University of Toronto, Faculty of Applied Science and Engineering
Department of Electrical and Computer Engineering

ECE 1387F - CAD for Digital Circuit Synthesis and Layout Handout #18

Exercise #3
The Synopsys Behavioral Compiler: A Tutorial

November 1998 Jason Anderson/J. Rose

Assignment Date: November 26
Due Date: December 3,
Late Penalty: **-1 mark per day late, with total marks available = 10**

The purpose of this exercise is to gain familiarity the behavioural level synthesis in the Synopsys behavioural compiler. What to hand in:

1. Report that you successfully walked through the entire tutorial.
2. Offer 1 page of comments on what you thought was good and bad about the behavioural compiler.

This tutorial uses a lab exercise to illustrate some of the features of the Synopsys Behavioral Compiler (BC). The BC performs behavioral synthesis (high-level synthesis). According to Synopsys, this is a transformation "from an algorithmic specification of the behavior of a circuit to a register-transfer level (RTL) structure that implements the behavior."[1]

The BC reads behavioral VHDL or Verilog code and can synthesize the HDL to an RTL circuit that consists of functional units (adders, multipliers, etc.), memory elements and a finite state machine (FSM) for control. The Synopsys Design Compiler can then be used to map the RTL to a gate-level circuit in the target technology.

Some of the advantages of behavioral synthesis are:

- Shorter code (rely more heavily on the synthesis tool to make architectural decisions)
- No coding for control FSM (BC does automatic FSM generation for dathpath control)
- No need to schedule or allocate a specific number of functional units in HDL code
- Fast, automated exploration of architectural alternatives

For those unfamiliar with high-level synthesis, a reference is given at the end of this lab to a review paper which will help you get up to speed.

The fascinating thing about the Behavioral Compiler is that different hardware architectures may be created and evaluated automatically by the tool without modifying the HDL description.

Lab: An Introduction to Behavioral Compiler

1. Behavioral Compiler Public Workshop Notes, April 3rd and 4th, 1996

This lab shows how the Behavioral Compiler can be used to synthesize a piece of HDL code. The HDL code describes a hardware sort engine that reads in eight 8-bit elements, sorts them, and writes the elements out in sorted order. The Verilog HDL code and a timing diagram of the behavior is given at the end of this document. This lab shows how constraints can be set in BC to create architectures possessing varying amounts of speed or area.

Begin by creating a new directory for yourself. Then, copy all the files in the directory: `~jayar/1387/e3` to your new directory. Change directories to ensure that your current directory is the one containing the files you have copied.

Make sure that the following line is in your `.cshrc` file:

```
source /nfs/vrg/cmc/cmc/tools/synopsys.1997.01/source_vrg.csh
```

Fire up the Synopsys Behavioral Compiler by typing (you may need to re-login):

```
bc_shell
```

Once the BC has finished loading you will see a prompt like: `bc_shell>`

Set up the Behavioral Compiler environment by including a special script that was created for this lab. Type:

```
include lab.syn
```

This script sets the synthesis *target* library and sets the *synthetic* library. The synthetic library is set to allow us to use Design Ware components. These special Synopsys components are large macros such as multipliers, adders, and comparators.

We begin by performing a syntactic check on the HDL code and translating the code into an intermediate format. Do this using the analyze command:

```
analyze -format verilog sort.v
```

Assuming the HDL description was analyzed without error, we use the elaborate command to build an internal Synopsys model from the analyzed HDL. Some compiler optimizations are also performed including dead code elimination and constant propagation. Elaborate by typing:

```
elaborate -s sort
```

The ‘`-s`’ switch indicates that the HDL module is going to be *scheduled* with the Behavioral Compiler. `sort` is the name of the top-level HDL module.

Constraints are set to let Behavioral Compiler know a designer’s expectations for a synthesized design. For now, let’s tell BC that we wish to clock the design at 20 MHz. Type:

```
create_clock clk -period 50
```

Check the design by typing:

```
bc_check_design -io superstate_fixed
```

BC supports three different I/O modes. Different I/O modes allow varying degrees of freedom for I/O to move with respect to the clock cycle boundaries specified in the HDL description. The different I/O modes are: `cycle_fixed`, `superstate_fixed`, and `free_float`. The first two maintain the order of the I/O given in the HDL description while `free_float` allows the BC to re-order I/O operations in order to produce more optimal schedules. In `cycle_fixed`, the precise cycle-by-cycle I/O behavior of the HDL description is maintained. In `superstate_fixed`, clock cycles other than those in the HDL may be introduced during scheduling.

