# Database Accelerator

*Codename:dpu*

CSEE 4840 Embedded System Design
Spring 2014
Project Report

Andrea Lottarini (al3125)

## Abstract

For my project, I designed and implemented a database processing unit in hardware. This project is complementary to a software based compiler and runtime system that I already developed for another project. Using the aforementioned compiler we are able to translate SQL queries into a sequence of hardware instructions which are then processed by our custom logic which I developed for this class. This project can be divided into two major components: a set of hardware “tiles” corresponding to most of relational algebra operators, e.g. Aggregation, Join, etc and the software (drivers and test programs) that enable execution of queries on the hardware.

# I. Introduction

## Motivation

We are in the era of big data. User data (social networks, cloud storage etc) is currently produced at a rate which makes data mining almost infeasible. Nonetheless this data is extremely valuable.

Furthermore, the end of dennard scaling is stalling the performance of computer chips. A larger and larger portion of a chip will have to be powered off in order to be within an acceptable power budget. This creates an opportunity for acceleration of interesting workloads. We can trade off chip area for specialized hardware that can accelerate specific application of interest. Therefore we decided to implement a prototype of an accelerator targeting sql queries.

## Scope of this report

We limit our report to the parts that are pertinent for the Embedded Systems class. We will focus on the hardware and drivers, as the compiler and runtime system are beyond the scope of this class.

# II. Hardware Design and Specification

## A case study

Our hardware is based on a set of heterogenous hardware blocks which we call “tiles”. These tiles closely match relational algebra operators, with some extensions to support certain specific sql operations. All these operators work on a streaming fashion - receiving some data on a set of input streams and producing data on a set of output streams. For the purposes of this project, we have implemented the following set of tiles: Boolgen, Filter, Joiner, Aggregator, Sorter, and Joiner.

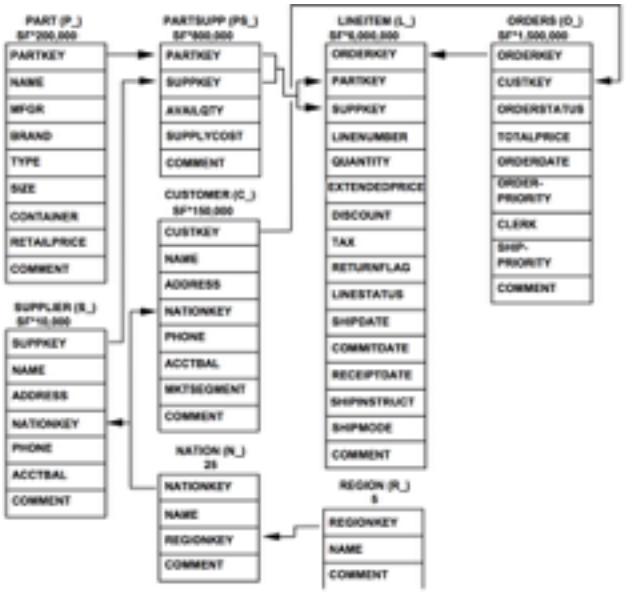



**FIG1: SCHEMA FOR THE TPC-H BENCHMARK**

We are going to introduce their functional specifications using a simple SQL query as example:

```
select  o.orderkey, sum( (l.extendedprice*(1-
l.discount) ) as revenue
  from    lineitem l,order o
    where l_discount > 0.1 and o.orderkey =
l.orderkey
            group by o.orderkey
```

**FIG2: EXAMPLE QUERY**

This query computes, for every order, the revenue coming from items discounted more than 10%.

