Bubble Sort IP Core Specification

Author: avram ionut
avramionut@opencores.org

Rev. 0.1
March 28, 2014

This page has been intentionally left blank.

Revision History

Rev.	Date	Author	Description
0.1	28.03.14	Avram Ionut	*First release*
0.2			
0.3			

Contents

1. Introduction

This is an extreme useful and clear specifications document wannabe.

It describes the interface, internal structure and functionality of Probably the Best Sorting Module in the World. I am aware that there are people that may not agree to that so I decided to call it: Bubble Sort. This is in part due to the fact that it implements the bubble sorting algorithm.

The module is written in Verilog; no VHDL version planned.

Probably the Best Sorting Module in the World in intended to be independent of any bus so there is no code to ensure the data transfers. It is left to the user of the module to attach it to the preferred bus.

The internal structure is very simple. It resembles a chain and it is designed for fast clk.

Simple structure brings simple functionality: standard bubble sort algorithm is in place, no fancy optimizations, only limitations: the module is capable of sorting only unsigned words of N_BITS bits.

The code is released under LGPL as recommended by opencores.org.

2. Interface

The interface of the module is:

Signal	***Direction***	***Comments***
clk	*input*	*No comments*
rst	*input*	*Active high*
load_i	*input*	*Array. Active high. Control the load of new data inside the sorting module.*
writedata_i	*input*	*Array. Provides new data to the sorting module.*
start_i	*input*	*Active high. Triggers the sorting process.*
abort_i	*input*	*Not used.*
readdata_o	*output*	*Array. Expose sorted data outside the sorting module.*
done_o	*output*	*Active high. Indicates the end of sorting process.*
interrupt_o	*output*	*Active high. Indicates the validity of the output data.*

The user should make sure there are no additional start_i pulses from the start of sorting process until the end of sorting indicated by a pulse on interrupt_o.

The user is advised to not read readdata_o until the end of process indicated by a pulse on interrupt_o. Information on readdata_o wires is garbage during the sorting process as it reflects the content of registers during shifting process.

The user should consider the ending of sorting process indicated by a pulse of interrupt_o signal. The signal done_o is exposed to provide an early indication of the process end.

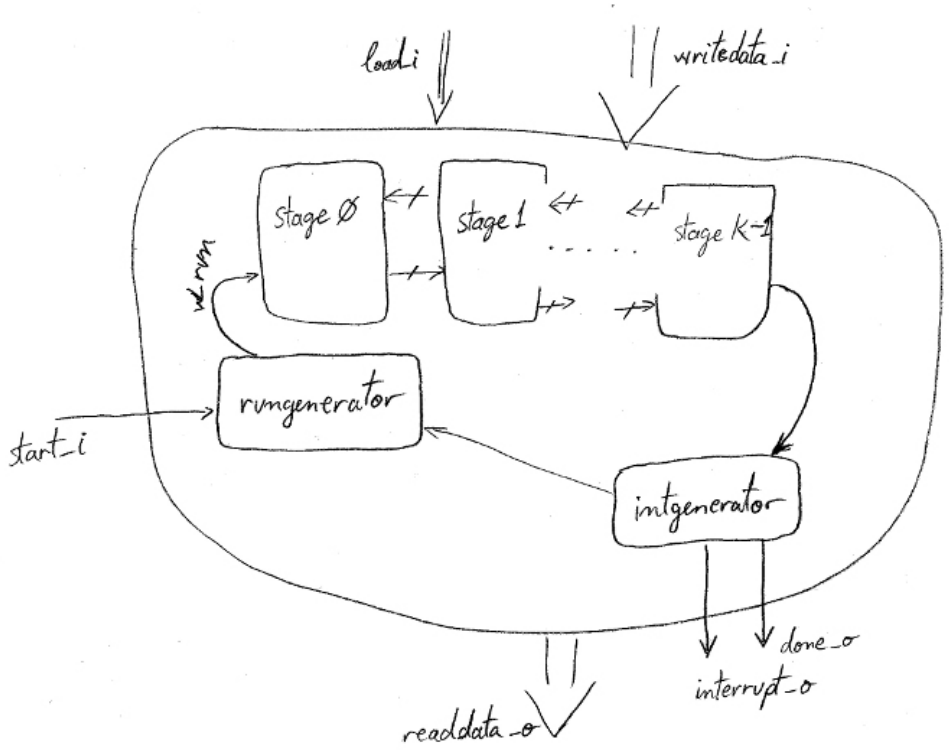
The user can tie the done_o signal to the interrupt controller and use it to launch the data reading procedure on a slow system or for systems that have significant latency.

Please see below an imaginary use case of the sorting module:

1. *bring valid data to the writedata_i;*
2. *load data with a positive pulse on load_i;*
3. *start sorting with a positive pulse on start_i;*
4. *watch done_o or interrupt_o for positive pulse;*
5. *retrieve data from readdata_o;*

3. Architecture

The architecture is a chain of simple modules alongside 2 control blocks. The number of modules is set by parameter K_NUMBERS.



