DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 391 – Computer Systems Design Studio
Fall 2015/2016 Term 2

Tutorial 1.8
Adding a Custom Hardware Component to your NIOS System

In Tutorial 1.6, you added a pre-designed QSYS component to communicate with the 16x2 LCD Display. Altera already provides QSYS components for most of the I/O devices on the DE-2 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 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 perform a computational task, perhaps because a custom circuit can do it faster than a processor (such as encryption/decryption), or to simply offload some work from the processor so it can do other work in parallel.

In this tutorial we will create a component that can perform a very simple computational task. Our component will have an Avalon Memory-Mapped Interface, which allows the NIOS processor to communicate with our circuit through memory reads and writes.

References

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

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

The full specification of the Avalon interface can be found here:

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

Our device specification

Our device will provide the following functions:

1. You can write to the device to save a 32-bit value
2. You can read from the device to return the saved value with the bits in reverse order.
3. You can read from the device to return the original saved value.
4. You can write to the device to increment the saved value by 1.

5. You can read from the device to return the complement of the original value.

Note: This probably isn't something worth implementing as a separate circuit in practice, as it could quickly be done by the processor. It is just meant as an example.

Memory Map

By now, you should be familiar with memory/register maps from the specification documents of other Qsys components. We have arbitrarily assigned the above functions to the registers map shown below. Our register map consists of three 32-bit registers. Next we will design the HDL circuit that will provide this behavior.

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	Unflipped	R/W	WRITE – Writing to this address increments the saved value by 1. 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 circuit need to conform to the Avalon Memory-Mapped Interface standard. At bare minimum we need the top-level signals described in the following table.

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 registers, the valid values are (0, 1, 2), and it only requires 2 bits.
Read Enable	In	1	The read enable signal. When asserted, it indicates that the processor is requesting a read.
Write Enable	In	1	The write enable signal.
Read Data	Out	32	The data we provide back to the processor during a read operation.
Write Data	In	32	The data the processor provides for a write operation.

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 HDL with active-high or

active-low reset, and read/write enable signals. At a later point we indicate the function of each of our top-level ports, 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.

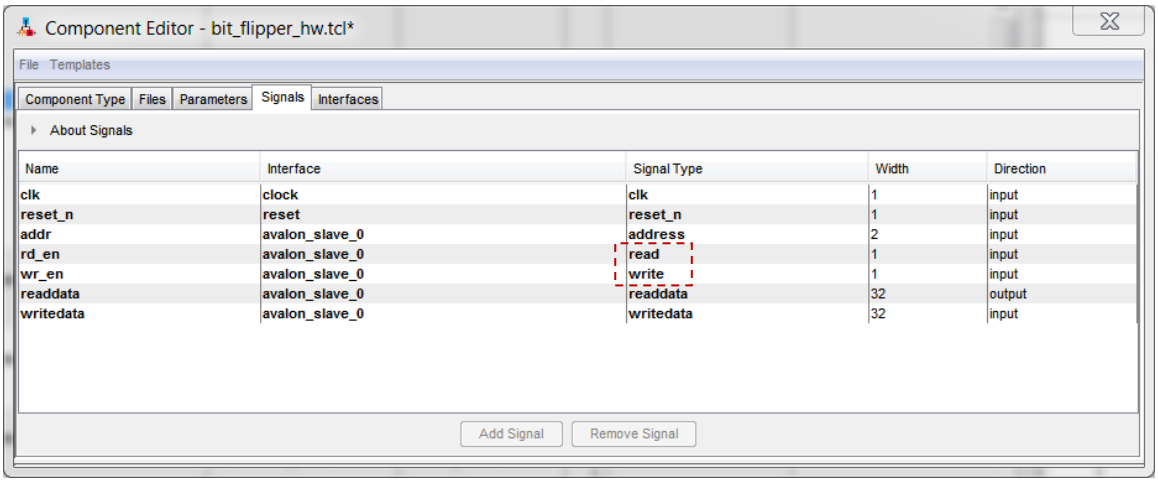
The full VHDL for our circuit is provided in Appendix A. Make sure you look it over and understand its behaviour.

