The ABAQS Manual

Alan Ehret
ASCS Lab, Boston University
abaqs.feedback@gmail.com

June 8, 2018

Abstract

The Agent Based Architecture for Quorum sensing Simulation (ABAQS) allows researchers to create and launch Quorum Sensing (QS) simulations. New simulations can be made by incorporating custom cell models. ABAQS uses a shift register that runs along rows of processing elements to load or store simulation states. The shift registers can be connected to a host processor or another system to manage simulations. The architecture is highly modular and separates the functional model from control logic. It has a simple interface to enable users to readily connect their custom models to the simulation platform. The open-source license of this project allows other researchers to contribute and improve the architecture to (a) better fit their quorum sensing simulations and (b) give the community a flexible simulation acceleration tool.

1 Getting Started

1.1 Quorum Sensing Overview

Quorum Sensing (QS) is defined as the detection of extra-cellular chemical molecules, which, at certain levels, will alter the behavior of a cell by activating specific genes. By sensing the external concentration of a specific chemical that other cells produce, a cell can infer the number of surrounding cells. QS acts as a form of indirect communication between cells and is sometimes thought of as chemical "wires" in a broadcast communication. In QS communication, cells do not have a mechanism to directly communicate. Instead, any communication or coordination is achieved by sensing chemical molecules outside the cell. Figure 1 depicts several cells in a varying concentration of chemical molecules. The concentration of chemical molecules is proportional to the number of cells nearby.

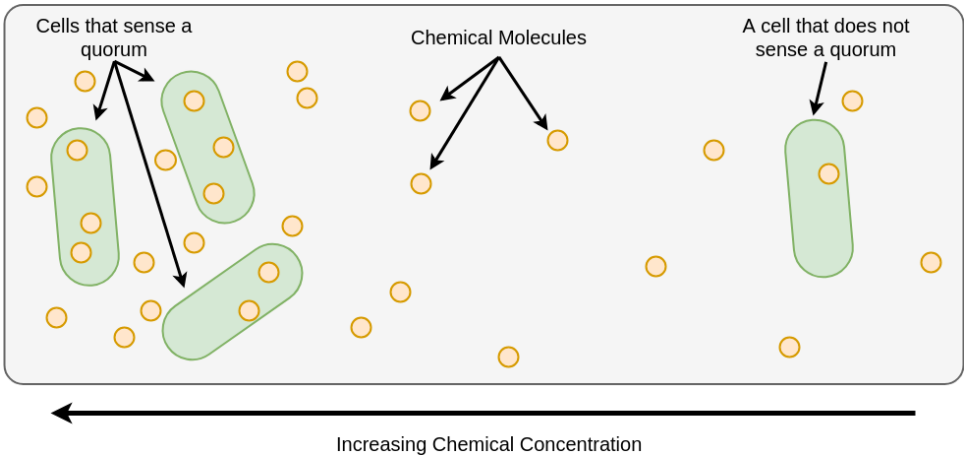

Figure 1: Four cells surrounded by chemical molecules. Each cell is sensing and outputting quantities of the chemical. The cells on the left are in a higher concentration of chemical allowing them to sense a quorum, while the cell on the right is in a lower concentration and does not sense a quorum.

ABAQS, the Agent Based Architecture for Quorum sensing Simulations, is an open-source architecture that accelerates bacterial quorum sensing (QS) simulations. The current version of ABAQS requires a decent understanding of Hardware Descriptor Languages (HDL), specifically Verilog. ABAQS is written in the Verilog 2001 (IEEE 1364-2001) specification. So far ABAQS targets Altera Cyclone V FPGAs but it should work fine on other FPGAs. For synthesis and simulations, ABAQS has been tested with Quartus Prime Lite Edition 17.1 and ModelSim Intel FPGA Starter Edition 10.5b (the version distributed with Quartus). This document will use the specified versions of Quartus and ModelSim but other tools should work as well.

1.2 Module Overview

This section provides a brief overview of the modules in the ABAQS architecture. See the sections below for more documentation on the top level module interface and each module in the design. Figure 2 shows the module hierarchy in the ABAQS architecture.