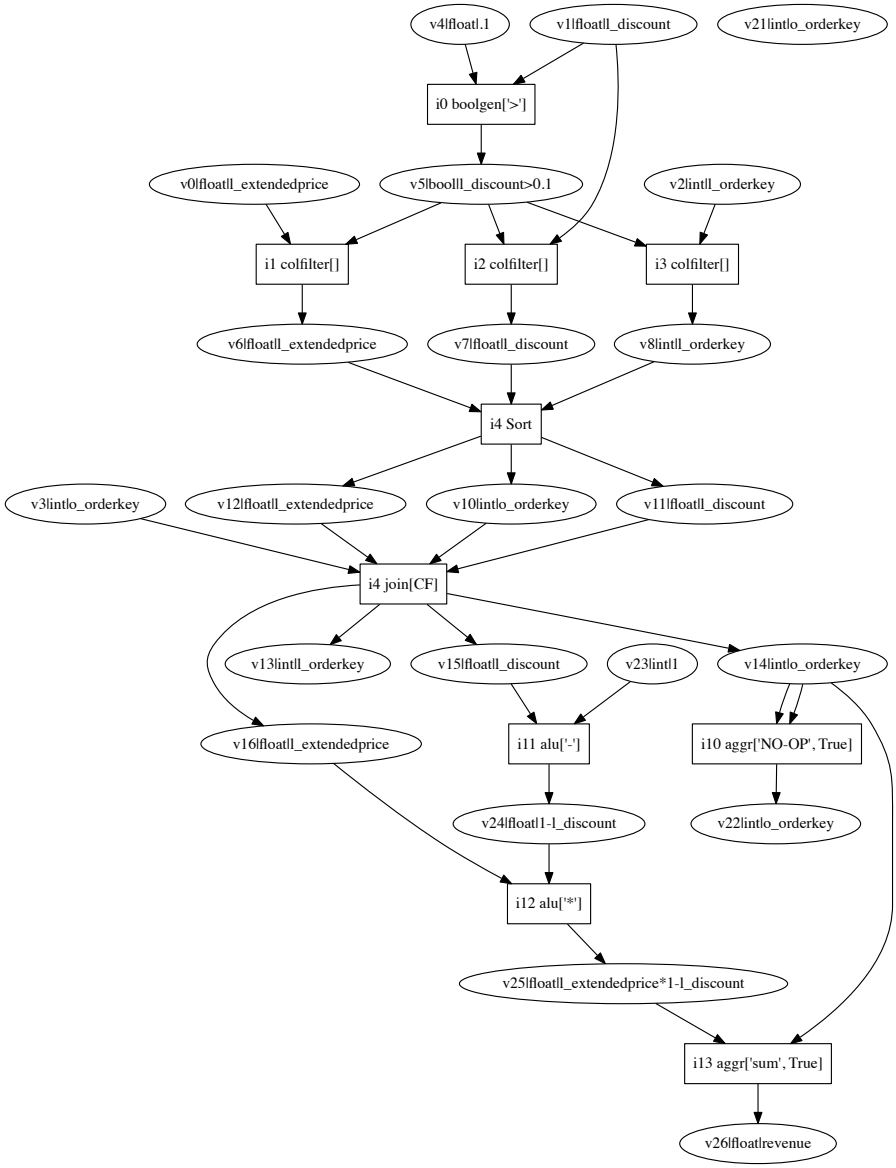



**FIG3: QUERY PLAN RELATIVE TO THE QUERY IN FIG2. RECTANGULAR BOXES INDICATE INSTRUCTIONS THAT CAN BE EXECUTED ON HARDWARE TILES. TILES LABELS HAVE THE FOLLOWING MEANING: *<UNIQUE_ID TILE_NAME [CONFIGURABLE ATTRIBUTES]>*. CIRCLE BOXES INDICATE DATA STREAMS FLOWING BETWEEN TILES. V0,V1,V2,V3 ARE THE INPUT COLUMNS WHILE V22 AND V26 ARE THE OUTPUT COLUMNS. VALUE LABELS HAVE THE FOLLOWING MEANING: <UNIQUE_ID | DATATYPE | VALUE NAME>**

## Tiles functional specification

By processing the query in FIG2 using a custom SQL compiler it is possible to obtain the query plan presented in FIG3. We are going to introduce the functional specification of each tile in the order in which it appears on the query plan. We can identify three phases that occur in the query plan: filter, join, and aggregate phase.

FILTER PHASE

In this phase the lineitem table has to be filtered to extract only the records where the condition *l_discount > 0.1* is satisfied.

### Boolgen

The first operation to be performed in the example query is to compute which records in the lineitem table has a discount greater than 0.1. This is done using the boolgen tile which will produce a bit for every record indicating whether it satisfies this property.
A boolgen tile has two inputs (in1,in2) and one output (out1). For every input element pair (in1[i],in2[i]) it test a programmable condition C(in1[i],in2[i]) and produce a single bit out1[i] indicating whether the condition evaluates to true or false[1]. It is possible to configure the boolgen at runtime to ignore the second input and use an internal constant ( also configurable ).
In the case of the query plan in FIG3 the boolgen tile i0 has been configured to use an internal constant set to 0.1. It will output 1 if the discount value received in the first input stream is bigger than 0.1.