Compute area and timing estimates for the functional units needed to implement the design. This is done by typing:

```
bc_time_design
```

The BC considers a variety of architectures for the functional units needed to implement the operations in the HDL. For example, for an adder, it will compute the area and delay of a ripple-carry implementation and a carry-look-ahead implementation. These area and delay estimates are using during scheduling to make area/speed trade-offs.

Check the results of `bc_time_design` by typing:

```
report_resource_estimates
```

Note that the area and timing estimates that result from `bc_time_design` do not need to be recomputed every time a design is scheduled. This information can be saved and re-used. Do this by typing:

```
write -hier -o sort_timed.db
```

Now it’s time to make our first attempt at scheduling the hardware sort circuit. As a default, the Behavioral Compiler performs ASAP scheduling. This creates the fastest possible schedule (shortest) without any concern for the area of components. The BC does this by instantiating multiple components to perform operations in parallel. Find the fastest architecture by typing:

```
schedule -io superstate_fixed -effort zero
```

`zero` effort is acceptable because according to Synopsys, the effort has no effect on the speed of resultant designs, only their area. We will perform *resource constrained* scheduling shortly (to find the architecture with the smallest area).

Save the scheduled (RTL) version of the sort engine:

```
write -hierarchy -o sort_sch_fastest.db
```

Look at the results of scheduling by writing the report file to disk:

```
report_schedule -var -oper -summary -abstract > sort_sch_rpt
```

The switches have the following meanings:

`var` = report showing variable lifetimes and register usage
`oper` = report showing how operations were scheduled into clock steps
`summary` = report showing timing of loops, estimates of area, and hardware resources
`abstract` = report showing state table of finite state machine used to control the datapath

Look at the `sort_sch_rpt` file using a text editor. The first part of the report describes the register resources needed to implement the design and provides a table of variable lifetimes. You should see something like:

```
Storage resource types
======================
 r92.......8-bit register
 r212......8-bit register
 r213......8-bit register
 r215......64-bit register ...
```

Note that one 64-bit register has been allocated. This is certainly to hold the eight 8-bit data values that must be sorted.

The variable lifetimes table shows which variables are in which registers during particular clock cycles. You will see something like the following:

```
-------+------+------+------+-----+-------+------+------+------+------+------+------
 cycle | r215 | r212 | r217 | r92 | r213 | r262 | r268 | r233 | r237 | r232 | r221
-------------------------------------------------------------------------------------
       | (64) | (8)  | (8) | (8) | (8)  | (8)  | (8)  | (1)  | (1)  | (1)  | (1)
=====================================================================================
   0   |......|......|......|.....|......|......|......|......|......|......|......
   1   |......|..v4..|......|.....|......|......|......|......|......|......|.v39..
   2   |..v5..|..v4..|......|.....|......|......|......|......|......|......|.v39..
   3   |..v5..|..v4..|..v7..|.....|......|......|......|......|......|......|.v39..
   4   |..v5..|..v4..|..v7..|.v8..|......|......|......|......|......|......|.v39..
   5   |..v5..|..v4..|..v7..|.v8..|..v9..|......|......|......|......|......|.v39..
   6   |..v5..|..v4..|..v7..|.v8..|..v9..|......|.v12..|......|......|.v33..|.v39..
   7   |..v5..|..v4..|..v7..|.v8..|..v9..|.v10..|.v12..|......|......|.v33..|.v39..
   8   |..v3..|.v11..|......|.....|......|......|......|.v21..|.v28..|......|.v39.. ...
```

