TKJ electronics
Development with Ease

**TUT001**
**Tutorial**

# Porting the LM32 to Xilinx FPGAs

**How to port the LatticeMico32 soft core processor to a Xilinx FPGAs**

## Introduction

In this tutorial we will guide you thru the steps required to port the LatticeMico32, the 32-bit soft core processor from Lattice, to a Xilinx FPGA. In this tutorial we will be using the Spartan 6, XC6SLX25 (on a ZTEX FPGA Module).

Before we start it is required to download and install the necessary software tools. You would also need a proper JTAG-programmer, to transfer the design to the FPGA. Here is a list and download link of the software tools we are going to use:

- Xilinx ISE >12.2 – the WebPACK is fine
  - http://www.xilinx.com/support/download/index.htm
- LatticeMico32 System
  - http://www.latticesemi.com/dynamic/index.cfm?fuseaction=view_documents&document_type=65&sloc=01-01-08-11-48-02
- Verilog replacement files
  - http://www.tkjelectronics.dk/downloads/fpga/lm32/REPLACE
- Intel HEX to Xilinx Coefficient (.COE)
  - http://www.tkjelectronics.dk/downloads/fpga/lm32/IHEX2COE

It is also possible to download the final LatticeMico32 project for the ZTEX FPGA Module: http://www.tkjelectronics.dk/downloads/fpga/lm32/ZTEX_PROJECT

The ZIP-file includes everything needed to test the LED blinking application, using the LatticeMico32 core on the ZTEX FPGA Module. The ZIP-file contains:

- The LatticeMico32 project
- The Xilinx project
- The LatticeMico32 Managed C-code project.

## Steps

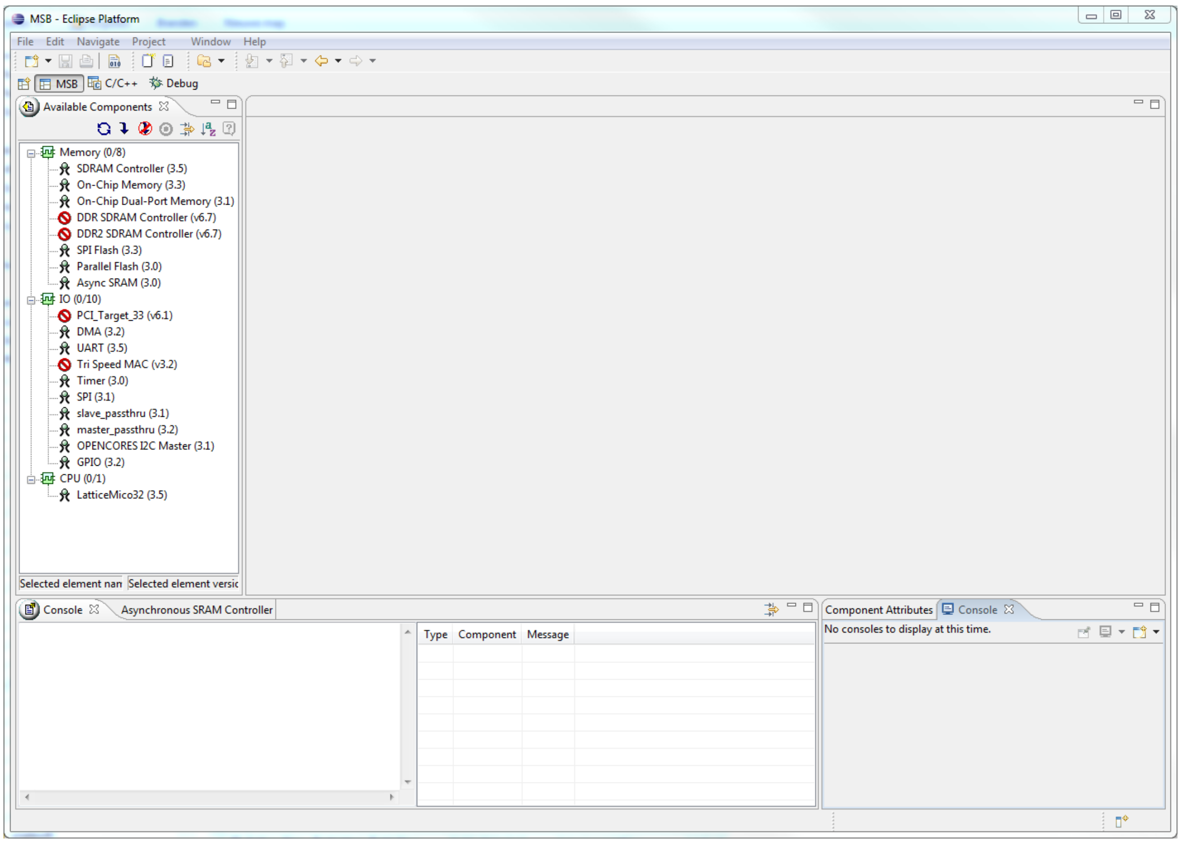
The overall steps in this tutorial are the following:

1. Create a LatticeMico32 project
2. Modify the generated Verilog files to match Xilinx FPGAs
3. Import the Verilog files into Xilinx ISE
4. Add the Xilinx Block RAM (BRAM)
5. Hello World example for the processor

## 1. Create a LatticeMico32 project

### Create the MSB

- Open the LatticeMico32 System IDE.

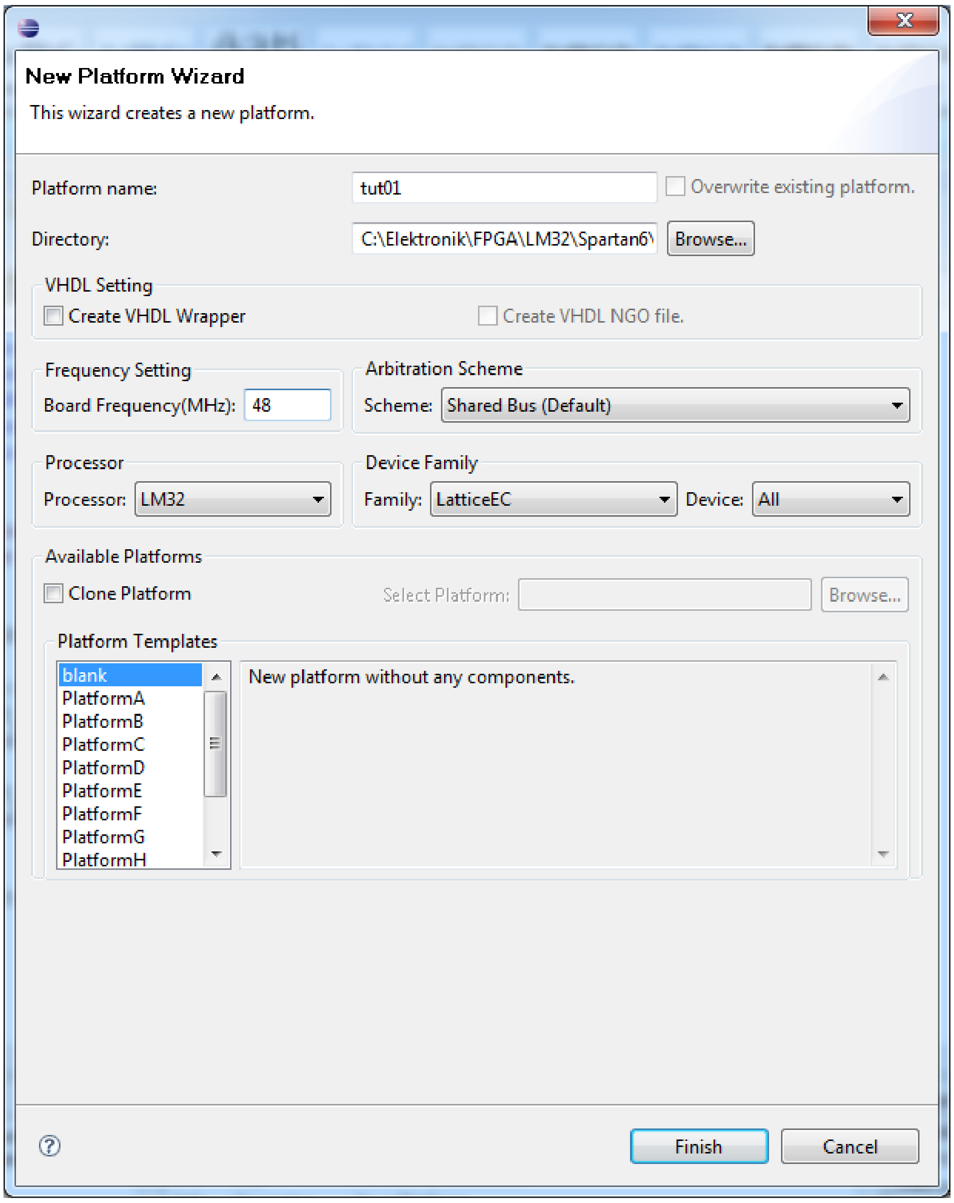


No consoles to display at this time.

