UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Processor Design
(PD)

Module Implementation

3rd Report

Jordi Solà, Ying hao Xu

November 14, 2018

Contents

1 Specification update

In the previous report, we shown the structure of a single PE (Processing Element) and how they are interconnected in order to form the actual core of the systolic array. Something we did not cover in the last report is how this systolic array is going to be feed or drained. For this task, we assumed that our internal cache memory of the accelerator has a limited number of read / write ports.

1.1 Feed array

As the Systolic Array has to be feed following a specific pace, we decided to add what we call a feed array. This array is basically an array of flip-flops connected such in a way that each i element is delayed exactly one cycle after the previous element $(i-1)$. This mechanism lets us feed N elements into the Systolic Array in the same cycle, without having to worry about the rate in which the Systolic Array expects the data to enter.

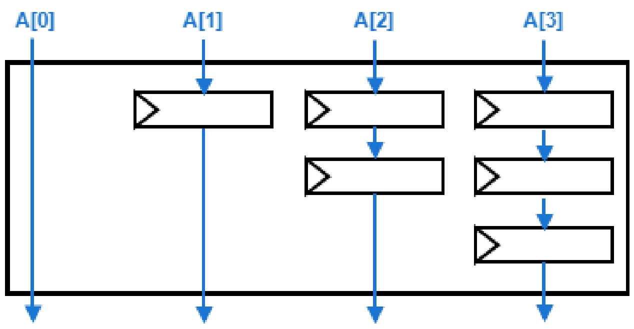


Figure 1: Feed array for $N = 4$

The basic structure can be seen in the Figure 1, in this case, as the HDL language we chose supports the `for` syntax, we defined our feed array as a macro that automatically generates the correct number of delay flip-flops given a `SIZE` parameter (Figure 2).

```
‘define DELAY_ARRAY(CLK, RESET, EN, SIZE, ARRAY_DATA_I, ARRAY_DATA_O) \
    for (genvar gv_i=0; gv_i < SIZE; gv_i++) begin \
        if (gv_i == 0) begin\
            assign ARRAY_DATA_O[0] = (RESET == 1’b1) ? ’0 : ARRAY_DATA_I[0];\
        end\
        else begin\
            logic [$bits(ARRAY_DATA_I[0])-1:0] delayer [gv_i:0];\
            for (genvar gv_j=0; gv_j < gv_i; gv_j++)\
                ‘FF_RESET_EN(CLK, RESET, EN, delayer[gv_j], delayer[gv_j+1], ’0)\
            assign delayer[0] = ARRAY_DATA_I[gv_i];\
            assign ARRAY_DATA_O[gv_i] = delayer[gv_i];\
        end\
    end
```

Figure 2: Delay array macro in SystemVerilog

The number of required flip-flops given an n can be expressed using the following formula:

$$A(n) = \frac{n^2 - n}{2}$$

1.2 Drain array

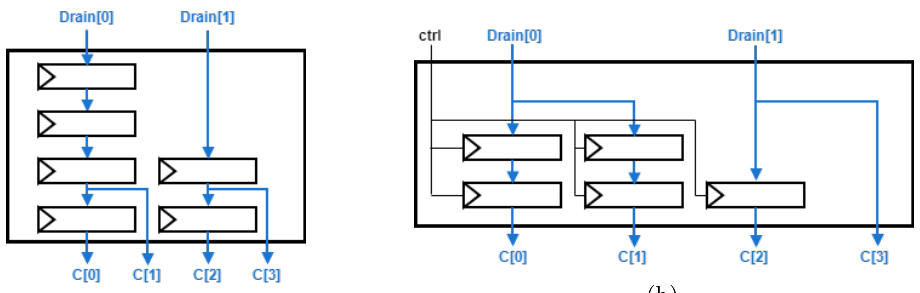
Similarly as in the feed array, our Systolic Array design does not outputs all the data in the same cycle. Furthermore, as we are using a shared drain channel between two consecutive columns, it is critical to read and store the data in the exact cycle. For this task, we followed a similar approach as in the feed array by using a delay array in order to abstract to the other functional units, the pace in which the results are available.

In our systolic array, the order in which the results flow out through the drain channel can be seen in Table 1. Each element of the table represents in which cycle it is going to be available on the output ports. For example, for the element (0,0), we are considering that it is going to be available in the cycle 1, the element (0,1) in the cycle 2 and so on. Notice that the elements flow out in a interleaved fashion, so for example, in the cycle 3 and 4, the drain channel outputs elements from the first and the second row.