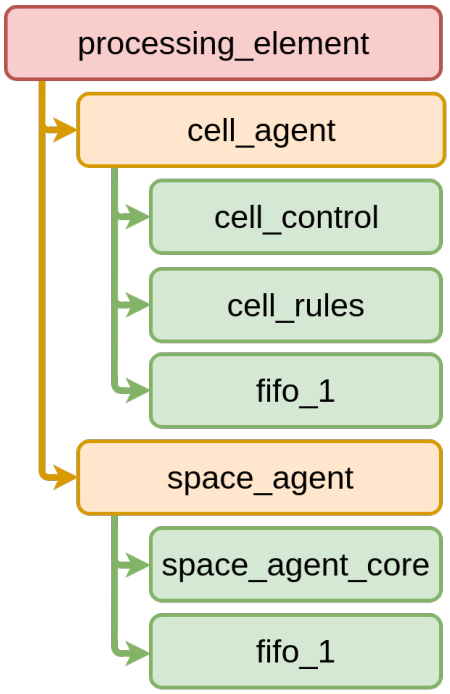

Figure 2: A depiction of the module hierarchy in ABAQS. Modules are only shown once at each level in this hierarchy even if their parent module instantiates them several times.

The ABAQS architecture is made up of two execution units, the cell execution unit and the chemical execution unit (also called the space execution unit). The cell execution unit is all of the logic inside the *cell_agent* module. The chemical execution unit is all of the logic inside the *space_agent* module. A cell execution unit and a chemical execution unit together make a processing element. Figure 3 shows a block diagram of the ABAQS architecture. The expanded gray oval section represents a single processing element. Note that the dotted lines between the cell execution units are only for cell movement. Communication between cells is done through the chemical execution units (along the solid lines).

A custom quorum sensing simulation can be created by making changes to the *cell_rules* module. For now the diffusion model used to simulate chemical concentrations is fixed. The next section describes how to change the *cell_rules* module to create a custom simulation.

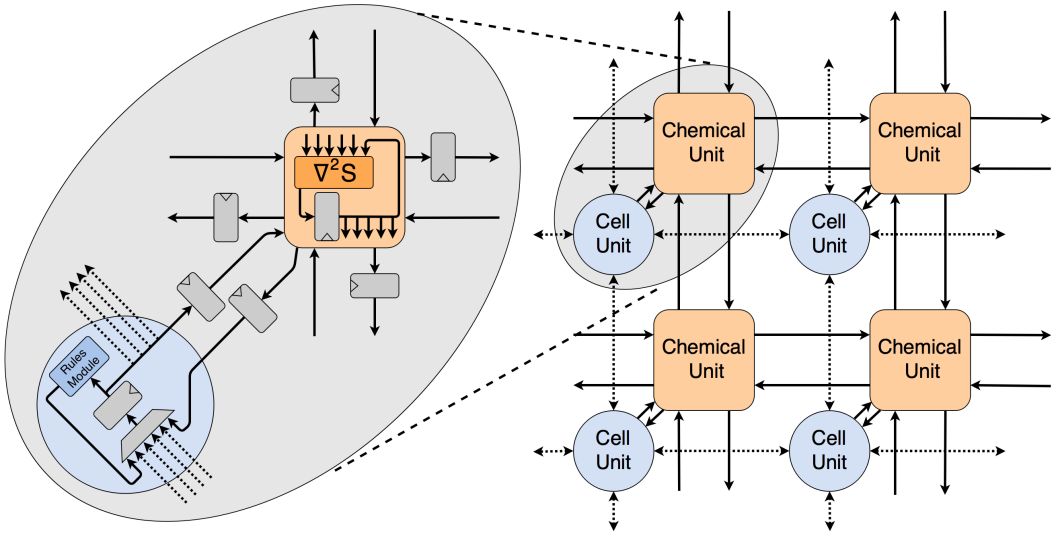

Figure 3: A block diagram of the ABAQS architecture. The expanded section shows a processing element with chemical and cell execution units and the single word FIFOs between them.

