# Implementing a Simple PicoBlaze Design in Vivado

Ken Chapman – 25th June 2014

**This Document**

This document is a worked example of a procedure that shows how to set up a PicoBlaze project in Vivado. The example is based on the 'uart6_kc705' reference design provided in the KCPSM6 package but it is presented in a way that pretends that it is a design being created from scratch as if it were your own design.

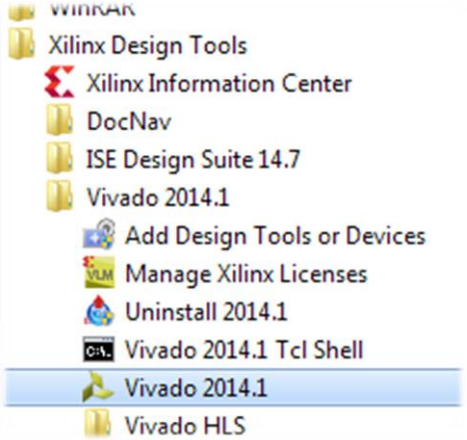
This document is NOT intended to be replacement for all of the formal Vivado documentation or training courses. Neither is it intended to teach you how to write VHDL, Verilog, XDC or PSM files. It is purely focussed on mechanisms to implement a design containing PicoBlaze successfully when using Vivado.

| | **This is my first experience of using PicoBlaze** | **I have implemented PicoBlaze designs before** |
|---|---|---|
| **This is my first experience of using Vivado** | Whether you are an experienced designer or a novice, following the steps presented in this document should be a useful exercise. However, don't expect 'PicoBlaze' to teach you everything about FPGA design or Vivado. Before you start, look at the first 30 pages of the 'KCPSM6 User Guide' provided in the KCPSM6 package. These introduce you to PicoBlaze and show you a step by step guide to creating a PicoBlaze design. When you then use Vivado you can decide whether to use one of the reference designs provided or try to implement something of your own. | Following the steps presented in this document should give you a feel for Vivado and help to get you started. The 'uart6_kc705.xdc' reference design constraints file could well be your first experience of XDC (rather than UCF). Take time to appreciate the directory structure of a Vivado project and see the scheme presented so that your assembled program is used by the project (starting on page 14). You've probably used JTAG Loader in the past so read about the issue with 'Hardware Manager' on page 23. |
| **I have implemented designs using Vivado.** | You probably don't need most of this document! Begin with learning a bit more about PicoBlaze; the first 30 pages of the 'KCPSM6 User Guide' provided in the KCPSM6 package introduce PicoBlaze and show you how to include PicoBlaze in your hardware design. The one area that you need to consider most when incorporating PicoBlaze into your Vivado design flow is the way in which the Assembler generates the program memory file. Review pages 14 to 18 of this document to appreciate the requirement so that your flow is also suitable. | As you know, the KCPSM6 Assembler generates the program memory definition file. Each time you modify your PSM code and re-run the assembler that file is updated. Establishing a scheme in which Vivado uses the updated file is really the key to success. Take a look at pages 14 to 18 of this document to appreciate the requirement so that your flow is also suitable. Also be aware of the issue with 'Hardware Manager' described on page 23. |

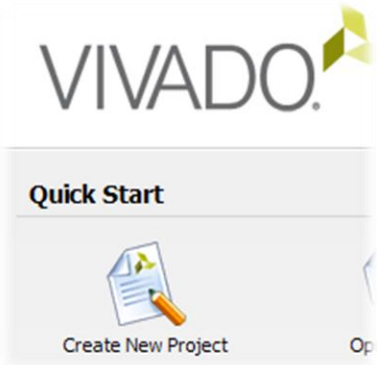
**Please note:** The images shown throughout the following pages will only show the significant areas of each screen and describe the particular things that you are required to do. Except for the example below, screens that only present supplementary information will not be shown and it is assumed that you will just observe them and continue. Likewise, except for the example below, a 'Next' button will not be shown and it is expected that you will just click 'Next' to continue when you are ready to do so.

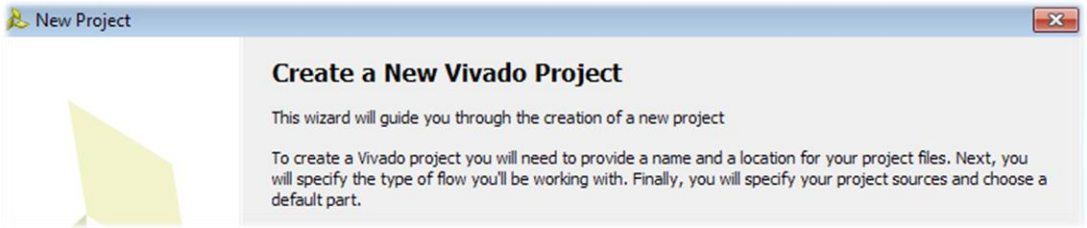
This sequence was captured using Vivado 2014.1 so there may be some differences when using a later version.

Start Vivado from the desktop icon **or** the Start Programs menu…


Select 'Create New Project'

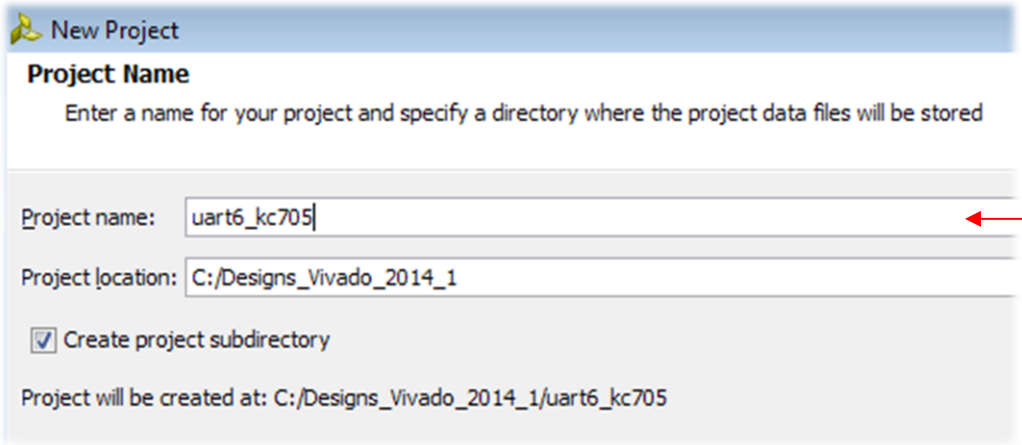




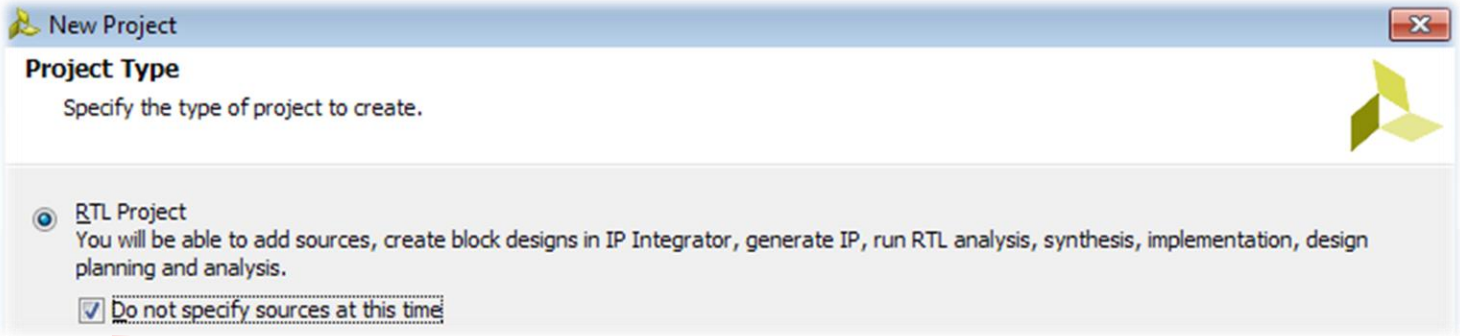
Click 'Next' to continue.

This sequence illustrates the **creation** of a design called 'uart6_kc705' which is a UART reference design for the KC705 Evaluation Platform. The source files are provided for you in the 'UART_and_PicoTerm\KC705_design' directory of the PicoBlaze package. Although the source files are provided (and you can make use of them), the object is to show you how you would create a PicoBlaze design from scratch in a Vivado project. So please pretend that you are creating the design from scratch (and then cheat by copying from the files provided as you play along ☺). Seriously, there are key points to be observed and learnt from rebuilding the reference design in this way.




Enter a name for your project.

This example is the UART reference design that is presented on the KC705 Evaluation Platform and provided in the PicoBlaze package contained in the 'UART_and_PicoTerm\KC705_design' directory.

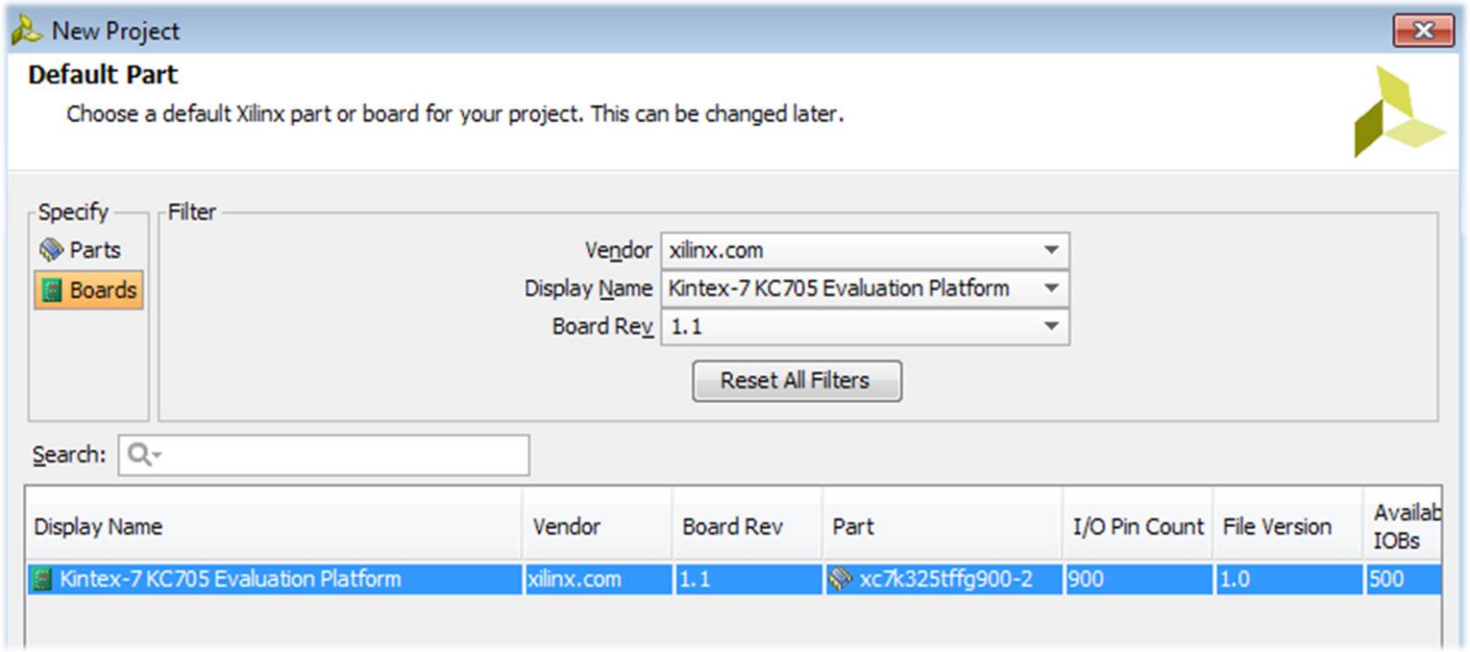


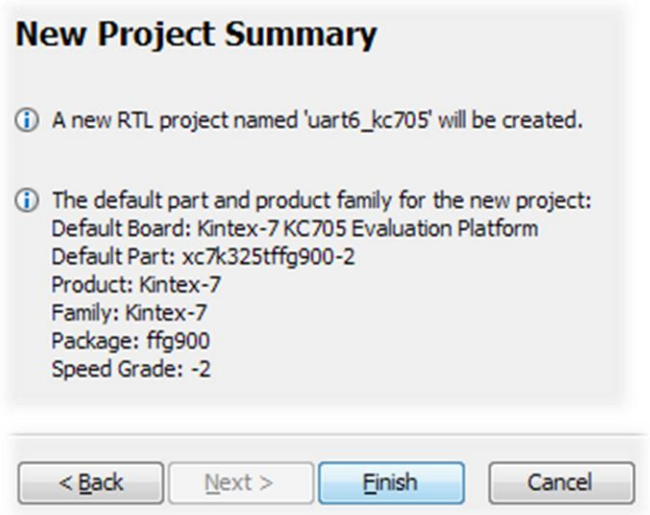

This will be an RTL project (VHDL in this example).

Even though you might be going to reuse code provided in the reference design we will pretend that we are creating a design from scratch. So do check this box before you continue.

Observation – Selecting this option and creating design files later will have a distinct impact on the directory structure that Vivado creates for your project and where each source file will be located. This becomes relevant when we reach the point of assembling the PicoBlaze as you will see later.

Select the target device (under ‘Parts) or a target board (under ‘Boards’).
In this case the design will target the XC7K325T device on the KC705 Evaluation Platform so it is easier to select the board.

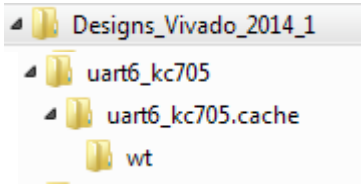


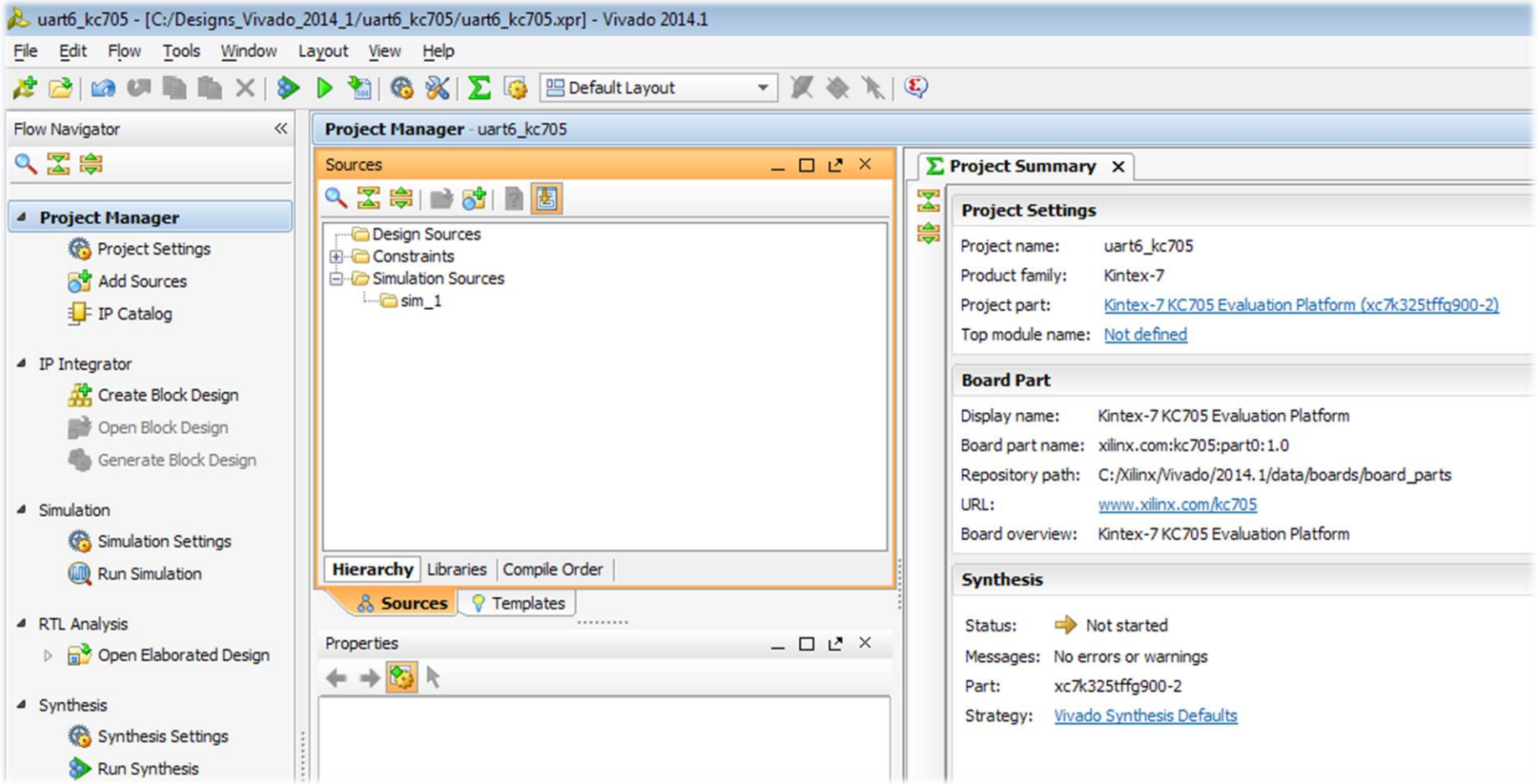

The Vivado project has been created and we are ready to start designing.