After you have opened the IDE you will see the main window. As you probably notice, it is an eclipse based IDE. On the left the “hardware” part of our LatticeMico32 controller is listed, the so called IP Cores. The ones with a RED mark is licensed, and you would have to buy a license from Lattice to use it.

Let's start creating a project.

- File → New Platform


**New Platform Wizard**

This wizard creates a new platform.

Platform name: tut01 ☐ Overwrite existing platform.

Directory: C:\Elektronik\FPGA\LM32\Spartan6\ Browse...

VHDL Setting
☐ Create VHDL Wrapper ☐ Create VHDL NGO file.

Frequency Setting
Board Frequency(MHz): 48

Arbitration Scheme
Scheme: Shared Bus (Default)

Processor
Processor: LM32

Device Family
Family: LatticeEC Device: All

Available Platforms
☐ Clone Platform Select Platform: Browse...

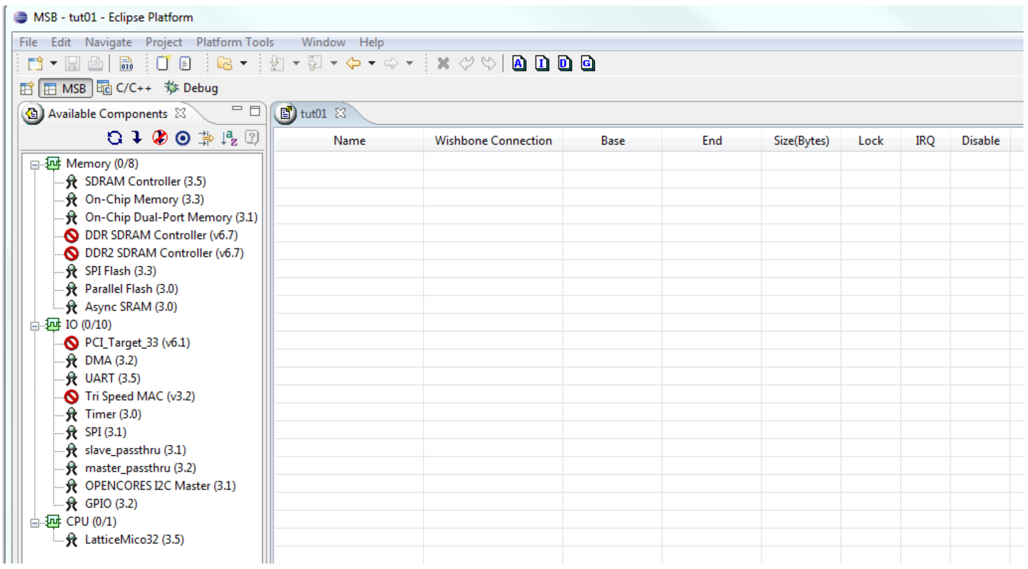
Platform Templates
blank
PlatformA
PlatformB
PlatformC
PlatformD
PlatformE
PlatformF
PlatformG
PlatformH

New platform without any components.

The "New Platform Wizard" will pop-up. In the screen you would need to enter the following:

- Platform name: The name of you SoC
- Frequency Setting: The clock frequency of the CPU
    - This should be the input frequency ($f_{cpu} = f_{input}$)

In this tutorial I will use "tut01" as the name and 48MHz as the frequency (48MHz input). After you have finished entering the settings, you will get to the main window, but now with a new "view".



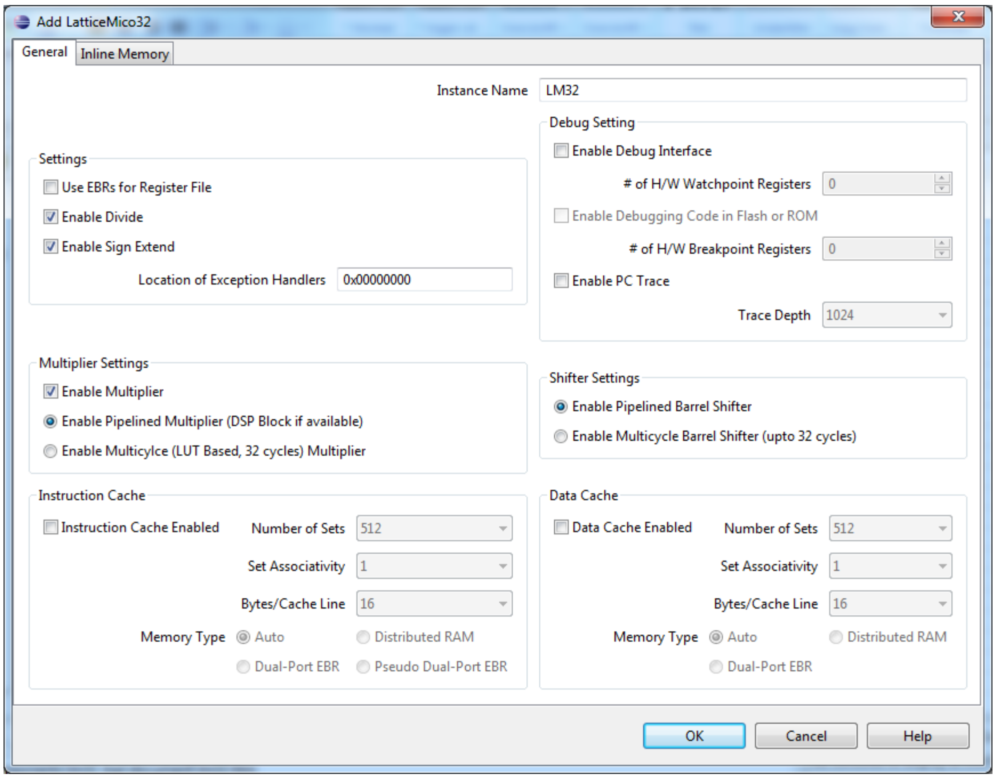
- Memory (0/8)
  - SDRAM Controller (3.5)
  - On-Chip Memory (3.3)
  - On-Chip Dual-Port Memory (3.1)
  - DDR SDRAM Controller (v6.7)
  - DDR2 SDRAM Controller (v6.7)
  - SPI Flash (3.3)
  - Parallel Flash (3.0)
  - Async SRAM (3.0)
- IO (0/10)
  - PCI_Target_33 (v6.1)
  - DMA (3.2)
  - UART (3.5)
  - Tri Speed MAC (v3.2)
  - Timer (3.0)
  - SPI (3.1)
  - slave_passthru (3.1)
  - master_passthru (3.2)
  - OPENCORES I2C Master (3.1)
  - GPIO (3.2)
- CPU (0/1)
  - LatticeMico32 (3.5)

Now you've created the project and the "motherboard" where we will place the SoC.

## Adding the CPU

Every System on a Chip (SoC) needs an cpu, so we will now add the LatticeMico32.

- Double click on the LatticeMico32 in the list to the left.



Instance Name: LM32

Settings
- [ ] Use EBRs for Register File
- [✓] Enable Divide
- [✓] Enable Sign Extend
- Location of Exception Handlers: 0x00000000

Debug Setting
- [ ] Enable Debug Interface
- # of H/W Watchpoint Registers: 0
- [ ] Enable Debugging Code in Flash or ROM
- # of H/W Breakpoint Registers: 0
- [ ] Enable PC Trace
- Trace Depth: 1024

Multiplier Settings
- [✓] Enable Multiplier
- [✓ Enable Pipelined Multiplier (DSP Block if available)]
- Enable Multicylce (LUT Based, 32 cycles) Multiplier

Shifter Settings
- [✓ Enable Pipelined Barrel Shifter]
- Enable Multicycle Barrel Shifter (upto 32 cycles)

Instruction Cache
- [ ] Instruction Cache Enabled
- Number of Sets: 512
- Set Associativity: 1
- Bytes/Cache Line: 16
- Memory Type: [✓ Auto] Distributed RAM Dual-Port EBR Pseudo Dual-Port EBR

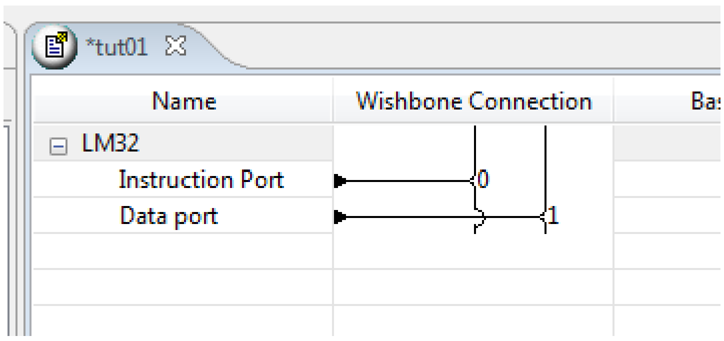
Data Cache
- [ ] Data Cache Enabled
- Number of Sets: 512
- Set Associativity: 1
- Bytes/Cache Line: 16
- Memory Type: [✓ Auto] Distributed RAM Dual-Port EBR

A new window will open giving you the change to customize the CPU. We will have to make some minor adjustments to get it working on a Xilinx FPGA.

