DSP in VLSI

Homework 1

Sorter

I. Purpose

In this homework., we will learn to use **sufficient** hardware that meets the throughput requirements to sort a sequence.

II. Related Algorithm

In signal processing algorithm, sometimes we need a sorter to arrange the elements of a sequence in a descending or ascending order. There are many famous sorting algorithms which have different time complexity, such as bubble sort ($\mathcal{O}(n^2)$), merge sort ($\mathcal{O}(n\log(n))$) or heap sort ($\mathcal{O}(n\log(n))$). Of course, we would like to implement a sorter with good time complexity. Here, we consider the merge sort algorithm.

The merge sort is a recursive algorithm, which actually contains two phases, split and merge. In the split phase, the input sequence is partitioned into several groups due to the recursive call and the order in each group must be determined. Thus, the split phase is usually continued until one element in one group as shown in Fig. 1. Of course, the size of the smallest group still can be set by the designer.

In the merge phase, the candidates that are the maximum in the respective groups are compared and merged. If the candidate of one group is chosen as the desired one in the current run of the merge phase, that group must generate a new candidate for the comparison in the next run. When all the groups are merged successfully, the input sequence are arranged in the given order and the sorting is completed as shown in Fig. 3.

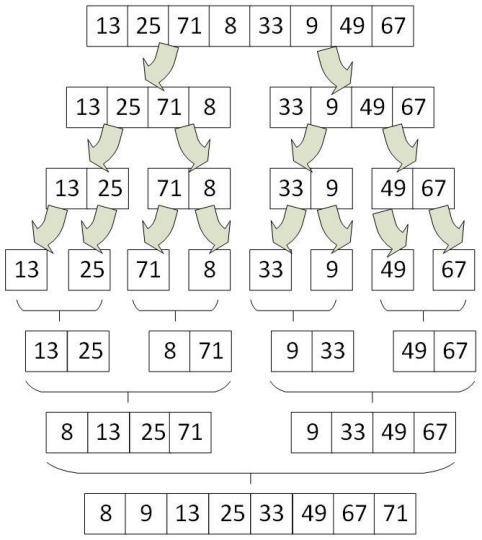

Fig. 1 Merge sort.

The sorting algorithm can be changed to select top M among N inputs. Assume that input sequence is partitioned into G groups. Each group contains N/G inputs arranged in order. When we proceed the candidate selection for M times, the top M elements are determined and the sorting operation is completed. Fig. 2 shows an example with $M = 3,\ N = 8,\ G = 4$. Practically, in the implementation we can use a pointer to point the target element for comparison in each group during the merge phase. When a new candidate is desired, the address of the pointer will be increased by 1 until the end of the group.

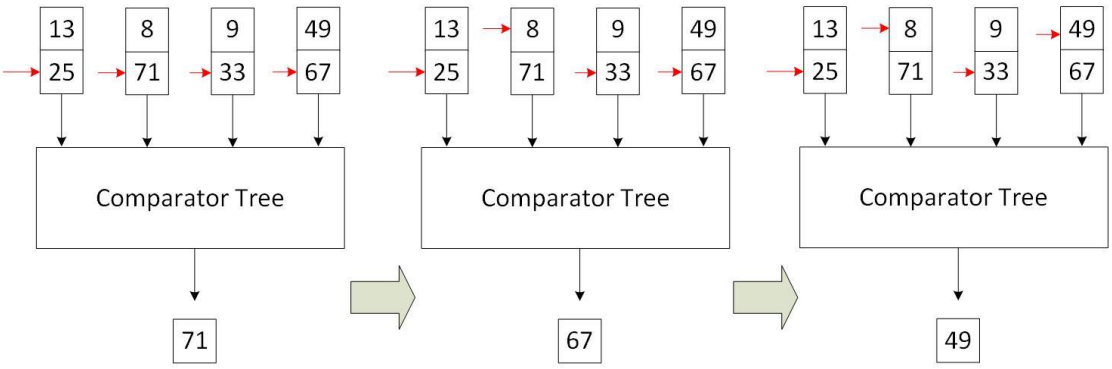

Fig. 2 Selection of top 3 among 8.

III. Function Requirements

Four parallel inputs are entered each clock cycle. One block contains 32 inputs. Namely, it takes 8 clock cycles to send the 32 input data of one block. Assume that the first block is sent from clock cycle 1 to clock cycle 8 and the second block is sent consecutively from clock cycle 9 to clock cycle 16. You need to sort the 32 data and

obtain the top 7 using the merger sort algorithm with only the sufficient number of adders/subtracts for comparison. The latency is not constrained, but the top 7 among 32 must be generated every 8 clock cycles. Of course, to simplified design, we choose to generate the outputs in descending order, with one value produced per clock cycle as shown in the following.

We aim to implement the required function by $M = 7$, $N = 32$, $G = 8$, and now each group contains 4 elements to match the parallel input ports.