Click ‘Finish’ to continue

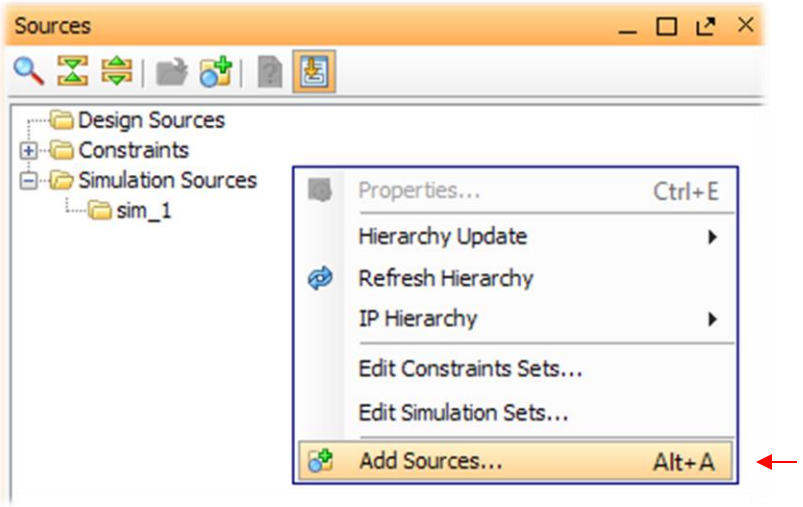
You may be interested to use Windows Explorer to see the start of a directory structure for your project.



You should now have an empty looking project that looks something like this…

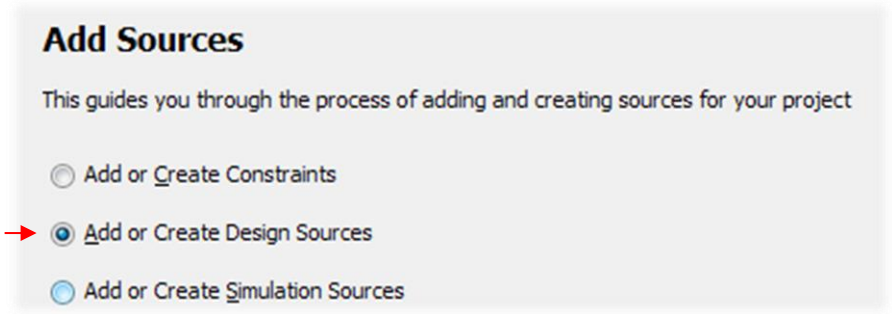


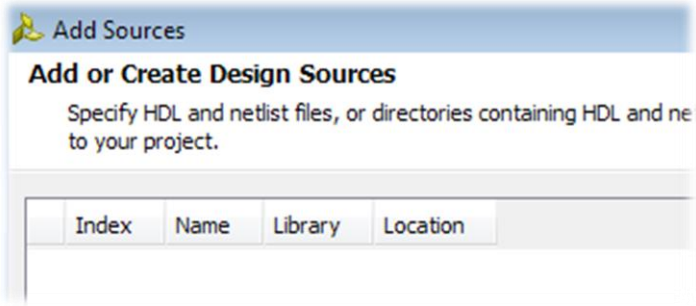

We will now create some design files. Even if you are going to adopt some reference code, let's continue to pretend that we are creating a design from scratch. As previously stated, the ways files are created and added to a project will impact the directory structure of a Vivado project and where Vivado locates the files. So until you become familiar with using Vivado it is probably best to continue following the sequence presented so that everything works out the same way the first time.



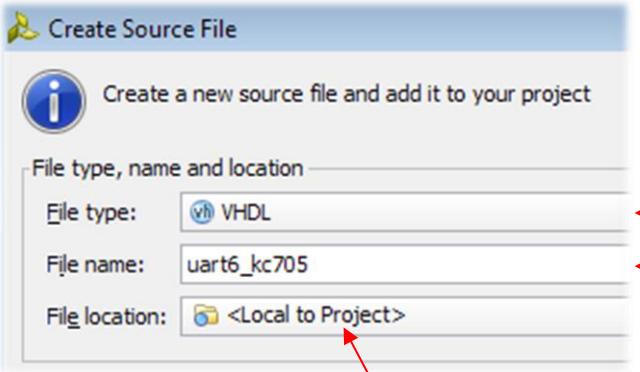

Right click in the 'Sources' window and select 'Add Sources…'

Then select 'Add or Create Design Sources'



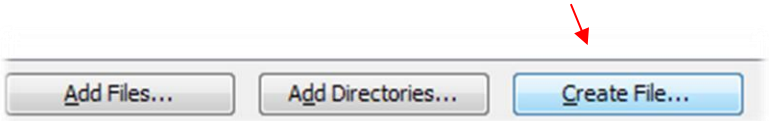




To begin with we will only define the top level file for the design.



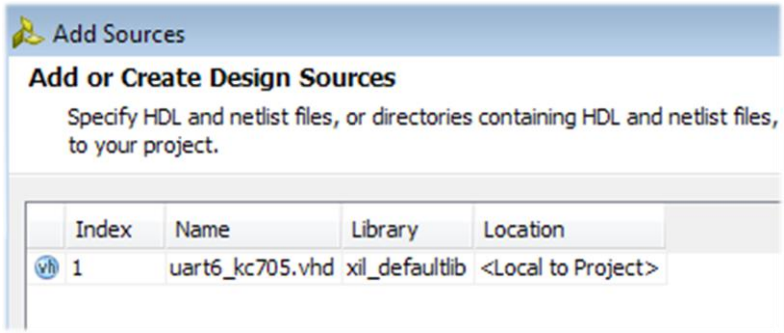

Specify HDL language and a top level file name for the design

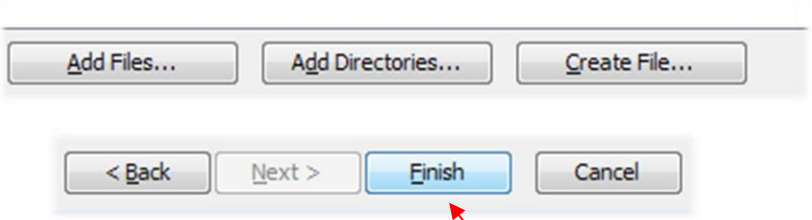
Select the '**Create File**' option to make a new file.



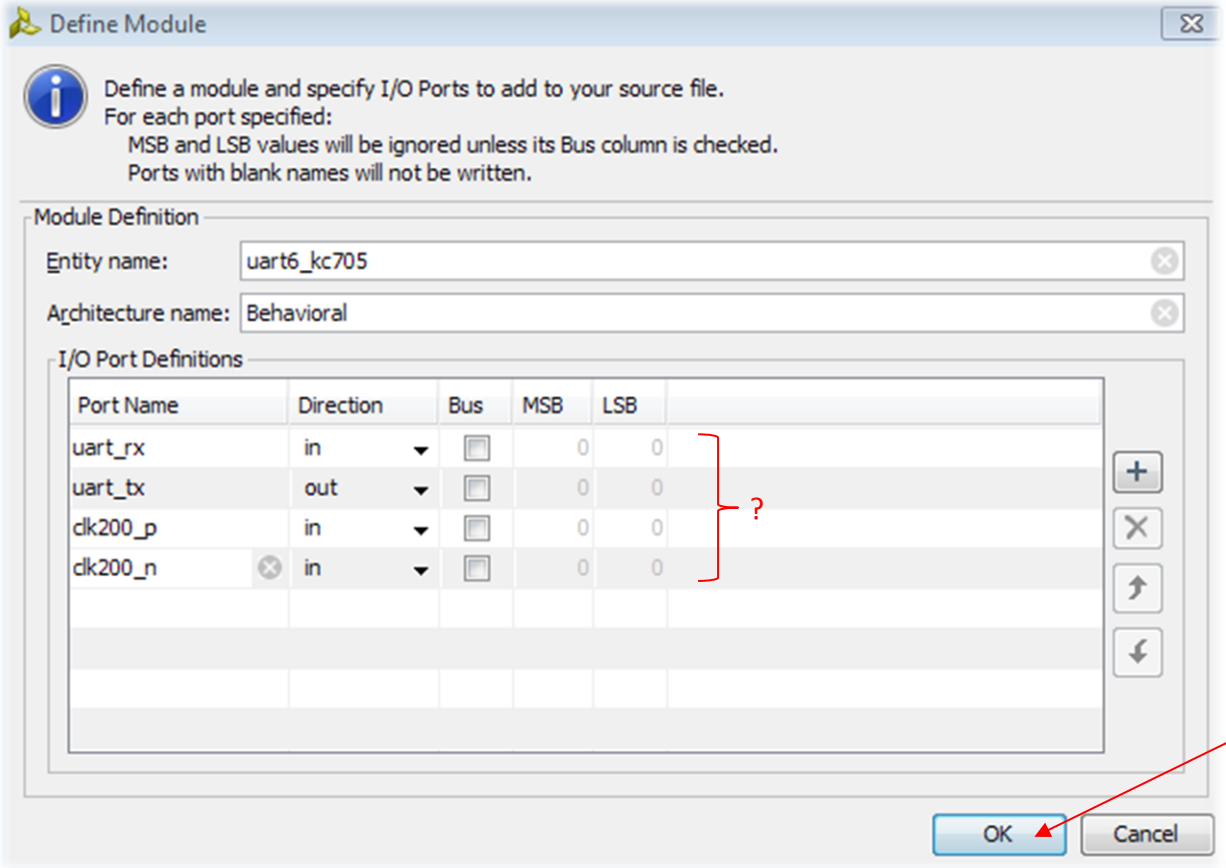

We are using the 'Create File' option as we are at least pretending to be creating a design from scratch.

We will continue with the default option of files being stored locally within the project and then we will see where Vivado puts them. However, you can see that this is where you may choose to locate your source file somewhere else (but then you will become responsible for managing them!).



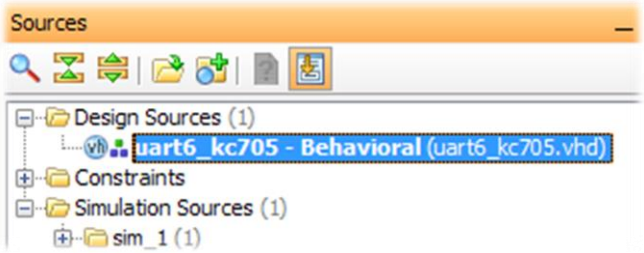




The new file is listed and we could go on to create some more files before we continue. In this case we just click 'Finish' to continue with the one file we have defined.




When creating a file from scratch the module definition GUI can help you create the initial structure for your code but this is completely optional.

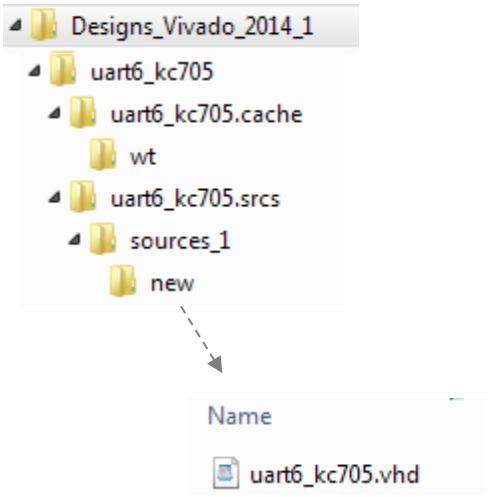
In this example all the simple I/O of the UART reference design have been entered but you can clock 'Ok' at anytime to continue. You can always add or modify your I/O by editing the HDL in the usual way later on.



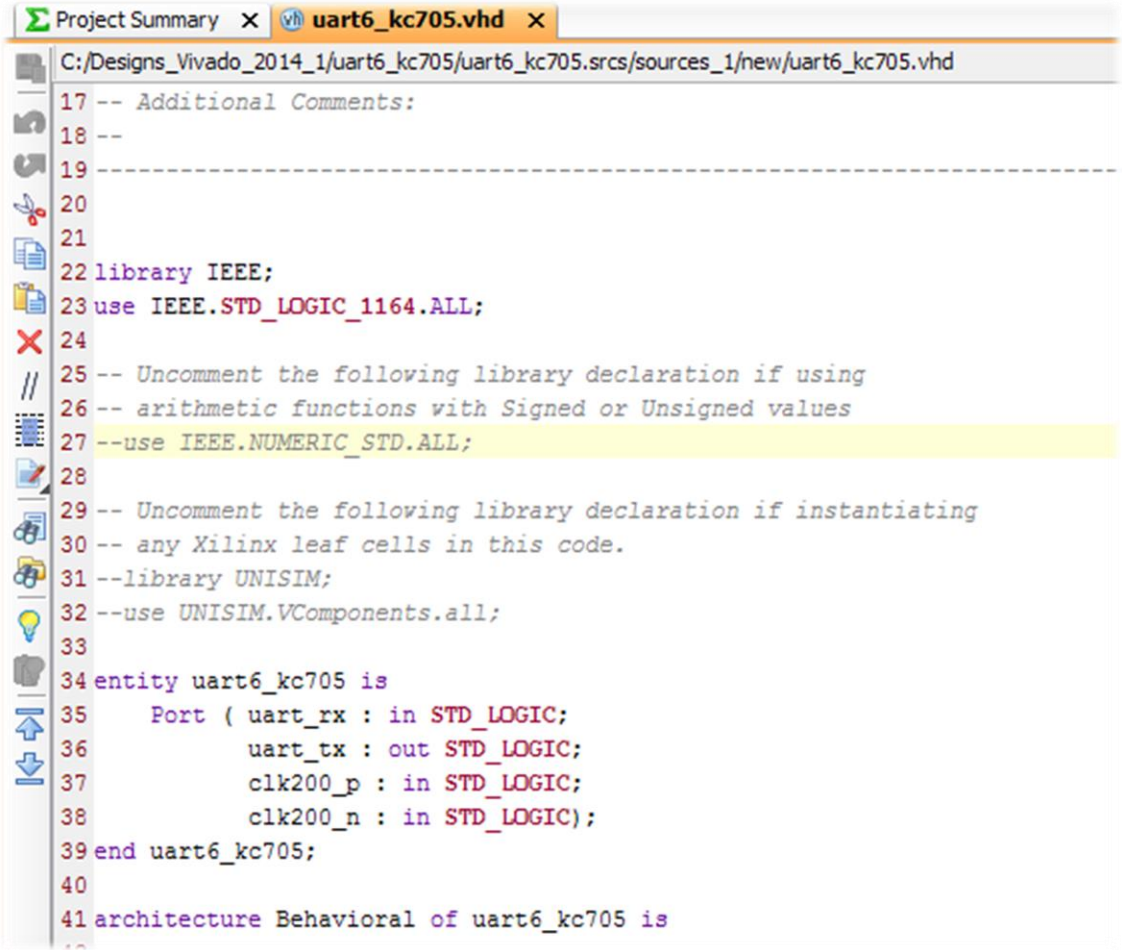

Your file will then appear as the top level file in the 'Sources' window of the project.