- Disable Debug Interface (JTAG eg.)
- Also Disable the Instruction Cache and Data Cache
    - This is big, and can cause problems on some FPGAs, especially the smaller ones

The “Location of Exception Handlers” will be the address in the RAM where the CPU starts executing code from.

- Now press “OK”.

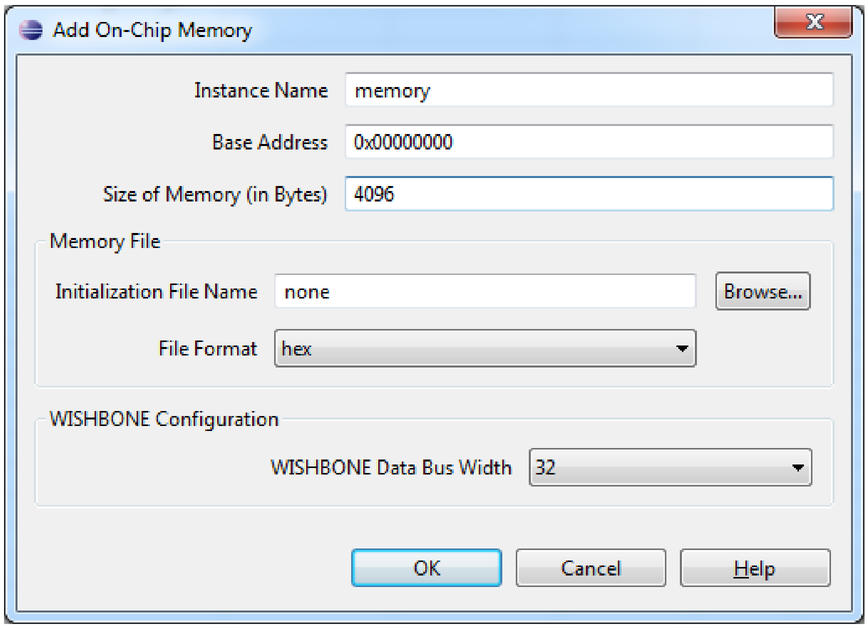


The LM32 will now be added to the view on the right. As you will notice the CPU has two Wishbone connections: One RW for RAM/memory and other peripherals, and one for the instruction memory.

## Adding memory

Our CPU will also need to have some RAM to store the code and runtime variables.

- Double click on “On-Chip Memory” in the list to the left

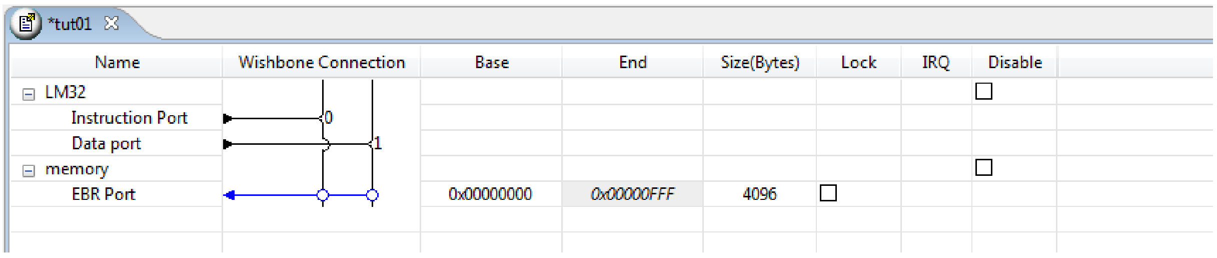


Instance Name: memory
Base Address: 0x00000000
Size of Memory (in Bytes): 4096
Initialization File Name: none
File Format: hex
WISHBONE Data Bus Width: 32

Another window will open, giving you the chance to change the settings of the On-Chip Memory, which will be the Xilinx Block RAM (BRAM).

The only settings that are important for us are “Instance Name”, “Base Address” and “Size of Memory”

- Instance Name: Choose a logic name for the memory like, like “memory”
- Base Address: Has to be the same as the “Location of Exception Handlers”, so the CPU will start executing code from this memory
- Size of Memory: 4KB (4096) will be enough for this test
- Press “OK”

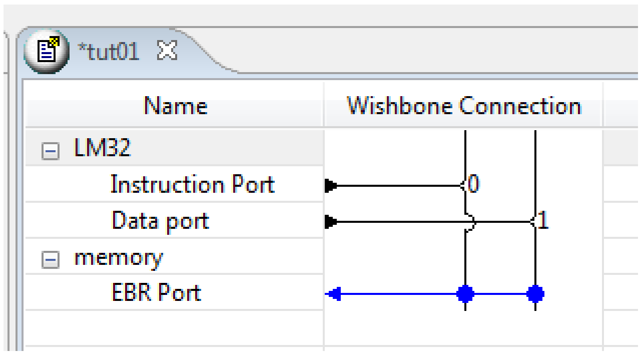


*tut01

| Name | Wishbone Connection | Base | End | Size(Bytes) | Lock | IRQ | Disable |
|---|---|---|---|---|---|---|---|
| ⊟ LM32 | | | | | | | ☐ |
| Instruction Port | 0 | | | | | | |
| Data port | 1 | | | | | | |
| ⊟ memory | | | | | | | ☐ |
| EBR Port | | 0x00000000 | 0x00000FFF | 4096 | ☐ | | |

At this moment, the memory isn't connected to the CPU though.

- So click on the two BLUE circles.



*tut01

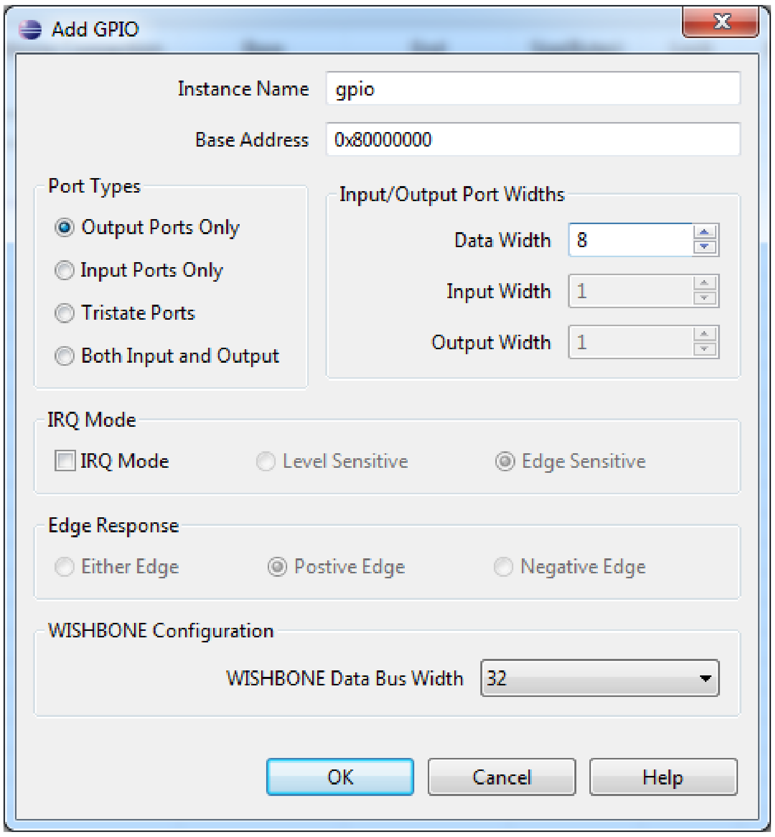
| Name | Wishbone Connection |
|---|---|
| ⊟ LM32 | |
| Instruction Port | 0 |
| Data port | 1 |
| ⊟ memory | |
| EBR Port | |

Now the memory is connected to the Instruction port and the Data port. This is the best solution to start with, but it makes a risk: It IS possible to overwrite the memory!

## Adding the GPIO

To do something useful with our SoC, we need to provide some connection to the outside world.

- Double Click on "GPIO" in the list to the left



Add GPIO

Instance Name: gpio

Base Address: 0x80000000

Port Types: [✓ Output Ports Only] Input Ports Only / Tristate Ports / Both Input and Output

Input/Output Port Widths
Data Width: 8
Input Width: 1
Output Width: 1

IRQ Mode: ☐ IRQ Mode; Level Sensitive [✓ Edge Sensitive]

Edge Response: Either Edge [✓ Postive Edge] Negative Edge

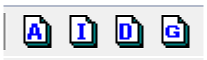
WISHBONE Configuration
WISHBONE Data Bus Width: 32

The only settings that are important for us are "Instance Name" and "Data Width". For the tutorial we will just need 8 outputs, so leave the rest as default, but:

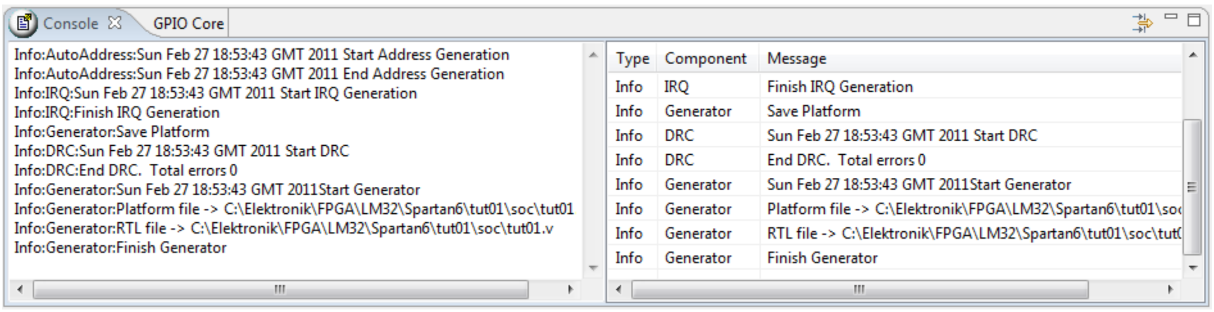
- Instance Name: Choose a logic name for the GPIOs, like "GPIO"
- Data Width: As we need 8 outputs, write 8
- Now press "OK".