1	‖	2	3	‖	4
3	‖	4	5	‖	6
5	‖	6	7	‖	8
7	‖	8	9	‖	10

Table 1: Systolic Array drain order for $N = 4$. Each cell represents the cycle when it is available.

As the drain channel is half the size of the systolic array, consecutive elements are available within one cycle of distance and they will flow out from the same output port. When we were thinking about this unit, we came up with two different approaches (Figure 3a and Figure 3b).



(a) (b)

Figure 3: Drain array proposals for $N = 4$

Figure 3a shows the most natural way to design the delay array. In this case, each element is processed sequentially and during the last stage, the element in the previous stage is bypassed, so we can read the N result elements at once.

Figure 3b focuses on reducing the number of flip-flop toggles. In this design, the delay array granularity is a pair (two elements) and only when both elements are ready, the data is passed to the next stage. This behavior forced us to use a control signal (`ctrl`) that must be in sync with the rest of the system in order to write the data to the correct flip-flop.

If we compare Figure 3a and Figure 3b, the number of flip-flops needed in both cases is almost the same. However, for the second one, as we are passing the elements to the next stage every two cycles, the number of toggles is significantly reduced. This can be easily seen if we analyze it from the point of view of a single element. While in the Figure 3a approach, in the worst case an element has to traverse N flip-flops, in the Figure 3b approach, also in the worst case, the number of flip-flops to traverse is reduced by half ($N/2$), as the elements are passed to the next stage every two cycles instead of every cycle.

As power is something important in our design, even if the final design becomes more complex, we decided to go with the Figure 3b approach, as it requires almost the same number of flip-flops as the Figure 3a approach, but consuming half of the dynamic power. The number of flip-flops that are needed for the 3b approach can be expressed using the following formula:

$$D(n) = \frac{n^2 + 2n}{4} - 1$$

1.3 Systolic Array Wrapper

Now that we have defined the feed array and the drain array, we joined them and created the Systolic Array Wrapper module. This module is basically the Systolic Array plus 2 feed arrays (as we have two inputs) and a single drain array (as we only have one output). Figure 4 shows how the systolic array is going to be connected to the new arrays and which input/output ports will be exposed to the rest of components.

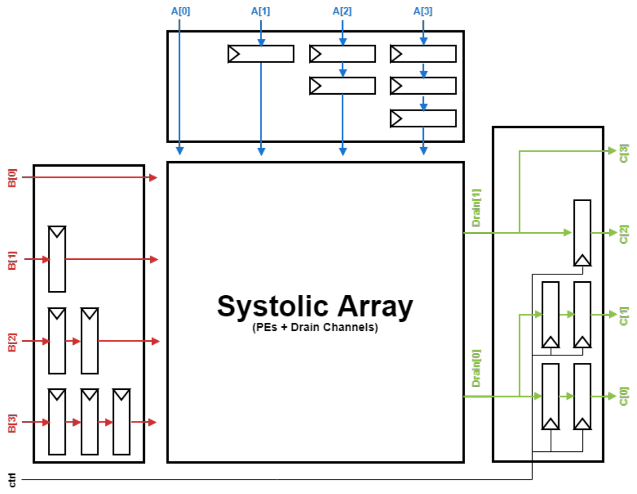


Figure 4: Systolic Array wrap module

This new wrap module let us forget about the pace in which the systolic array read / writes the data, simplifying the interaction between the systolic array and the other units. Considering that the wrap module has 2 feed arrays and 1 drain array, the total number of extra flip-flops that we need is given by the addition of the aforementioned elements, thus it grows lineally with the array area:

$$T(n) = 2 \cdot A(n) + D(n) = 2 \cdot \frac{n^2 - n}{2} + \frac{n^2 + 2n}{4} - 1 = \frac{5}{4} \cdot n^2 - \frac{1}{2} \cdot n - 1$$

2 Development status

2.1 Infrastructure

One of our main goals for the project development was to build and maintain an infrastructure that allow us to automatize the tests and perform some kind of validation of the project.

This implies two main tasks, first, we need to develop some testing tools allowing us to compile and execute our testbenches, and second, we need to design a testplan and build the corresponding tests for each component.

2.1.1 Test tools status

2.1.1.1 Structure of the system

We wanted the system to be easily reproducible without a complicated setup in as many diverse environments as possible, and this conditioned our design decisions.

Given that, we decided to use Linux containers (in particular Docker) in order to isolate our infrastructure environment, thus allowing us to run out-of-the-box, independently of the host environment. There is a lot of literature on the benefits of containerization on reproducible research, and one of the easiest to understand is Carl Boettiger's introduction [1].

Keeping that on mind we created a Docker image containing all our software dependencies. All our scripts spawn an instance of the container mounting our project folder and doing its job inside our isolated environment.