Double click on the file name and the file will be opened in a window where you can view and edit it. In this example we can see how the entity and the fundamental structure of the VHDL file has already been generated by Vivado based on the I/O previously specified.

Windows Explorer shows us how the directory structure of the project has developed and where our newly created file has been located.




Unsurprisingly and logically Vivado has created the file in a directory called 'new'. At least we know where it is ☺



At this point we would start to write code to define the PicoBlaze design. Pages 5 to 20 of the KCPSM6 User Guide show you the fundamental steps to include KCPSM6 in a design. Likewise, the UART6 User Guide show you how to include the UART macros in a design and connect them to KCPSM6. To make your life easier the KCPSM6 package contains reference code and reference designs so most of your design work is reduced to simple copy-and-paste tasks.

The ultimate copy-and-paste! In this example the entire contents of the 'uart6_kc705.vhd' reference design were copied and pasted into in to the Vivado editor window and then the file was saved.

The alternative way to achieve this complete replacement would be to physically replace the 'uart6_kc705.vhd' file in the 'new' directory of the Vivado project with a copy of the reference design provided in the KCPSM6 package.

Project Summary | uart6_kc705.vhd *

C:/Designs_Vivado_2014_1/uart6_kc705/uart6_kc705.srcs/sources_1/new/uart6_kc705.vhd

```
74 --
75
76 entity uart6_kc705 is
77   Port (  uart_rx : in std_logic;
78           uart_tx : out std_logic;
79         clk200_p : in std_logic;
80         clk200_n : in std_logic);
81 end uart6_kc705;
```

Whether (like me) you cheated or really did write your design from scratch then the keys points are that you will now have a design file in the 'new' directory of the Vivado project that contains instantiations of KCPSM6 and a program memory with a name that will be associated with a PSM program. Like in this reference design, you may also have instantiated the UART macros if you need them.

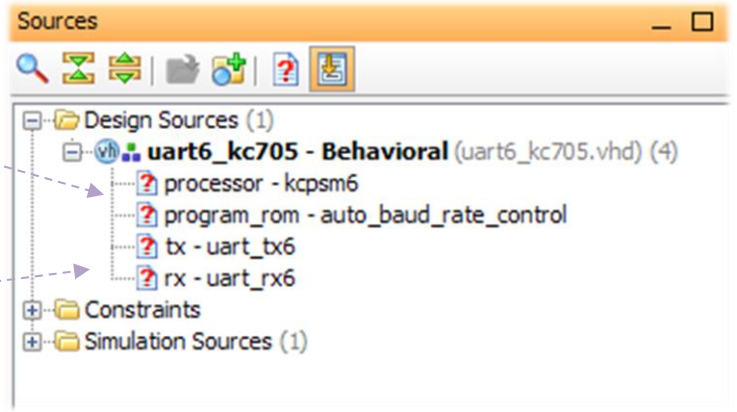
kcpsm6

```
processor: kcpsm6
  generic map (                 hwbuild => X"41",
                       interrupt_vector => X"7FF",
                scratch_pad_memory_size => 64)
  port map(      address => address,
             instruction => instruction,
             bram_enable => bram_enable,
                 port_id => port_id,
            write_strobe => write_strobe,
          k_write_strobe => k_write_strobe,
                out_port => out_port,
             read_strobe => read_strobe,
                 in_port => in_port,
               interrupt => interrupt,
           interrupt_ack => interrupt_ack,
                   sleep => kcpsm6_sleep,
                   reset => kcpsm6_reset,
                     clk => clk);
```

Program ROM which in this reference design is called 'auto_baud_rate_control'

```
program_rom: auto_baud_rate_control
  generic map(              C_FAMILY => "7S",
                         C_RAM_SIZE_KWORDS => 2,
                      C_JTAG_LOADER_ENABLE => 1)
  port map(      address => address,
                 instruction => instruction,
                      enable => bram_enable,
                         rdl => rdl,
                         clk => clk);
```

Having saved the design file containing the instantiations of these components the 'Sources' window of the Vivado project will update to show the hierarchy of the design but also indicate that there is no definition for these components.



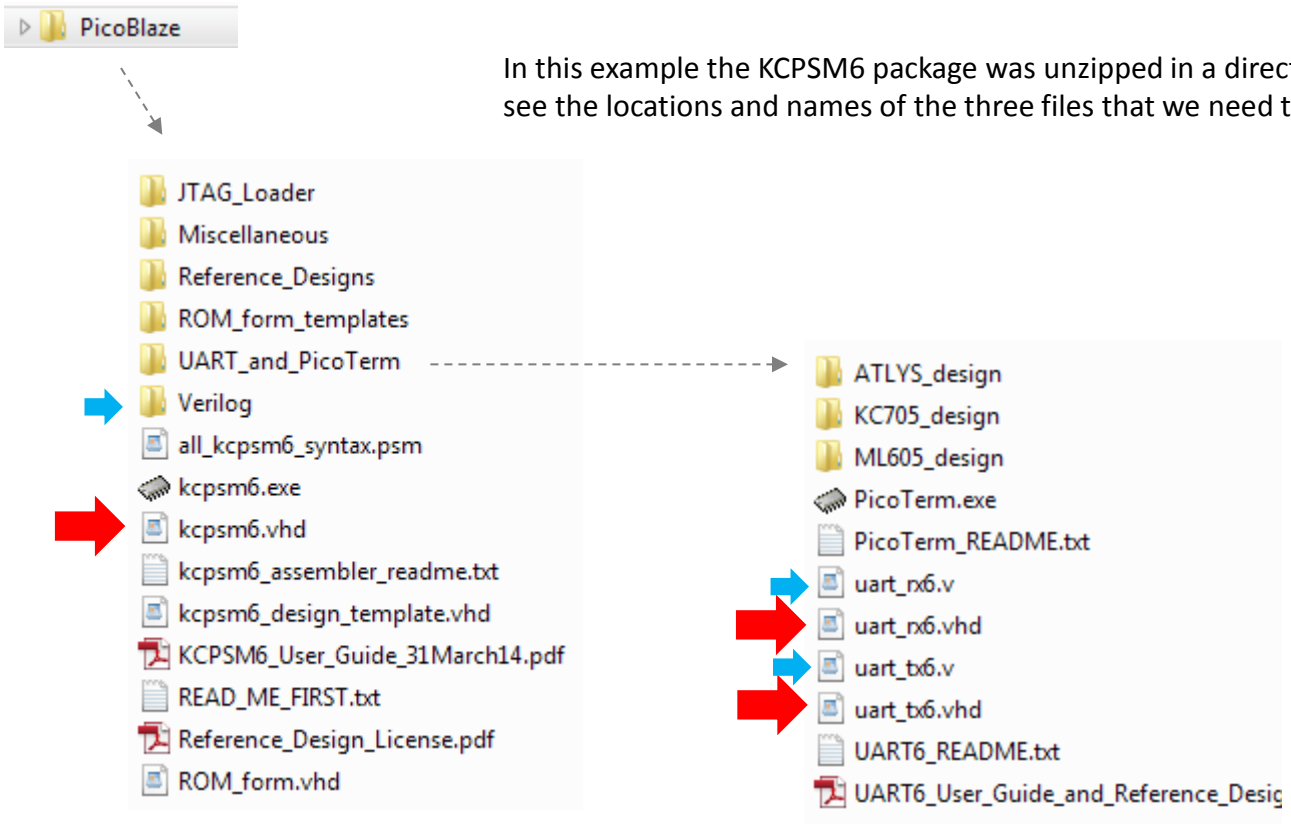

uart_tx6

```
tx: uart_tx6
port map (              data_in => uart_tx_data_in,
                     en_16_x_baud => en_16_x_baud,
                       serial_out => uart_tx,
                     buffer_write => write_to_uart_tx,
              buffer_data_present => uart_tx_data_present,
                 buffer_half_full => uart_tx_half_full,
                      buffer_full => uart_tx_full,
                     buffer_reset => uart_tx_reset,
                              clk => clk);
```

uart_rx6

```
rx: uart_rx6
port map (            serial_in => uart_rx,
                   en_16_x_baud => en_16_x_baud,
                       data_out => uart_rx_data_out,
                    buffer_read => read_from_uart_rx,
             buffer_data_present => uart_rx_data_present,
                 buffer_half_full => uart_rx_half_full,
                      buffer_full => uart_rx_full,
                     buffer_reset => uart_rx_reset,
                              clk => clk);
```

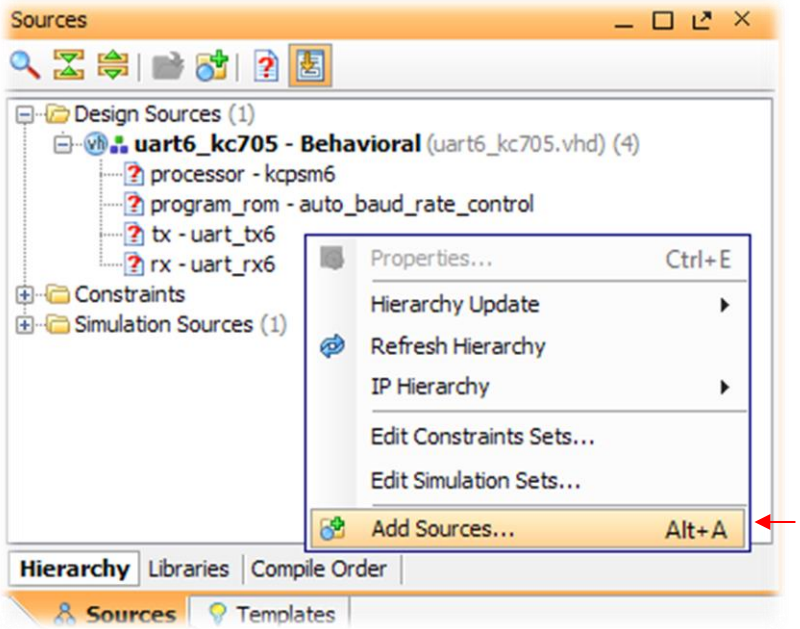
First we will add the files that define KCPSM6 and the UART macros to the Vivado project. We would never want to create these files from scratch so we will add them to the project as pre-existing files . The files are provided in the KCPSM6 package (i.e. ZIP file) but obviously you need to know where they are located on your PC so that you can tell Vivado where to find them.




In this example the KCPSM6 package was unzipped in a directory called 'PicoBlaze' and we can see the locations and names of the three files that we need to add to the Vivado project.

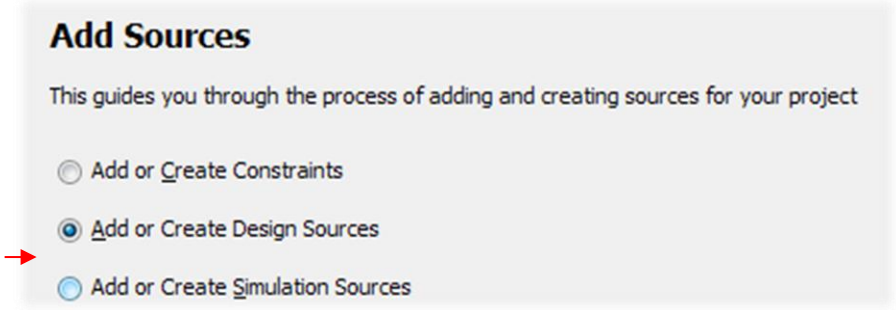
The reason for showing you this directory structure will become apparent after the files have been added to the project.

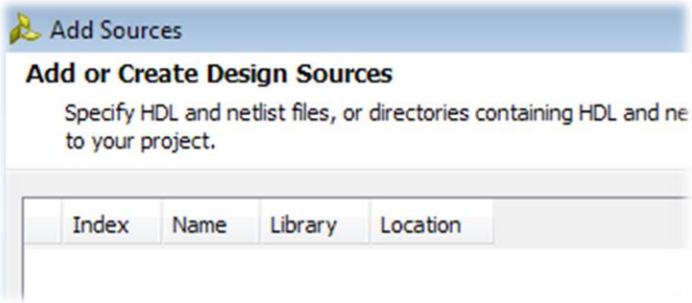
- 'kcpsm6.v' is contained the Verilog subdirectory.



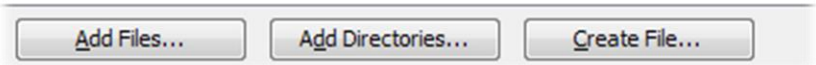

As before, Right click in the 'Sources' window and select 'Add Sources…'

Then select 'Add or Create Design Sources'

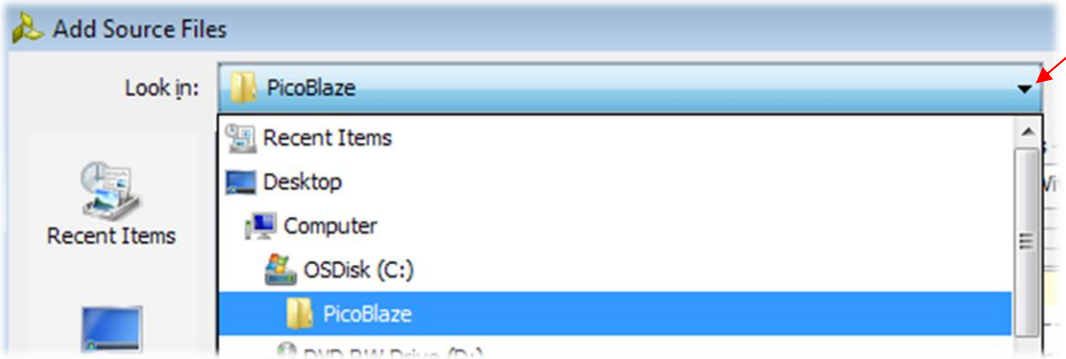


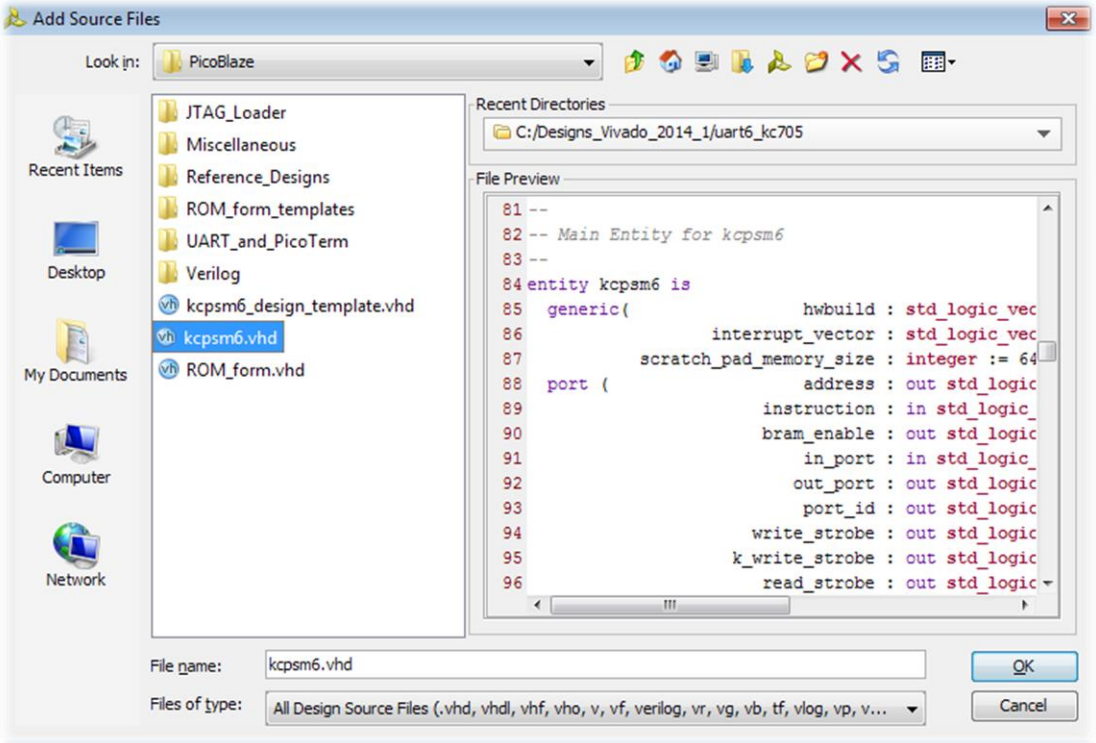

But this time we choose the 'Add Files' option



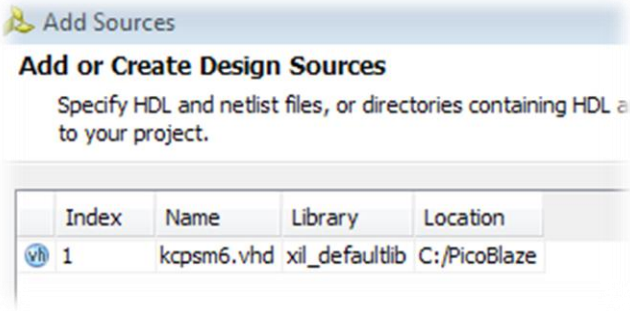

Use the 'Look in' tab to locate the directory of the next file to add.