One of the more interesting parts of the report is the operation schedule. This shows which operations were scheduled in which clock steps:

```
                                D       D       D       D       D       D       D
                                W       W       W       W       W       W       W
                                0       0       0       0       0       0       0
                                1       1       1       1       1       1       1
                                _       _       _       _       _       _       _
                    p       p   c       c       c       c       c       c       c       p       p
                    o       o   m       m       m       m       m       m       m       o       o
                    r       r   p       p       p       p       p       p       p       r       r
                    t       t   2       2       2       2       2       2       2       t       t
-------+------+-----+------+------+------+------+------+------+------+------+-----+------+------
 cycle | loop | p0  |  p1  | r220 | r219 | r223 | r224 | r225 | r115 | r226 | p2  |  p3
-------------------------------------------------------------------------------------------------
   0   |..L0..|.....|......|......|......|......|......|......|......|......|.W21.|......
   1   |..L3..|.R24.|.R28..|......|......|......|......|......|......|......|.W61.|......
   2   |......|.....|.R28a.|......|......|......|......|......|......|......|.....|......
   3   |......|.....|.R28b.|......|......|......|......|......|......|......|.....|......
   4   |......|.....|.R28c.|......|......|......|......|......|......|......|.....|......
   5   |......|.....|.R28d.|......|......|......|......|......|......|......|.....|......
   6   |......|.....|.R28f.|......|......|......|......|......|.o38p.|......|.....|......
   7   |......|.....|.R28e.|......|......|......|......|......|......|......|.....|......
   8   |......|.....|.R32..|.o38l.|.o38A.|.o38c.|.o38d.|.o38h.|.o38t.|.o38v.|.....|......
   9   |......|.....|......|.o38b.|.o38a.|.o38m.|.o38g.|.o38e.|.o38i.|.o38q.|.....|......
  10   |......|.....|......|.o38..|.o38n.|.o38j.|.o38r.|.o38o.|.o38z.|.o38y.|.....|......
  11   |......|.....|......|.o38u.|.o38w.|.o38f.|......|.o38s.|.o38k.|.o38x.|.W48.|.W51d. ...
```

Notice in the report file that Port p1 is an 8-bit port. The eight reads of port p1 have been scheduled during cycles 1 - 8. Some comparisons have been scheduled in the same clock cycle as the eighth read of port p1. In clock cycles 8 and 9, many comparisons are being performed in parallel.

The summary report follows the operations schedule:

```
*************************************
*  Summary report for process procA:  *
*************************************
--------------------------------------------------------------------------------
     Timing Summary
--------------------------------------------------------------------------------
 Clock period 50.00
 Loop timing information:
    procA.........................................20 cycles (cycles 0 - 20)
        main_loop.................................19 cycles (cycles 1 - 20)
--------------------------------------------------------------------------------
     Area Summary
--------------------------------------------------------------------------------
 Estimated combinational area    4130
 Estimated sequential area       1300
 TOTAL                           5430

 21 control states
 22 basic transitions
 3 control inputs
 24 control outputs

--------------------------------------------------------------------------------
     Resource types
--------------------------------------------------------------------------------
        Register Types
=======================================
        1-bit register.....................2
        8-bit register.....................6
        64-bit register....................1
```

```
        Operator Types
=======================================
        (8_8->1)-bit DW01_cmp2.............1
        (8_8->1_1)-bit DW01_cmp2...........6

        I/O Ports
=======================================
        1-bit input port...................1
        1-bit registered output port.......1
        8-bit input port...................1
        8-bit registered output port.......1
```

The area of the design is given as well as the number and type of resources that have been allocated to implement the design. In the example above, the latency of the `main_loop` is 19 cycles. Several `DW01_cmp2` components have been allocated. These are comparators that are used to compare the elements to be sorted.

Make a note of the area of your design and the latency of the `main_loop`. By doing this, you can make comparisons between the fastest architecture and the minimum area architecture.

Scheduling completes the generation of an RTL version of the sort engine. You could now perform a functional simulation to verify the correctness of the RTL. Write out the RTL version of the hardware sort circuit in VHDL by typing:

```
vhdlout_levelize = true
vhdlout_equations = true
write -hierarchy -format vhdl -out sort_sch.vhd
```

You can use VSS (Synopsys VHDL System Simulator) to simulate the `sort_sch.vhd` RTL description, though this is beyond the scope of this lab. The timing diagram at the end of this document depicts the results of simulating a scheduled version of the hardware sort circuit.

Now, let's find the architecture with the smallest area. To do this, we will re-use the `sort_timed.db` that we created after we executed the `bc_time_design` command. We must remove the current architecture and perform scheduling again to create a smaller architecture. To do this, type:

```
remove_design -design
read sort_timed.db
schedule -io superstate_fixed -effort zero -area
```

The `-area` switch directs BC to perform resource constrained scheduling. After scheduling is complete, save the new architecture:

```
write -hierarchy -o sort_sch_smallest.db
```

Use the `report_schedule` command as before and investigate the area of the new architecture. You can see that the schedule uses a smaller number of functional units and there are fewer parallel operations. Notice the latency of the `main_loop` is longer than before and the

design consumes less area.