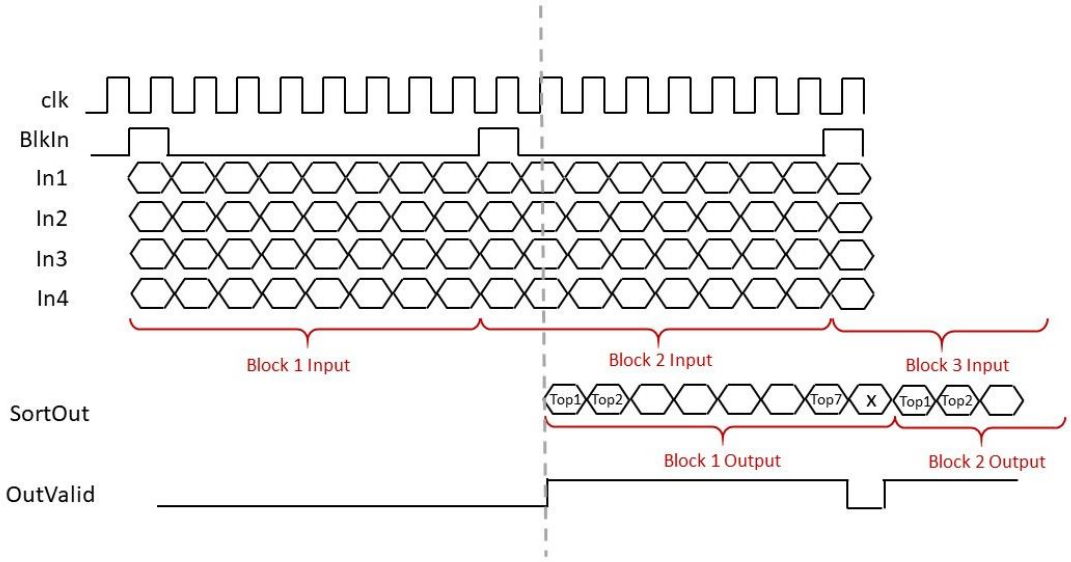

Fig. 3 Timing diagram

IV. Procedures

1. First, generate several random sequences with 32 elements $[x_1\ x_2\ \ldots\ x_{32}]$ in a block between -128 and 127 as your test inputs and prepare as the testbench. (10%)
2. Please use Verilog to implement **Sort4**. Feed the 4 inputs $[x_1\ x_2\ x_3\ x_4]$ simultaneously in one clock cycle. Observe the outputs to realize sorting among four elements in one group. Use $[x_1\ x_2\ x_3\ x_4]$ as the input and check for the correctness of the function "**Sort4**", which can generate sorted output from maximum to minimum as shown in the following Fig. 4. Indicate this function output in the timing diagram of your behavior simulation results. (10%)

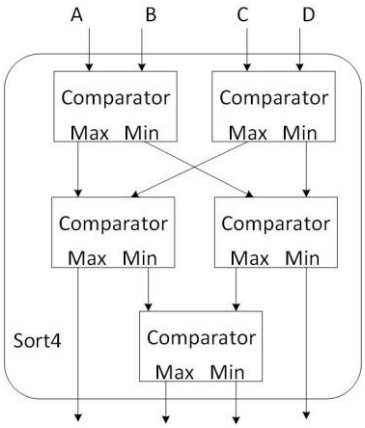

Fig. 4 Block diagram of "**Sort4**".

3. Construct the module (**SelectTopK**) of merger sort to select top 7 among 32 input elements ($[x_1\ x_2\ x_3\ ...\ x_{32}]$) under your control. One example is given as in the following Fig. 5. Use a comparator tree to feed the sorted results from each group and then activate your comparator tree to select top 7 values.
   Draw the block diagram of your own implementation and calculate the number of adders/subtractors/comparators in your block diagram. (20%)
   Show the timing diagram of your behavior simulation results (10%)
   Show the hardware output is correct by comparing it to your Matlab results. (10%)

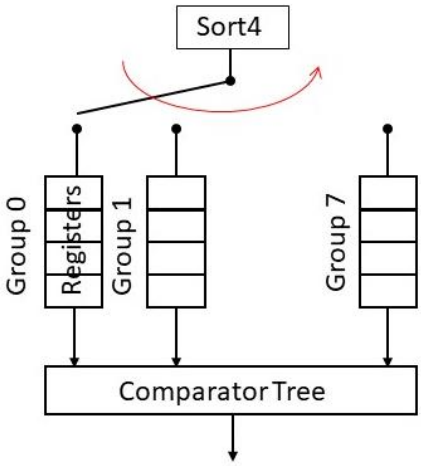

Fig. 5 Block diagram of "**SelectTopK**"

4. Show that your design can generate required top-7 output in descending order every 8 clock cycles as indicated in Fig. 3 in the timing diagram with proper input signal "**BlkIn**" and output indicator "**OutValid**". If it is too long, please cut it down into several segments to make the numerical expressions in the timing diagram clear. If the numerical expressions are hard to distinguish, correct evaluation may not be

given. (30%)

5. Please use Matlab command “sort” to verify your results of your randomly-generated sequence and compare to the Verilog simulation results (10%).
6. Synthesis your design in Q6 and Q8. Show the number of adders/subtractors/comparators in your design. Sum them up together to see if it matches with your block diagram. (10%) (Because you may use counters to control your circuits, we will not ask that the two values should be exactly the same. However, there should not be a large difference.)