### Colfilter

After computing which records in the lineitem table satisfies the *l.discount > 0.1* condition the records have to be filtered.
A filter tile has two inputs and one output. The first input stream is boolean while the other can be of arbitrary type. For every input element pair (in1[i],in2[i]) the filter produces out1[j] = in2[i] if in1[i] = 1. The number of elements produced by the filter element will be less or equal than the number of elements received at its inputs.
In the case of the query plan in FIG3 the values v6,v7,v8 (produced by colfilter tiles i1,i2,i3) corresponds to the records (for three different columns) of the line item table where the condition l_discount > 0.1 is satisfied.

JOIN PHASE

The join has to be performed between the order and the filtered lineitem table. This is a equi-join between two tables using the primary key of a candidate table (*o_orderkey* in the *orders* table) and a foreign key in a foreign table (*l_orderkey* in *lineitem*). There is no tile that can perform **arbitrary** joins however there is a joiner tile that can perform such equi-joins assuming all the columns from both tables are sorted according to the attribute used in the equality. This is the case for the *orders* table which is sorted on its primary key but it is not the case on the lineitem table which has to be sorted on the *l_orderkey* attribute first.

[1] We are referring to the i-th element received/produced on a input/output stream s with s[i]

### Sorter

The Sorter tile has a configurable number of inputs and outputs and it also has an internal parameter k which corresponds to the block size it can operate on. For this project we instantiated a 4 inputs 4 outputs sorter with a block size of 32.
Each block is sorted on each column where the first input has the highest priority and the last input has the smallest priority. As an example consider the following case of a 2x2 sorter operating on a block size of 8. Given the following two inputs:

in1: 6,5,3,1,8,12,1,45, 5,8,7,3,3,2,10,14 …
in2: 1,1,1,3,2,2 ,2,2 , 1,2,3,9,8,6,7 ,8 …

it all produce the following two outputs:

out1: 1,1,3,5,6,8,12,45, 2,3,3,5,7,8,10,14 …
out2: 2,3,1,1,1,2,2 ,2 , 6,8,9,1,3,2,7 ,8 …

Notice how the output is sorted on the in1 column first and then according to the in2 column.

### Joiner

The joiner tile performs equi-joins and has two preconditions:

- Input data is sorted on the attribute used for the equality testing
- Input1 attribute is a primary key,i.e. there exists at most a single instance of every possible value.

This tile has a configurable number of inputs and outputs, set for this project to 4.
The first input corresponds to the column which is primary (candidate) key while the second input corresponds to the column which is the foreign key. The other input columns are payload columns which can be either belong to the candidate table or the foreign table. As an example consider the case where in3 belongs to the candidate table and in4 belongs to the foreign table; the joiner will output each tuple ( in1[i] , in2[j] , in3[i], in4[j] ) such that i < len(in1) , j < len(in2) , in1[i] = in2[j].
Payload column configuration can be changed at runtime, e.g. it is possible to execute a join operation where all payload columns belong to the candidate table and subsequently reconfigure the tile to execute a join where all payload columns are from the foreign table.

Notice that the *sorter* tile will not be sufficient to sort tables of length bigger than the sorter block size. It is assumed that the software can perform a merge of sorted runs. On the other hand it is possible to sort tables which have a number of columns bigger than the number of sorter inputs simply by repeating the sorting procedure.

## AGGREGATE PHASE

After the join phase we have a table composed of all the tables specified in the from clause and satisfied all the condition specified in the where clause. We now have to perform all the operations specified in the select clause and in the group by clauses. In the case of the query plan in FIG3 we have to compute the revenue of every single item from the columns *l_discount*

and *l_extendedprice*. Afterwards we have to compute the total revenue by summing up the revenue of items in the same order.

### ALU

The ALU is used to compute basic arithmetic operations, it has two inputs and one output. For every input par (in1[i],in2[i]) it produces an element out[i] such that out[i] = f(in1[i],in2[i]) where f can be chosen at runtime between addition, subtraction, multiplication, and division. There is no control for overflow, so the runtime must handle whether output data is valid or not. The ALU is configurable so that it ignores the second input and produces out[i] = f(in1[i],constant) where the constant is a signed int configurable at runtime.

