DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 391 – Computer Systems Design Studio
Fall 2019/2020 Term 2

Tutorial 1.8
Adding a Custom Hardware Component to your ARM System

Altera already provides QSYS components for most of the I/O devices on the DE1 board. However, there are times you might want to design your own hardware component and integrate it into a QSYS system. There are at least two reasons you might do this:

1. If you are interfacing ARM to a new hardware component, you may want to create your own interface circuit in hardware.
2. You may wish to create a hardware component to accelerate a computational task, e.g. encryption/decryption, or to simply **offload** some work from the processor to dedicated hardware so it can do other work in parallel.

In this tutorial we will look at an approach to creating a new hardware "slave" component that can perform a very simple computational task but do it considerably faster than the ARM processor could carry it out in software, i.e. accelerate software. Our new slave component will interface to the ARM processor via Altera's **Avalon Memory-Mapped Interface**.

This interface permits the ARM processor to exchange data with the slave via a mixture of read and write operations. In fact the interface is configured to look like memory to the ARM processor, so we can write software to use pointers to talk to the component (*just like we did in software for the LCD, LEDs and Switches*).

References

If you want more information, than is provided in this tutorial, a more verbose tutorial can be found here:

ftp://ftp.altera.com/up/pub/Altera_Material/12.0/Tutorials/making_qsys_components.pdf

Note that the **Avalon Memory-Mapped Interface**, while being the most common interface, is just one of a number of different interfaces we could make use of for our slave devices, depending upon their capabilities and performance. You can read about these other interface here:

http://www.altera.com/literature/manual/mnl_avalon_spec.pdf

Our Custom Component Specification

Our new H/W component will provide the following functions:

1. A **write** operation to **store** a 32-bit value
2. A **write** operation that will **increment** the stored value by 1.
3. A **read** operation to return the **original stored value**.
4. A **read** operation to return the saved value with the **bits reversed**.
5. A **read** operation to return the **complement** of the original value.

Note: This is a relatively simple problem that is meant to serve only as an example or template for some more complex hardware that you might want to add to your system.

Memory Map

By now, you should be familiar with memory/register maps from the specification documents of other Qsys components and you 2nd year course CPEN 211 and project course CPEN 311.

In implementing the above functionality (operations 1-5 above). We arbitrarily assign the functions to the register map shown below. That is our circuit will interpret read and write operations to these addresses as "actions" that it should carry out. Some of those actions will involve the input or output of data.

Given the 5 operations above, our register map for our component interface will consist of three 32-bit locations i.e. an address range from 0x0 to 0x0C (i.e. 12 decimal locations). From this we will design the HDL circuit to provide this the required behavior when the processor reads or writes to those addresses.

Offset in bytes	**Register Name**	**Read/Write**	**31..0**
0	Flipped	R/W	WRITE – Writing a value to this address saves the value. READ – Reading from this address returns the saved value in reverse bit order.
4	Un-flipped	R/W	WRITE – Writing to this address increments the saved value READ – Reading from this address returns the saved value.
8	Complement	R	READ – Reading from this address returns the complement of the saved value.

Creating our HDL Circuit

In order to create a circuit that will respond to *reads* and *writes* from the processor, the top-level ports of our component need to conform to the **Avalon Memory-Mapped Interface standard** which includes the signals in the table below.

At a bare minimum, these signal describe an interface with a 32 bit read/write data bus and other signals such as a **clock**, **reset** and signals to tell our component if the CPU is ***reading*** *from it* or ***writing*** *to it*.

Because our component occupies 3 x 32 bit locations in memory (or 12 bytes), we'll round this up to a power of 2 and assume our bit flipper component occupies 4 locations (16 bytes) in the memory map of the system. Thus our component needs a 2 bit address bus (to select 1 from 4 locations), separate read and write enable signals, a 32 bit data bus in and out, as well as a clock and a reset.

Signal	In/Out	Width	Description
Clock	In	1	The clock that read/write operations are aligned to.
Reset	In	1	The reset signal.
Address	In	2	The address that the processor is requesting for the read/write. Since we have three locations, the valid values are (0, 1, 2), and it only requires 2 bits.
ReadEnable	In	1	The read enable signal. When asserted, it indicates that the processor is performing a read.
WriteEnable	In	1	The write enable signal. When asserted, it indicates that the processor is performing a write.
ReadData	Out	32	The data we provide back to the processor during a read operation.
WriteData	In	32	The data the processor provides during a write operation.

Table 1: Avalon Memory-Mapped Interface Standard Signals

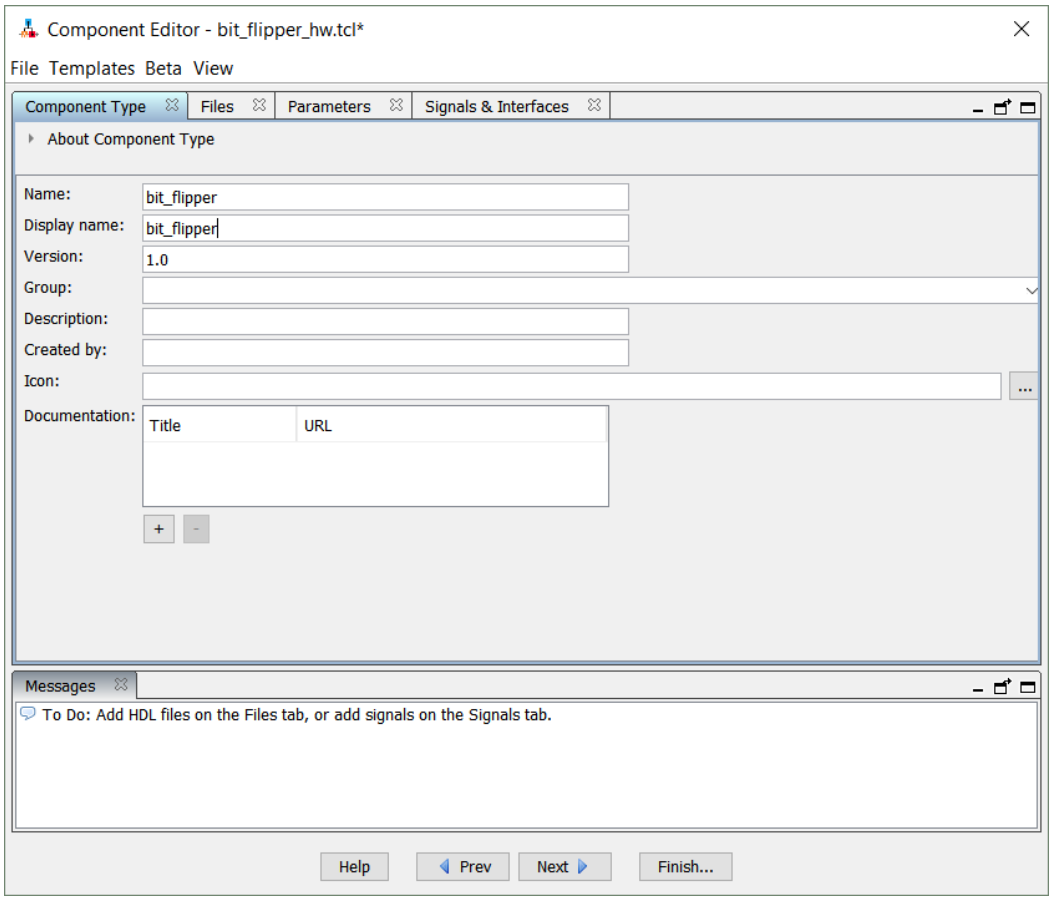
Note: The names we choose for these signals don't matter. Likewise, the polarity of the reset and enable signals don't matter. We could design our component with **active-high** or **active-low** reset, as well as active high or low read/write enable signals. At a later point we indicate the *function* of each of our top-level signal names, including whether they are active-low or active-high.