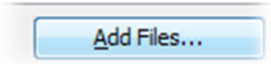
Select a file and click ‘Ok’…



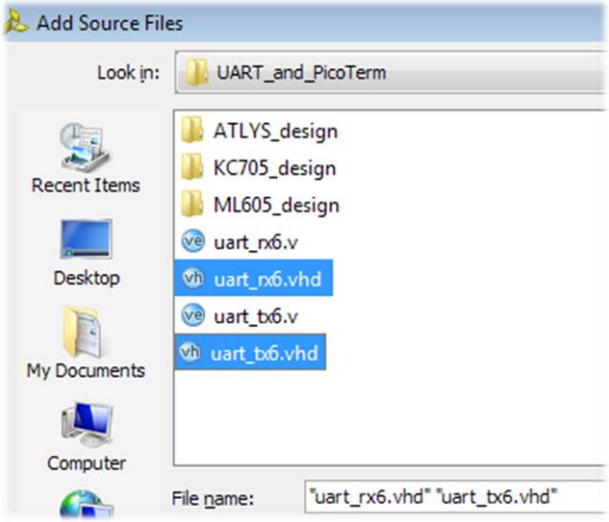

The file will appear on the list of files to be added to the project.




Click on ‘Add Files’ again to repeat the procedure until you have added all the files needed to the list.



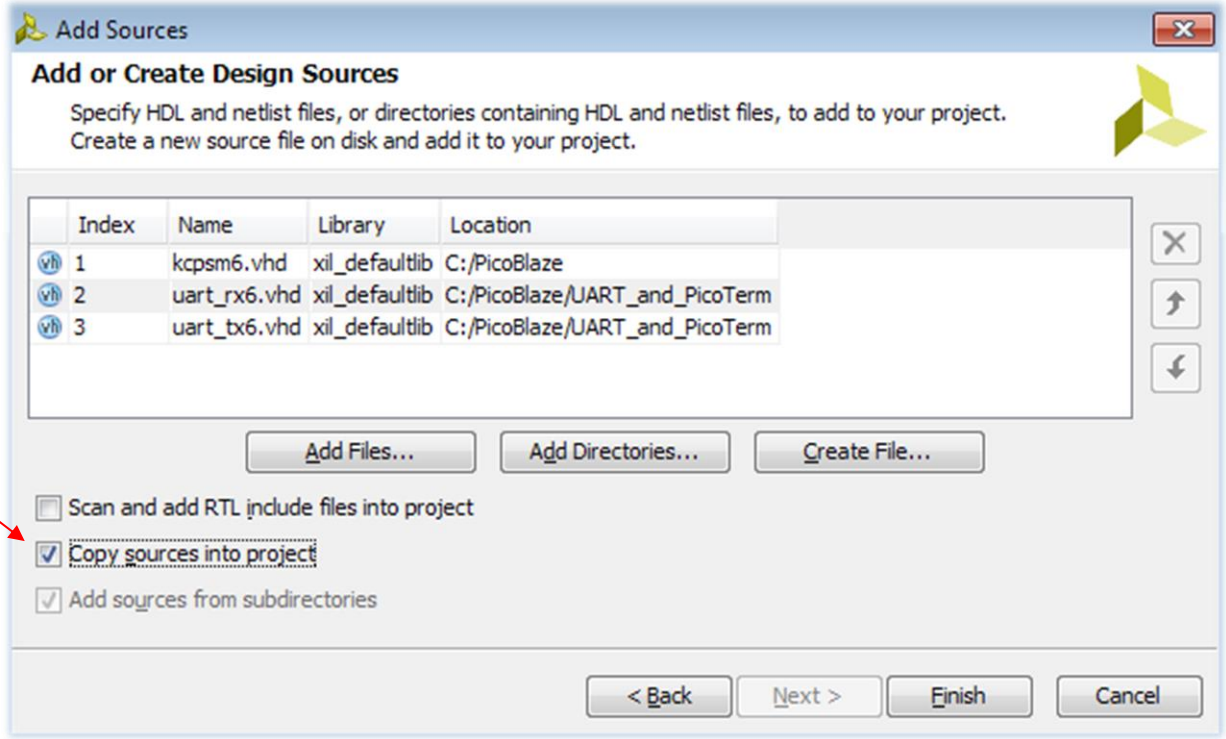

Hint – If more than one to be added to the project is contained in the same directory then you can cold the ‘Ctrl’ and click to select additional files and add them in one go. This example shows both of the UART macros being selected at the same time.



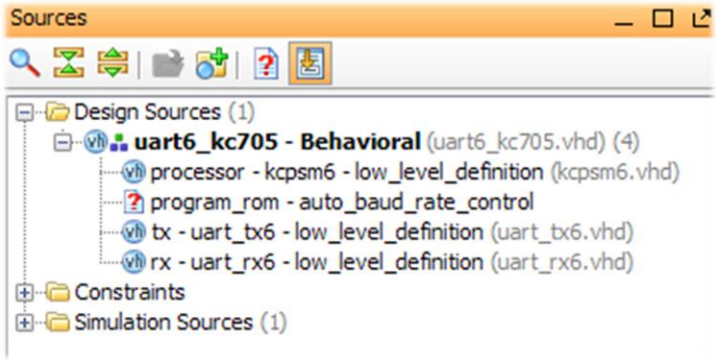
Here we see the list showing the three files to be added to the project.

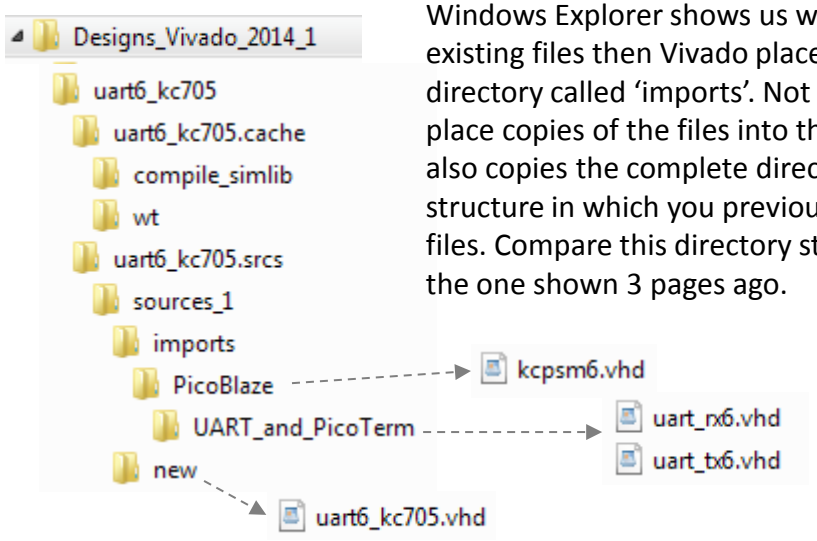
Check this box so that copies of these files will be made and copied into the project. This is optional but if you decide to keep files in other locations do remember that you become responsible for managing them.

Click ‘Finish’ to continue




The hierarchy of the project updates to reflect that three of the components have been resolved and now have sources



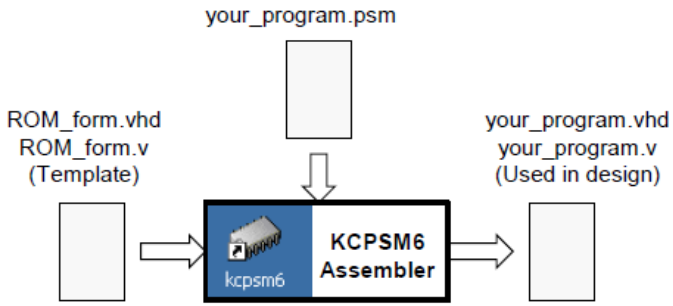




Windows Explorer shows us when we add existing files then Vivado places them in a directory called ‘imports’. Not only does it place copies of the files into this area, it also copies the complete directory structure in which you previously had the files. Compare this directory structure with the one shown 3 pages ago.

Hopefully you are beginning to appreciate that the way in which you create or add files to a Vivado project will determine where Vivado will store them in the project directory structure. This does become significant as we move on to defining a PicoBlaze program memory.

## PicoBlaze Program Memory

As described on page 11 of the KCPSM6 User Guide, the KCPSM6 Assembler will read and assemble your PSM code and generate a VHDL or Verilog file of the same name that defines the programme memory for your design. So once we have written some PSM code we can assemble and generate the last HDL source file to complete the definition of this design. (with the help of a 'ROM_form' template)




The easiest and most straightforward way to assemble PSM code is to place the PSM code together with a copies of the KCPSM6 Assembler and a 'ROM_form' template into the same directory and simply run the assembler . Depending on the 'ROM_form' template provided, the assembler will generate a VHDL or Verilog file of the same name as the top level PSM file. Although rather obvious, it is important to recognise that the HDL file is also generated in the same directory.

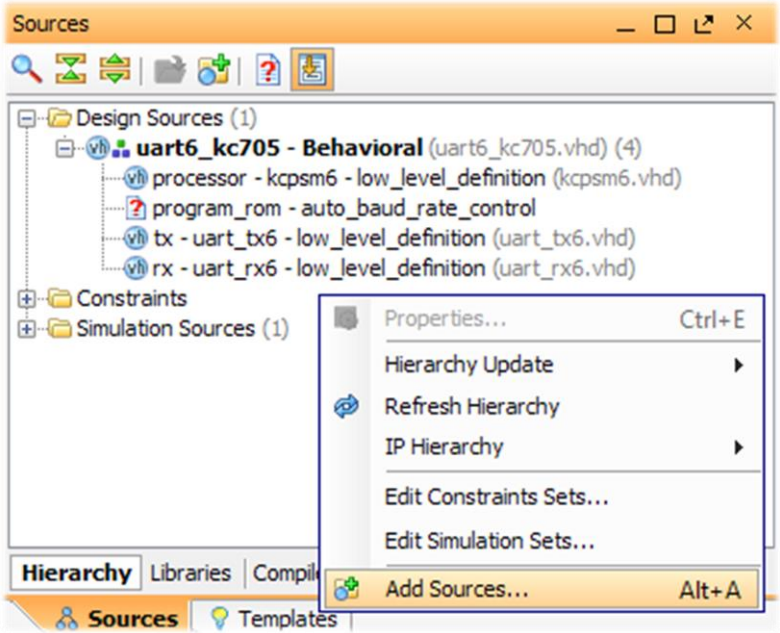
ROM_form_JTAGLoader_Vivado_2June14.vhd
ROM_form_JTAGLoader_Vivado_2June14.v

When using Vivado, the default 'ROM_form' template should be a copy of one of these files (or possibly one with a later date) provided in the KCPSM6 package. Make a copy of the file of the desired language and name it 'ROM_form.vhd' or 'ROM_form.v' as appropriate.

Although it is easy to generate an HDL memory definition file and then to add it to a Vivado project, we should remember that it is also highly likely that we will want to modify the PSM code in the future (i.e. as a program is being developed). Each time the assembler is run, the HDL file in the same directory is updated (overwritten). However, we also need Vivado to use the updated file; not to continue using the a copy of an older version that it imported the first time. This is the reason why it has been useful to know where Vivado creates and stores files. With this knowledge it is possible to set up a simple scheme in which the file generated by the assembler is the one used by Vivado.

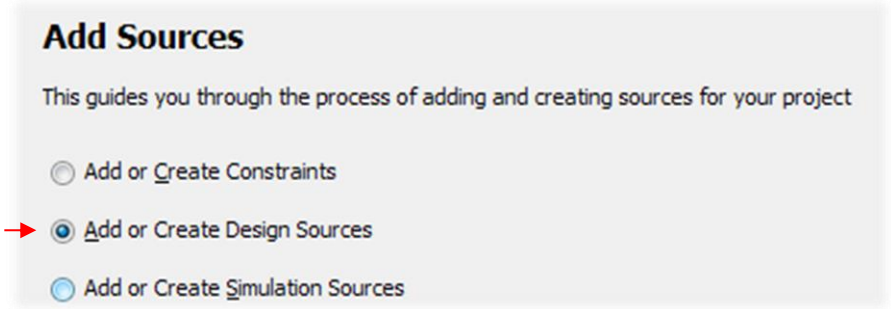
Please note that the scheme described is just one way to implement a PicoBlaze design when using Vivado. As you become more familiar with using Vivado and the KCPSM3 Assembler then you may decide that a different scheme is better suited to your way of working. Just make sure that whatever scheme you choose to use in the future correctly supports updates as well as the initial build.

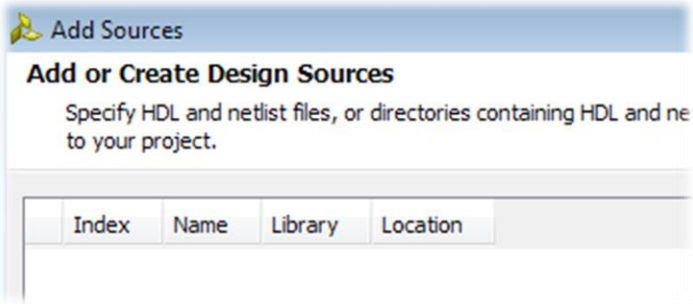
In the same way that we created a top level design file, this scheme will initially 'create' a file as if it were being written from scratch. In this way the file will be added to the project and located in the 'new' directory.