### Aggregator

The aggregator tile is used to implement the “Group By” clauses in sql. It has two inputs and one output and it is configurable at run time to perform either count, sum, min, max, or average for a given set of records associated with a given group.
The first is input is the group input while the second is the payload. For each input pair (g,d) the g element indicates the group to which data d belongs. The tile will output a single element o = f(d[i], .., d[j]) such that i < j , g[k] = g[i] for i < k <= j where f is chosen among sum,min,max,count and average.
The group datatype can be anything since only equality is tested within the tile to discern group boundaries. The data field must have numeric type for the result to be meaningful.

## Tile Design and Implementation

All the tiles share many similarities, at a minimum they have to handle input and output streams which we assume not synchronized, i.e. it is not guaranteed that the tile will receive from all input streams the input elements at the same time; only ordering within a stream is guaranteed. Similarly it is not expected that all receiving tiles of the output streams will be ready at the same clock cycle. Therefore there must be a mechanism to handle back pressure in a structured way among all these tiles. Moreover, if back-pressure is not handled correctly, e.g., by having a single long combinational path going backwards in the entire design, it could affect performance negatively.

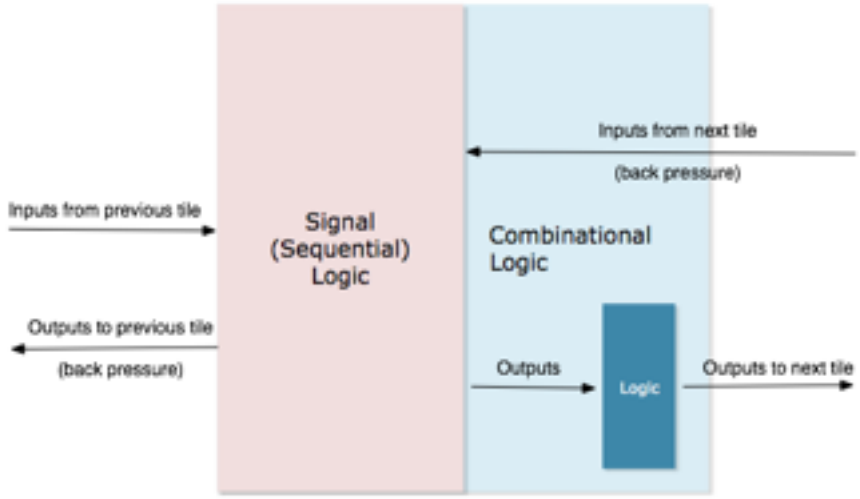



**FIG4: STRUCTURE OF A TILE**

We have settled for a design which we reused in all tiles. A sequential logic block at the input of the tiles handles back pressure and synchronization of the different input streams. When all inputs are valid and receiving tiles are all ready, some combinational logic, which is tile specific, will produce the next output stream elements. The sequential logic part of a tile (which we called a *buffer*) is similar to a *relay station*[2] which we modified for our purposes. By adding such buffers no single combinational path can exists due to back-pressure signals.

Another important design details is handling the termination of a stream, tiles have no notion of the length of a stream and rely instead on a single bit that we called *done* to signal termination. This bit is contained in the error field of avalon ST links; whenever a tile receive a done bit it will forward it on every outgoing link in order for downstream tiles to know about termination.

## Tile Implementation Details

We are going to describe here in more details the implementation of the hardware elements that compose our design. We are going to start with the buffer module, which is very important since it is present at the input of every tile.

### Buffer

The buffer has one input which is connected to an input stream of a tile and one output interface to the tile combinational logic. For the purposes of this project all link between tiles are Altera Avalon ST with ready latency 1. We are going to describe this communication protocol in order to understand the behavior of the buffer module.