This interface is known as a **Memory Mapped Slave**. It is a slave because it just sits there and waits for read/write requests from a master (the processor). A master has the ability to initiate read/write requests. If you are interested in creating a Master, check out the Avalon document for more info.

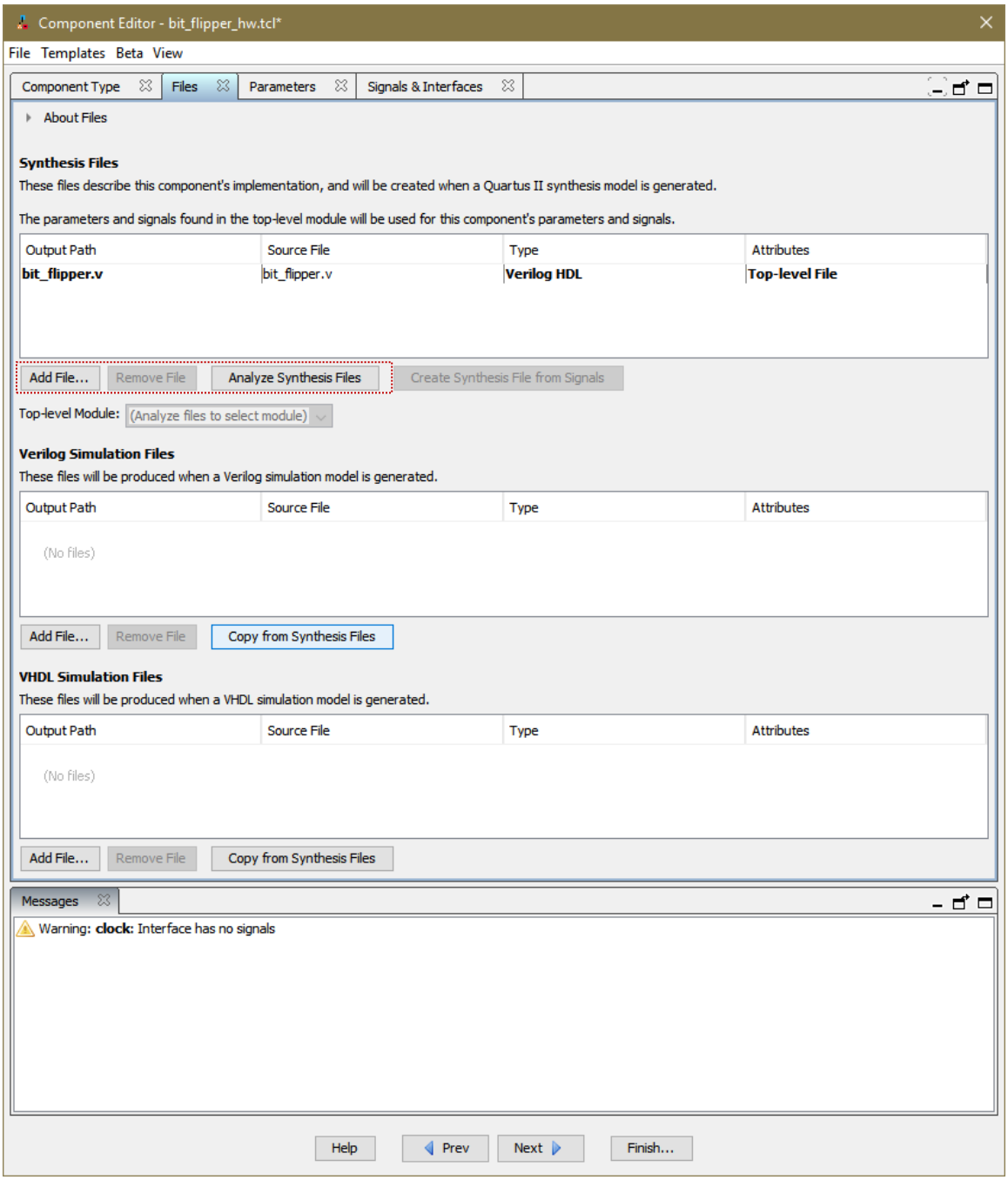
The full **Verilog** and **VHDL** for our circuit are provided in **Appendix A**. Make sure you look it over and understand its behaviour.

Creating the Qsys Component

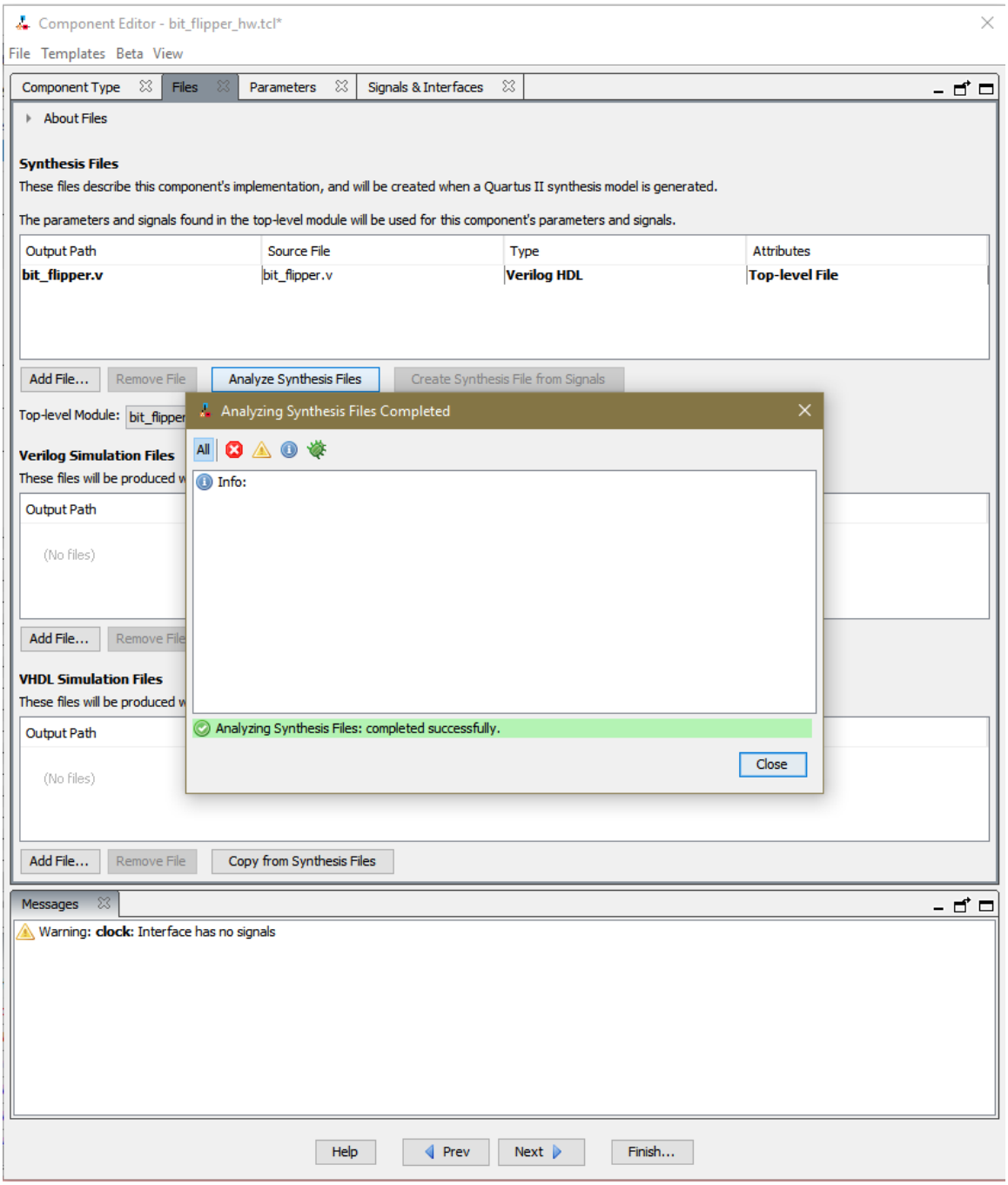
1. Start with your QSYS design from your previous tutorial
2. Copy the **bitflipper** code (in VHDL or Verilog) from **Appendix A** or from **Canvas** into your working project directory, and **add** it to your Quartus II project.
3. Start **Qsys** and open our current project.
4. In the '**Component Library**', click '**New component**'
5. Go into the '**Component Type**' tab.
6. Give the new component the 'name' and 'display name' '**bit_flipper**' to match the name of the module/entity in the Verilog or VHDL file.