This image is published at the docker store under jsola/verilator and can be imported with the following command:

```
docker pull jsola/verilator
```

The Dockerfile can be found in the Appendix A.1.

2.1.1.2 Installation and environment setup

We wanted to automatize as much as possible the setup, thus we created a script to automatically detect and configure the environment variables needed to run our tests and pulling our docker image as well. This script can be found at the Appendix A.2.

Thus, given that we have Docker installed on our machine, the only thing needed to setup the environment for the project is to clone it from github and source this script:

```
git clone https://github.com/ying27/matrix-multiplication-accelerator.git
cd matrix-multiplication-accelerator
source env.sh
```

2.1.1.3 Specification of a testbench and compilation

In order to perform tests of specific parts of the design, we need to be able to create different testbenches instantiating the top module of the hierarchy we want to test.

In order to simplify the compilation flow, we also created a script that given a parameter specifying the name of the testbench, it automatically looks for the testbench top at the path `tb/TB_NAME/TB_NAME.sv` and compiles the model.

One extra parameter to consider when compiling the model is whether we want to see a trace of the result or not. One could think that it makes no harm to always compile it with trace support, but it adds some specific constraints to the compiler resulting both in a larger compilation and execution time. Therefore our script has an optional parameter `-trace` used to specify that we want a model with support for traces.

Also, some specific testbeches might use extra files to execute (due to `$fopen` or `$freadmem` in the code), thus, our tool lets us specify a test name using `-test` and all the files under the directory are linked to the future build folder. This parameter looks for all the files inside the folder `tb/TESTBENCH/test/TESTNAME`.

With this, in order to create a testbench of a specific element, one can simply create the file `element_tb.sv` in the folder `tb/element_tb` with the following baseline testbench:

```
`ifndef TIMEOUT
    `define TIMEOUT 100
`endif

module element_tb(
    input clk,
    input reset
    );

    element element_name(
        .clk_i(clk),
        .rst_i(reset),
        .* //Automatically capture the defined signals
    );

    //Timeout control
    int cycle_counter;
    `FF_RESET(clk, reset, cycle_counter+1, cycle_counter, '0);

    always_comb begin
        if(cycle_counter == `TIMEOUT) $finish;
    end

endmodule
```

To compile the testbench, we have to execute the following command:

```
compile -top element_tb
```

Or, if we want to be able to produce a trace afterwards:

```
compile -top element_tb -trace
```

This will perform the aforementioned flow and create a folder with the build at `tb/element_tb/build`. Both, the compile script as well as the generic `main.cpp` file for verilator can be found at appendices A.3 and A.4.

2.1.1.4 Execution

Following the criteria we've seen before, we also have created a script in order to simplify the execution of the model. This script has one mandatory argument which is the name of the testbench that needs to be run. If the testbench was compiled for tracing we can pass as an argument `--trace target.flt` to produce a tracefile. The script can be found at appendix A.5.

Once the test execution finishes, a soft-link at `$REPOROOT/last_run` pointing to the execution directory will be created. So, in order to get a trace of our previous model `element_tb` we do it like this:

```
run element_tb --trace trace.flt
```

2.1.1.5 Viewing the trace *(unstable)*

There's a tool in development to be able to launch gtkwave inside the docker environment in order to make it generic too, however, as forwarding the X display to the host is not host independent, and so far this only works on OSX with XQuartz 2.7 and it's still unstable. The script can be found at the appendix A.6.
This can be done by using:

```
view trace.flt
```

2.1.2 Next steps

Given our current status, the following steps would be to further automatize the testing and consolidate a standard for test specification. This implies also to build the necessary structures for applying the correspondent stimulus to the simulations, making them UVM compliant if possible.

2.2 Model

One of the main goals of our project is to develop HDL that is scalable and easy to understand. For this reason, we have defined some key parameters that can be found in `inc/common_pkg.sv`:

- **DATA_WIDTH:** This defines the width of the data that the systolic array operates with.
- **SYS_ARRAY_SIZE:** This defines the number of rows/columns of the systolic array.
- **ADDR_WIDTH:** This is the width of the memory addresses used to read/write data into the accelerator.

Following the same idea, all the data types are well defined inside the `inc/common_pkg.sv` as a `struct packed` type, which let us make the design even more generic and easy to modify.