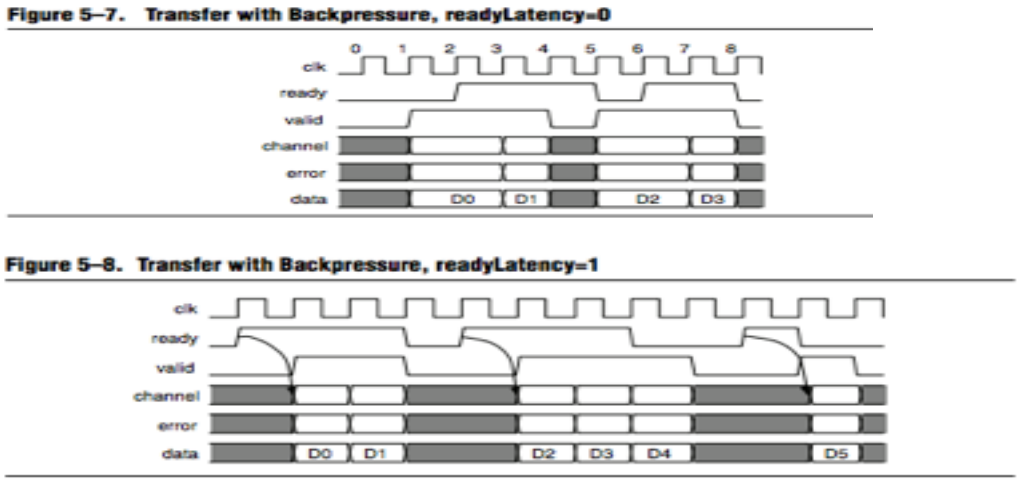



**FIG5: TIMING DIAGRAM OF AN EXCHANGE OF DATA BETWEEN TWO MODULES CONNECTED USING AN AVALON ST LINK. ON THE TOP FIGURE MODULES COMMUNICATE USING BACK PRESSURE WITH READY LATENCY 0 WHILE IN THE BOTTOM FIGURE READY LATENCY 1 IS USED.**

[2] L. Carloni et al., The Role of Back-Pressure in Latency-Insenstive Systems
http://www.cs.columbia.edu/~luca/research/rbilsENTCS06.pdf

Altera Avalon ST is a unidirectional synchronous interface for streaming links[3]. They are composed of a set of signals, in the case of FIG5:

- *Data*[DATA_WIDTH]: data flowing from source to sink.
- *Error*[ERROR_WIDTH]: possible error messages/ out of band signals.
- *Channel*[CHANNEL_WIDTH]: virtual channel on which data is transferred.
- *Valid*: single bit indicating whether the data transmitted in this clock cycle is valid.
- *Ready*: single bit **opposite** to the direction of the link, i.e. going from the sink to source. It indicates whether the source can accept more messages. It does not distinguish between data coming from different channels. The *ready latency* parameter corresponds to the latency (in terms of clock cycles) that the source has available to react to a change in the ready bit from the sink. Notice from FIG5 that the implications of using *ready latency 0* is that there should be a combinational path between source and sink modules - we wanted to avoid such condition therefore we used links with *ready latency 1*.

DATA_WIDTH, ERROR_WIDTH and CHANNEL_WIDTH are all parameters that can be configured when instantiating the link between modules.

On top of this, Altera Avalon ST has additional signal for the transmission of packets of data which are bigger than a single flit.

Buffer tiles have as input an Avalon ST sink and they store received data in a buffer with two slots. Buffered data is made available to the tile with a similar interface:

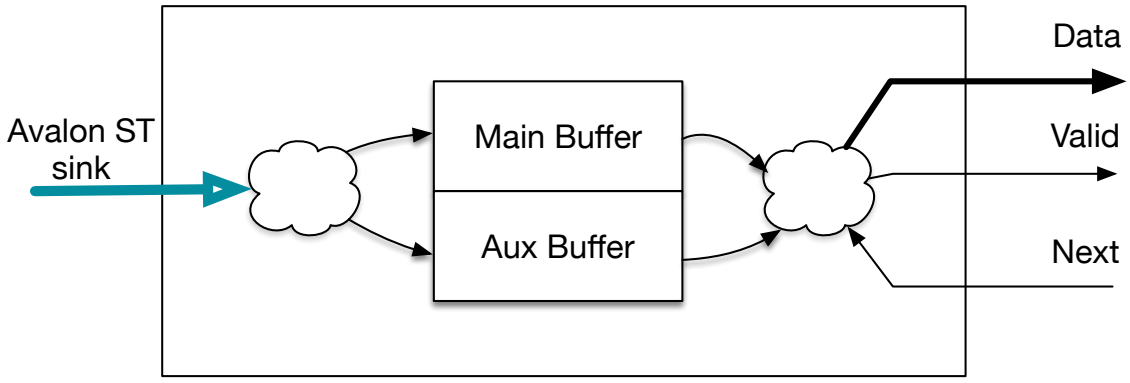



**FIG6: DIAGRAM OF A BUFFER**

In a steady state condition, the source keeps producing a new flit every clock cycle and the tile consume it therefore, only a single slot is used.
The second slot is used in case the source produces a new element that the tile can not consume, in this case the new element is stored in the second slot and the buffer dessert the outgoing ready bit, ensuring no data will be received from the source in the next clock cycle.