7. Go to the '**Files**' tab.
8. Add the **Verilog or VHDL version** of the bit flipper code to the list of Synthesis Files. In the example below I've used the **Verilog** version, but feel free to use the **VHDL** version if you wish.



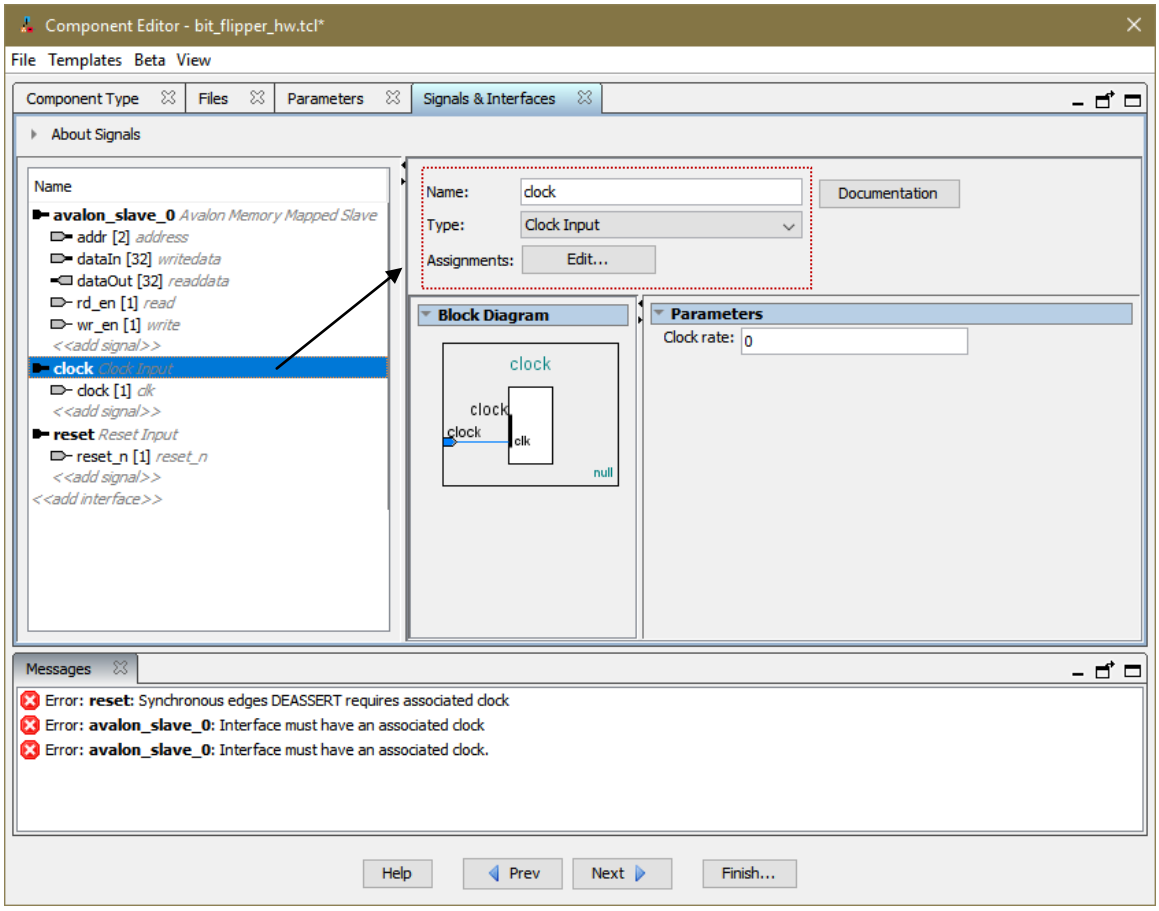
9. Click the '**Analyze Synthesis Files**' button above. Qsys will examine the fill to figure out what its *inputs* and *outputs* are. It will then synthesise a circuit from the HDL.



10. **Note** there will be errors at the bottom of the Component Editor, but we will deal with those in the next steps.
11. Go to the '**Signals and Interfaces**' tab.
12. Configure the list according to the following table. This is where you indicate the *function* of each of your custom component's top-level signals (*including their polarity – i.e. whether they are input or output signals, active high or active low etc.*).

Note that signals for the component are labeled from the perspective of the component. That is, a signal identified as an "*input*" is an "*input*" to the new component etc. Set up the interface signals as shown below. You may have to *remove* or *rename* signals to match those in your **bit_flipper** file.

Also some times Qsys is not very good at removing the error messages when in fact the cause of the error has been cured, so if in doubt save/quit Qsys and restart it if you think the error message is incorrect.