1.3 Creating a Quorum Sensing Simulation

This section assumes that you are comfortable with ModelSim and know how to simulate a verilog design. If you do not know how, other resources are available online to teach you.

The default *cell_rules* module implements a simple counter and shift register with the cell state registers. The default *cell_rules* module is not meant to represent a real cell. Instead it demonstrates the minimum functionality needed to verify the features in the architecture. The move_rq signal is held low in the default version of *cell_rules* meaning that the modeled cell will never move.

Figure 5 shows the ModelSim output after compiling and running the *torus_tb* test bench. On the left, the concentration of the chemical field is shown in hex. On the right, the type register of each PE is shown. In this case, there is only one type of cell. A type register value of '1' means that a cell is present in the PE, a type of '0' means that there is no cell present in the PE. Notice that the quorum sensing simulation is initialized with one cell in the center PE and that it does not move because the move_rq signal is held low. The chemical field is initialized to be empty, except for the center PE which is initialized to 0x000F0000 (15 in base10 if there are 16 bits of fixed point precision).

As the simulation progresses, the high concentration of chemical in the center PE begins to diffuse to the neighboring PEs. The center PE will maintain a slightly higher concentration of chemical because the cell model adds a small amount of chemical to the chemical concentration each simulation tick.

After running the default simulation, try changing the *cell_rules* module to change the cell's behavior. Commenting out line 69 and uncommenting line 68 will assert the move_rq signal every 7 clock cycles. Running the simulation again yields the output shown in Figure 6. Note that the '1' in the type registers moves between PEs, indicating that the cell is moving.

It is also important to note that the simulation state cannot be saved every simulation tick. The frequency of simulation state snapshots is limited by the length of the shift register used to output the state data. In the *torus_tb* test bench, a simulation snapshot is saved every two or three simulation ticks (the snapshot period does not line up perfectly with the tick period).

2 Top Level Interface

The *torus* module is the top level module in the ABAQS architecture with a simple interface to save or initialize a simulation state. Exposed at the top level module are shift register inputs and outputs as well as separate read and write signals. Table 1 describes each of the ports in the *torus* port list. Table 2 describes each of the parameters in the torus module. These parameters will be passed down to the modules that need them. The *torus_tb* test bench provides an example of when to assert the read and write signals.

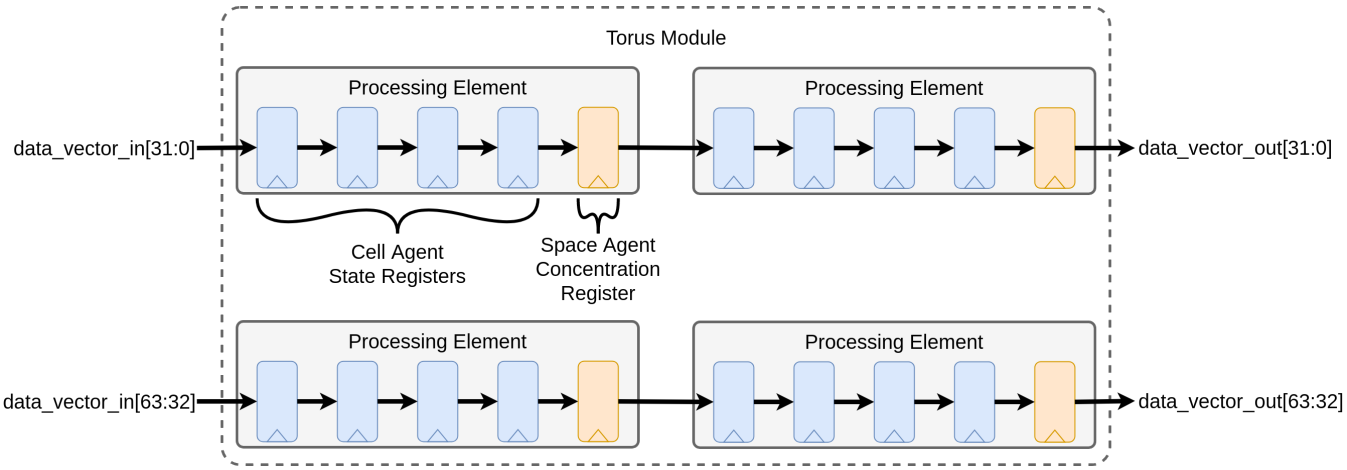

Figure 4: The data shift register in a two high by two wide simulation. Each PE has four cell state registers and one space state register. Type shift registers have been omitted from this diagram.