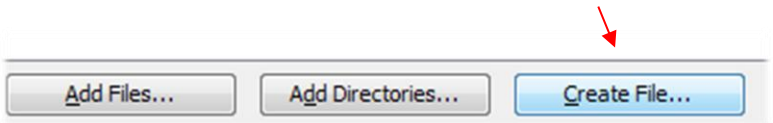

Right click in the 'Sources' window and select 'Add Sources…'

Then select 'Add or Create Design Sources'



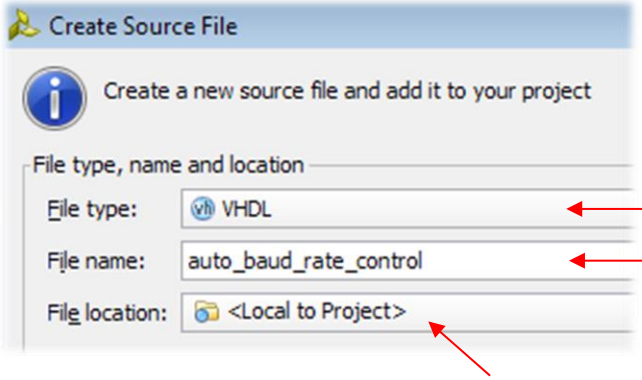

Select the '**Create File**' option to make a new file.




Specify the name for the program definition file. This name *must* match…

i) The name of the top level PSM file name.

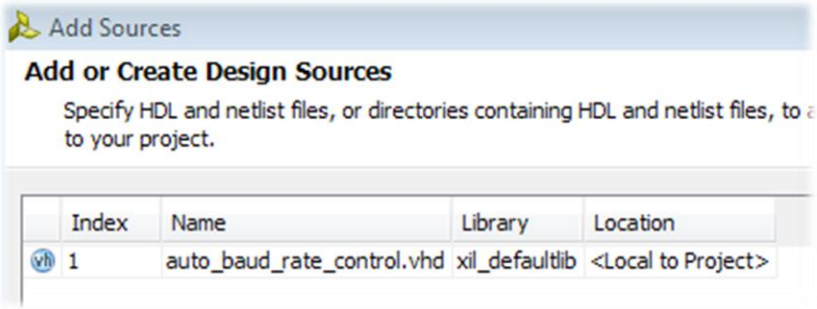
ii) The name of the component defined in the design.



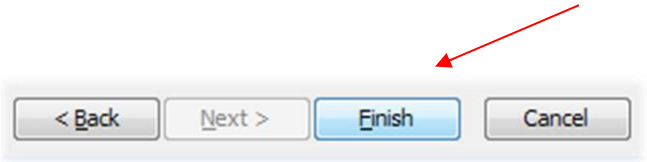

HDL language

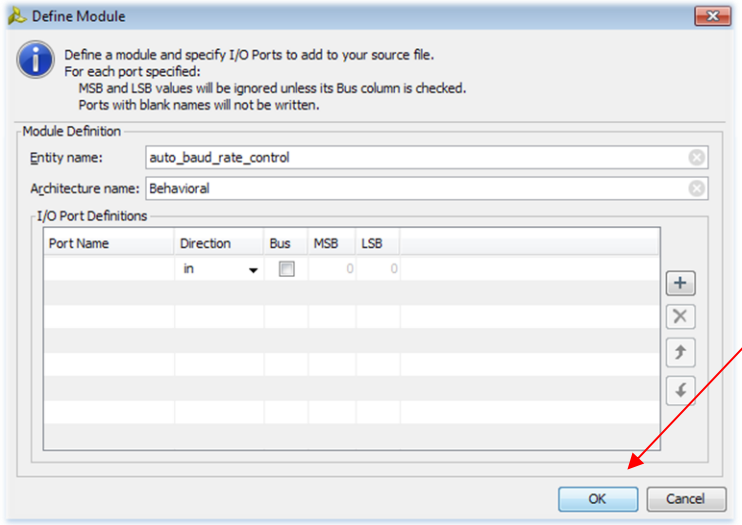
Filename

We will continue to use the default option but clearly Vivdao is presenting you with options to modify the scheme.

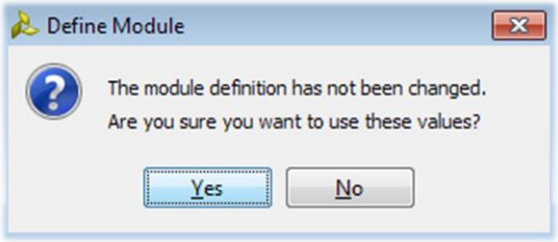



The new program memory file is listed so just click 'Finish' to continue.

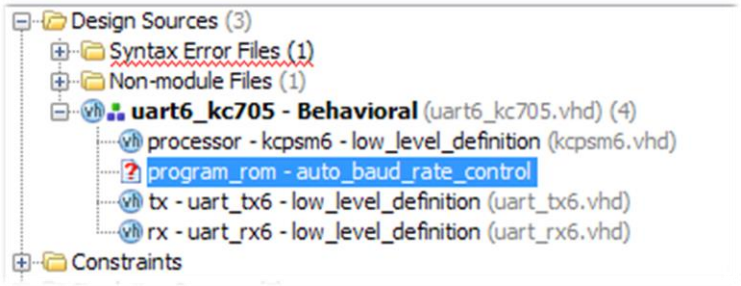


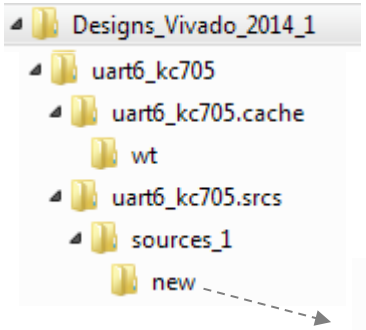




The KCPSM3 assembler is really going to define the program memory so we don't need to waste time specifying the I/O ports and can just click 'OK' and then 'Yes' to continue.

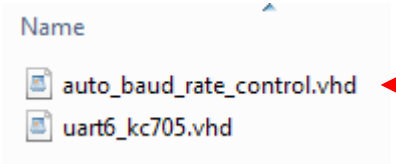


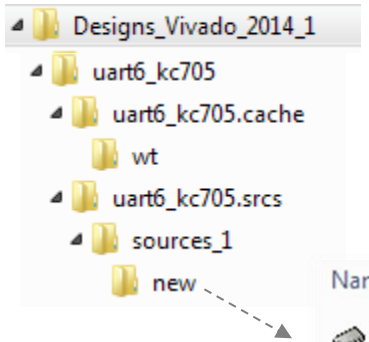

Initially the hierarchy in the 'Sources' window reflects that the program memory is undefined but this is only because the file is still almost empty.



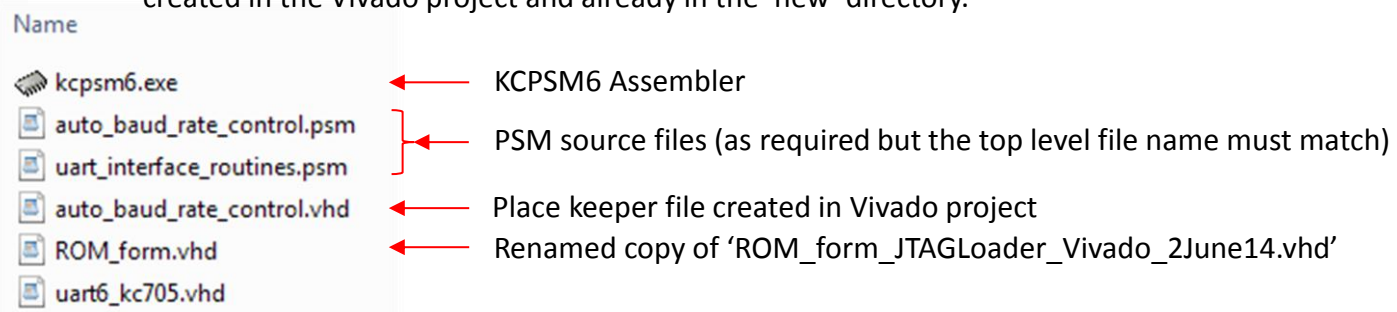

The important thing is that a place keeper for the program memory definition file has been created in the 'new' directory and that file is associated with the Vivado project.






Using Windows Explorer, place copies of the KCPSM6 Assembler and the desired 'ROM_form' template into the 'new' directory.

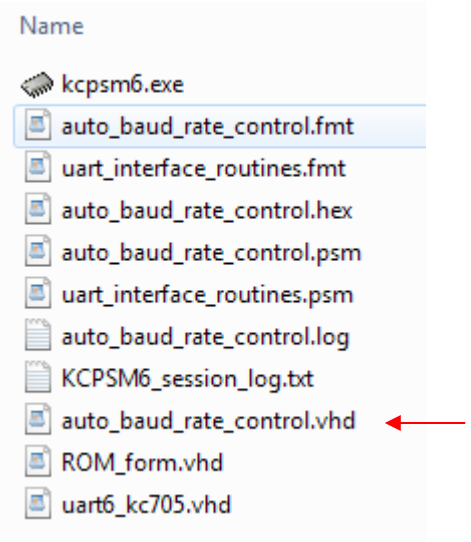
Also provide the PSM code that needs to be assembled. If this was a real design then you would typically create and write a new file using WordPad or other suitable text editor. In this case the PSM files provided with the reference design in the KCPSM6 package have simply been copied into the 'new' directory. The only requirement is that the top level PSM file name must match with the name of the place keeper file created in the Vivado project and already in the 'new' directory.




Assemble the PSM file…

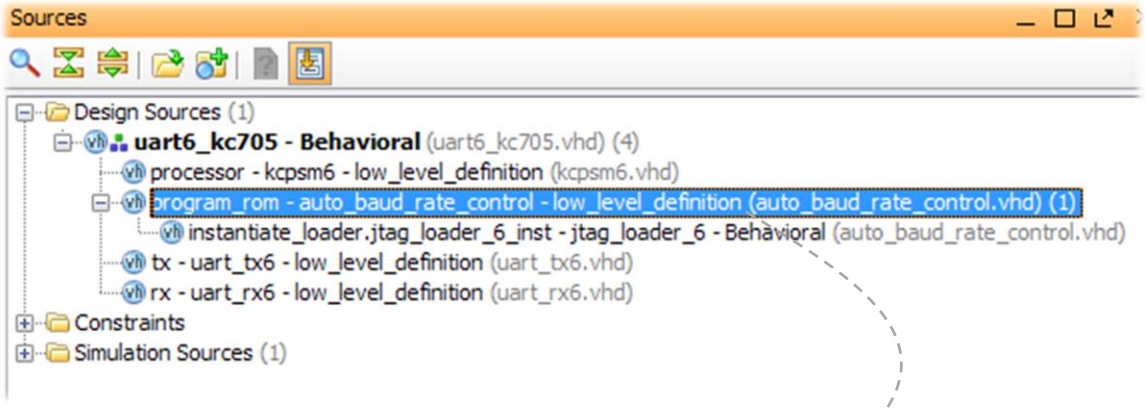
```
kcpsm6.exe
KCPSM6 Assembler v2.70
  Ken Chapman - Xilinx Ltd - 16th May 2014

Reading top level PSM file...
  C:\Designs_Vivado_2014_1\uart6_kc705\uart6_kc705.srcs\sources_1\new\auto_baud_rate_control.psm

Including PSM files...
  C:\Designs_Vivado_2014_1\uart6_kc705\uart6_kc705.srcs\sources_1\new\uart_interface_routines.psm

A total of 777 lines of PSM code have been read

Checking line labels
Checking CONSTANT directives
Checking STRING directives
Checking TABLE directives
Checking instructions

Writing formatted PSM files...
  C:\Designs_Vivado_2014_1\uart6_kc705\uart6_kc705.srcs\sources_1\new\auto_baud_rate_control.fmt
  C:\Designs_Vivado_2014_1\uart6_kc705\uart6_kc705.srcs\sources_1\new\uart_interface_routines.fmt

Expanding text strings
Expanding tables
Resolving addresses and Assembling Instructions
  Last occupied address: 294 hex
  Nominal program memory size: 1K (1024)    address(9:0)
  Occupied memory locations: 661
  Assembly completed successfully

Writing LOG file...
  C:\Designs_Vivado_2014_1\uart6_kc705\uart6_kc705.srcs\sources_1\new\auto_baud_rate_control.log
Writing HEX file...
  C:\Designs_Vivado_2014_1\uart6_kc705\uart6_kc705.srcs\sources_1\new\auto_baud_rate_control.hex
Writing VHDL file...
  C:\Designs_Vivado_2014_1\uart6_kc705\uart6_kc705.srcs\sources_1\new\auto_baud_rate_control.vhd
Complete with 0 Errors
```

Hint – A very quick way to run the assembler is to 'drag and drop' the top level PSM file over 'kcpsm6.exe'. The Assembler will open, assemble the program and close automatically (as soon as there are no syntax errors in the PSM code).

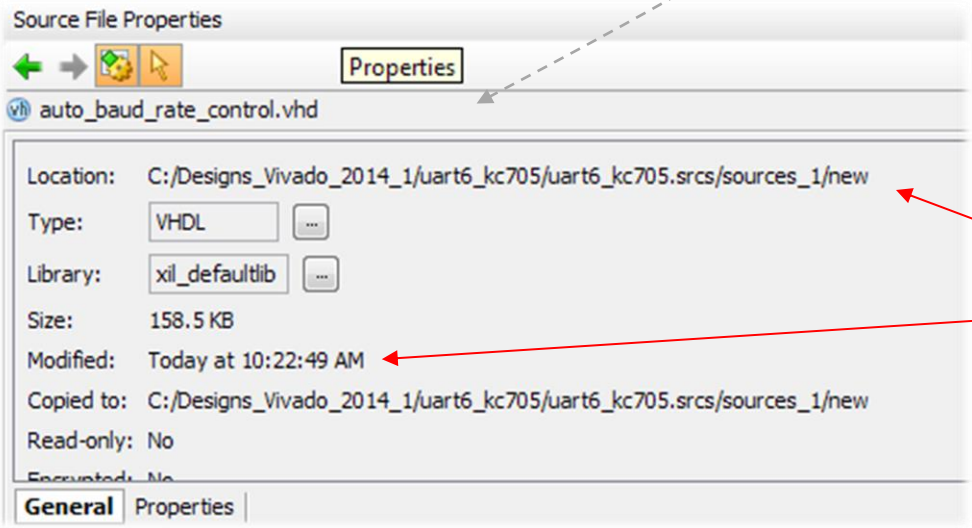



The assembler will generate LOG, HEX and FMT files, but most significantly of all, it will overwrite the HDL memory definition file.

Now that the assembler has generated a real program memory definition file the hierarchy in the 'Sources' window reflects that the design is complete. The default 'ROM_form' template includes JTAG Loader so that has also appeared in the hierarchy.

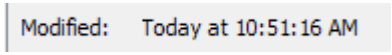



Useful to know….




If you select a file in the 'Sources' window then the 'Source File Properties' window tells you where the file is located and the date and time that it was last modified. This is a convenient way to check that program memory definition file is being updated when you re-run the assembler.