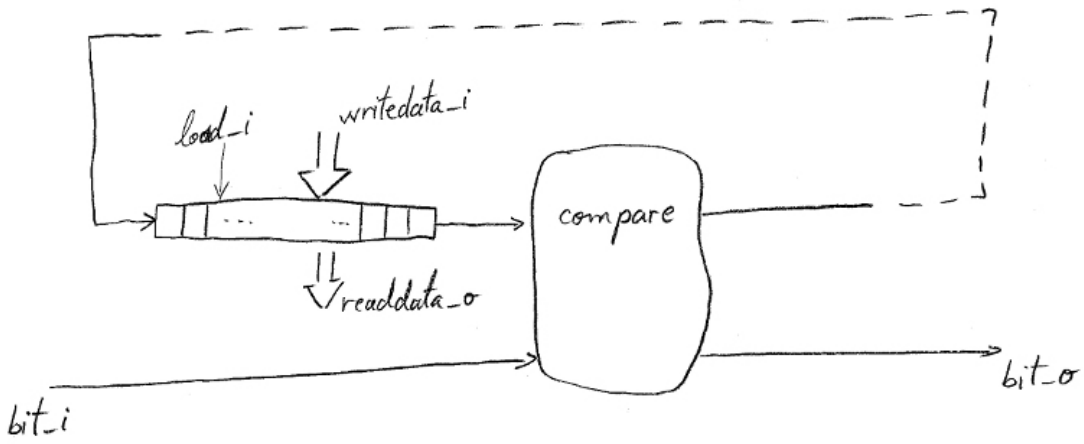
The control blocks are module intgenerator and module rungenerator.

Rungenerator is a shift register controlled by start_i and all_sorted_i signals.

Intgenerator is made from an edge select and a counter.

Stage modules are made from a shift register and a comparator. The size of the shift register is set by parameter N_BITS. The shift register has parallel load capabilities.

That's all; simple works best.



4. Operation

Here is described the normal operation of Bubble Sort hardware, assuming the unsorted data is already loaded in the module, until the moment when all data is sorted. Reset and abort are not discussed. Load and retrieve of data are straightforward and no further details are needed.

At the arrival of start_i positive pulse the rungenerator module starts to generate a continuous train of wide run pulses until brought to stop by the all_sorted_i signal. The run pulses are delivered to the K_NUMBERS stages by run_o.

During the run pulse, at each clk rising edge, each stage module is taking one bit from the shift register and compare it with the bit from bit_i input. The compare module inside each stage is self locking when a valid compare decision can be done. Lets name An the number stored in internal register of the stage and Bn the number arriving at bit_i input. The bits of the largest number of An and Bn will be sent out through bit_o output to reach bit_i input of the next stage. The bits of the smallest number will be sent through value_o to be stored back in a shift register. This makes a An number that is not moved by the sorting process to be shifted through next stage module before being brought back to the holding shift register.

This way, at each run pulse, a stage module is comparing an input number with the one stored inside and deliver to the next stage the largest of the 2 just as in bubble sort software algorithm.

Each stage module is also propagating the run signal to the next stage ensuring parallel operation of stages. The second propagated signal is the swap signal. This is active high and indicates a swap of numbers taking place during current or a previous stage.

Both run and swap signals provided by the last stage module are used by the intgenerator module to decide the end of sorting process. Just as in software the bubble sort, it is considered achieving the goal when no swapping takes place. The intgenerator stops the run wide pulses by done_o signal generated by missing swap indication at falling edge of the run signal.

The done_o output of intgenerator linked to the all_sorted_i input of the rungenerator module is only stopping the generation of run signal, while, due to the chain architecture, the run pulses previously generated will continue to propagate from stage to stage. Even

if no swapping will take place, the stage modules will be shifting data so the readdata_o will become valid word by word as the run signal moves away. To indicate all readdata_o words are ready, the intgenerator module counts the done pulses and when it reaches the estimated value it generates a pulse on the signal interrupt_o. At this point the data should be read from the module before writing any new unsorted array.

5.

Clock

This sorting module was built with speed in mind so the clk can be run at high speeds. Running from the clock of the bus will probably be a bad ideea. To benefit from the design, a faster clk should be dedicated to this module.

So, even if this module is presented as a synchronous single clk design in practice it will end as a multi clock project.

6.

Final chapter

The sorting module is simple and this makes it probably the best sorting module in the world, alongside other features:

- *small area required;*
- *area increases linearly with word size and number of words;*
- *fclk_max does not depend on the word size or number of words (ok, there is not absolute independence as far as the placement is affecting fclk_max);*

There are some issues; some are by design, like:

- *variable duration of sorting dependent on the data to be sorted;*
- *clock problems due the fast clock imposed by the serial nature of the design;*

Some of the limitations are easy to overcome, please see the list below.

For the enthusiastic student/hobbyist:

- *can you make it faster (any of: clk speed or number of clk periods is ok as long as the area does not grow excessively)?*
- *can you make it work with signed numbers?*
- *can you extend it to {key,value} ?*
- *can you implement abort_i functionality?*

I hope you will have at least as much fun playing with it as I had writing it.

Thank you.

PS: There is a trick in the code so any student willing to drop it on the homework without studying it, will have a surprise.

PS PS: Let me rephrase: there is a chance that the module have more than a trick – nobody is perfect. If you spot any issues please use the email on the front page. Thank you.