As one of the potential extensions is to apply clock gating to the flip-flops, we have defined 3 macros that mimics the behaviour of a flip-flop:

- `FF_RESET(CLK, RESET, DATA_I, DATA_O, DEFAULT)`
- `FF_EN(CLK, RESET, EN, DATA_I, DATA_O, DEFAULT)`
- `FF_RESET_EN(CLK, RESET, EN, DATA_I, DATA_O, DEFAULT)`

By using these macros, we can easily change the flip-flops to a gated one but it also makes the files easier to read and more close to the actual implementation. These macros can be found in the file `inc/common.svh`

2.2.1 HDL modules

Currently, the core of the systolic array is working. Thus is a module that is capable of successfully complete a matrix matrix multiplication. In order to accomplish this task, we have implemented the following modules:

- **PE:** Processing Element that computes the MAC (Multiply Accumulate) operation (`pe.sv`).
- **Drain channel:** implementation of a single drain channel that connects two PEs from the same row (`drain_channel.sv`).
- **Systolic array:** module that instantiates the PEs and the drain channels given a size and interconnects them properly (`systolic_array.sv`).
- **Drain array:** implementation of the drain array described in the previous section (`drain_array.sv`).
- **Systolic array wrap:** module that connects the systolic array and the feed/drain arrays (`systolic_array_wrap.sv`).

However, the control logic is not ready yet and the signals must be carefully driven by a behavioural model.

2.2.2 Next steps

As mentioned above, we currently have implemented the core of the accelerator. As part of the next steps, we will have to implement the controller. This unit will be responsible of accessing to the accelerator's cache and feed the systolic with the desired data. On the other hand, as the controller knows at any time the state of the systolic array, it will also be responsible of driving the `last` and `ctrl` signals.

Once we have a controller, a cache will be implemented too. This will store both, the data required for feeding the accelerator and the results of the matrix matrix multiplication. As the project main focus is not the memory subsystem, for this task we will assume an ideal memory model.

And last but not least, an instruction decoder will be required too. This unit will interpret a mini-ISA (Instruction Set Architecture) that follows the RISC (Reduced Instruction Set Computer) style. The instructions it will cover are described in the previous report, but basically are read, write and operate instructions.

3 Validation

"Validation. The assurance that a product, service, or system meets the needs of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers. Contrast with verification."[2]

3.1 Testplan

In this section we will try to describe a testplan to verify that our DUTs (Design Under Test) works properly, following the expected behaviour.

3.1.1 PE

For the PE, we intend to test the following situations:

	Test	Status
1	Send a request to a PE without the `last` bit set.	PASSING
2	Send a request to a PE with the `last` bit set.	PASSING
3	Send a sequence or requests to a PE without any request with the `last` bits set.	PASSING
4	Send a sequence of requests to a PE finishing it with a request with the `last` bit set.	PASSING
5	Send it a sequence of requests with interleaved requests with the `last` bit set.	PASSING
6	Send it a request which produces a multiplication overflow.	PENDING
7	Send it a sequence of request which produce an addition overflow.	PENDING
8	Send it a request with the `last` bit set only on one of the inputs.	PENDING

3.1.2 Systolic array

This are the ones we though could be good for our array model:

	Test	Status
1	Send a request through each of the channels without the `last` bit set.	PASSING
2	Send a request through each of the channels with the `last` bit set.	PASSING
3	Send a sequence of requests through each channel without any of them having the `last` bit set.	PASSING
4	Send a sequence of requests through each channel with the last of them having the `last` bit set.	PASSING
5	Send a sequence of requests through each channel with interleaved requests with the `last` bit set.	PASSING

3.1.3 Wrapper

This are the tests considered to be run for the wrapper module:

	Test	Status
1	Send a request to the wrapper without the `last` bit set.	PASSING
2	Send a request to the wrapper with the `last` bit set.	PASSING
3	Send a sequence of requests to the wrapper without any of them having the `last` bit set.	PASSING
4	Send a sequence of requests to the wrapper with the last of them having the `last` bit set.	PASSING
5	Send a sequence of requests to the wrapper with interleaved requests with the `last` bit set.	PASSING

3.2 Next steps

On the validation side, this are the next steps planned for the next release:

- Creation of checkers for each relevant module following UVM standards.
- Extension of the testing environment taking into account the feedback provided by the checker, detecting execution errors.
- Adition of assertions for explicit implementation error detection.
- Designing the testplan for the forthcoming modules, and do an intense revision and possible extension of the current ones.
- Implement and monitor line coverage, toggle coverage, and functional coverage metrics.

4 Verification

"Verification. The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process. Contrast with validation."[2]

4.1 Test specification

4.1.1 Hardware

In order to simplify the analysis of the trace while keeping some complexity, we decided to test the model with the following configuration:

- Data width of 1 byte.
- Array of 4x4 PEs.

As we are not using the memory yet, the value for the address length is irellevant.