## Generate the Verilog files

We are now finished setting up our SoC with the CPU, RAM and peripheral. The next step is to generate the Verilog files, which will be imported into Xilinx ISE.



To start the generation process, click the blue G in the top bar.



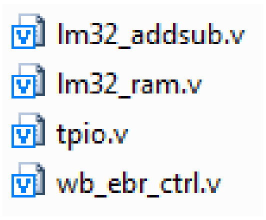
```
Info:AutoAddress:Sun Feb 27 18:53:43 GMT 2011 Start Address Generation
Info:AutoAddress:Sun Feb 27 18:53:43 GMT 2011 End Address Generation
Info:IRQ:Sun Feb 27 18:53:43 GMT 2011 Start IRQ Generation
Info:IRQ:Finish IRQ Generation
Info:Generator:Save Platform
Info:DRC:Sun Feb 27 18:53:43 GMT 2011 Start DRC
Info:DRC:End DRC.  Total errors 0
Info:Generator:Sun Feb 27 18:53:43 GMT 2011Start Generator
Info:Generator:Platform file -> C:\Elektronik\FPGA\LM32\Spartan6\tut01\soc\tut01
Info:Generator:RTL file -> C:\Elektronik\FPGA\LM32\Spartan6\tut01\soc\tut01.v
Info:Generator:Finish Generator
```

| Type | Component | Message |
|---|---|---|
| Info | IRQ | Finish IRQ Generation |
| Info | Generator | Save Platform |
| Info | DRC | Sun Feb 27 18:53:43 GMT 2011 Start DRC |
| Info | DRC | End DRC.  Total errors 0 |
| Info | Generator | Sun Feb 27 18:53:43 GMT 2011Start Generator |
| Info | Generator | Platform file -> C:\Elektronik\FPGA\LM32\Spartan6\tut01\soc |
| Info | Generator | RTL file -> C:\Elektronik\FPGA\LM32\Spartan6\tut01\soc\tut( |
| Info | Generator | Finish Generator |

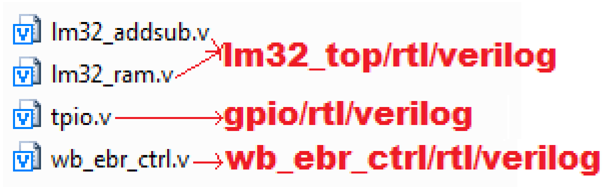
When you at the bottom of the screen, in the Console window, see the "Finish Generator" message, it means that the Verilog files have now been generated.

## 2. Modify the generated Verilog files

Before we can import the Verilog files into Xilinx ISE, we have to modify some of them, to make them match with the Xilinx FPGAs. To do so you need to download our replacement files, listed on the first page of this guide.



When this is done, you should open the project folder of the LatticeMico project. In there you would see two folders, "components" and "soc". Enter the "components" folder. In there you would see 3 folders, "gpio", "lm32_top" and "wb_ebr_ctrl". Those are the folders which contains the Verilog code for the three different things we added to our SoC.



You should now replace the 4 files in these 3 folders with the corresponding replacement file, like shown in the image above. The last thing we should do is to remove this line from the "lm32_include_all.v" inside the "lm32_top" folder:

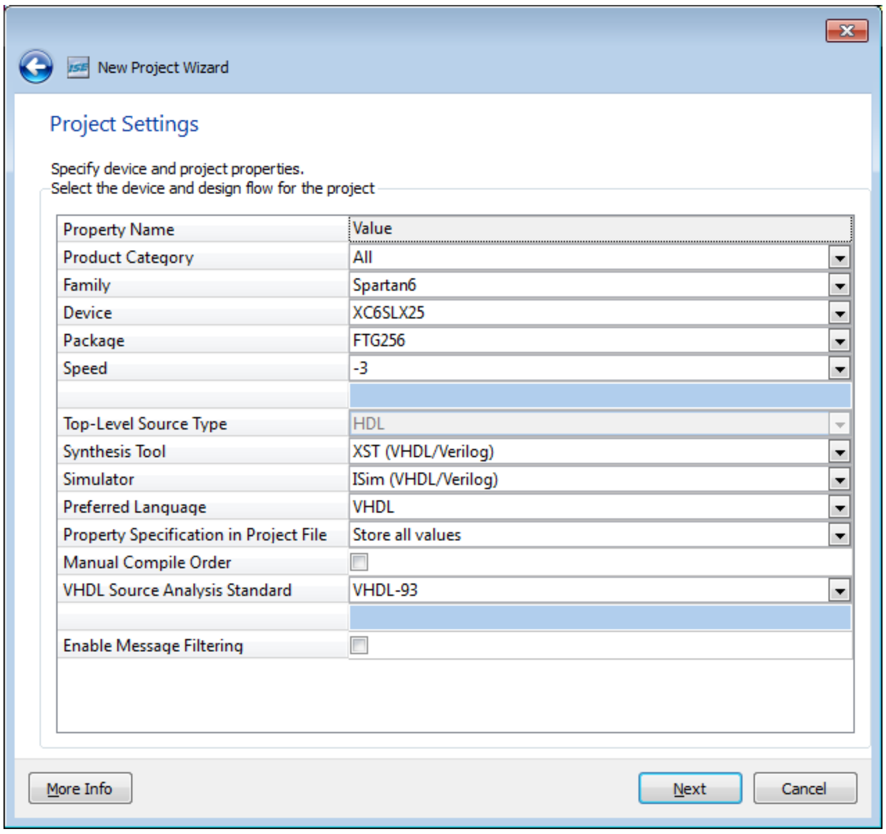
```
`include "pmi_def.v"
```

The "pmi_def.v" file, which this line is including, is a simulation file, though this simulation is only possible with Lattice FPGAs! Now let's move on and get the files imported into Xilinx ISE.

# 3. Import the Verilog files into Xilinx ISE

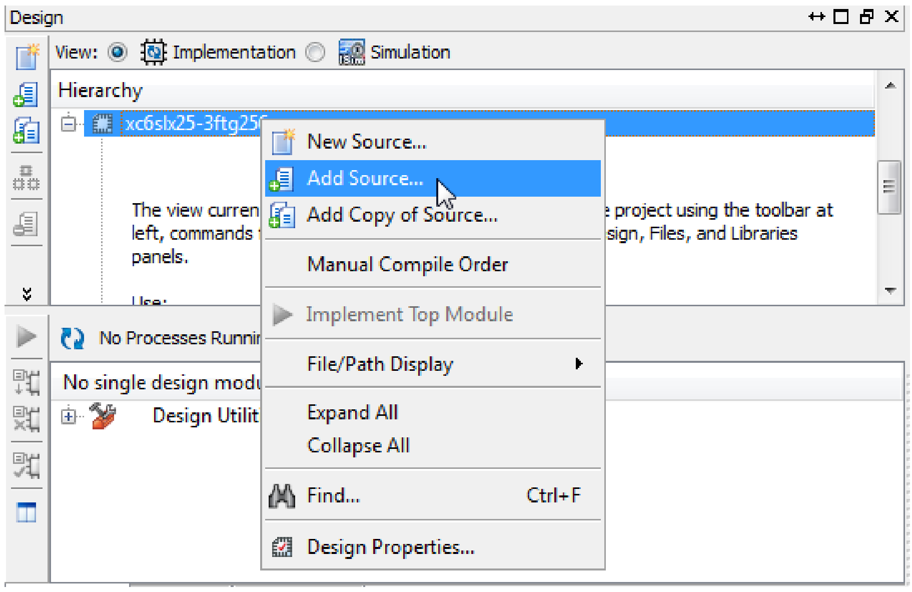
## Import the Verilog files

As we have now modified the files, we are ready to import them into Xilinx ISE. First you would have to create a project for you specific FPGA. In this tutorial we will be making a project for the Xilinx Spartan 6 FPGA XC6SLX25 device.