[3] Avalon ST reference…

Unless the buffer holds no valid data it will assert the outgoing valid signal, if the tile assert the ready_in bit the buffer will discard the current element since it has already been processed by the tile. In any case the presence of the buffer is transparent to the tile.
The behavior of the buffer is summarized by the state machine in the next figure:

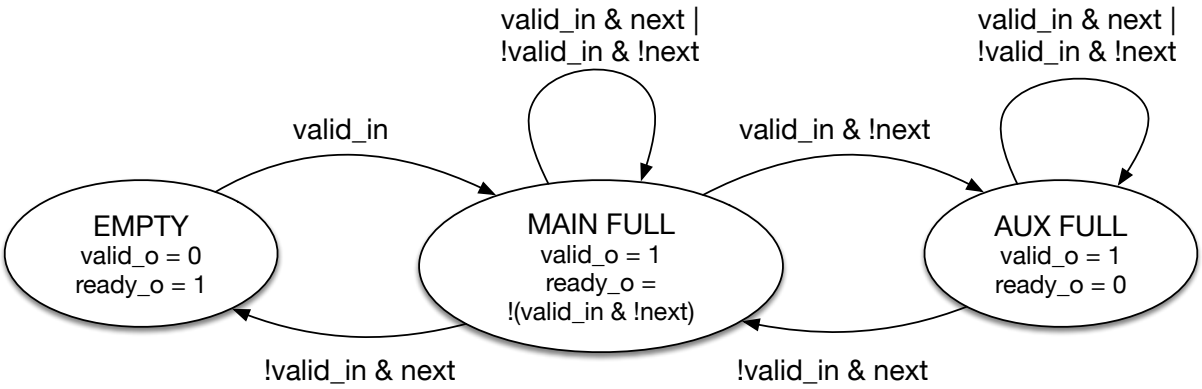



**FIG7: FINITE STATE MACHINE OF THE BUFFER MODULE**

## ALU

We are going to use the ALU tile as an example for the implementation of a generic tile. Both inputs of the ALU have buffers and the tile behavior is specified in a single *always_comb* block (as mentioned in FIG4). Notice how the done flag is forwarded to downstream tiles as soon as it is received, and the buffering of back pressure signal to avoid long wires going upstream on the pipeline. The presence of the input buffers make the tile internals very simple, no logic is present to handle back pressure signals directly only to selectively ask for a new element to the buffers. Tiles logic is composed of a single guarded action that it is executed when all inputs are ready as well as the downstream tile. Finally notice that we are using inferred multipliers and dividers so this design is not bind to a specific board or manufacturer.