You can try some different scheduling efforts in a further attempt to reduce area. The scheduling efforts are: `zero`, `low`, `medium`, or `high`.

The number of cycles used to implement the `main_loop` can be explicitly specified. This allows the designer to make latency/area trade-offs and generate architectures in between the fastest and the smallest. Try the following:

```
remove_design -design
read sort_timed.db
find -hier cell main_l*
label = dc_shell_status
set_cycles 22 -from_beginning label -to_end label
schedule -io superstate_fixed -effort zero -area
```

Then save the design:

```
write -hierarchy -o sort_sch_22.db
```

Use the `report_schedule` command as before to see the results of specifying a specific number of cycles for the `main_loop`. Look at the area of your design and the number and types of allocated resources. Compare the area with the two architectures you previously synthesized. By writing Synopsys scripts, many different architectures can be generated automatically through the use of *for* loops that vary loop latencies.

When we performed ASAP scheduling, we found that the minimum number of cycles to implement the `main_loop` was 19. Using the automatic loop pipelining capabilities of the Behavioral Compiler we can create an architecture wherein we may initiate a new sort every 12 clock cycles. To do this type:

```
remove_design -design
read sort_timed.db
find -hier cell main_l*
label = dc_shell_status
pipeline_loop label -initiation_interval 12 -latency 24
schedule -io superstate_fixed -effort zero
```

The `pipeline_loop` command tells the BC to use a latency of 24 cycles for the `main_loop` and that we wish to initiate the `main_loop` every 12 cycles. Save your design:

```
write -hierarchy -o sort_sch_pipelined.db
```

Use `report_schedule` to evaluate the results of generating a pipelined sort engine. In the summary portion, you will notice the following:

```
************************************
*  Summary report for process procA:  *
```

```
************************************
------------------------------------------------------------------------
     Timing Summary
------------------------------------------------------------------------
 Clock period 50.00
 Loop timing information:
    procA.....................................25 cycles (cycles 0 - 25)
        main_loop...........(initiation interval)..12 cycles
                            (pipeline latency)..24 cycles (cycles 1 - 25)
------------------------------------------------------------------------
     Area Summary
------------------------------------------------------------------------
 Estimated combinational area    4386
 Estimated sequential area       1320
 TOTAL                           5706
```

The timing summary indicates that the `main_loop` is pipelined. The timing diagram that resulted from simulating the pipelined sort engine RTL description is given at the end of this document.

Another dimension for architectural exploration is to change the clock period. Recall that we specified a 50 ns clock period. This can be changed using the `create_clock` command.

BC allows designers to realize architectures with different characteristics without modifying the HDL description.

That's it for this lab! Any of the RTL designs you created could be synthesized to gates using the Design Compiler and re-verified through gate-level simulation. Placement and routing tools would take the gate-level netlist and complete the design's journey through the design hierarchy.

To exit the Behavioral Compiler type:

```
quit
```

To access the Synopsys help type:

```
iview
```

The documentation for Behavioral Compiler can be found in the chapter on synthesis.

The Behavioral Compiler User's Guide contains information on the HDL coding style recommended by Synopsys for behavioral synthesis.

A future lab may include a discussion of how to use Behavioral Compiler to instantiate pipelined components from the Design Ware library, and a discussion of some of the constraints that may be applied before scheduling.

For a good general introduction to high-level (behavioral) synthesis including scheduling and resource allocation, see:

Daniel D. Gajski and Loganath Ramachandran, “Introduction to High-Level Synthesis,” in *IEEE Design and Test of Computers*, Winter 1994, pp. 44-54.