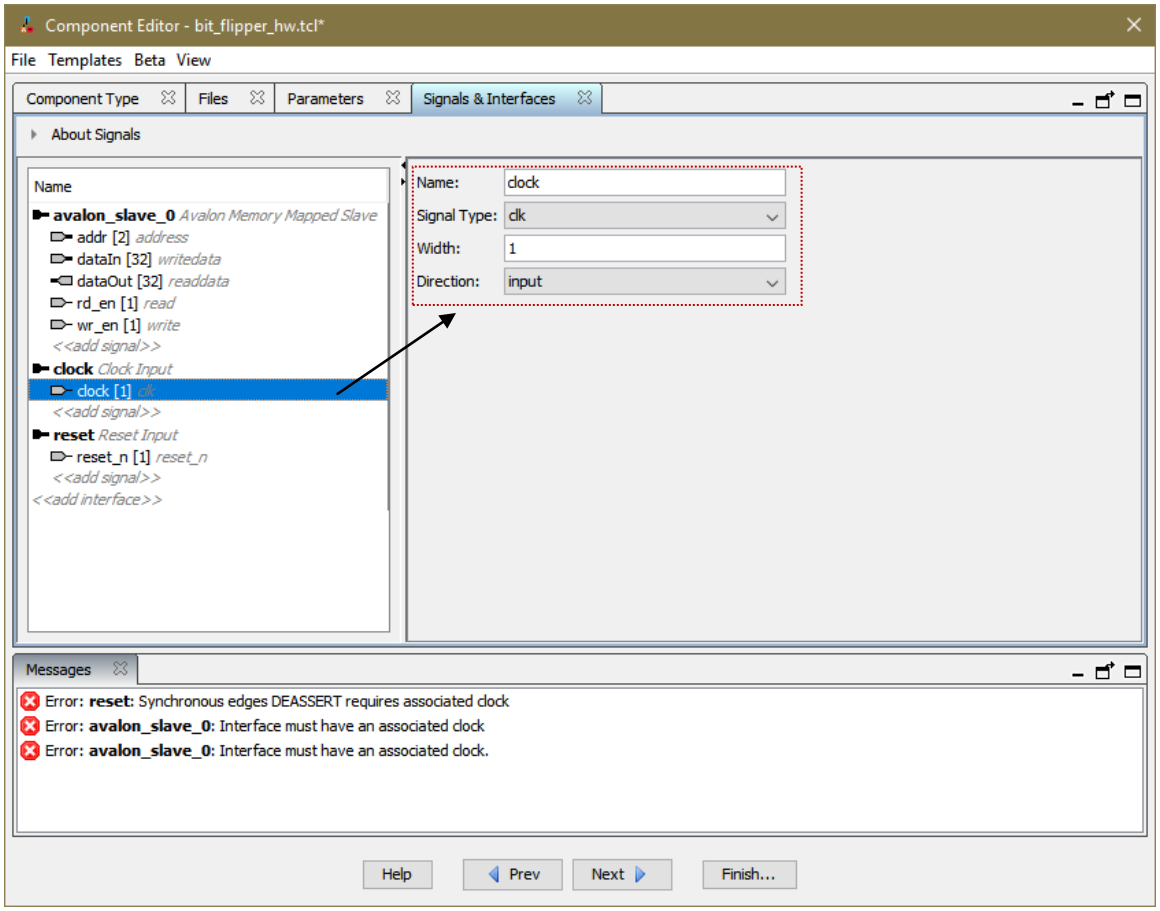


Fig 1.0 – above, ***clock*** *is a signal of type* ***Clk (or clock) input*** *to the component*

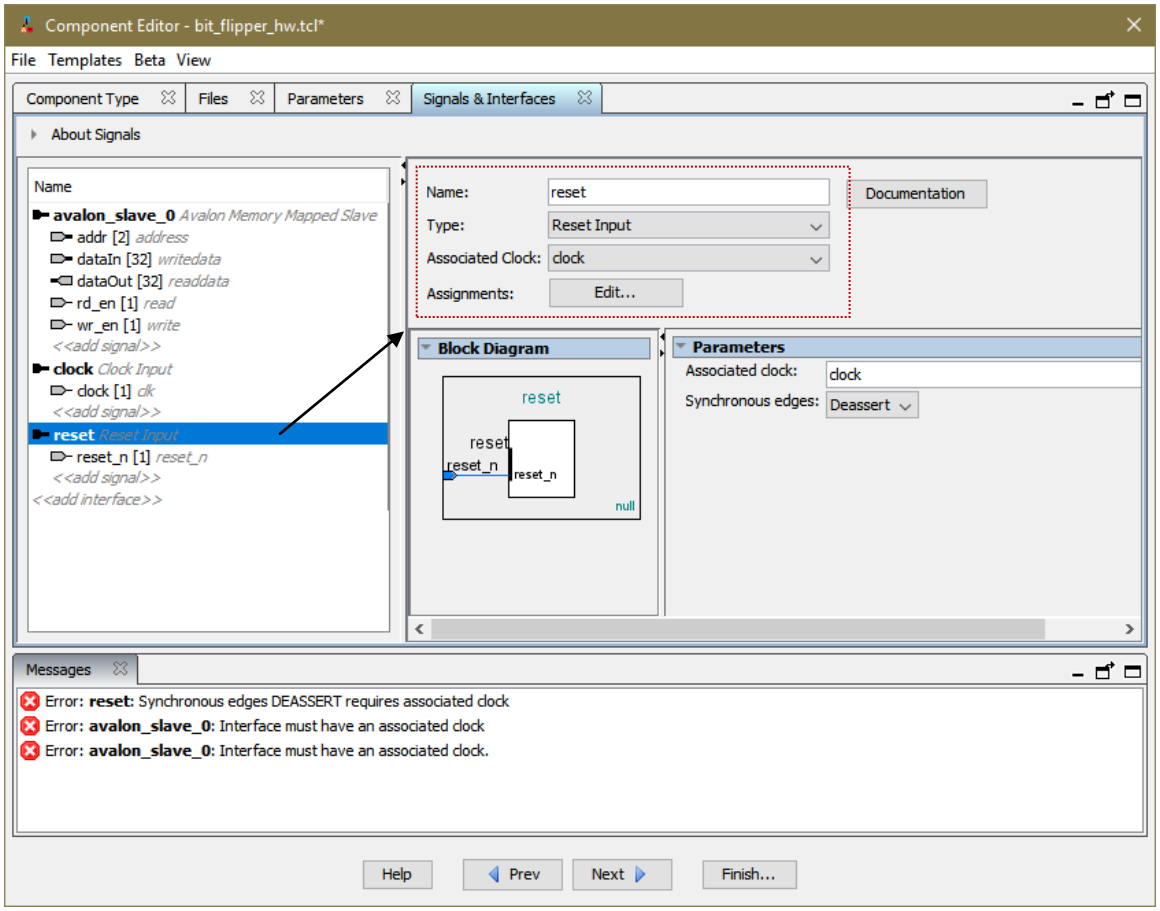


*Fig 2.0 – above, **reset** is an active low signal of type **Reset_n** and an **input** to the component.*