The shift register used to save or initialize a simulation state in the ABAQS architecture runs along the rows of processing elements (PE). In the Verilog, each row of PEs has its own shift register and the torus module concatenates them together, presenting a single (very wide) shift register in the interface of the top level (*torus*) module. The bit width of the shift register is determined by the height of the simulation (and the BIT_WIDTH parameter) while the depth of the shift register is determined by the width of the simulation and the number of state registers in each PE. Figure 4 represents a two wide by two tall simulation. Each cell agent has four state registers and each space agent has one state register (to hold chemical concentration). The shift register inputs and outputs on each PE are connected to the PEs on their left and right. In this example, BIT_WIDTH is set to 32. The type shift registers are omitted for clarity. The type shift register length is equal to the data shift register length, this way each data word has a type word associated with it. Future versions of the architecture may include the type word in the data shift register so that type words are associated with PEs rather than with data words.

Port Name	Direction	Width	Description
clock	input	1	Clock signal for the entire ABAQS architecture.
reset_n	input	1	Active low reset signal for the entire ABAQS architecture.
write_shift_reg	input	1	Load values from the execution unit registers into the shift registers. Asserting this signal saves the current simulation state into the shift register so it can be shifted out of the torus module and stored somewhere.
read_shift_reg	input	1	Store shift register values in the execution unit state registers. Asserting this signal initializes the simulation state to the current state of the shift register.
data_vector_in	input	HEIGHT * BIT_WIDTH	The data shift register input vector. This port concatenates the input words for each row-wise shift register.
data_vector_out	output	HEIGHT * BIT_WIDTH	The data shift register output vector. This port concatenates the output words for each row-wise shift register.
type_vector_in	input	HEIGHT * TYPE_BIT_WIDTH	The type shift register input vector. This port concatenates the input words of the type mask row-wise shift register. Each data word has an associated type word to indicate the type of agent it belongs to. In reality, only one type mask is needed for each PE module given that the PE only stores one agent. For now a separate shift register is used for simplicity. The area penalty will be small because the TYPE_BIT_WIDTH is small.
type_vector_out	output	HEIGHT * TYPE_BIT_WIDTH	The type shift register output vector. This port concatenates the output words of the type mask row-wise shift register. Each data word has an associated type word to indicate the type of agent it belongs to. In reality, only one type mask is needed for each PE module given that the PE only stores one agent. For now a separate shift register is used for simplicity. The area penalty will be small because the TYPE_BIT_WIDTH is small.

Table 1: A description of each port in the *torus* module, which is the top level module in the ABAQS architecture.

Parameter Name	Default Value	Description
BIT_WIDTH	32	The bit width of a data word. This is the bit width used throughout the architecture, including for space and cell state registers.
NUM_REGS	4	The number of registers to store variables or data in the *cell_control* module. This is the number of cell state registers.
NUM_BITS	$NUM\_REGS * BIT\_WIDTH$	The total number of bits used to store the cell state in the *cell_control* module. This should not be directly changed. Instead change the NUM_REGS parameter.
TYPE_BIT_WIDTH	1	The number of bits in the *cell_control* type type register.
RULE_DELAY	1	The number of clock cycles needed for valid rule output. A delay of 1 means that the *cell_rules* module does not latch the cell state between the input and output.
WIDTH	3	The width of the simulation. This is represents the number of *processing_element* modules in each simulation row.
HEIGHT	3	The height of the simulation. This represents the number of *processing_element* modules in each simulation column.
START_TYPE	{TYPE_BIT_WIDTH{1'b0}}	The value of the type register in *cell_control* after a reset signal. This can be used in simulation to avoid using the shift register to initialize simple simulations.