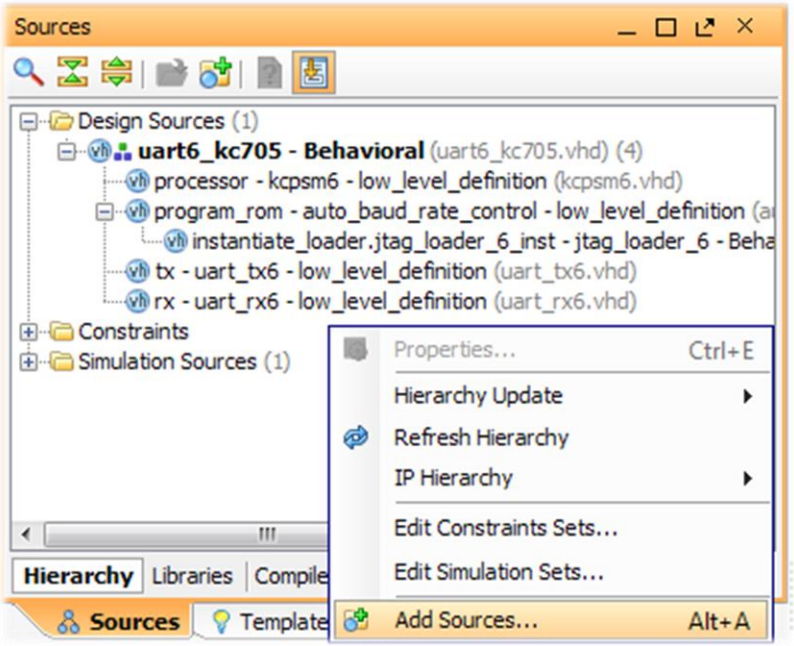
In this example the 'program_rom' defined by 'auto_baud_rate_control.vhd' has been selected . As expected, it is located the 'new' directory of the project and the 'Modified' date and time stamp (Today at 10:22:49 AM) .




Re-running the assembler updates the 'auto_baud_rate_control.vhd' file and, after a few seconds, Vivado reacts to the change in one of its source files and this is reflected by a new 'Modified' date and time stamp (Today at 10:51:16 AM) .

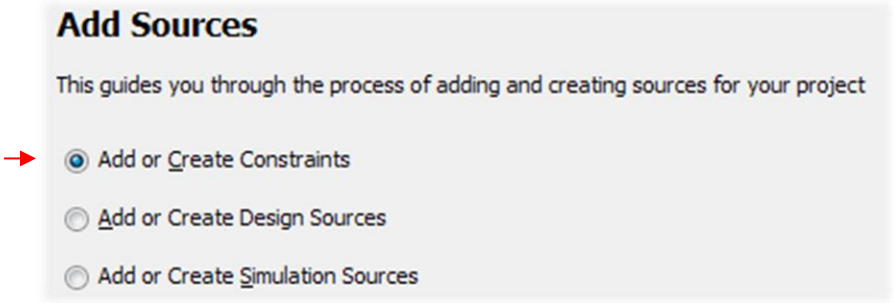
## Constraints

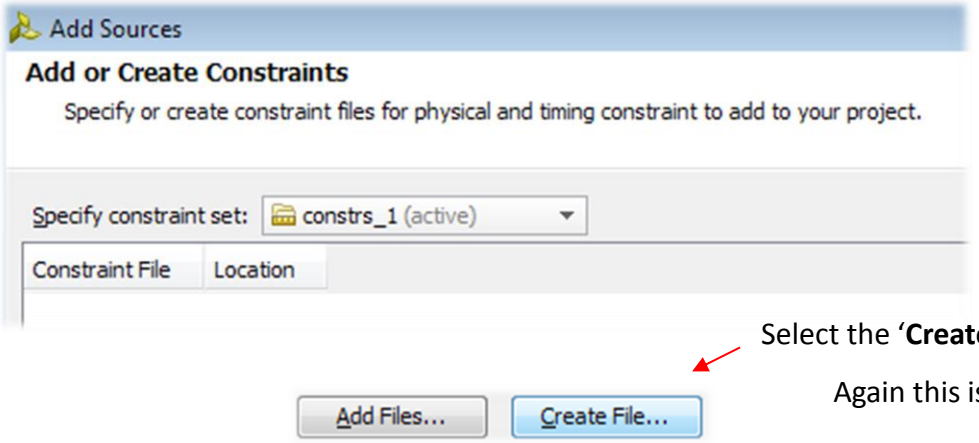
Finally, we need to provide some design constraints. This document only shows how to create an XDC file for your Vivado project. It is beyond the scope of this document to explain the details and syntax of XDC constraints, but in simple terms, constraints must at least define which device pins inputs and outputs will be connected to and the performance that the design must meet.




Right click in the 'Sources' window and select 'Add Sources…'

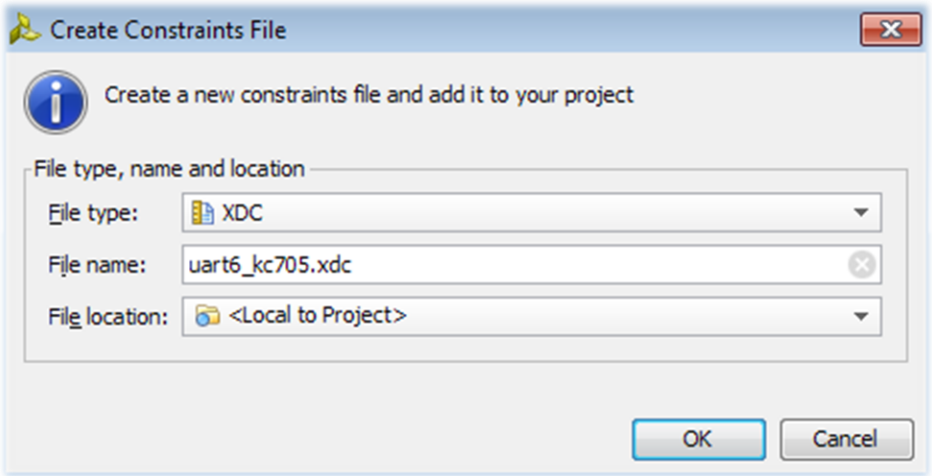
This time select 'Add or Create **Constraints**'



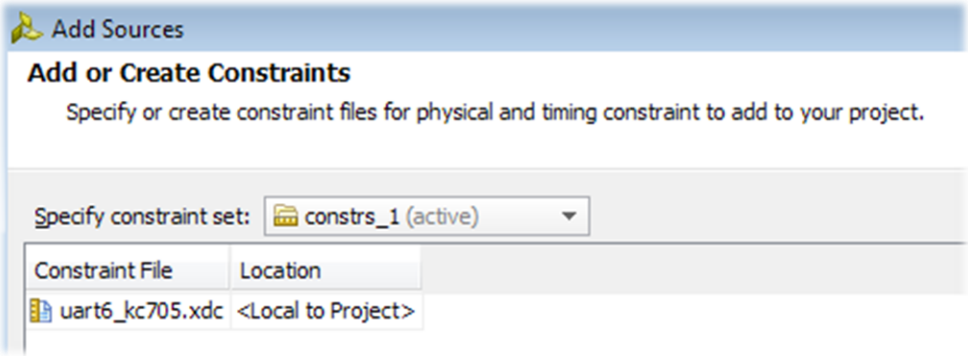

Select the '**Create File**' option to make a new file.

Again this is at least pretending to be creating a design from scratch.

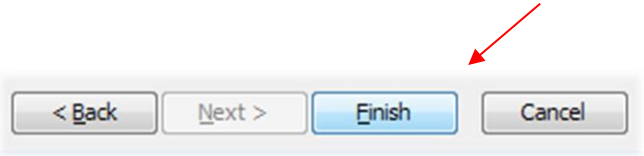



Define a name for your XDC constraints (typically the same name as your top level design)

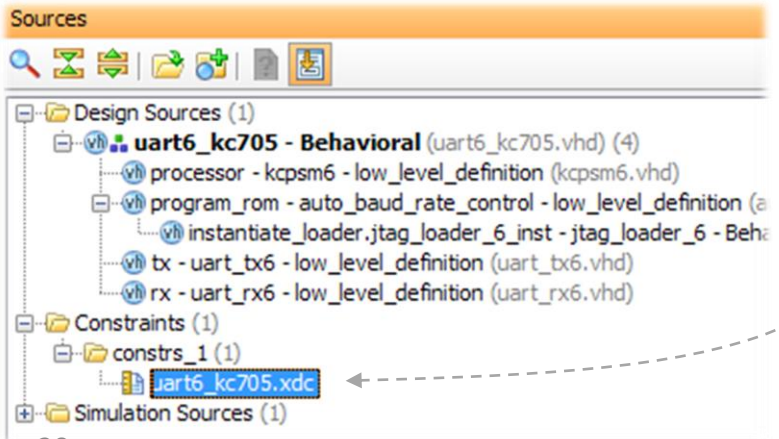
Use the default location.



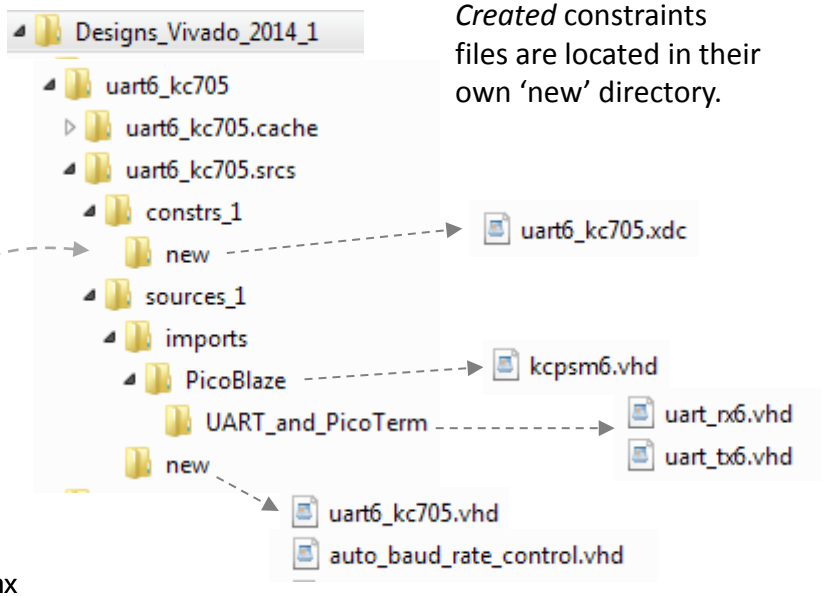

The constraints file appears in the list. Click 'Finish' to continue.

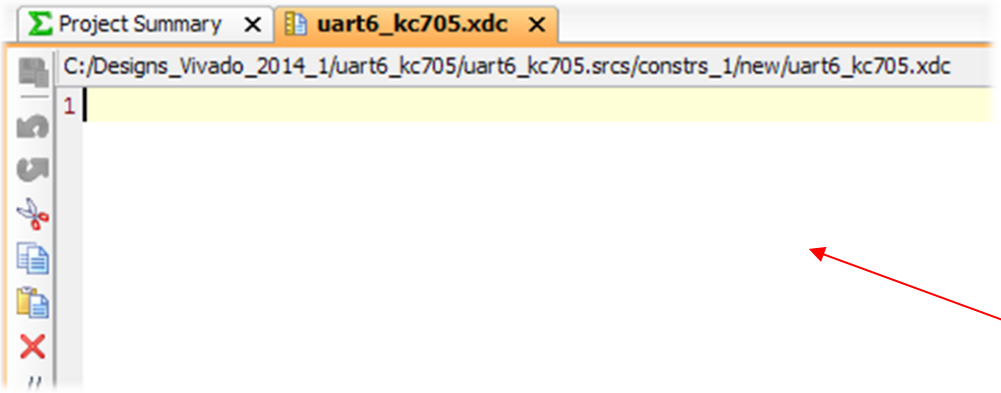



The constraints file can be seen in the 'Sources' window.



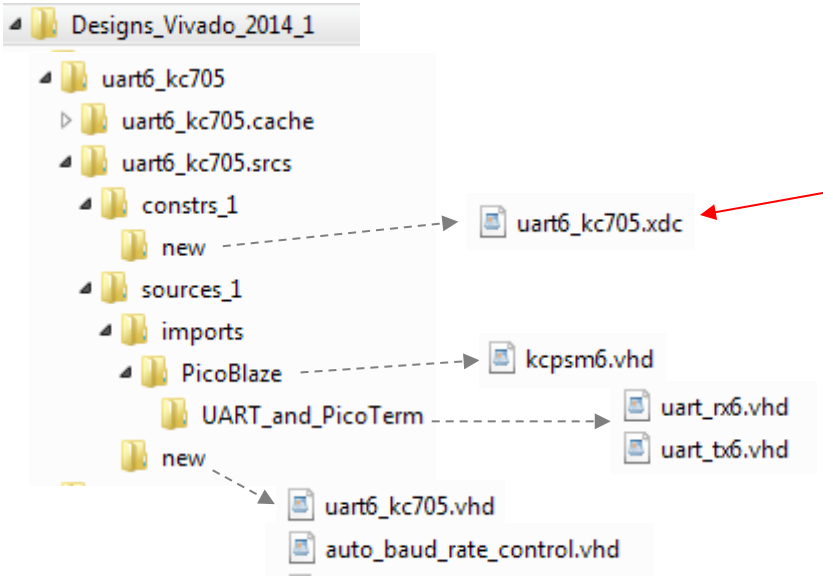

*Created* constraints files are located in their own 'new' directory.





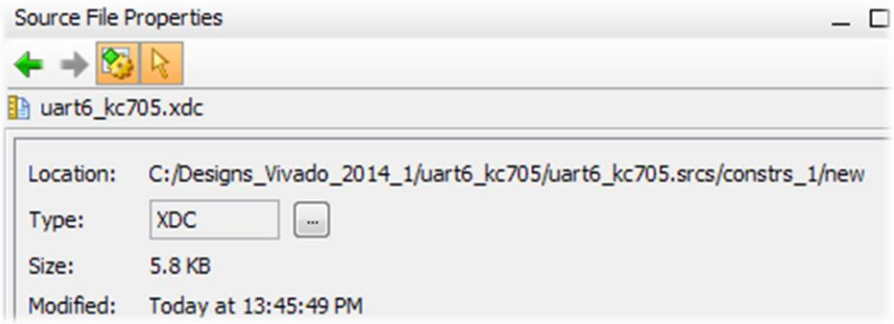

The created XDC file will be completely empty. You can double click on the file name in the 'Sources' window to open it in an editing window.

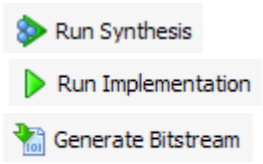
```
# USB-UART
# --------
#
set_property PACKAGE_PIN M19 [get_ports uart_rx]
set_property IOSTANDARD LVCMOS25 [get_ports uart_rx]
#
set_property PACKAGE_PIN K24 [get_ports uart_tx]
set_property IOSTANDARD LVCMOS25 [get_ports uart_tx]
set_property SLEW SLOW [get_ports uart_tx]
set_property DRIVE 4 [get_ports uart_tx]
```




In this case an XDC file has been provided with the reference design so you can either cut-and-paste the constraints provided in 'uart6_kc705.xdc' into to the editor and save the file or you can directly replace the XDC file in the project directory.

Hint – Remember that the location and date/time stamp of any source files, including the constraints file, can be seen and checked in the 'Source File Properties' window.





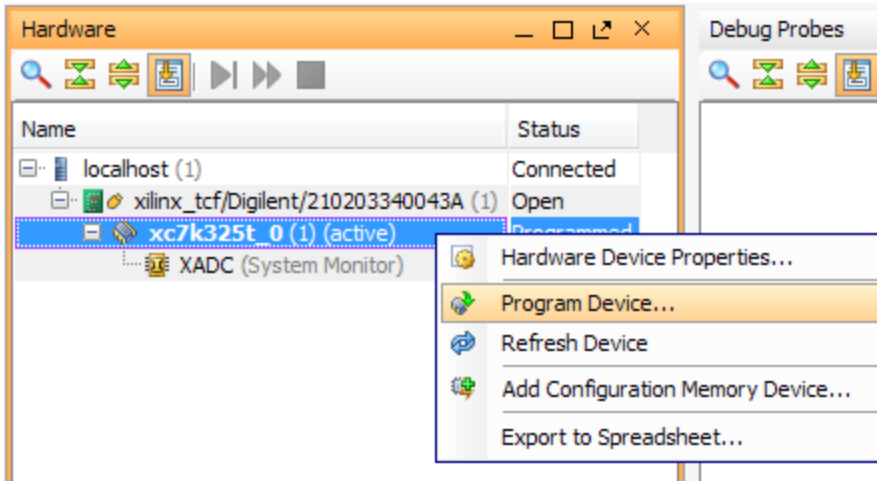

**Implementing the design and generating the configuration BIT file**

Assuming that your design synthesises, implements and generates a bit stream successfully then you are ready to configure your device.