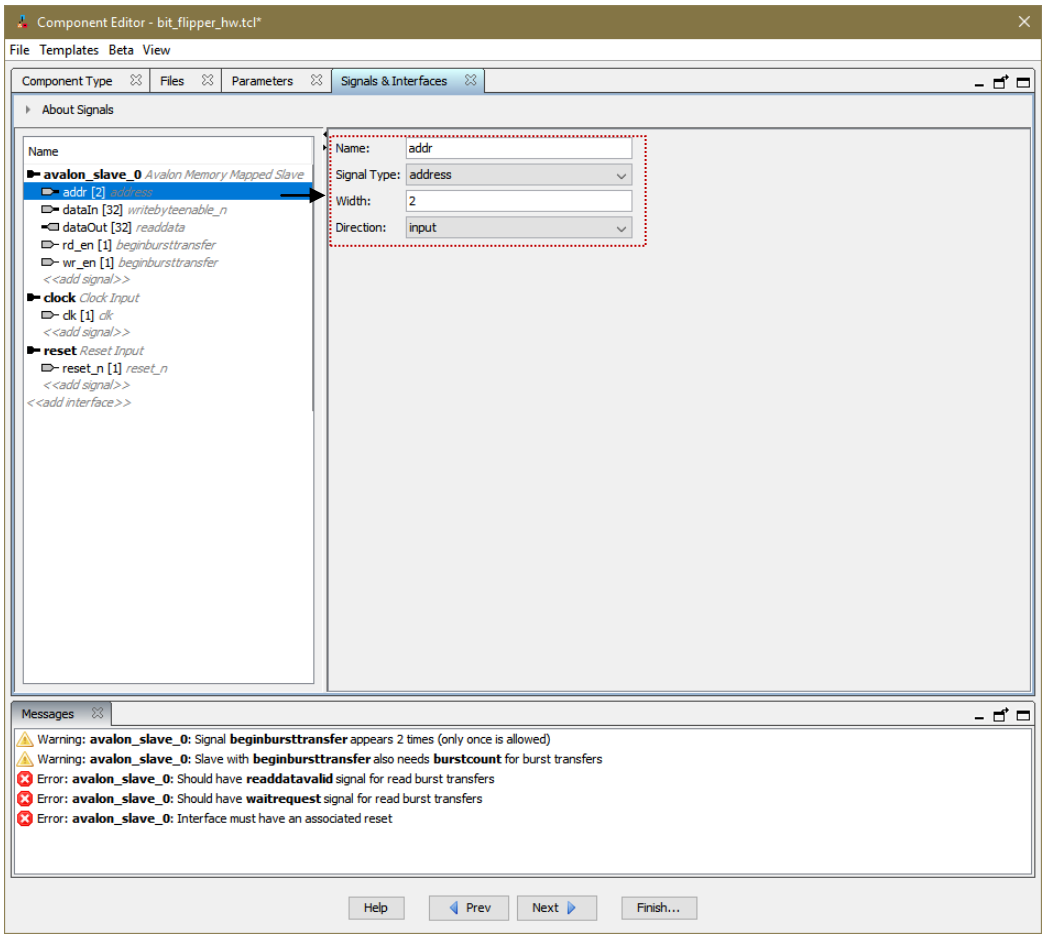


*Fig 3.0 – above, **addr** is a 2 bit **Address input** signal to the component*

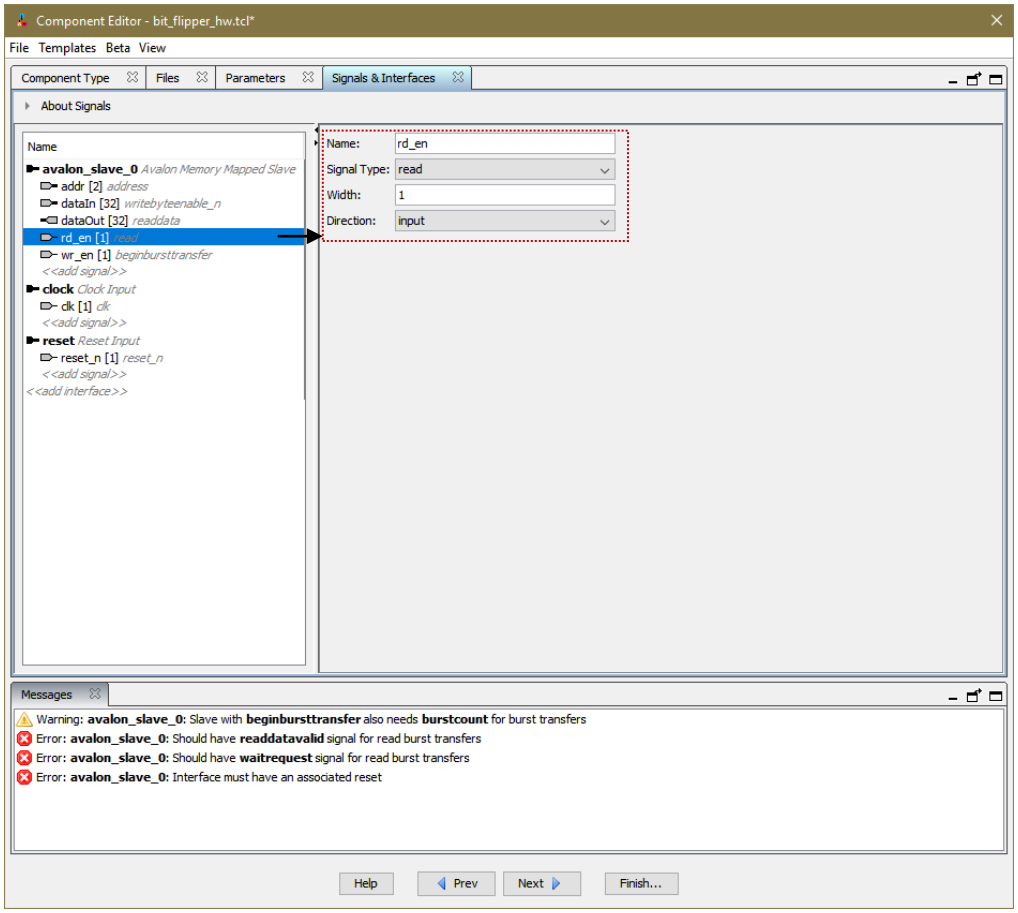


*Fig 4.0 – above, **rd_en** is a **read** input signal to the component*

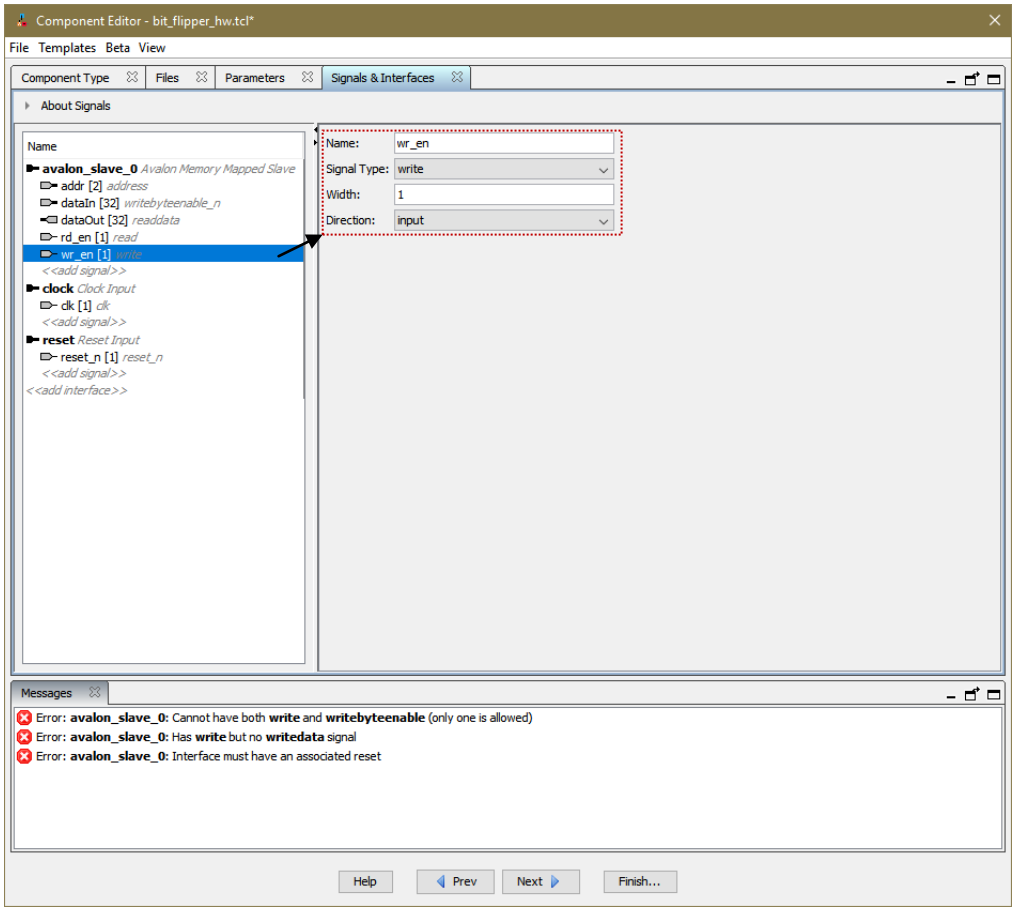


*Fig 5.0 – above, **wr_en** is a **write input** signal to the component*