Table 2: A description of each parameter in the *torus* module, which is the top level module in the ABAQS architecture.

3 Module Description

This section describes each module in the design. See Figure 2 for a depiction of the module hierarchy.

3.1 torus

The *torus* module is considered the top level module of ABAQS. It implements a torus of *processing_element* modules and provides inputs and outputs to the shift register used to load and store simulation states. Extra logic will need to be added to interface ABAQS to the rest of a system (a simple PCIe wrapper is in the early stages of development).

3.2 processing_element

A *processing_element* (PE) is made up of a single *cell_agent* and *space_agent* module. The PE module connects the two types of agent modules creating an interface to represent a single point in space in the simulation.

3.3 cell_agent

The *cell_agent* module connects the *cell_control* module to the *cell_rules* module. It also includes a *fifo_1* module for sending data to the *space_agent* module. This module (with its submodules) contains all of the logic needed to represent a single cell agent.

3.4 cell_control

The *cell_control* module creates the output interface for the *cell_agent*. All of the *cell_agent* output signals run through this module (except for the connections to the *space_agent* which run through the *fifo_1* module first).

This module handles cell movement, checking that the adjacent cell is empty before copying the cell state. All of the control signals for the *cell_agent* (including the ones for cell movement) are generated here.

3.5 cell_rules

The *cell_rules* module must be customized to represent the given cell model. The provided module can be used as a template for custom cell models. Changes should be made to the code below the generate statement (marked by the multi-line comment).

The custom part of the rules module will generally be a pipeline for each type of cell followed by a multiplexer controlled by the type register to output the appropriate state to the *cell_control* module. Note that in the provided *cell_rules* template, there is no type multiplexer because there is only one type of cell. An empty cell is represented by a type of zero. The *cell_control* will automatically prevent state updates with a type of zero, so no logic is needed for it in this module.

3.6 space_agent

The *space_agent* module wraps up all of the logic needed to compute the concentration of a chemical at the current point in space. This module connects to the four adjacent *space_agent* modules (through the *processing_element* modules) as well as one *cell_agent* module.

All of the connections are made with two *fifo_1* modules (one for each direction). Only the outbound *fifo_1* modules are instantiated in the module. The inbound FIFOs are an adjacent module's outbound FIFOs.

3.7 space_agent_core

The *space_agent_core* module reads the chemical concentration from each of the available inbound FIFOs and computes the chemical concentration for the next time step. The *space_agent_core* module then writes the new chemical concentration to the outbound FIFOs.

3.8 fifo_1

This module is a simple FIFO with a depth of one word. This means that the empty and full signal are guaranteed to be inverted and can be combined into a single signal.

4 Creating a Custom Rules Module

In order to change the simulation, a user must add their custom cell model to the *cell_rules* module in cell_rules.v. It is important that the port list is not changed to ensure compatibility with the other modules and future designs. Table 3 describes each of the ports to the *cell_rules* module.

A typical rules module will have a pipeline for each type of cell and a multiplexer controlled by the type_in port. The user defined cell model will read the current cell state and compute the cell's state for the next timestep.

Port Name	Direction	Width	Description
clock	input	1	Clock signal for the rules module.
reset_n	input	1	Active low reset signal for the rules module.
space_data_in	input	BIT_WIDTH	The concentration from the local *space_agent* module.
cell_state_in	input	NUM_BITS	A vector representing the current state of the cell. This input is the concatenation of each of the cell state registers. Use the cell_state_in_array wire to access each one of the state registers individually.
type_in	input	TYPE_BIT_WIDTH	The current type of this cell. The type indicates which cell model should be used. A type of zero means the cell is empty and no updates will be made. The type value can be used to determine which hardware to use to update the cell state if there are several different species of cells in the model.
cell_state_out	output	NUM_BITS	A vector of bits representing the new state of the cell. This vector is a concatenated version of the cell_state_out_array. Use the cell_state_out_array wire to directly assign values to the next cell state.
quantity	output	BIT_WIDTH	This is the amount of chemical added to the *space_agent* in this timestep. This value is a quantity and not a rate, it is not multiplied by dt in the *space_agent*.
move_rq	output	1	Assert this signal to trigger cell movement. If the desired adjacent PE is empty, the control logic will copy the cell state to the destination PE and mark this PE as empty.
move_direction	output	2	This indicates the direction of the desired movement. It is ignored if move_rq is 0. The encoding is: 0x0 = North, 0x1 = South, 0x2 = East, 0x3 = West.