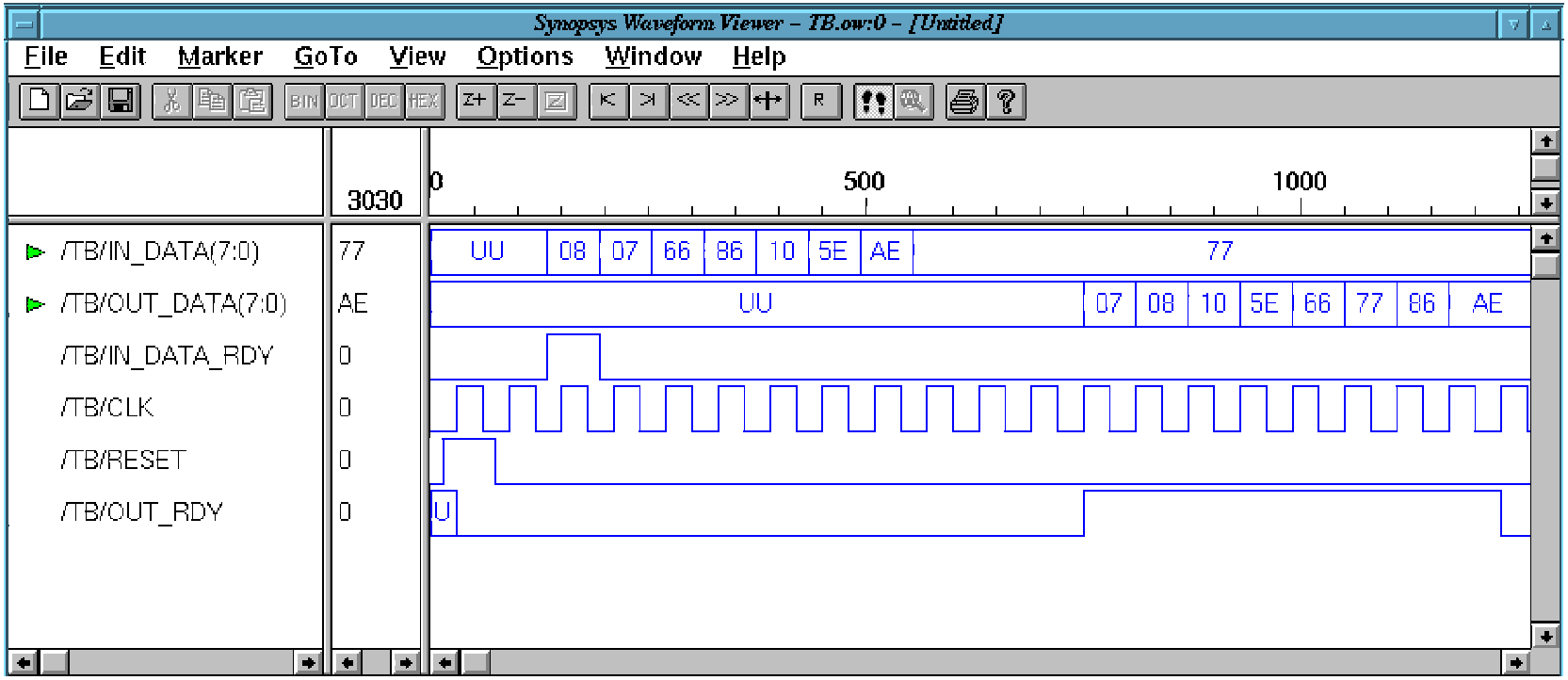
This document was prepared by Jason Helge Anderson. In you find any problems in this lab, please e-mail: *janders@eecg.toronto.edu*