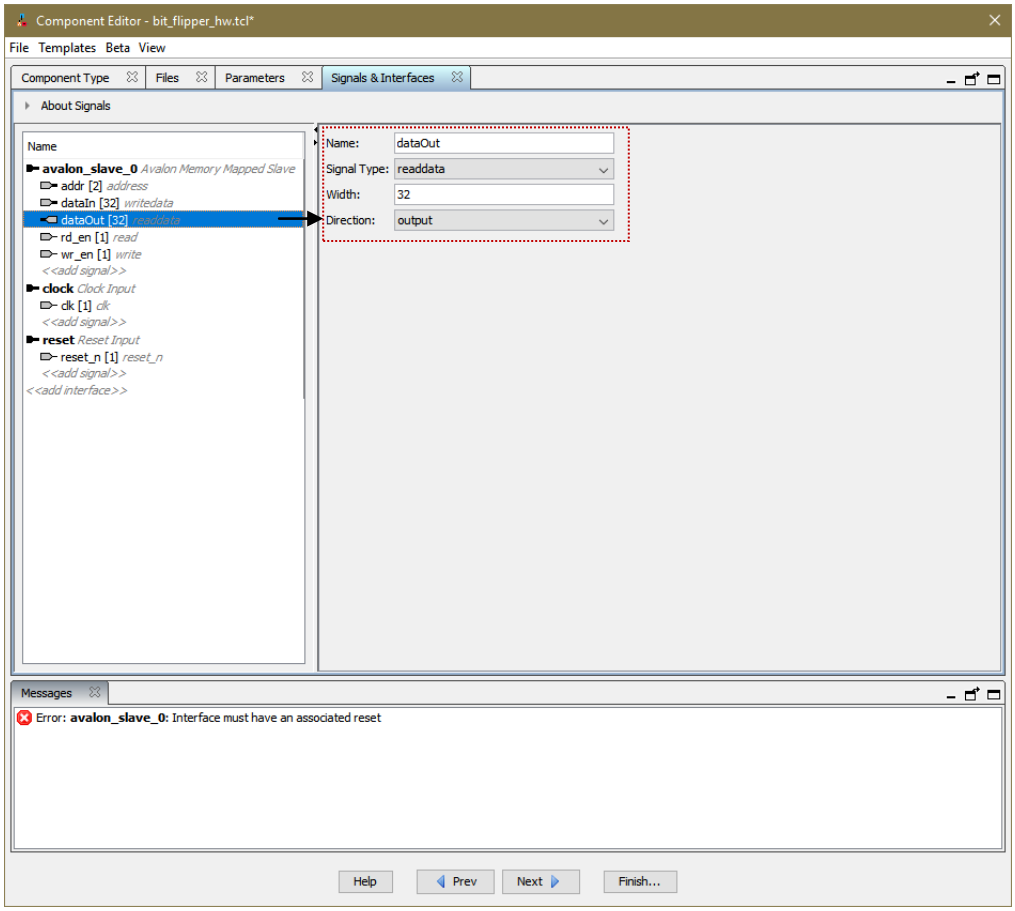


*Fig 6.0 – above, **dataOut** is a 32 bit read data output signal (i.e. one that can be read by the CPU) from our component.*

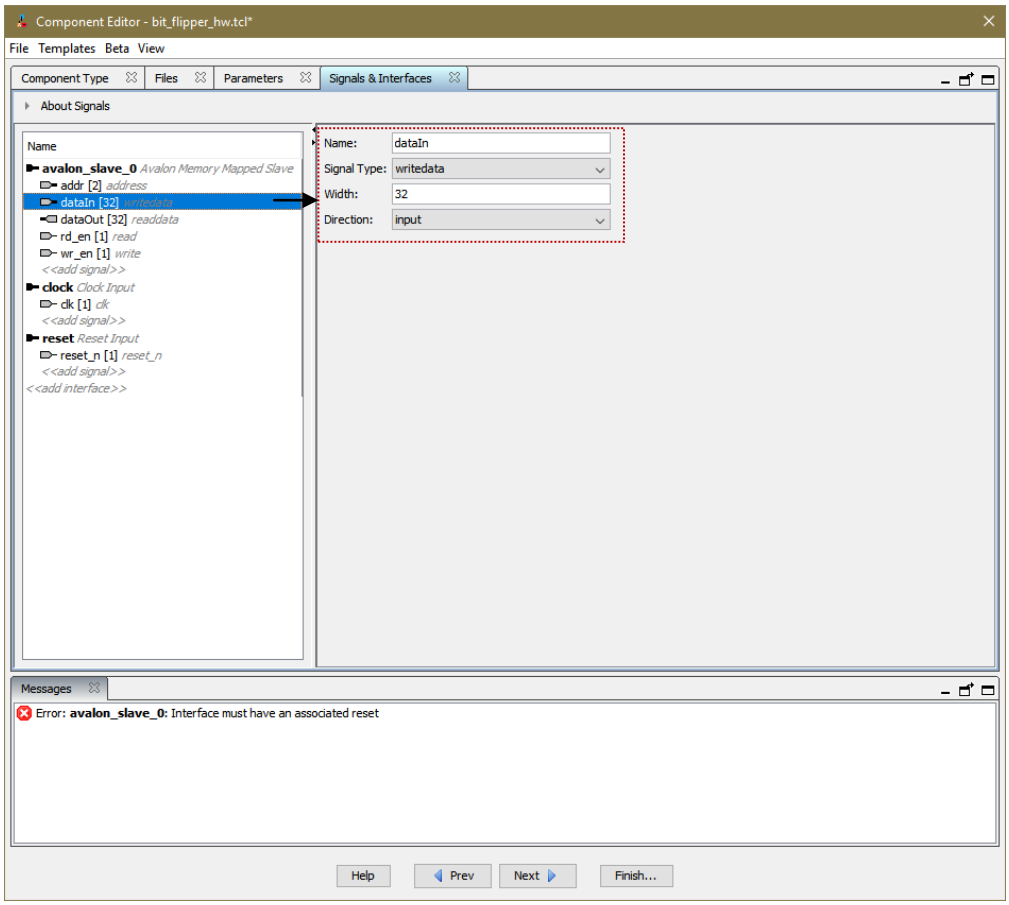


*Fig 7.0 – above, **dataIn** is a 32 bit written data <u>input</u> signal (i.e. one that can be written to by the CPU) to our component.*