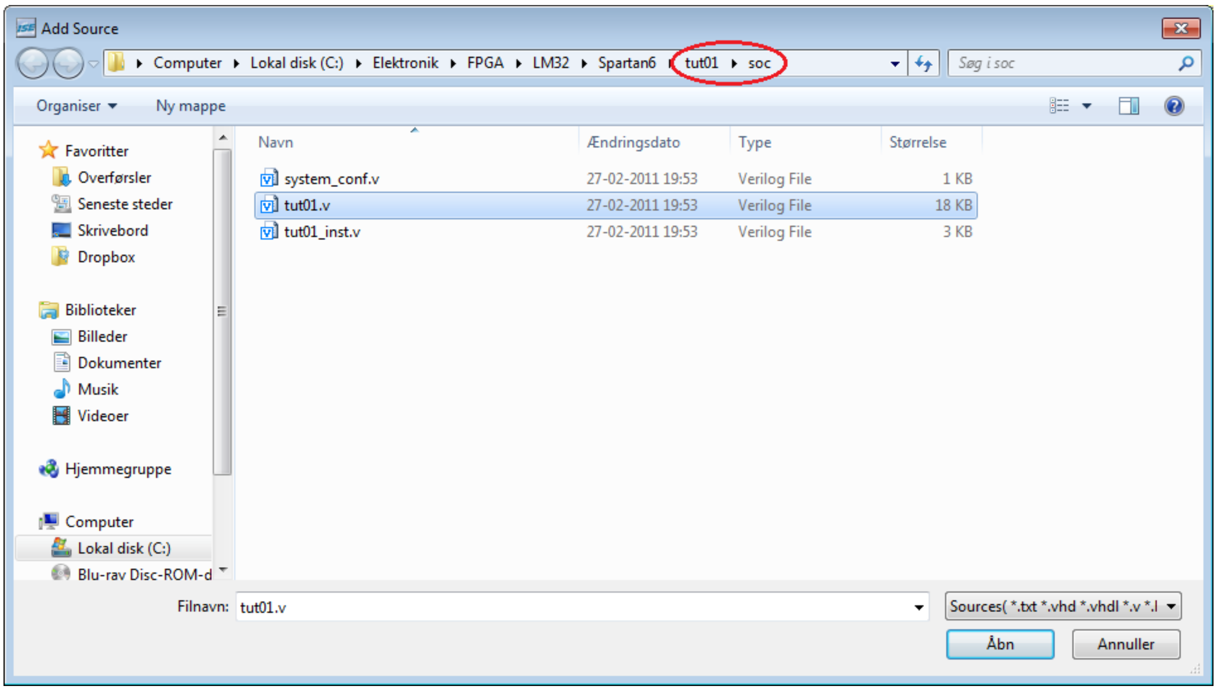

New Project Wizard
Project Settings
Specify device and project properties.
Select the device and design flow for the project
More Info
Next
Cancel

| Property Name | Value |
|---|---|
| Product Category | All |
| Family | Spartan6 |
| Device | XC6SLX25 |
| Package | FTG256 |
| Speed | -3 |
| | |
| Top-Level Source Type | HDL |
| Synthesis Tool | XST (VHDL/Verilog) |
| Simulator | ISim (VHDL/Verilog) |
| Preferred Language | VHDL |
| Property Specification in Project File | Store all values |
| Manual Compile Order | ☐ |
| VHDL Source Analysis Standard | VHDL-93 |
| | |
| Enable Message Filtering | ☐ |

After you have created a project, right click on the device name in the Hierarchy view, and select “Add Source”.



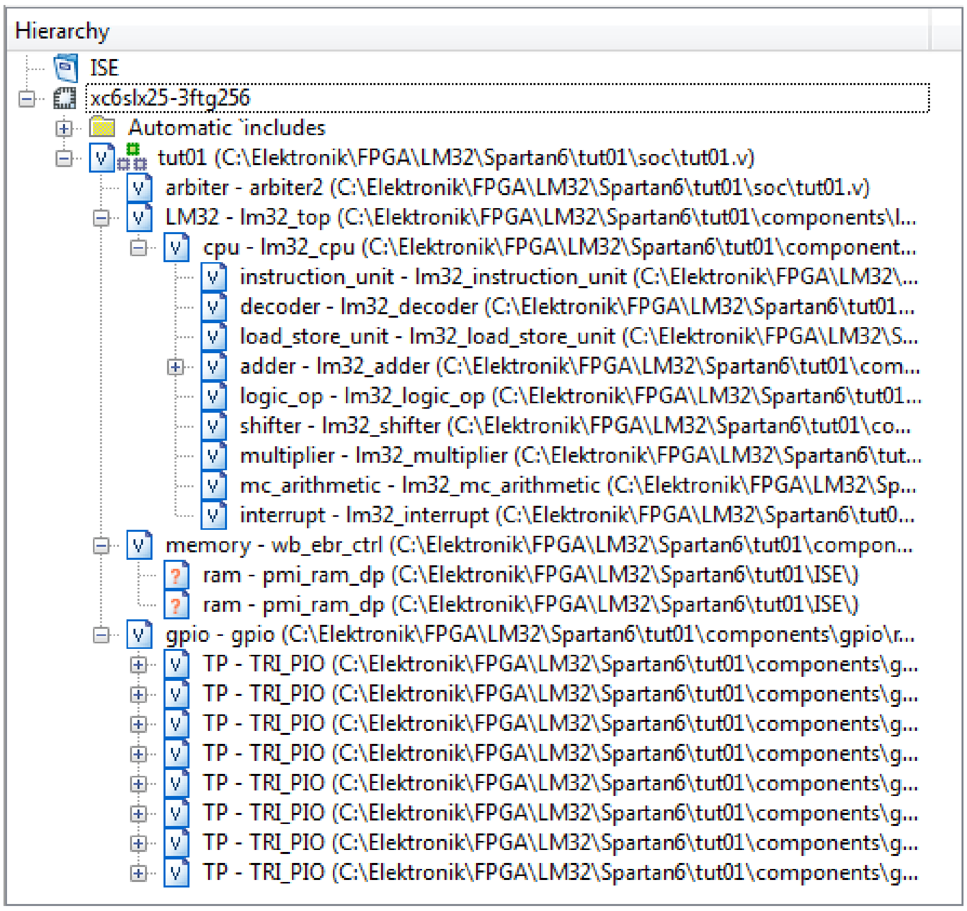
Next you should find the tut01.v file, in the “soc” directory in the Lattice project folder.



| Navn | Ændringsdato | Type | Størrelse |
|---|---|---|---|
| system_conf.v | 27-02-2011 19:53 | Verilog File | 1 KB |
| tut01.v | 27-02-2011 19:53 | Verilog File | 18 KB |
| tut01_inst.v | 27-02-2011 19:53 | Verilog File | 3 KB |

Filnavn: tut01.v
Sources( *.txt *.vhd *.vhdl *.v *.l

Press “OK” to the popup box, telling you if the source can be added. If everything went well, you should now have a Hierachy looking like this.



```
Hierarchy
  ISE
  xc6slx25-3ftg256
    Automatic `includes
    tut01 (C:\Elektronik\FPGA\LM32\Spartan6\tut01\soc\tut01.v)
      arbiter - arbiter2 (C:\Elektronik\FPGA\LM32\Spartan6\tut01\soc\tut01.v)
      LM32 - lm32_top (C:\Elektronik\FPGA\LM32\Spartan6\tut01\components\l...
        cpu - lm32_cpu (C:\Elektronik\FPGA\LM32\Spartan6\tut01\component...
          instruction_unit - lm32_instruction_unit (C:\Elektronik\FPGA\LM32\...
          decoder - lm32_decoder (C:\Elektronik\FPGA\LM32\Spartan6\tut01...
          load_store_unit - lm32_load_store_unit (C:\Elektronik\FPGA\LM32\S...
          adder - lm32_adder (C:\Elektronik\FPGA\LM32\Spartan6\tut01\com...
          logic_op - lm32_logic_op (C:\Elektronik\FPGA\LM32\Spartan6\tut01...
          shifter - lm32_shifter (C:\Elektronik\FPGA\LM32\Spartan6\tut01\co...
          multiplier - lm32_multiplier (C:\Elektronik\FPGA\LM32\Spartan6\tut...
          mc_arithmetic - lm32_mc_arithmetic (C:\Elektronik\FPGA\LM32\Sp...
          interrupt - lm32_interrupt (C:\Elektronik\FPGA\LM32\Spartan6\tut0...
      memory - wb_ebr_ctrl (C:\Elektronik\FPGA\LM32\Spartan6\tut01\compon...
        ram - pmi_ram_dp (C:\Elektronik\FPGA\LM32\Spartan6\tut01\ISE\)
        ram - pmi_ram_dp (C:\Elektronik\FPGA\LM32\Spartan6\tut01\ISE\)
      gpio - gpio (C:\Elektronik\FPGA\LM32\Spartan6\tut01\components\gpio\r...
        TP - TRI_PIO (C:\Elektronik\FPGA\LM32\Spartan6\tut01\components\g...
        TP - TRI_PIO (C:\Elektronik\FPGA\LM32\Spartan6\tut01\components\g...
        TP - TRI_PIO (C:\Elektronik\FPGA\LM32\Spartan6\tut01\components\g...
        TP - TRI_PIO (C:\Elektronik\FPGA\LM32\Spartan6\tut01\components\g...
        TP - TRI_PIO (C:\Elektronik\FPGA\LM32\Spartan6\tut01\components\g...
        TP - TRI_PIO (C:\Elektronik\FPGA\LM32\Spartan6\tut01\components\g...
        TP - TRI_PIO (C:\Elektronik\FPGA\LM32\Spartan6\tut01\components\g...
        TP - TRI_PIO (C:\Elektronik\FPGA\LM32\Spartan6\tut01\components\g...
```

## Create a wrapper for the LM32 Verilog files

As the files are now imported into our project, we should make a wrapper in either Verilog or VHDL for the LM32 Verilog code. Right click on the device name in the Hierarchy view, and select “New Source”. Next select “VHDL Module” or “Verilog Module”.