4.1.2 Input

In order to verify the behaviour of our build model, we have performed a simple matrix multiplication:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 8 & 7 & 6 \\ 5 & 4 & 3 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 8 & 7 & 6 & 5 \\ 4 & 3 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 51 & 47 & 43 & 39 \\ 123 & 119 & 115 & 111 \\ 129 & 133 & 137 & 141 \\ 57 & 61 & 65 & 69 \end{bmatrix}$$

In order to do so, we had to add to the testbench the capacity to read values from a pre-formatted file. This was achieved by adding several `$scanf`'s to the code and creating a file with the data we wanted to feed at `tb/array_wrap_tb/test/base`. The test we created feeds the array two times with the aforementioned matrices, in order to see the interaction of several operations.

4.1.3 Reproducibility

Our test can be reproduced by doing the following:

```
git clone https://github.com/ying27/matrix-multiplication-accelerator.git
cd matrix-multiplication-accelerator
git checkout 85f4337f9e83f69d2ce303fee20c6891757cef91
source env.sh
compile -top array_wrap_tb -trace -test base
run array_wrap_tb --timeout 10000 --trace test.fst
```

This will produce the trace file `test.fst` at `$REPOROOT/last_run/test.fst` which can be later opened with gtkwave. The signal files used to see the screen captures can be found at `$REPOROOT/tb/array_wrap_tb/profiles`.

4.2 Analysis

4.2.1 Processing Element

First, let us take a look at the behaviour of the most basic unit, the PE. For this, we've picked as an example the first element of the array. Each PE is in charge of computing the element of the resultant matrix with its same indicies, thus, each PE is in charge of computing the following operation:

$$value_{i,j} = \sum_{k=0}^{N} a_{i,k} * b_{k,j}$$

The PE structure is targeted to perform an accumulate each of the products each consecutive cycle. With this, after running the test, Figure 6 shows the signals of $PE_{0,0}$. There, we can see that the PE is correctly feed, receiving each value of the matrix at the specified cycle and performing a multiply plus addition operation, accumulating the result in `result_n` (which is the name for the wire feeding the partial register). We can see how the element starts to get feed at marker A which causes it to start performing computation and storing partial results, and receives its `last` control signal at marker B, which produces the emission of the results to the drain channel and the resetting of the partial stored value.

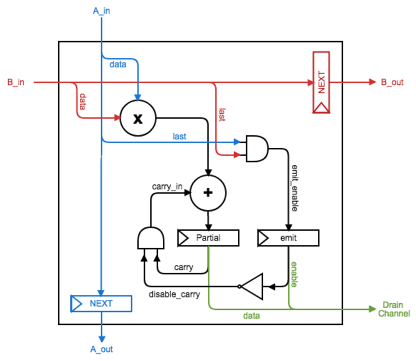


Figure 5: Reminder of the internal structure of a PE

4.2.2 Drain channel

As for the Drain Channel, looking at the trace for the first drain channel (Figure 7) we can see how the PE's finish their computations in a way that they do not collide with each other and leave the pipeline as specified in Section 1.2.

4.2.3 Wrapper

For the top level, our feed and drain arrays allow us to feed each line of data at once, as well as retrieve the results also line by line. On Figure 8 we can see a trace of the behaviour of our wrapper. First of all, we can see how it is feed, line by line in case of input A, and column by column in case of B, and how we retrieve it line by line at output C. We can see the first operation being fed between markers A and B, and how the result of that operation is emitted every two cycles after marker C until marker D.