Settings for the Bit Flipper Component

Finally fill in **all** the timing related info and **other options** (Avalon Memory Mapped Slave), address units = **8 bit words** as shown below.

Setup and hold timing (expressed as multiples of clock periods) for both the data and address signals relative to the rising edge of clock for read and write operations can be defined here, as well as the number of clocks required per read/write operation.

That is, we can introduce delays or wait states, if our component is either slow or requires multiple clocks to compute its answer. You will see the read/write timing diagrams appear below to reflect the parameters we define here.

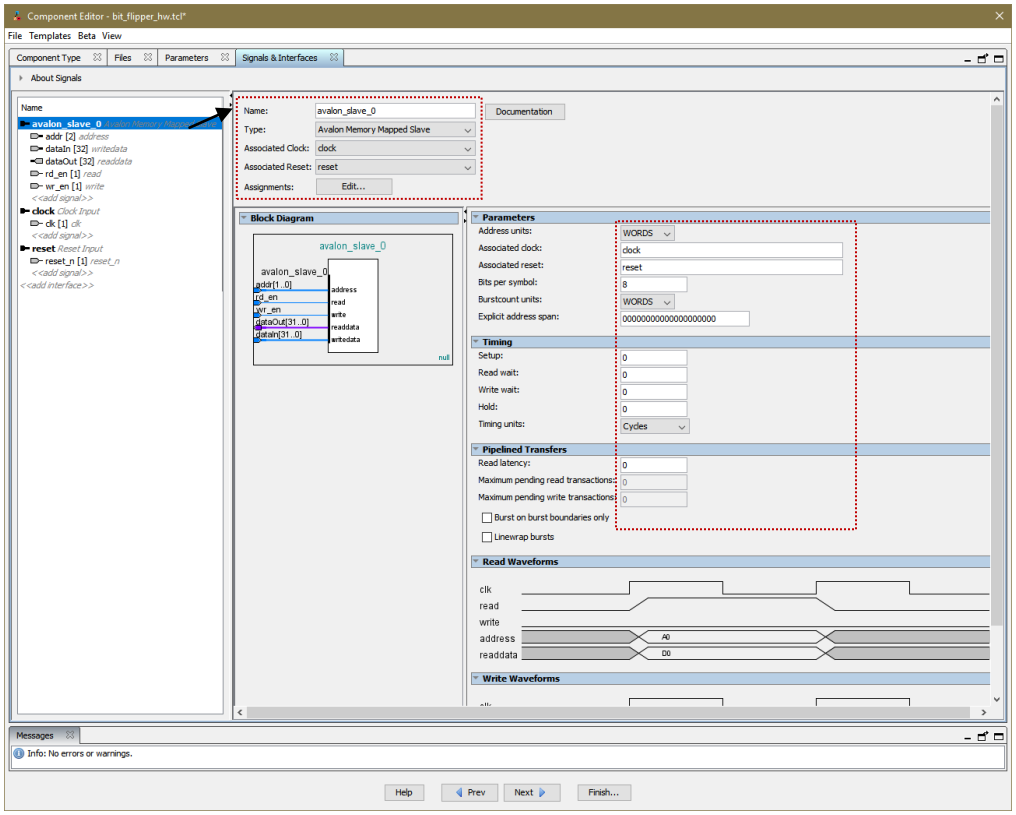


Fig 8.0 – Other Settings and Timing Waveforms

13. Notice that we changed the 'Read wait' to 0 cycles, because our circuit provides the read data in the same cycle as the processor requests it.
14. At this point you could check the 'Read Waveforms' and 'Write Waveforms' and check that the behavior is what your circuit is expecting or requires.
15. Click '**Finish**…' and then '**Yes, Save**…'.

Instantiating the Bit Flipper with Qsys

Double click the ***bit flipper*** circuit in Qsys (top left corner) and wire up as shown below just like we did with the other peripherals. If you cannot see where to connect the Clock for our component, make sure you expand the **System_PLL** component at the top.

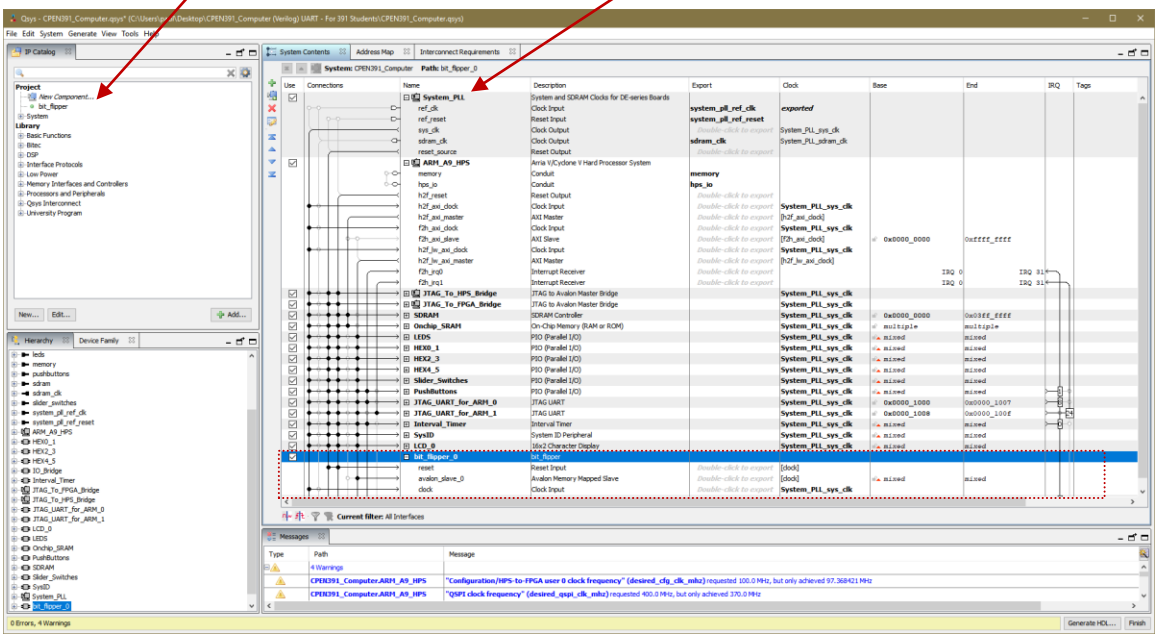


Fig 9.0 – Adding the Bit Flipper to our circuit

Setting the Address of the Component in the CPU Memory Space

Click on the **Address Map** tab and set an **address** for the bit flipper which does not conflict with any other peripherals and components. I used address **0x0000_2040** and then set the address in the **JTag_To_FPGA_Bridge master column** as **0xFF20_2040** (see below). This last address is the important one we will use in our software to talk to our new component.