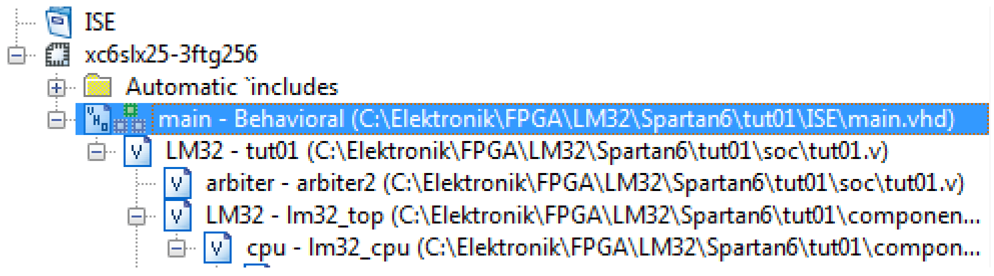
```
`timescale 1ns / 1ps

module main(
          input clk,
          input rst,
   output [7:0] led
           );


tut01 SOC (
    .clk_i(clk),
    .reset_n(rst),
    .gpioPIO_OUT(led)
    );


endmodule
```

*Verilog-wrapper*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity main is
   port (
      clk: in std_logic;
      rst: in std_logic;
      led: out std_logic_vector(7 downto
0)
   );
end main;

architecture Behavioral of main is
   component tut01
      port (
         clk_i: in std_logic;
         reset_n: in std_logic;
         gpioPIO_OUT: out
std_logic_vector(7 downto 0)
      );
   end component;
begin

   LM32: tut01
      port map (clk_i => clk, reset_n =>
rst, gpioPIO_OUT => led);

end Behavioral;
```

*VHDL-wrapper*

This is the wrapper code in either Verilog or VHDL. When you have saved the wrap-per file, you should notice that the wrapper file will be the Top Module, and that the tut01 will be shown underneath.



```
ISE
xc6slx25-3ftg256
   Automatic `includes
   main - Behavioral (C:\Elektronik\FPGA\LM32\Spartan6\tut01\ISE\main.vhd)
      LM32 - tut01 (C:\Elektronik\FPGA\LM32\Spartan6\tut01\soc\tut01.v)
         arbiter - arbiter2 (C:\Elektronik\FPGA\LM32\Spartan6\tut01\soc\tut01.v)
         LM32 - lm32_top (C:\Elektronik\FPGA\LM32\Spartan6\tut01\componen...
            cpu - lm32_cpu (C:\Elektronik\FPGA\LM32\Spartan6\tut01\compon...
```

### Create a constraint file for the FPGA

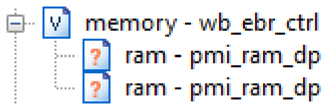
Now our wrapper is done, so let's make the constraint file too. Right click on the Top Module file in the Hierarchy view, and select "New Source". Next select "Implementation Constraints File".

The constraint file for the board I have (the ZTEX Spartan 6 module) contains the following lines, but remember this can be different from board to board.

```
# 48 MHz EZ-USB clock
NET "clk" TNM_NET = "clk";
TIMESPEC "ts_clk" = PERIOD "clk" 20.833 ns HIGH 50 %;
NET "clk"  LOC = "K14" | IOSTANDARD = LVCMOS33 ;

NET "led<0>"  LOC = "A14" | IOSTANDARD = LVCMOS33 | DRIVE = 12 ;
NET "led<1>"  LOC = "C13" | IOSTANDARD = LVCMOS33 | DRIVE = 12 ;
NET "led<2>"  LOC = "D11" | IOSTANDARD = LVCMOS33 | DRIVE = 12 ;
NET "led<3>"  LOC = "D12" | IOSTANDARD = LVCMOS33 | DRIVE = 12 ;
NET "led<4>"  LOC = "F10" | IOSTANDARD = LVCMOS33 | DRIVE = 12 ;
NET "led<5>"  LOC = "E11" | IOSTANDARD = LVCMOS33 | DRIVE = 12 ;
NET "led<6>"  LOC = "E10" | IOSTANDARD = LVCMOS33 | DRIVE = 12 ;
NET "led<7>"  LOC = "C10" | IOSTANDARD = LVCMOS33 | DRIVE = 12 ;