**Configuring the device**

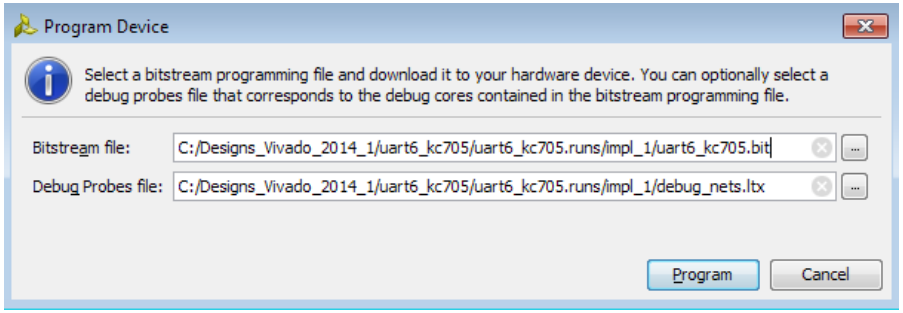
Vivado provides 'Hardware Manager' which is shown below. However, **there is an issue that you should be aware of** when using Vivado (up to and including at least version 2014.2).

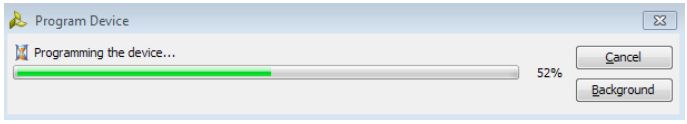

Make sure your target is connected and powered and then Click on 'Open Target' and then select 'localhost…'



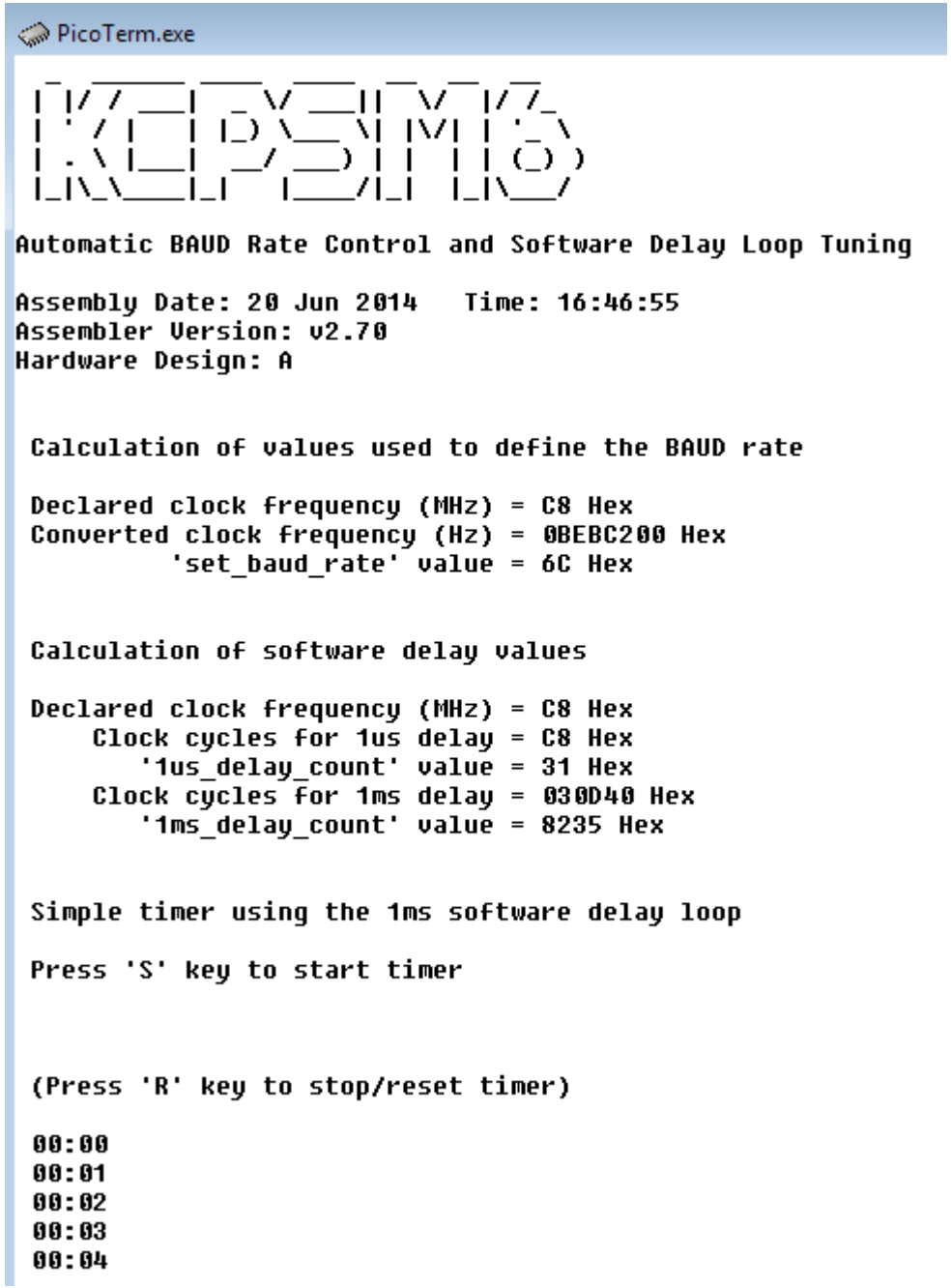

Once the connection is made, right click on the target device and select 'Program Device'.

Check that Vivado has identified the correct BIT file (or manually select the file you want to use) and then click 'Program'.




The device should be configured and PicoBlaze should become active.
In this example, communication via the USB/UART should result in something being displayed on the terminal (e.g. PicoTerm).




If you have been reconstructing the ‘uart6_kc705’ reference design ‘as is’ then with PicoTerm suitably connected to a KC705 board the design should come to life as shown here.

The ‘auto_baud_rate_control.psm’ program provided with the ‘uart6_kc705’ reference design ultimately implements a simple minutes and seconds timer. However, this timer is really just a way to demonstrate the more significant code which sets the BAUD rate of the UART and adjusts software delay loops to reflect the frequency of the clock provided to KCPSM6.

For more details please review the descriptions contained in the source files and refer to the following document provided in the KCPSM6 package….

‘ UART6_User_Guide_and_Reference_Designs_30Sept14.pd’

*If your own design doesn’t do exactly what you were expecting then you may have encountered the issue caused by ‘Hardware Manager’ which is covered on the following pages.*

## **Known Issue** - **Post-configuration issue caused by Vivado 'Hardware Manager'** (up to and including at least version 2014.2)

'JTAG Loader' provides a way to upload a new PicoBlaze program to program memory within an active device. It facilitates rapid development of PSM code as well as ad hoc experiments (e.g. loading special programs to test, monitor and diagnose hardware issues etc.). It s for these reasons that JTAG Loader is provided as an option in the development 'ROM_form' template and it is generally recommended that you should enable this feature when developing designs.

The 'uart6_kc705' reference design illustrated in this document does enable the JTAG Loader feature as shown in this abstract of the 'uart6_kc705.vhd' file.

```
--
-- Development Program Memory
--   JTAG Loader enabled for rapid code development.
--

program_rom: auto_baud_rate_control
  generic map(              C_FAMILY => "7S",
                  C_RAM_SIZE_KWORDS => 2,
               C_JTAG_LOADER_ENABLE => 1)
  port map(       address => address,
              instruction => instruction,
                   enable => bram_enable,
                      rdl => rdl,
                      clk => clk);
```

JTAG Loader enabled

After 'Hardware Manager' has finished configuring the device it interrogates the internal BSCAN ports of the device to discover if there are any Xilinx IP cores present in the design. Unfortunately this process does not respect the 'USER' addresses assigned to each BSCAN primitive and this interrogation interferes with the 'JTAG Loader' circuits in such a way that it results in corruption of the program memory contents. Hopefully this undesirable behaviour will be resolved in later version of Vivado. On a positive note, the corruption is limited to program address 003 which is always cleared. Op-code 00000 hex is a 'LOAD s0, s0' instruction which is ultimately has no effect on the contents of registers or flags but that is of little comfort if the instruction it replaced was means to do something!

**Workarounds**

There is nothing fundamentally wrong with the configuration image (BIT file) so there would be no issues when a production design is configured in an embedded system (e.g. from an image held in Flash memory). As such, this issue only tends to impact operation during design development but that can lead to confusion exactly when you don't need to be confused! Fortunately there are some workarounds...

a) Adjust the PSM code such that memory location 003 is not used by the program. A example of suitable code to place at the start of your program is shown below.

```
            JUMP cold_start   ;Avoid address 003 on start up
            JUMP cold_start
            JUMP cold_start
            JUMP cold_start   ;Address 003
            ;
cold_start: <normal program code starts here>
```

b) Configure the device using iMPACT from ISE. (Note that ChipScope Analyser has the same issue as 'Hardware Manager').

c) Refresh the program memory using JTAG Loader. Just be aware of anything nasty the initial execution of the corrupted program may have done.

d) Disable JTAG Loader. Of course this also inhibits rapid code development!

## Updating your PicoBlaze program using JTAG Loader

More about JTAG Loader on pages 25-29 of 'KCPSM6_User_Guide_30Sept14.pdf'

When you are developing a KCPSM6 program it only takes a few seconds to assemble the modified PSM code and generate a new VHDL or Verilog file defining the new contents of the program memory. Although it doesn't take tool long to run a small design through Vivado synthesis and implementation again process a small design it soon becomes tiresome and unproductive. JTAG Loader is a mechanism that enables you to load a newly assembled program (in the form of a HEX file) directly into the program memory inside the 7-Series or UltraScale device whilst it remains configured and active with your design. This ability to assemble and execute a modified program in less than 15 seconds enables you to develop code in a very interactive way as well as facilitating experiments (e.g. loading different programs on a temporary basis simply to test something).

As shown on the previous page, your design (like the reference design) needs to instantiate the development program memory with JTAG Loader enabled (i.e. C_JTAG_LOADER_ENABLE => 1). This will insert the small JTAG Loader circuit in the design that connects the second port of the BRAM(s) that form the program memory to the BSCAN primitive. The 'JTAG_Loader' utility program then communicates with the device using JTAG to control the BSCAN primitive and load (or read back) a program HEX file.

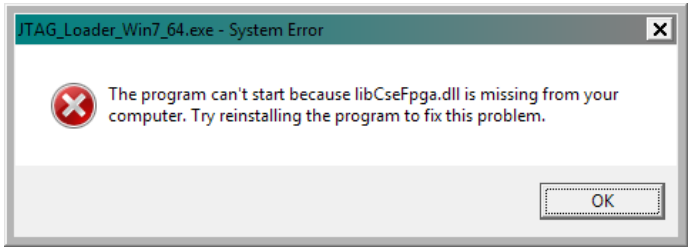
Note – The JTAG Loader utility uses the drivers and files associated with ChipScope which are in tern part of an ISE installation. So at the time of writing it is required that you have an installation of ISE as well as Vivado.

In order that JTAG Loader can operate correctly it must also know where ISE has been installed and this is achieved by setting the PATH and a XILINX environment variable.

As shown in this example, it is probably easiest if you create a batch file which sets the environment variables and then runs JTAG Loader specifying the assembled HEX file which is to be loaded into the program memory.

```
REM Setting environment variables to define location of ISE v14.7
PATH=%PATH%;C:\Xilinx\14.7\ISE_DS\ISE\lib\nt64
set XILINX=C:\Xilinx\14.7\ISE_DS\ISE

REM Upload program HEX using JTAG Loader
JTAG_Loader_Win7_64.exe -l auto_baud_rate_control.hex
```

Adjust as required




If you see a message like this then you still need to define your PATH appropriately. For example, with a default installation of ISE v14.7 your PATH is required to include C:\Xilinx\14.7\ISE_DS\ISE\lib\nt64;

```
JTAG_Loader_Win7_64.exe
The XILINX environment variable is not set or is empty.
```

If you see a message like this then you still need to set a XILINX environment variable appropriately. For example, with a default installation of ISE v14.7 you should set XILINX=C:\Xilinx\14.7\ISE_DS\ISE

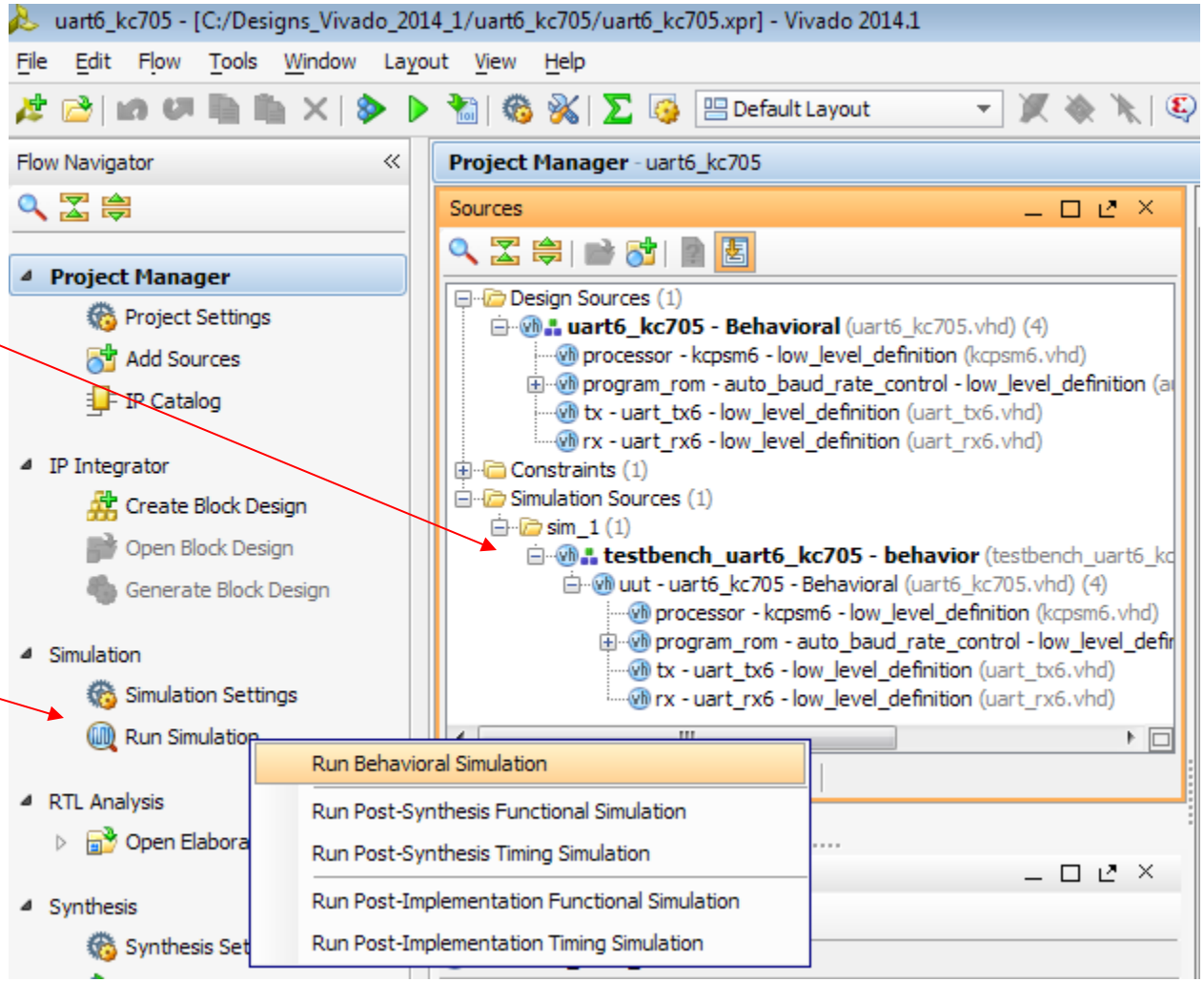
**KCPSM6 Simulation**

Along with the rest of a design, KCPSM6 can be simulated in Vivado.Simulation .