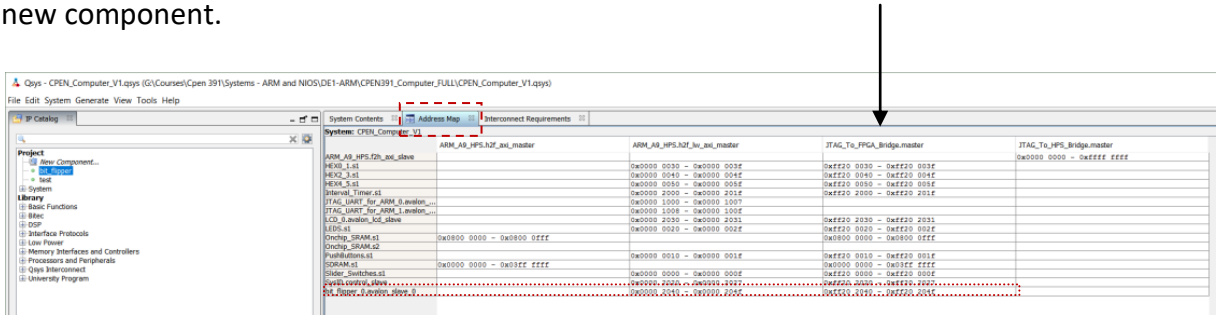


Fig 10 – setting the address of the Bit flipper in CPU address space.

After these steps, your bit flipper component should be listed in your Qsys component library. All of the settings for the component are stored in a file named **bit_flipper_hw.tcl**.

If you want to use this in other Qsys projects, just copy the *_hw.tcl file to the new project directory. Likewise you can delete it to remove the component from the library

Save and **Generate** your QSYS design, ensuring there are **no errors** and then **compile the Quartus Project** and download to the DE board.

Testing your Component

You can now write a simple C program to allow ARM to write and read the bit flipper registers. As an example, consider the following C code:

```
#include <stdio.h>


#define  bit_flipper_base          (volatile int *) 0xFF202040

int main()
{
  int v = 0x05;                                          // arbitrary value

  *bit_flipper_base = v;                                 // write the value to component
  printf("value is now %x\n", *(bit_flipper_base+1));

  // increment what is in the component
  *(bit_flipper_base+1) = 0;                             // does not matter what you write
  printf("value is now %x\n", *(bit_flipper_base+1));

  // increment it again
  *(bit_flipper_base+1) = 0;                             // does not matter what you write
  printf("value is now %x\n", *(bit_flipper_base+1));

  // get the value in reverse bit order
  printf("reverse bit order %x\n", *(bit_flipper_base));

  // get the complement of the value
  printf("complement is %x\n", *(bit_flipper_base+2));

  return(0);
}
```

In this case, 0xFF202040 is the base address of my **bitflipper** component (defined in QSYS - as far as ARM Cpu is concerned). Be sure to change this to match **the** base address you set in QSYS. If you run this program, you should see something like this:

```
value is now 5
value is now 6
value is now 7
reverse bit order e0000000
complement is fffffff8
```

As an aside, if you are observant, you will notice that a write to ***(bit_flipper_base+1)** writes to location **0xFF202044** (which, from the above, is the second word in the memory map of this component). This might be confusing.

The value of bit_flipper_base is **0xFF202040**, so by adding 1, you might expect to get the result 0xFF202041. But in this case, the addition gives the sum **0xFF202044**.

Think about why this might be so, and discuss with your TA if you can't figure it out (**hint**: think of the "type" of bit_flipper_base: is it an integer? Is it something else?). By the way, this is a great/common interview question for a C or C++ job.

If you reach this point, congratulations. You have just created an embedded system with a processor, some custom hardware, and embedded software all working together.

Appendix A: The Bit Flipper VHDL code (Verilog code is beneath)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity bit_flipper is
port (
   clk: in std_logic;
   reset_n: in std_logic;
   addr: in std_logic_vector(1 downto 0);
   rd_en: in std_logic;
   wr_en: in std_logic;
   readdata: out std_logic_vector(31 downto 0);
   writedata: in std_logic_vector(31 downto 0)
);
end bit_flipper;

architecture rtl of bit_flipper is
   signal saved_value: std_logic_vector(31 downto 0);        -- temp reg for result
begin

   -- writing
   process (clk)
```

```
   begin
      if rising_edge(clk) then
         if (reset_n = '0') then                              -- synchronous reset
            saved_value <= x"00000000";                       -- clear result to hex 0 on reset
         elsif (wr_en = '1' and addr = "00") then             -- if write to address 0
            saved_value <= writedata;                         -- save the written data
         elsif (wr_en = '1' and addr = "01") then             --  if write to address 1, increment
            saved_value <= std_logic_vector(unsigned(saved_value) + 1);
         end if;
      end if;
   end process;

   -- reading
   process (rd_en, addr, saved_value)
   begin
      readdata <= (others => '-');
      if (rd_en = '1') then                                   -- if reading
         if (addr = "00") then                                -- from address 0
            for i in 0 to 31 loop                             -- reverse bits
               readdata(i) <= saved_value(31-i);
            end loop;
         elsif (addr = "01") then                  -- if reading from address 1
            readdata <= saved_value;                          -- return saved value
         elsif (addr = "10") then                  -- if reading from address 2
            readdata <= not saved_value;                      -- return inverse of all bits
         end if;
      end if;
   end process;
end rtl;
```

The Bit Flipper VERILOG code

```
module bit_flipper (
   input clk,
   input reset_n,
   input unsigned [1:0] addr,
   input rd_en,
   input wr_en,
   output reg unsigned [31:0] readdata,
   input unsigned [31:0] writedata
);
   reg unsigned [31:0] saved_value;             // temp reg for result
          integer i;
```

```
  // writing
  always@(posedge clk) begin
    if (reset_n == 0)                          // synchronous reset
      saved_value <= 32'b0;            // clear result on reset
    else if (wr_en == 1 && addr == 2'b00)    // if write to address 0
      saved_value <= writedata;                // store result
    else if (wr_en == 1 && addr == 2'b01)    // if write to address 1
      saved_value <= saved_value + 1;        // increment result
  end

       // reading
  always@(*)  begin
    readdata <= 32'b0;                         // default value is 0
    if (rd_en == 1) begin
      if (addr == 2'b00) begin                 // if reading from address 0
        for(i = 0; i < 32; i = i + 1)
          readdata[i] <= saved_value[31-i];          // flip bit order
       end

      else if (addr == 2'b01)            // if reading from address 1
        readdata <= saved_value;               // give back result
      else if (addr == 2'b10)            // if reading from address 2
        readdata <= ~saved_value;  // give back inverse all bits of result
    end
  end
endmodule
```