NET "rst"     LOC = "B14"  | IOSTANDARD = LVCMOS33 ;
```

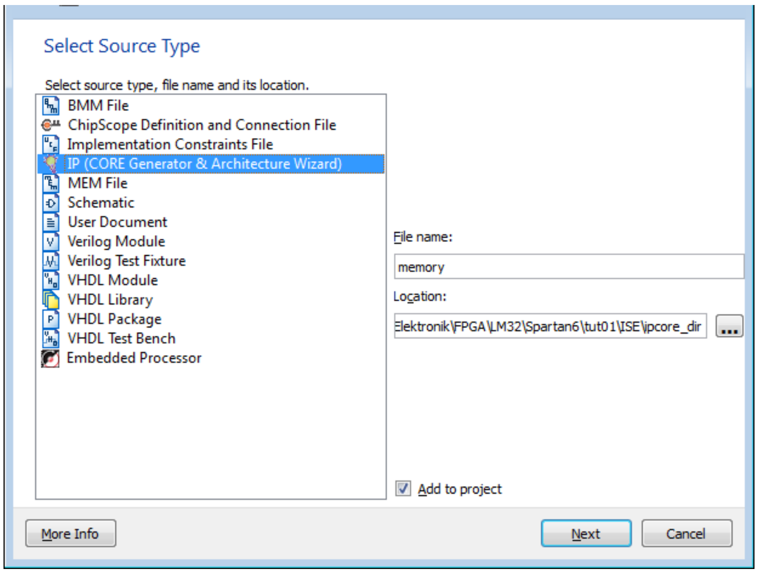
*Constraint file for the ZTEX Spartan 6 module*

## 4. Add the Xilinx Block RAM (BRAM)

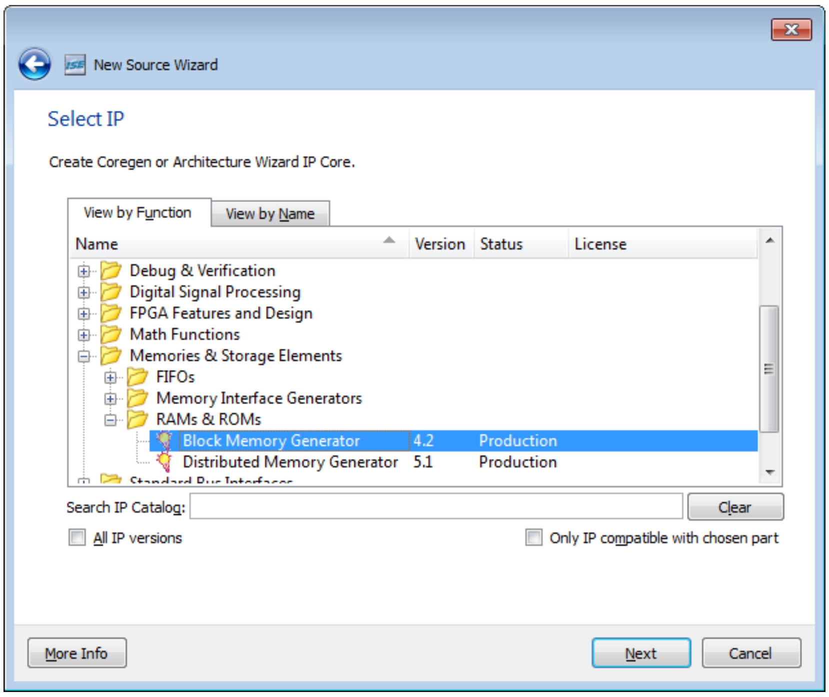


Now when we are done with the wrapper and the constraint file we are still missing the Xilinx Block RAM. You might have noticed the orange question marks? Those indicate that something is missing.

To add a Xilinx Block RAM we are going to use the Core Generator, so one more time right click on the Top Module file in the Hierarchy view, and select "New Source". Next select "IP (CORE Generator & Architecture Wizard)". As filename just write "memory", and then press "Next".



Now you should find and select "Memories & Storage Elements → RAMs & ROMs → Block Memory Generator".



New Source Wizard

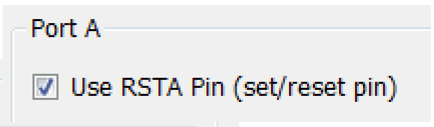
Select IP

Create Coregen or Architecture Wizard IP Core.

| Name | Version | Status | License |
|---|---|---|---|
| Debug & Verification | | | |
| Digital Signal Processing | | | |
| FPGA Features and Design | | | |
| Math Functions | | | |
| Memories & Storage Elements | | | |
| FIFOs | | | |
| Memory Interface Generators | | | |
| RAMs & ROMs | | | |
| Block Memory Generator | 4.2 | Production | |
| Distributed Memory Generator | 5.1 | Production | |

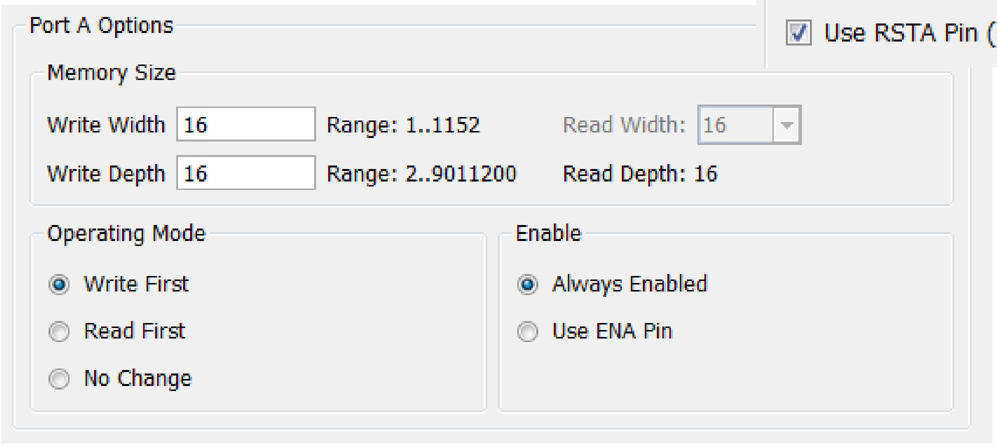
In the Wizard you should just leave everything as default except the following:

- Write Width (page 2) → 32
    - o We set this to 32 as we are using a 32-bit architecture, and therefor the RAM should also be 32-bit
- Write Depth (page 2) → 1024
    - o We set this to 1024, because we told the Lattice to have a RAM of 4096 bytes, which is equal to 1024 32-bit variables (4096 / 4)
- Use RSTA Pin (page 4) → Should be checked



Port A

[✓] Use RSTA Pin (set/reset pin)



Port A Options

Memory Size

Write Width 16 Range: 1..1152 Read Width: 16

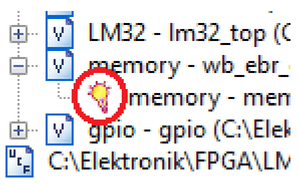
Write Depth 16 Range: 2..9011200 Read Depth: 16

Operating Mode: [✓ Write First] Read First No Change

Enable: [✓ Always Enabled] Use ENA Pin

When this is done, press "Generate" – this can take a while depending on your computer.

When the Block RAM generation has finished the question marks should have disappeared, and been replaced by a lightning bulb. Now we have generated the memory for our SoC, so the last thing to do now is to make the code that is going to be stored inside that RAM!

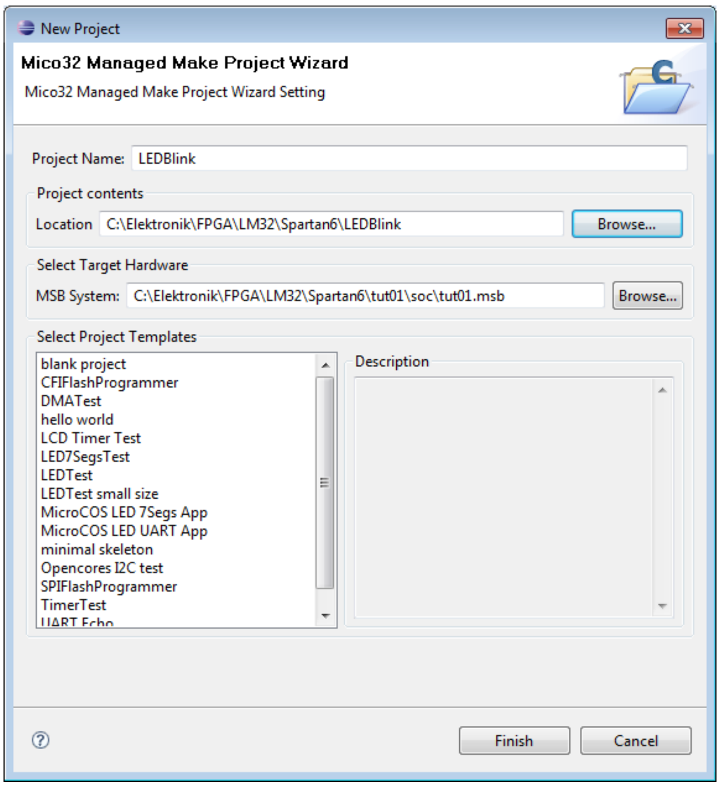


## 5. Hello World example for the processor

### Create and compile the Hello World example

Head back to the LatticeMico32 System IDE and select the C/C++ perspective. Next we should add a new C project.

- File → New → Mico32 Managed Make C Project



Mico32 Managed Make Project Wizard
Mico32 Managed Make Project Wizard Setting

Project Name: LEDBlink
Project contents
Location C:\Elektronik\FPGA\LM32\Spartan6\LEDBlink
Select Target Hardware
MSB System: C:\Elektronik\FPGA\LM32\Spartan6\tut01\soc\tut01.msb
Select Project Templates
blank project
CFIFlashProgrammer
DMATest
hello world
LCD Timer Test
LED7SegsTest
LEDTest
LEDTest small size
MicroCOS LED 7Segs App
MicroCOS LED UART App
minimal skeleton
Opencores I2C test
SPIFlashProgrammer
TimerTest
UART Echo

Name the system what you want, but make sure the MSB System is set to the MSB you made in the beginning. Press “Finish” when it’s correct.

In the left you will now see the new project. Right click to add the C file.

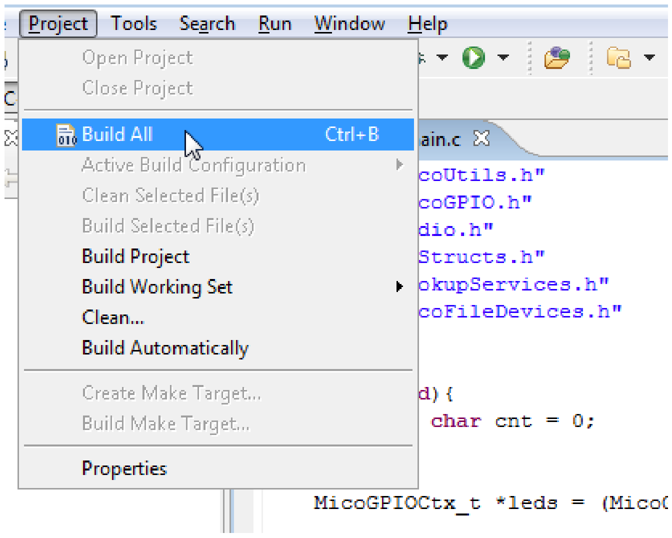
- New → Source File
- Name the source file main.c

At this moment you should now see a blank window where you can write. In there you should add the following code which makes the 8 LEDs, on our GPIO, alternate (blink).