A simple test bench simulation file called 'testbench_uart6_kc705.vhd' is provided with the reference design. This can be added to the Vivado project in what should now be a familiar way (hint – select Add or Create Simulation Sources when you 'Add Sources').

Test bench simulation file added to the project.

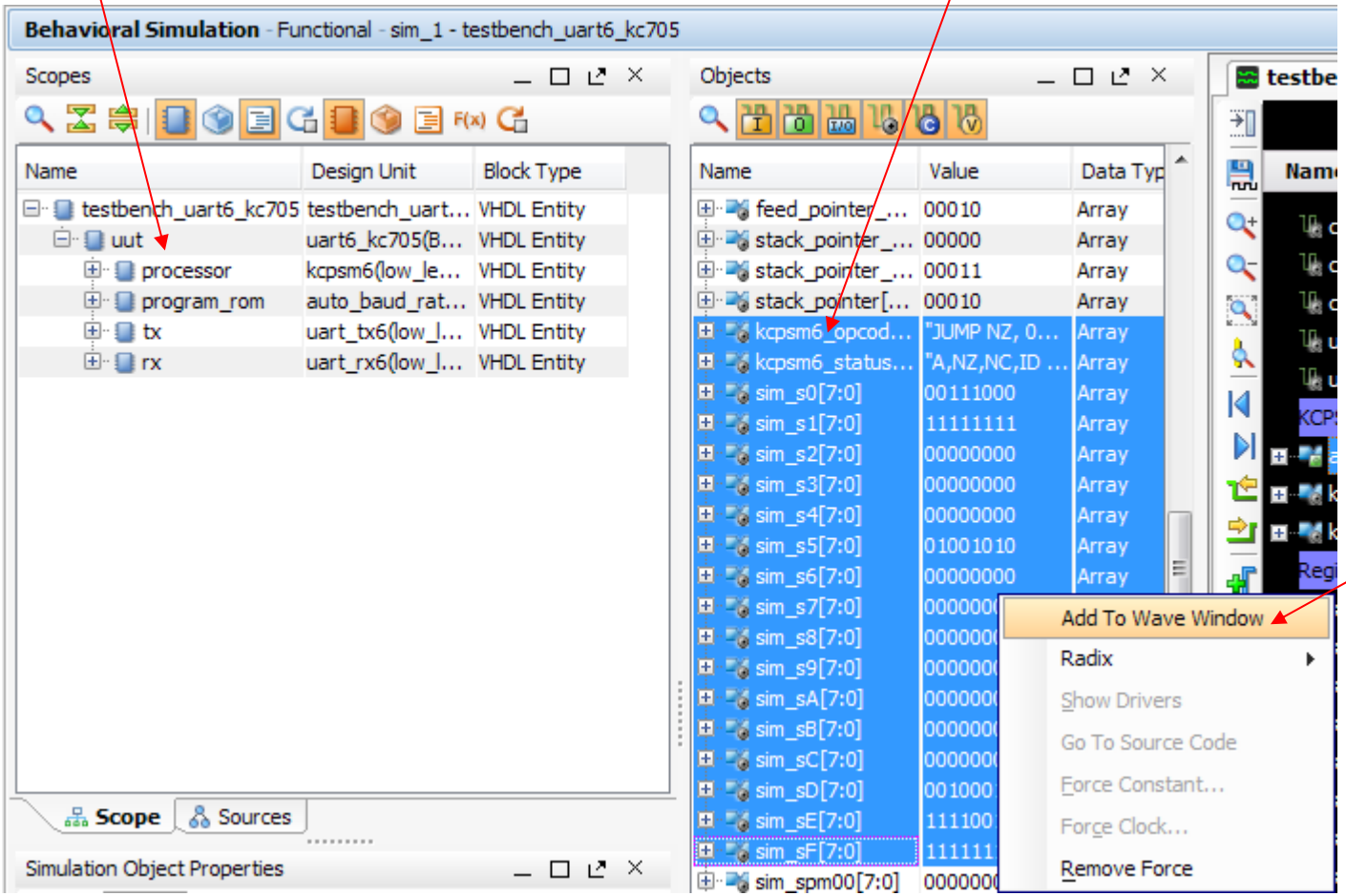
Run Behavioural Simulation



When the simulator opens for the first time the only signals shown in the waveform window are the inputs and outputs of the top level design (i.e. the I/O pins of the device). KCPSM6 simulation becomes more interesting and informative if you add internal signals to the waveform viewer. A very useful signal to observe is the program 'address' as that tells you what instructions KCPSM6 is executing and this can be compared with the Assembler LOG file. The KCPSM6 macro also includes some simulation only signals to help you more directly so look for 'kcpsm6_opcode' and 'kcpsm6_status' which are text strings that decode and display the instruction being executed and the status of the flags. 'sim_s0' through to 'sim_sF' enable you to observe the contents of the registers s0 through to sF, and likewise, 'sim_spm00' through to 'sim_spmFF' allow you to observe the contents of scratch pad memory locations 00 through to FF so pick the ones of relevance to the PSM code being examined. Note there is more about simulation pages 45-46 of 'KCPSM6_User_Guide_30Sept14.pdf'.

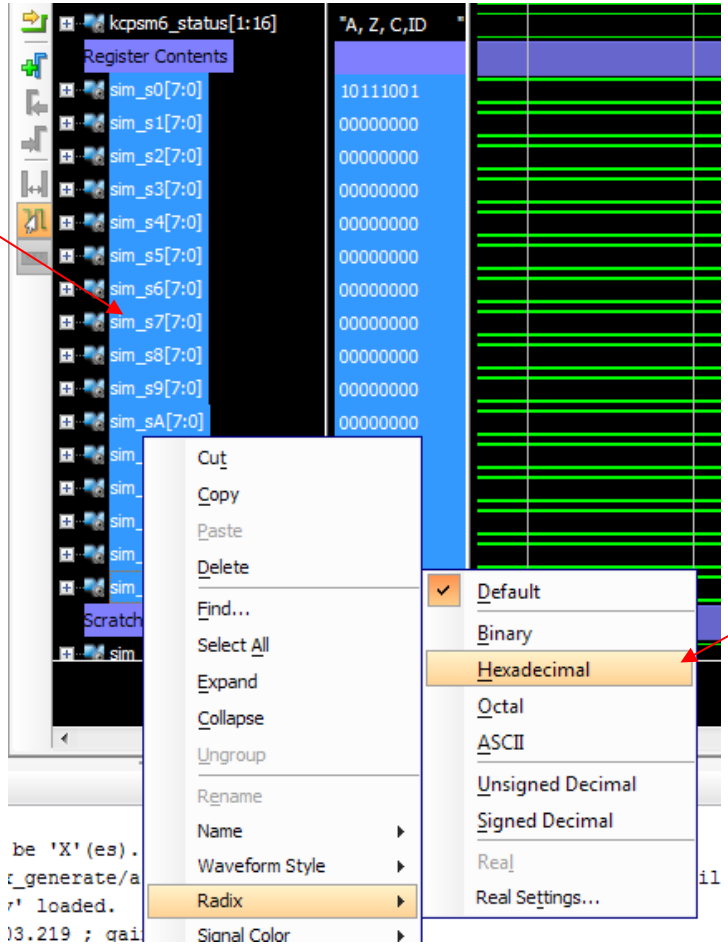
Locate and select the KCPSM6 'processor'

Locate and select the 'Objects' of interest




Right click and select 'Add To Wave Window'

Select signals and right click




Having added various signals to the waveform window then it is generally easy to digest the information if you specify a suitable 'Radix'. For example, it is common practice to display the KCPSM6 program address and the register and scratch pad memory contents in Hexadecimal.

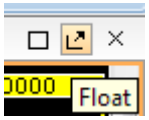
Hint – The assembler LOG file shows all values in Hexadecimal.

Values shown in hexadecimal followed by their original definitions enclosed in brackets

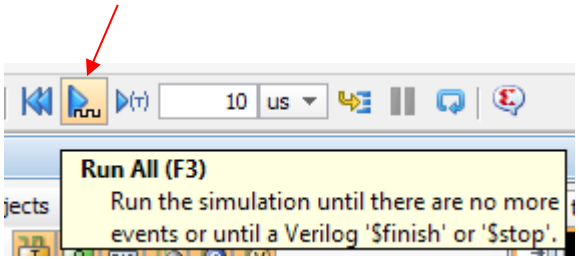
Address

```
 Addr Code                                  Instruction

 10A  00760        calc_1us_delay_count: LOAD s7, s6
 10B  19706                              SUB s7, 06[6'd]
 10C  01400                              LOAD s4, 00[0'd]
 10D  11401             1us_divide_loop: ADD s4, 01[1'd]
 10E  19704                              SUB s7, 04[4'd]
 10F  3E10D                              JUMP NC, 10D[1us_divide_loop]
```

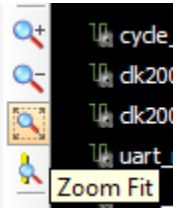
Hint – ‘testbench_uart6_kc705.vhd’ has been written in a style that has a pre-defined end after 5,000 clock cycles have been simulated to you can use the ‘Run All’ button to run the full simulation.



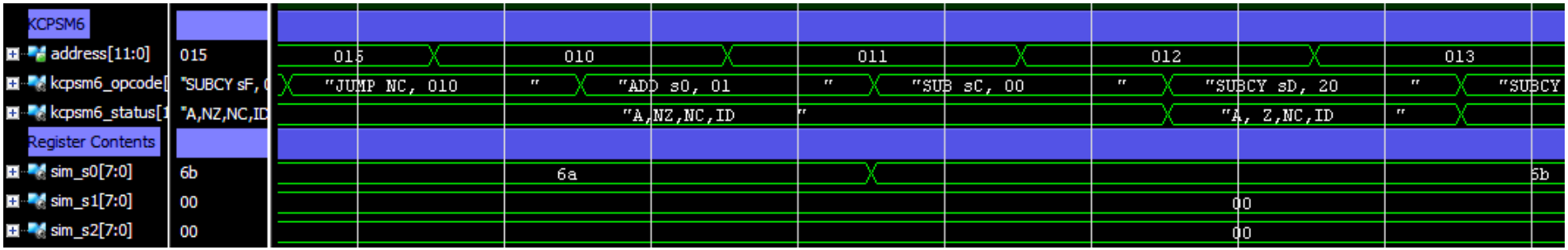

Hint – It can really help if you ‘Float’ the waveform window so that you can then make it much larger.




Hint – Zoom in or out using these controls. Start with ‘Zoom Fit’ to obtain an overall impression of design activity.



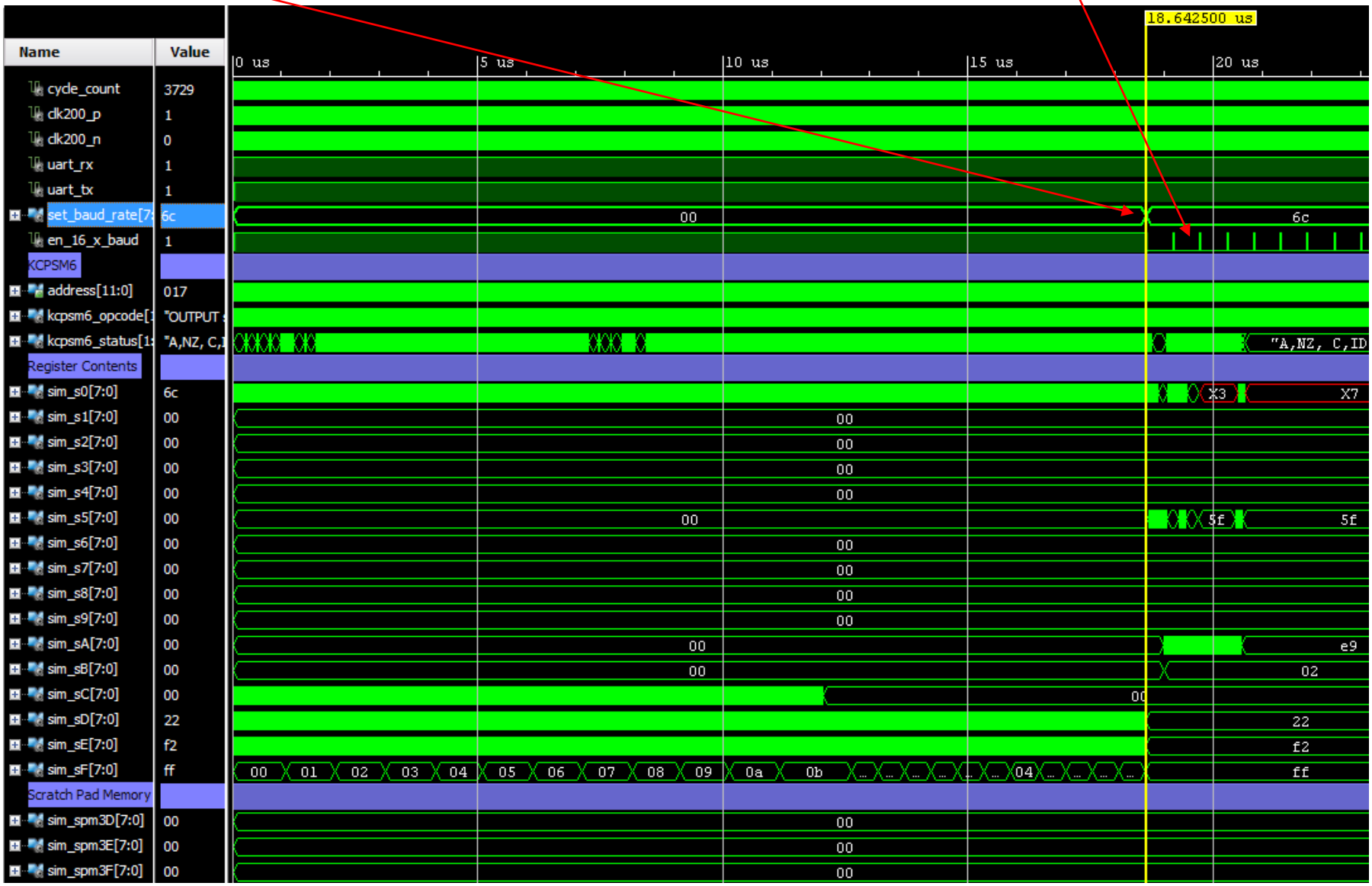

Hint – Using the ‘New Divider’ option can be used to organise your waveforms into a nice display like this example.




The same code being executed in the simulator as seen in the assembler LOG file

```
Addr Code                                  Instruction

010  11001          set_baud_rate_loop: ADD s0, 01[1'd]
011  19C00                              SUB sC, 00
012  1BD20                              SUBCY sD, 20
013  1BE1C                              SUBCY sE, 1C
014  1BF00                              SUBCY sF, 00
015  3E010                              JUMP NC, 010[set_baud_rate_loop]
```

This image shows the complete 5,000 clock cycle simulation. It can clearly be seen that KCPSM6 has taken 18.6425µs to compute the 'set_baud'rate' value which it has output to the BAUD rate generator circuit so that it can generate 'en_16_x_baud' pulses at the correct rate.




Questions – Why does 's0' sometimes contain unknown values (i.e. 'X')? Why is this valid and expected behaviour in this case?