Creating the Qsys Component

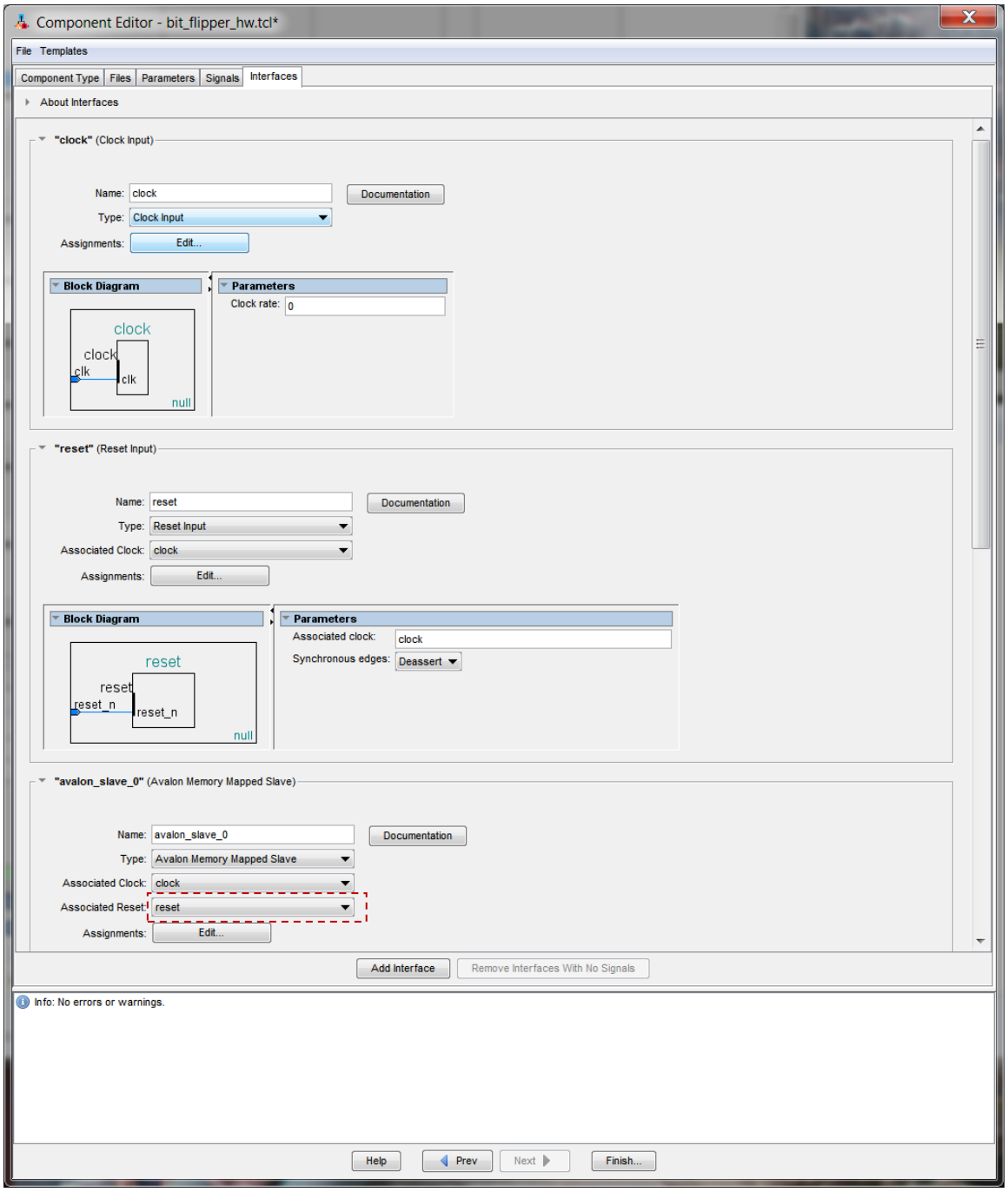
1. Start with your QSYS design from the previous tutorial, or create a new one (make sure you at least include the SDRAM as in Tutorial 1.3).
2. Copy the bitflipper.vhd from Appendix A or on-line into your working directory, and add it to your Quartus II project.
3. Open the Qsys project.
4. In the 'Component Library', click 'New component'
5. Go into the 'Component Type' tab.
6. For 'name' and 'display name' enter 'bit_flipper'
7. Go to the 'Files' tab.
8. Add the bit_flipper.vhd file to the list of Synthesis Files (use the '+' button)
9. Click 'Analyze Synthesis Files'. This should complete with the message "Analyzing Synthesis Files Completed. 0 Errors, 0 Warnings". If there are errors, you probably have problems in your HDL. (There will be errors at the bottom of the Component Editor, but we will deal with those in the next steps.) If necessary, you can create a new Quartus project and synthesize your HDL to see where the errors are.
10. Go to the 'Signals' tab.
11. Configure the list according to the following table. This is where you indicate the function of each of your top-level ports (including their *polarity*). You will find that much of this is preset for you, but you may have to change some (in particular, check the "signal_type" column for each entry).