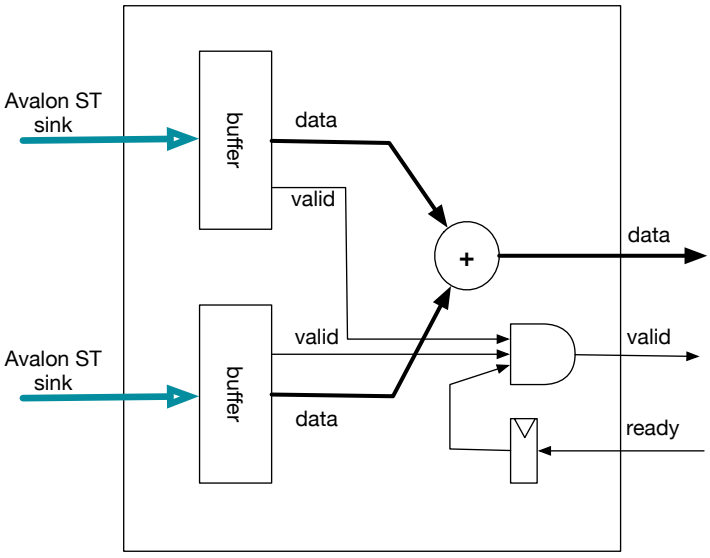



**FIG8: SIMPLIFIED DIAGRAM OF THE ALU**

```
always_ff @(posedge clk) begin

    was_rdy <= ready_i; //store the ready bit from the previous tile

end

always_comb begin
    if ( op < 3'd4 ) begin      //using both operands
      input_valid    = buf_valid1 && buf_valid2;
      done_output    = buf_done1 & buf_done2;
      second_operand = buf_col_2;
    end
    else begin                  //using internal constant
      input_valid    = buf_valid1;
      done_output    = buf_done1;
      second_operand = constant;
    end

    if ( was_rdy && input_valid )  begin
      valid_o = 1;       done_o = done_output;  //send result
      buf_ready1 = 1; buf_ready2 = 1;            //ask for new elements

      case(op[1:0])                              //switch on the operation
        //add
        2'b00:col_o = $signed(col_1) + $signed(second_operand);
        //minus
        …
      endcase
    end
    else begin
      done_o  = 0; valid_o = 0; col_o   = 0; //do not output
      ready1  = 0; ready2  = 0;              //do not shift inputs

    end // else: !if(was_rdy && input_valid)

end
```

**FIG9: CODE FROM THE ALU TILE.**
**NOTICE HOW THE READY INPUT FROM THE DOWNSTREAM TILE IS BUFFERED CONTINUOUSLY BREAKING POSSIBLE COMBINATORIAL PATHS.**
**TILE OPERATION IS GUARDED BY THE PRESENCE OF VALID INPUTS AND THE "READINESS" OF THE TILE CONNECTED DOWNSTREAM OF THE ALU.**

### Tile reconfiguration

Some tiles have the ability to reconfigure themselves at runtime, e.g. it is possible to have an ALU tile perform addiction between two columns and then reconfigure it do to multiplication between a column and a numeric constant.
For this design we decided to add an avalonMM interface to each tile, this interface is used to write from software the opcode of the operation to be performed.

Again consider the ALU tile as an example. It is possible to write an opcode in the range between 0 and 7 with the following meaning:

| | |
|---|---|
| 0 | Addition between two columns |
| 1 | Subtraction between two columns |
| 2 | Multiplication between two columns |
| 3 | Division between two columns |
| 4 | Addition between the first column and a constant |
| 5 | Subtraction between the first column and a constant |
| 6 | Multiplication between the first column and a constant |
| 7 | Division between the first column and a constant |

Other tiles have similar opcodes which can be summarized in the following way:

- Joiner: opcode will be a single bit for every payload columns; if the bit is 0 the associated column belongs to the foreign table otherwise to the candidate table.
- Aggregator: a different opcode for the operations: Min,Max,Count,Sum, and Average.

## Sorter

The *sorter* tile is more complicated than the others and we will briefly explain its differences. Sorting can not be strictly performed in a streaming fashion, it is necessary to buffer a chunk of elements from the stream to be sorted. We take the design of the sorter tile from [4].

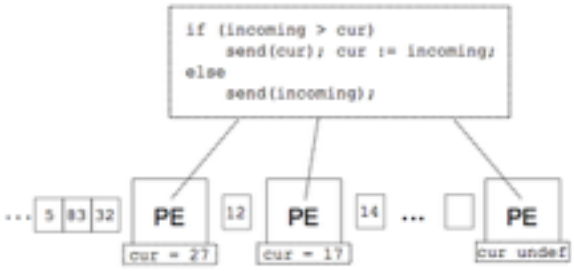



**FIG8: PIPELINE OF SORT TILES IN ACTION**

The design of the sorter is very simple. Each tile is equal and for each element received it checks whether it is bigger than the element it is currently holding. If that's the case it will store the newly received element, otherwise it will let it pass. Let's assume a pipeline composed of k equal tiles, after processing k elements the first tile will hold the biggest element, the second tile will hold the second biggest and so on.
This is sufficient to create sorted runs of a single column however what is necessary for our needs is to sort tables composed of many columns, possibly comparing multiple columns at once. We have seen in FIG3 a query plan which involved a sort operation on three columns using only a single column for the comparisons. Other queries might require comparison between multiple columns. Consider an equi-join between two tables which involves two columns, we would like to support this operation without having to stream all the columns in the table two times. Similarly an order-by or group-by operation can (and frequently does) operate on more than one column.