```c
#include "MicoUtils.h"
#include "MicoGPIO.h"
#include "stdio.h"
#include "DDStructs.h"
#include "LookupServices.h"
#include "MicoFileDevices.h"


int main(void){
           unsigned char cnt = 0;
           int i;

           MicoGPIOCtx_t *leds = (MicoGPIOCtx_t *)MicoGetDevice("gpio");

           while(1){
                      MICO_GPIO_WRITE_DATA(leds, 0xAA000000);
                      MicoSleepMilliSecs(100);

                      MICO_GPIO_WRITE_DATA(leds, 0x55000000);
                      MicoSleepMilliSecs(100);
           }

           return(0);

}
```

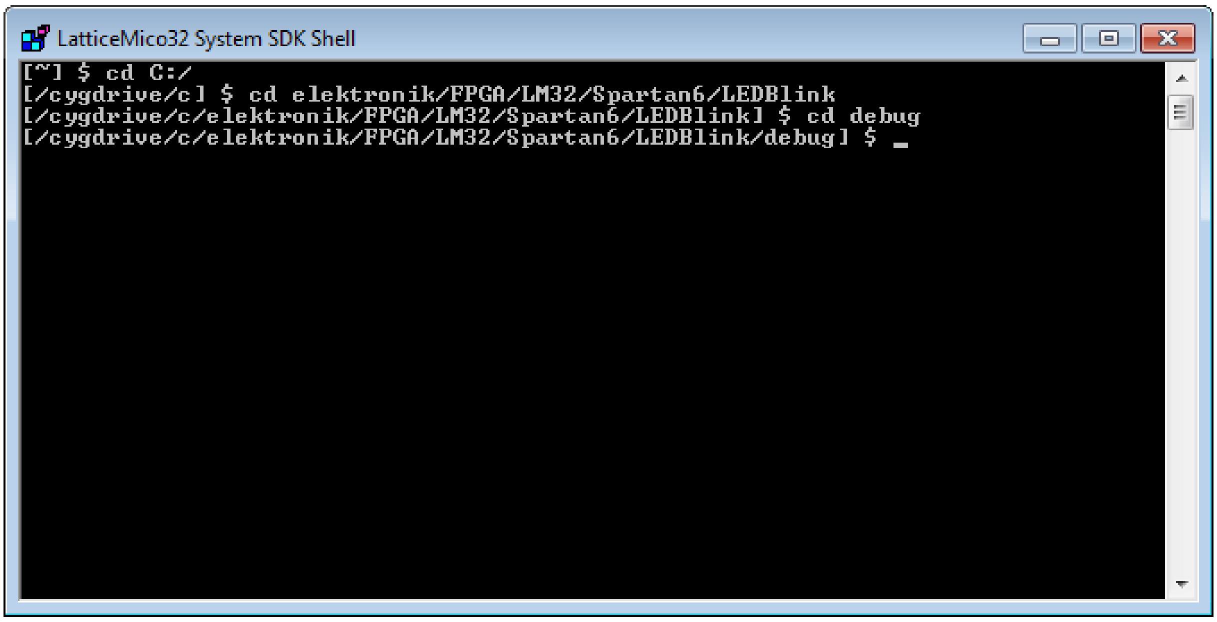
*Hello World code for the LM32 processor*

Make sure the "gpio" in the code is the same as the name of your GPIO (if you named it differently). When this is done, go to "Project → Build All" to compile the project. This would make an ELF file, which we are now going to convert into an Intel HEX file.



**Convert the ELF file into an Intel HEX file**

When the build is done ("Build complete for project…") you should open the "Lattice-Mico32 System SDK Shell", which has also been installed.



```
[~] $ cd C:/
[/cygdrive/c] $ cd elektronik/FPGA/LM32/Spartan6/LEDBlink
[/cygdrive/c/elektronik/FPGA/LM32/Spartan6/LEDBlink] $ cd debug
[/cygdrive/c/elektronik/FPGA/LM32/Spartan6/LEDBlink/debug] $ _
```

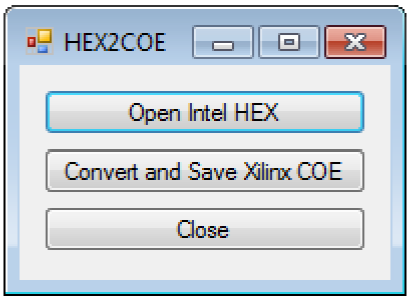
In that shell you navigate to the debug folder in the C-project directory. When you are in the debug folder, you write the following command to create an Intel HEX file.

*"lm32-elf-objcopy LEDBlink.elf -O ihex LEDBlink.hex"*

Replace the "LEDBlink" with the name of your C-project if different. Now should should have an Intel HEX file in the debug folder.

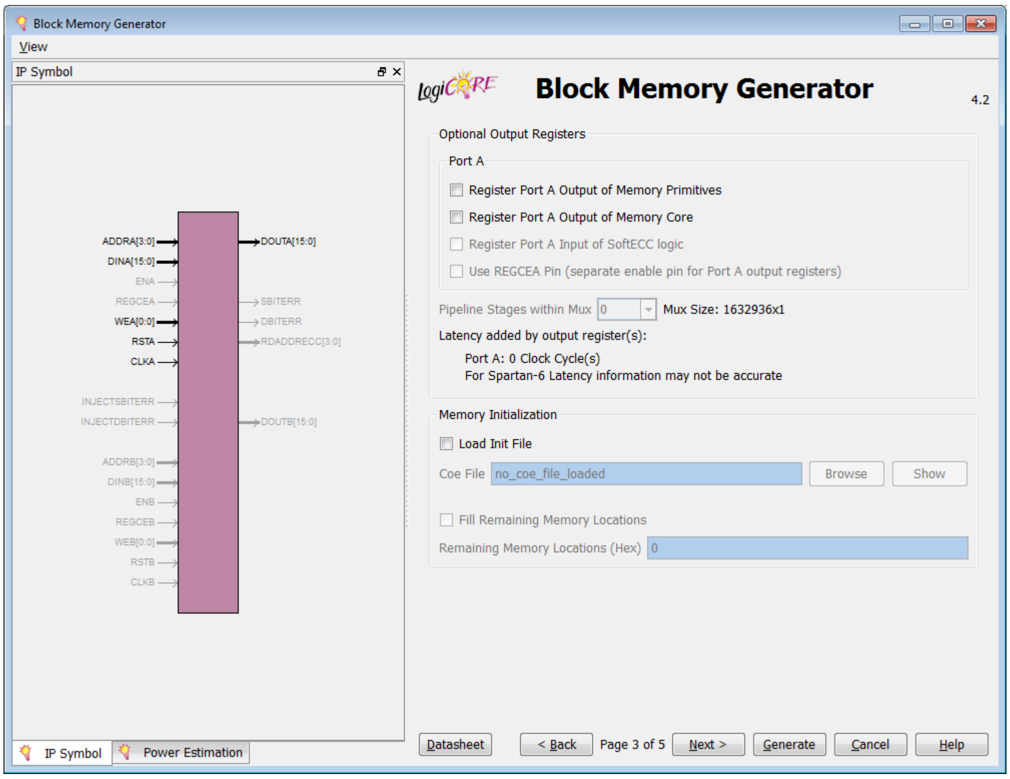
**Convert the Intel HEX file into a Xilinx Coefficient (.COE)**

Unfortunately we can't load a HEX file directly into the Xilinx Block RAM, so it has to be converted into a so called Xilinx Coefficient. This can be done with our "IHEX2COE" program. Start the program and press "Open Intel HEX". Now find and select the HEX file, generated previously. Next press "Convert and Save Xilinx COE" and save the Xilinx Coefficient file somewhere (remember the .COE extension)



## Loading the Xilinx Coefficient into the Block RAM

The last thing to do now is just to load the Coefficient file into the Block RAM and then synthesize. Reopen the Xilinx ISE and the project, and double-click on the "memory", the one with the lightning bulb at the left. The Core Generator would now open again.


Block Memory Generator
View
IP Symbol

LogiCORE Block Memory Generator 4.2

Optional Output Registers

Port A

- Register Port A Output of Memory Primitives
- Register Port A Output of Memory Core
- Register Port A Input of SoftECC logic
- Use REGCEA Pin (separate enable pin for Port A output registers)

Pipeline Stages within Mux 0 Mux Size: 1632936x1

Latency added by output register(s):

Port A: 0 Clock Cycle(s)
For Spartan-6 Latency information may not be accurate

Memory Initialization

- Load Init File

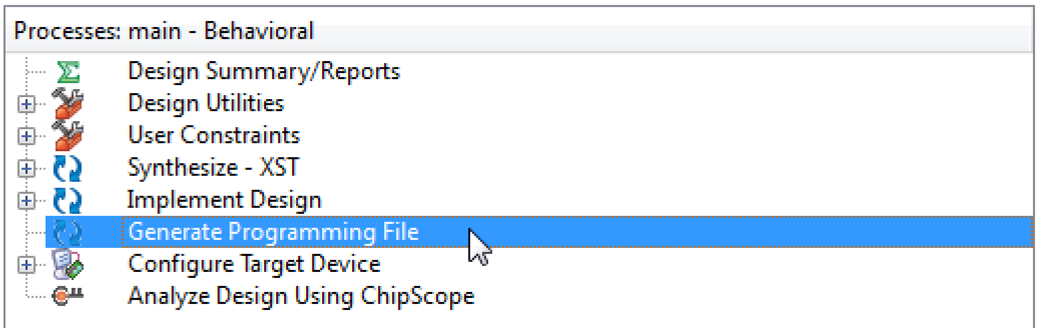
Coe File no_coe_file_loaded Browse Show

- Fill Remaining Memory Locations

Remaining Memory Locations (Hex) 0

Go to page 3 and check the "Load Init File". You should now be able to click "Browse" and select the .COE file generated previously. When the file has been selected just press "Generate" – again this could take a while depending on your computer. Press "OK" to the popup box, asking you to confirm overwriting.

Now there is only one thing left to do, and it is to Synthesize the project and Generate a bit-file. Do this by selecting the Top Module and then double-click on the "Generate Programming File".


Processes: main - Behavioral
Design Summary/Reports
Design Utilities
User Constraints
Synthesize - XST
Implement Design
Generate Programming File
Configure Target Device
Analyze Design Using ChipScope