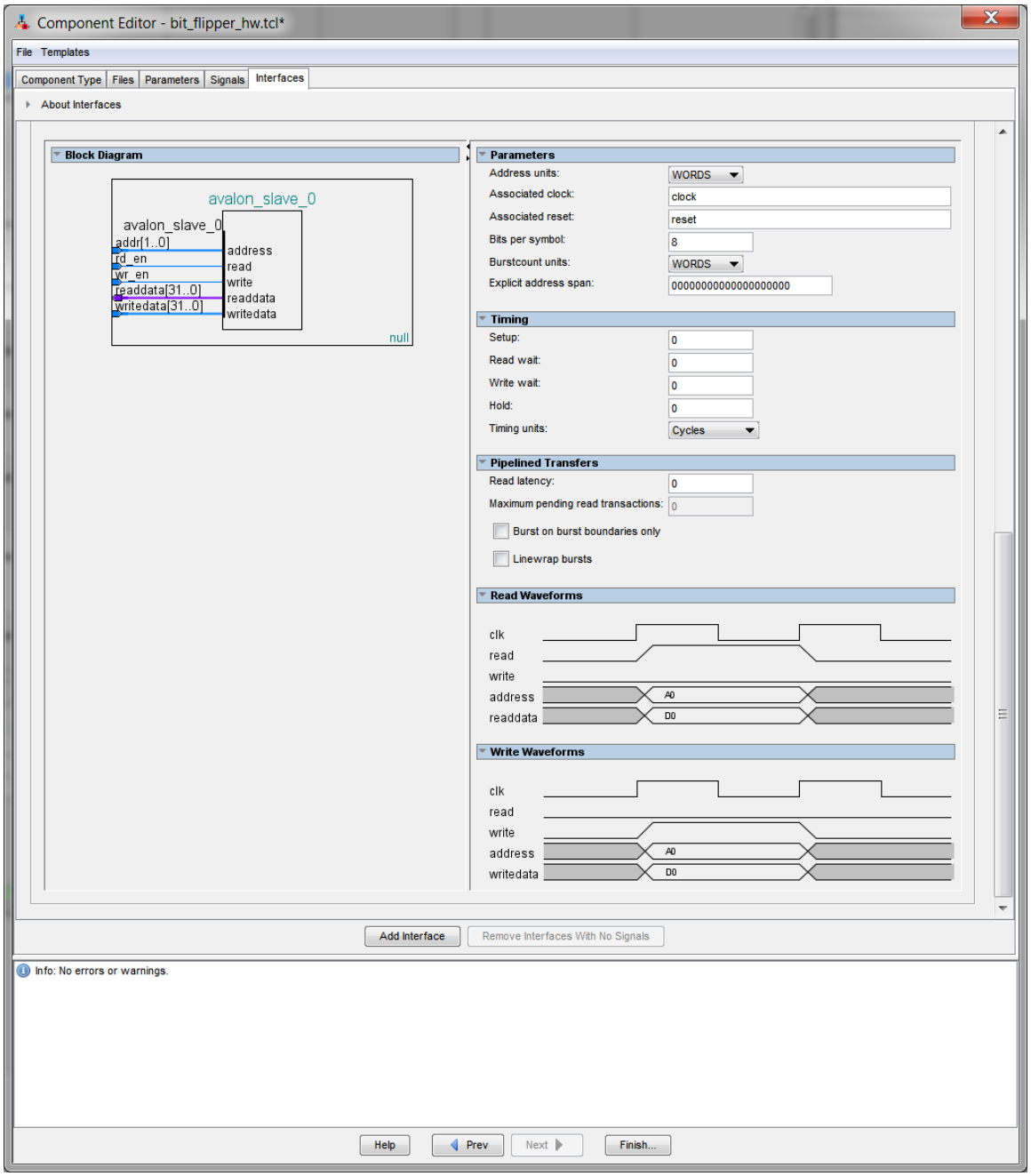
    The interface column is a drop-down list from the available interfaces. The Component Editor will probably create the interfaces for you, but if necessary, you would choose 'New Clock Input…', 'New Reset Input…', or