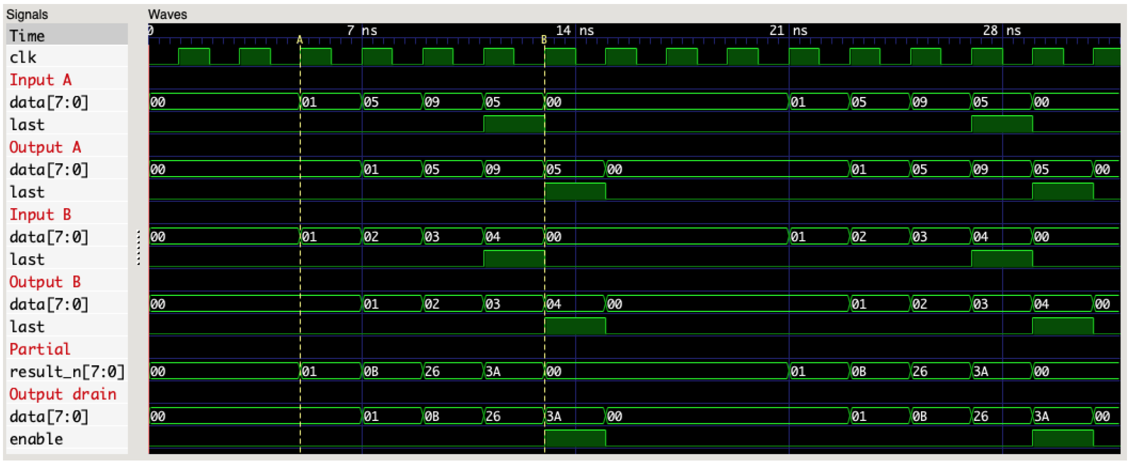


Figure 6: PE trace during execution

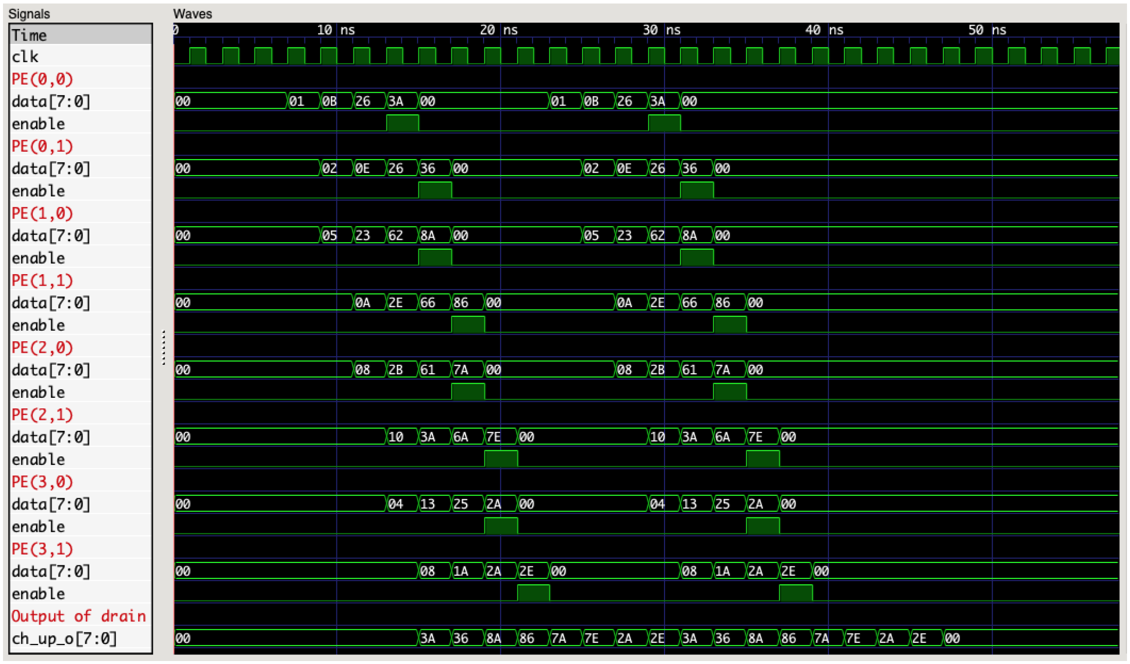


Figure 7: Drain channel trace during execution

Also, we can observe the delay we have for issuing the second operation. This will delay the emission of the result in order to avoid interfering with the previous operation's result, right as we predicted in our last assignment.

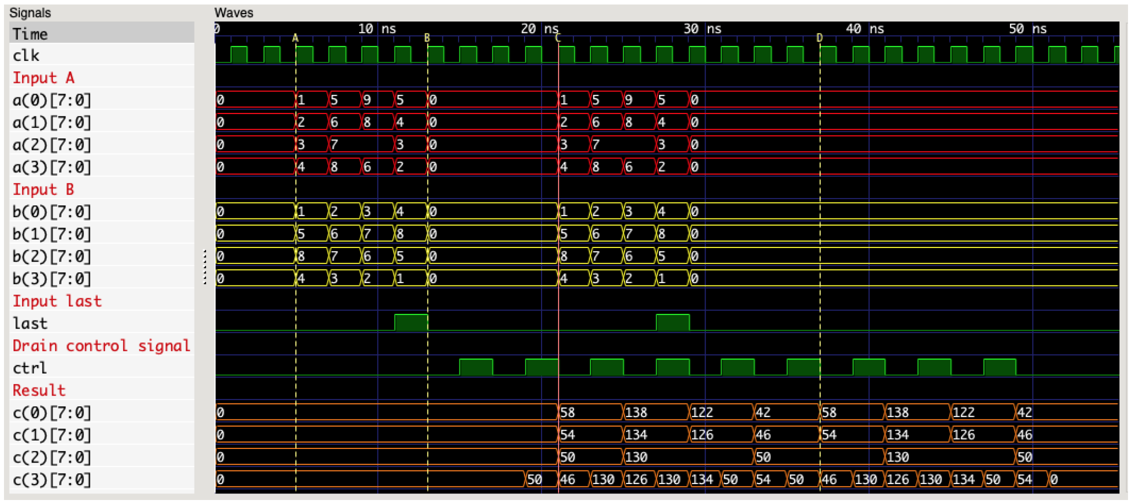


Figure 8: Systolic Array wrap module trace during execution.