Last edited: October 23, 1997

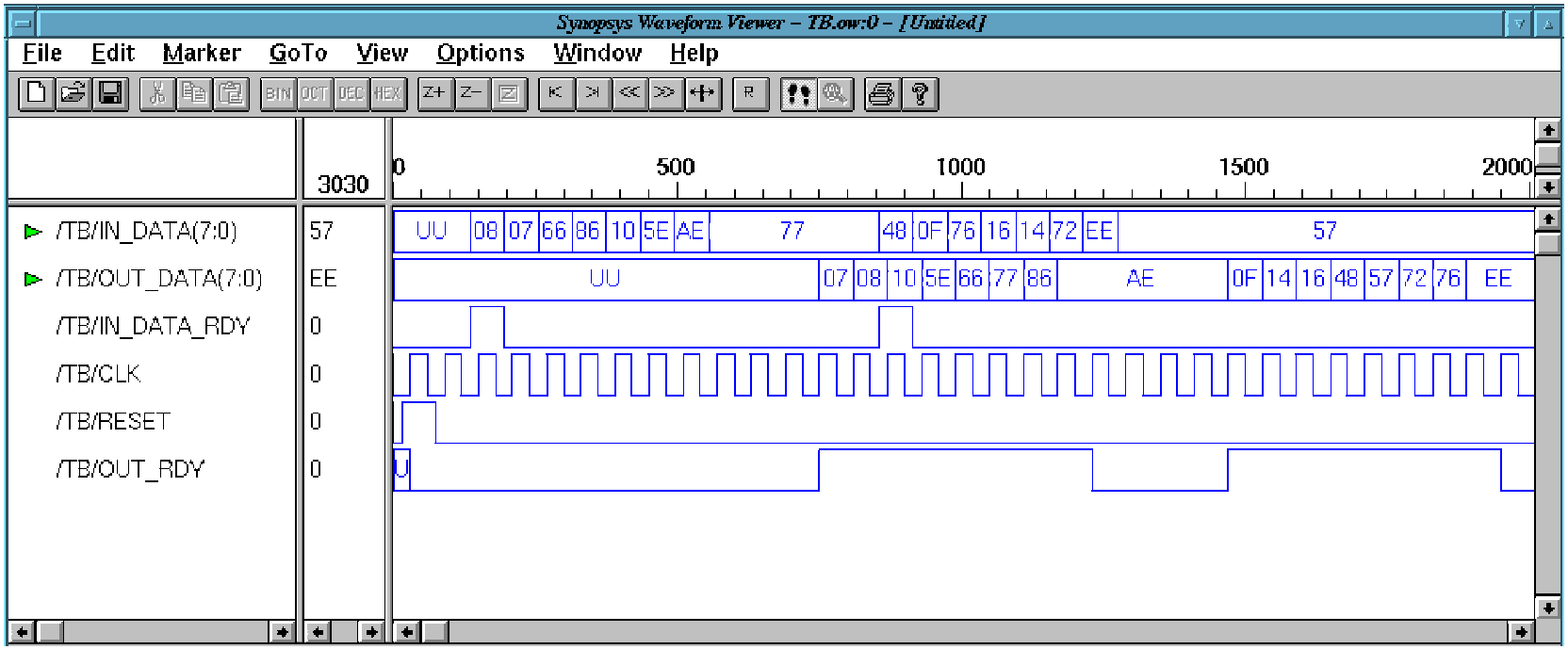
Verilog Code for Sort Engine

```
module sort (in_data,in_data_rdy,out_data,clk,reset,out_rdy);

        parameter               width=8;        //8-bit data items
        parameter               num_items=8;    //8-data items
        input   [width-1:0]     in_data;        //port for input data
        input                   clk;
        input                   reset;          //synchronous reset
        input                   in_data_rdy;    //input handshaking signal
        output  [width-1:0]     out_data;       //port for output data
        output                  out_rdy;        //output handshaking signal
        reg     [width-1:0]     out_data;
        reg                     out_rdy;
        integer                 i,j;            //loop indices
        reg     [width-1:0]     hold;
        reg     [width-1:0]     data_store      [num_items-1:0];
                                                //registers to hold data items

        always  begin: procA
                begin: reset_loop
                        out_rdy<=1'b0;  //on reset, out_rdy gets 0
                        @(posedge clk); if (reset==1'b1) disable procA;
                        //implement a synchronous reset behavior
                        forever begin: main_loop
                        if (in_data_rdy==1'b1)  //handshaking
                                begin   //read in the data items
                                        for (i=0;i<(num_items-1);i=i+1)
                                        begin: in_loop
                                        data_store[i]=in_data;
                                        @(posedge clk);
                                        if (reset==1'b1) disable procA;
                                end
                                data_store[num_items-1]=in_data;//read in last item

                                for (i=(num_items-1);i>=1;i=i-1)
                                //these 'for' loops sort the data
                                begin: sort_loop
                                        for (j=0;j<i;j=j+1)
                                        begin: inner_loop
                                                if (data_store[j]>data_store[j+1])
                                                begin
                                                        hold=data_store[j+1];
                                                        data_store[j+1]=data_store[j];
                                                        data_store[j]=hold;
                                                end
                                        end
                                end
                                @(posedge clk);
                                if (reset==1'b1) disable procA;
                                out_rdy<=1'b1;
                                for (i=0;i<num_items;i=i+1)
                                begin: out_loop //write sorted items
                                        out_data<=data_store[i];
                                        @(posedge clk);
                                        if (reset==1'b1) disable procA;
                                end
                                out_rdy<=1'b0;
                                @(posedge clk); if (reset==1'b1) disable procA;
                                end
                        else //if in_data_rdy is 0 then wait one clock cycle
                                begin: in_data_not_ready_loop
                                        @(posedge clk);
                                        if (reset==1'b1) disable procA;
```

```
                        end
                    end     //main_loop
                end     //reset
                end     //processA
endmodule
```



Timing Diagram for Sort Engine



Timing Diagram for Pipelined Sort Engine