handler, just cause lockup.
  0//...                    // Additional handlers would be listed here.
};
```

UCF File:

(extracted from the ucf file for the Nexys2 board available at Digilent's web site)

```
# clock pin for Nexys 2 Board
NET "Clock_In"   LOC = "B8"; # Bank=0, Pin name=IP_L13P_0/GCLK8, Type=GCLK, Sch name=GCLK0

# Leds
NET "Led0"  LOC = "J14"; # Bank=1, Pin name=IO_L14N_1/A3/RHCLK7, Type=RHCLK/DUAL, Sch name=JD10/LD0
NET "Led1"  LOC = "J15"; # Bank=1, Pin name=IO_L14P_1/A4/RHCLK6, Type=RHCLK/DUAL, Sch name=JD9/LD1
```

```
NET "Led2"  LOC = "K15"; # Bank=1, Pin name=IO_L12P_1/A8/RHCLK2, Type=RHCLK/DUAL, Sch name=JD8/LD2
NET "Led3"  LOC = "K14"; # Bank=1, Pin name=IO_L12N_1/A7/RHCLK3/TRDY1, Type=RHCLK/DUAL, Sch name=JD7/LD3
NET "Led4"  LOC = "E17"; # Bank=1, Pin name=IO, Type=I/O, Sch name=LD4 s3e500 only
NET "Led5"  LOC = "P15"; # Bank=1, Pin name=IO, Type=I/O, Sch name=LD5 s3e500 only
NET "Led6"  LOC = "F4";  # Bank=3, Pin name=IO, Type=I/O, Sch name=LD6 s3e500 only
NET "Led7"  LOC = "R4";  # Bank=3, Pin name=IO/VREF_3, Type=VREF, Sch name=LD7 s3e500 only
```