References

[1] Carl Boettiger. *An introduction to Docker for reproducible research.* Tech. rep.

[2] *IEEE P1490/D1, May 2011 - IEEE Draft Guide: Adoption of the Project Management Institute (PMI) Standard: A Guide to the Project Management Body of Knowledge (PMBOK Guide)-2008 (4th edition).* IEEE, 2011. ISBN: 9780738167886. URL: https://ieeexplore-ieee-org.recursos.biblioteca.upc.edu/document/5937011.

A Appendices

A.1 Dockerfile

```
FROM debian:stretch

WORKDIR /tmp
RUN apt-get -y update
RUN apt-get -y install apt-utils

#Installing pre-requisites

RUN apt-get -y install zlib1g-dev bash build-essential git make
RUN apt-get -y install autoconf perl gcc g++ flex bison

# Installation of Verilator

RUN git clone http://git.veripool.org/git/verilator
WORKDIR /tmp/verilator

RUN git pull
RUN git checkout stable      # Use most recent release
RUN autoconf
RUN ./configure
RUN make -j $(nproc)
RUN make install -j $(nproc)
WORKDIR /tmp

# Clean installation files
RUN rm -rf verilator

# Installation of gtkwave
RUN apt-get -y install gtkwave

# Adding the installed software to the path of the container
ENV PATH /usr/local/bin:$PATH
ENV SHELL /bin/bash
```

A.2 Environment script: **env.sh**

```
#Detect the path to the top of the repo from which we are calling the script
export REPOROOT=$( cd $(dirname "$0"); pwd -P)

#Add our scripts to the path
if [ -z ${OLD_PATH+x} ]; then
    export OLD_PATH=$PATH;
else
    export PATH=$OLD_PATH;
fi
export PATH="$REPOROOT/tools/scripts:$PATH"

#Specify the path of the rtl and testbenches
export RTLROOT="$REPOROOT/src"
export TBROOT="$REPOROOT/tb"

#Pull docker image if necessary
export DOCKERTAG="jsola/verilator:latest"
if [[ "$(docker␣images␣-q␣$DOCKERTAG␣2>␣/dev/null)" == "" ]]; then
    docker pull $DOCKERTAG
fi
```

A.3 Compile script: **compile**

```
#!/bin/bash

#Variable specification
export RTL_TOP
export DOCKER_NAME=compile
export EXTRA_VERLIATOR_ARGS
export CFLAGS

#Detect arguments
while [ $# -ne 0 ]
do
    arg="$1"
    case "$arg" in
        -top) #TB path
            RTL_TOP=$2
            CFLAGS+=\ -DCXX_TOP_LEVEL=V$2
            shift
            ;;
        -trace)#Trace or not
            EXTRA_VERILATOR_ARGS+=--trace-fst
            CFLAGS+=\ -DTRACE
            ;;
    esac
    shift
done

export TARGET_PATH=$TBROOT/$RTL_TOP/build
echo "RTL_TOP:␣$RTL_TOP"

#Clean
echo "Cleaning␣last␣build"
rm -rf $TARGET_PATH

#Docker container creation
echo "Creating␣compile␣container"
docker run -w $REPOROOT -i -t -d -v $REPOROOT:$REPOROOT \
--name=$DOCKER_NAME $DOCKERTAG > /dev/null

#Verilog compilation
echo "Compiling␣verilog"
compile_cmd="verilator␣-Wall␣$EXTRA_VERILATOR_ARGS␣--Mdir␣$TARGET_PATH␣-CFLAGS␣\
\"$CFLAGS\"␣--cc␣$RTL_TOP␣-f␣$REPOROOT/tools/conf/incdir.f␣-y␣$TBROOT/$RTL_TOP␣\
--exe␣$REPOROOT/tools/csrc/*.cpp"
echo $compile_cmd
docker exec -t -w $REPOROOT $DOCKER_NAME bash -c "$compile_cmd"

#C Compilation
echo "Compiling␣C"
compile_cmd="OPT=-DVL_DEBUG␣make␣-j␣-C␣$TARGET_PATH␣-f␣V$RTL_TOP.mk␣V$RTL_TOP"
echo $compile_cmd
docker exec -t -w $REPOROOT $DOCKER_NAME bash -c "$compile_cmd"

#Cleaning the environment
echo "Removing␣container"
docker stop $DOCKER_NAME > /dev/null
docker rm $DOCKER_NAME > /dev/null
```