'New Avalon Memory-Mapped Slave'. The interfaces can be renamed in the 'Interfaces' tab. Use the names below in particular read and write



12. Go to the 'Interfaces' tab.
13. If not greyed out, click 'Remove Interfaces with No Signals'. After this, you should have only your three interfaces: clock, reset and avalon_slave_0.
14. Configure your interfaces as shown in the diagrams on the next pages.
15. Notice that we changed the 'Read wait' to 0 cycles, because our circuit provides the read data in the same cycle as the request.
16. Look at the 'Read Waveforms' and 'Write Waveforms' and check that the behavior is what your circuit is expecting.
17. Click 'Finish…' and then 'Yes, Save…'.


Component Editor - bit_flipper_hw.tcl*
File Templates
Component Type Files Parameters Signals Interfaces
About Interfaces
"clock" (Clock Input)
Name: clock
Documentation
Type: Clock Input
Assignments: Edit...
Block Diagram
clock
clock
clk
clk
null
Parameters
Clock rate: 0
"reset" (Reset Input)
Name: reset
Documentation
Type: Reset Input
Associated Clock: clock
Assignments: Edit...
Block Diagram
reset
reset
reset_n
reset_n
null
Parameters
Associated clock: clock
Synchronous edges: Deassert
"avalon_slave_0" (Avalon Memory Mapped Slave)
Name: avalon_slave_0
Documentation
Type: Avalon Memory Mapped Slave
Associated Clock: clock
Associated Reset: reset
Assignments: Edit...
Add Interface
Remove Interfaces With No Signals
Info: No errors or warnings.
Help
Prev
Next
Finish...



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

18. Add the bit_flipper to your design and connect is as you have been connecting other components. Be sure to assign an unused base address (in my case, I used

0x2010) for your component. Generate your QSYS design, ensuring there are no errors.

19. Go back to Quartus II. Add the file nios_system/synthesis/submodules/bitflipper.vhd to your project.
Import your pin assignments, and recompile. Ensure there are no errors. Use the programmer to program your bitstream to the FPGA.

Testing your Component

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

```c
#include <stdio.h>
#include "io.h"

#define bit_flipper_base (volatile int *) 0x2010

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, 0x2010 is the base address of my bitflipper component (defined in QSYS). Be sure to change this to match your base address. If you run this program, you should see something like:

value is now 5
value is now 6

value is now 7
reverse bit order e0000000
complement is ffffffff8

As an aside, if you are observant, you will notice that a write to *(bit_flipper_base+1) writes to location 0x2014 (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 0x2010, so by adding 1, you might expect to get the result 0x2011. But in this case, the addition gives the sum 0x2014. 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.

Direct I/O:

There is one more issue that may arise if you are using the NIOS II/f variant of the processor (which most of you will end up using in your project). The differences between the variants will be explained in a later exercise, but for now, it is important to understand that the NIOS II/f contains a data cache. In previous years , many students spent many hours debugging during their projects because they didn't understand this, so be sure to read this carefully.

To understand the issue, consider this. Suppose you are reading location 0x0010000 from memory several times in your program (it is common that if you read a location from memory in your program, you are very likely to read that same location again… this is called *locality*). Memory accesses are relatively slow, so the first time you read the data, the *data cache* (which is a fast on-chip memory) stores the value so that the next time you read that memory location, it need not go to the slow memory to get it; the value can be obtained from the fast cache. If your program changes the value in 0x0010000, the cache entry is *invalidated*, meaning the next time you read that value, you will go back to main memory to get the new value (there are variants on this; sometimes the cache line itself is updated and the entry is not invalidated). For more information on caches, take EECE 476 next year.

However, now consider that the memory location is actually mapped to an input I/O port (such as switches or the I/O ports on this component). Consider the case of switches. Suppose your switches are mapped to location 0x2000 (as in the first tutorial). Suppose you initially read the switches and get a value of 0 (meaning all switches are off). Some time later, the user changes the value of the switches. The value that would be obtained by reading 0x2000 changes, however, the value in the data cache *does not change*. This is *not* a bug; this is how all data caches in virtually

all processors work. The problem occurs when you later read the value in 0x2000, and instead of getting the correct value, you get the value stored in the data cache (which is 0).

The problem is that reads from I/O ports must not be cached. All processors have a variant of a memory read instruction that bypasses the cache. This is typically called and I/O READ. There are also I/O WRITES which also bypass the cache.

To summarize the above: if you are using a processor with a data cache (the NIOS II/f), you must use an I/O READ instruction (rather than a direct memory access) to access data in an input I/O port. To do this in C, you would do something like the following:

```
#include <stdio.h>
#include "io.h"
void main()
{
    char my_char;

    while (1) {
        my_char = IORD_8DIRECT(0x0002000, 0);
        IOWR_8DIRECT(0x0002010, 0, my_char);
    }
}
```

In the above, **IORD_8DIRECT(address, offset)** reads an 8 bit quantity (a byte) from location **address**. Note that we have set the offset as 0, however if non-zero, it is added to the address before reading. There is also an IORD_16DIRECT and an IOREAD_32DIRECT to read 16 and 32 bit quantities respectively. Note that to use these functions, you must include "io.h".

In the above example, we use **IOWR_8DIRECT(address, offset, value)** to write **value** to **address**. This is not strictly necessary in our processor, since writes pass through the data cache anyway. However, it is good practice.

If you are using a processor without a data cache, you can use the direct memory addressing as in Tutorial 1.3. However, it is good practice to use the direct I/O version (just in case you change the processor later).

How does this apply to this tutorial? Ideally, the direct writes to and from memory should be done using the IORD and IOWR commands. This would lead to something like:

```
#include <stdio.h>
#include "io.h"

#define bit_flipper_base (volatile int *) 0x2010

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

   IOWR_32DIRECT(bit_flipper_base,0,v);  // write the value to the component
   printf("value is now %x\n", IORD_32DIRECT(bit_flipper_base,4));

   // increment what is in the component
   IOWR_32DIRECT(0x2010,4,0);   // does not matter what you write.  Note
offset is 4
   printf("value is now %x\n", IORD_32DIRECT(bit_flipper_base,4));

   // increment it again
   IOWR_32DIRECT(0x2010,4,0);   // does not matter what you write.  Note
offset is 4
   printf("value is now %x\n", IORD_32DIRECT(bit_flipper_base,4));

   // get the value in reverse bit order
   printf("reverse bit order %x\n", IORD_32DIRECT(bit_flipper_base,0));

   // get the complement of the value
   printf("complement sis %x\n", IORD_32DIRECT(bit_flipper_base,8));

  return(0);
}
```

Note that in the above, offsets are specified in terms of bytes (hence the “4” in the IORD_32DIRECT commands above).

APPENDIX A: bitflipper.vhd

```
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);
begin

    --saved_value
    process (clk)
    begin
        if rising_edge(clk) then
            if (reset_n = '0') then
                saved_value <= (others => '0');
            elsif (wr_en = '1' and addr = "00") then
                saved_value <= writedata;
            elsif (wr_en = '1' and addr = "01") then
                saved_value <= std_logic_vector(unsigned(saved_value) + 1);
            end if;
        end if;
    end process;

    --readdata
    process (rd_en, addr, saved_value)
    begin
        readdata <= (others => '-');
        if (rd_en = '1') then
            if (addr = "00") then
                -- bit-flip
                for i in 0 to 31 loop
                    readdata(i) <= saved_value(31-i);
                end loop;
            elsif (addr = "01") then
                readdata <= saved_value;
            elsif (addr = "10") then
                readdata <= not saved_value;
            end if;
        end if;
    end process;
end rtl;
```