Table 3: A description of each port in the *cell_rules* module.

```
##########################################
#                 Torus TB               #
##########################################
# Loading Sr at time                  180
#
# Value of rotated chem field at time                  180
#
# TR                    BR      Types
#
# 00000000, 00000000, 00000000   0, 0, 0
#
# 00000000, 000f0000, 00000000   0, 1, 0
#
# 00000000, 00000000, 00000000   0, 0, 0
#
# TL                    BL
#
#
# Loading Sr at time                  350
#
# Value of rotated chem field at time                  350
#
# TR                    BR      Types
#
# 00007800, 00023c00, 00007800   0, 0, 0
#
# 00023c00, 00063000, 00023c00   0, 1, 0
#
# 00007800, 00023c00, 00007800   0, 0, 0
#
# TL                    BL
#
#
# Loading Sr at time                  520
#
# Value of rotated chem field at time                  520
#
# TR                    BR      Types
#
# 00014120, 00024ef0, 00014120   0, 0, 0
#
# 00024ef0, 0004bfc0, 00024ef0   0, 1, 0
#
# 00014120, 00024ef0, 00014120   0, 0, 0
#
# TL                    BL
#
#
# Loading Sr at time                  690
#
# Value of rotated chem field at time                  690
#
# TR                    BR      Types
#
# 0001e5e2, 00027ecc, 0001e5e2   0, 0, 0
#
# 00027ecc, 00046d42, 00027ecc   0, 1, 0
#
# 0001e5e2, 00027ecc, 0001e5e2   0, 0, 0
#
# TL                    BL
```

Figure 5: The *torus_tb* test bench output with the default *cell_rules* module without cell movement.

```
##########################################
#                  Torus TB                #
##########################################
# Loading Sr at time                  180
#
# Value of rotated chem field at time                  180
#
# TR                    BR      Types
#
# 00000000, 00000000, 00000000   0, 0, 0
#
# 00000000, 000f0000, 00000000   0, 1, 0
#
# 00000000, 00000000, 00000000   0, 0, 0
#
# TL                    BL
#
#
# Loading Sr at time                  350
#
# Value of rotated chem field at time                  350
#
# TR                    BR      Types
#
# 00007800, 00023c00, 00007800   0, 0, 0
#
# 00023c00, 00053000, 00033c00   0, 0, 0
#
# 00007800, 00023c00, 00007800   0, 0, 1
#
# TL                    BL
#
#
# Loading Sr at time                  520
#
# Value of rotated chem field at time                  520
#
# TR                    BR      Types
#
# 00014120, 000212f0, 00017d20   1, 0, 0
#
# 00022ef0, 000313c0, 00027af0   0, 0, 0
#
# 00026120, 000232f0, 0001dd20   0, 0, 0
#
# TL                    BL
#
#
# Loading Sr at time                  690
#
# Value of rotated chem field at time                  690
#
# TR                    BR      Types
#
# 00026162, 0003b68e, 000236a0   0, 0, 0
#
# 0002521c, 00027794, 0002379a   0, 1, 0
#
# 00023992, 00025abe, 00021bd0   0, 0, 0
#
# TL                    BL
```

Figure 6: The *torus_tb* test bench output with the modified *cell_rules* module to enable cell movement every seven cycles. Note that the simulation ticks do not line up with the simulation snapshots. Each displayed snapshot is 2-3 simulation ticks.