A.4 Generic main file: **main.cpp**

```
#include "verilated.h"
#include <iostream>
using namespace std;
#include "verilated_fst_c.h"
#include <getopt.h>

#define PATH_LENGTH 1000

#define mStr(x) #x
#define mStr_(x) mStr(x)
#define topModuleInc mStr_(CXX_TOP_LEVEL.h)

```

```
#include topModuleInc

void usage(char* name) {
    cerr << "Usage:␣" << name << "␣[--trace␣traceFile.fst]␣[--timeout␣timeout]␣[--help]
    cerr << "\tTrace:␣off␣by␣default" << endl << "\tTimeout:␣default␣1" << endl;
    exit(EXIT_FAILURE);
}

void getArgs(int argc, char* argv[], uint64_t &timeout, bool &trace, char* traceFile){

    trace = false;
    timeout = 1;

    struct option argOptions[] = {
        { "trace", required_argument, 0, 1},
        { "timeout", required_argument, 0, 2},
        { "help", no_argument, 0, 3}
    };

    int idx;
    while ( (idx = getopt_long(argc, argv, "", argOptions, NULL)) != -1 ){
        switch(idx){
            case 1:
                trace = true;
                strncpy(traceFile, optarg, PATH_LENGTH);
                break;
            case 2:
                timeout = strtol(optarg, NULL, 0);
                break;
            default:
                usage(argv[0]);
        }
    }
}

int main(int argc, char** argv, char** env) {

    Verilated::commandArgs(argc, argv);

    bool trace;
    char traceFile[PATH_LENGTH];
    uint64_t timeout;

    getArgs(argc, argv, timeout, trace, traceFile);

    CXX_TOP_LEVEL* top = new CXX_TOP_LEVEL ();

#ifdef TRACE
    VerilatedFstC* tfp = new VerilatedFstC;

    if(trace){
        Verilated::traceEverOn(true);
        top->trace(tfp, 99);
        tfp->open(traceFile);
    }
#endif

    int main_time = 0;
    top->clk = 0;
    top->reset = 1;

#ifdef TRACE
    if(trace) tfp->dump(main_time);
#endif

    top->eval();
    while (!Verilated::gotFinish() && main_time < timeout) {
        main_time += 1;

        top->clk = 1;
        top->eval();

#ifdef TRACE
        if(trace) tfp->dump (main_time);
#endif

        main_time += 1;
        top->reset = 0;
        top->clk = 0;
        top->eval();

#ifdef TRACE
        if(trace) tfp->dump (main_time);
#endif
    }

    if(main_time >= timeout) cout << "ERROR:␣timeout" << endl;

#ifdef TRACE
    if(trace) tfp->close();
#endif
    delete top;

    exit(0);
}
```

A.5 Run script: **run**

```
#!/bin/bash

export DOCKER_NAME=run

export RTL_TOP=$1
shift
export TARGET_PATH=$TBROOT/$RTL_TOP/build

echo "Creating␣run␣container"
docker run -w $REPOROOT -i -t -d -v $REPOROOT:$REPOROOT --name=$DOCKER_NAME $DOCKERTAG
```

```
echo "Running␣Verilog"
run_cmd="$TARGET_PATH/V$RTL_TOP␣$*"
echo $run_cmd
docker exec -t -w $REPOROOT $DOCKER_NAME bash -c "$run_cmd"
echo "Removing␣container"
docker stop $DOCKER_NAME > /dev/null
docker rm $DOCKER_NAME > /dev/null
```

A.6 View script: **view**

```
#!/bin/bash

export DOCKER_NAME=run

export RTL_TOP=$1
shift
export TARGET_PATH=$TBROOT/$RTL_TOP/build

echo "Creating␣run␣container"
docker run -w $REPOROOT -i -t -d -v $REPOROOT:$REPOROOT --name=$DOCKER_NAME $DOCKERTAG
echo "Running␣Verilog"
run_cmd="$TARGET_PATH/V$RTL_TOP␣$*"
echo $run_cmd
docker exec -t -w $REPOROOT $DOCKER_NAME bash -c "$run_cmd"
echo "Removing␣container"
docker stop $DOCKER_NAME > /dev/null
docker rm $DOCKER_NAME > /dev/null
```