In order to satisfy this requirement we arranged the sorter tile as a mesh. Each row is in charge of sorting a single column however different rows have a different priority. The top row of the mesh will be the primary sorting column, in case of equal elements the data will be sorted according to the second row and so on.
In order to achieve this, every sorter tile has an additional input and output for "commands". Every top row tile will send commands to the tile below indicating whether they should swap the buffered element with the incoming element. In case of equal elements the top row will communicate that it doesn't know what to do therefore, it will be for the second row to determine it by comparing the stored and received element. Similarly all tiles in the second row will communicate the action they performed to the third row and so on.

---

[4] Parashar et al., *Triggered Instructions: A Control Paradigm for Spatially-Programmed Architectures*, ISCA 2013

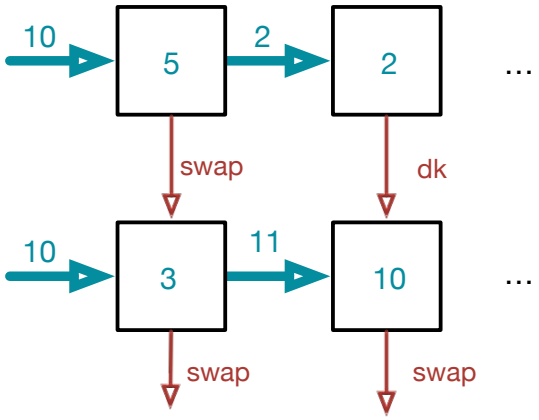



**FIG8: MESH OF SORT TILES.**
**NOTICE HOW TILES IN THE TOP ROW SEND COMMANDS TO THE SECOND ROW.**

### Other Tiles

The design of the remaining tiles ( boolgen, colfilter, aggregator and joiner ) is very similar to the design of the ALU.
It is worth noting that the design of the ALU and the aggregator somewhat intersect since they both perform arithmetic operations on the incoming stream of data. Similarly the aggregator needs to perform comparisons between elements of the data stream, a function which the boolgen tile already performs.
It would be interesting to consider an aggregation of all these functionalities in a single tile that can be configured to perform each one of this operations dynamically at runtime.

# III. System Design and Integration

The tiles described in the previous section can be arbitrarily combined to form complex queries. Instead of sticking to a single design we decided to implement a system where a single instance of each tile is offered. In order to pass values from software to the hardware we attached an Altera FIFO to every input and output interface of the tiles.

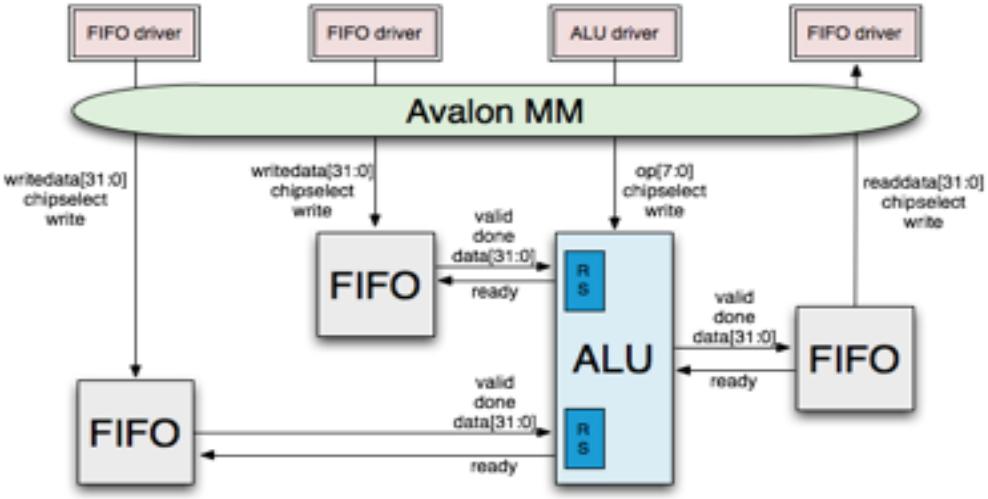
## Altera FIFO IP

The Altera FIFOs can buffer up to 8192 records. It can receive data from an Avalon MM and send it to an Avalon ST link (and viceversa). For our base implementation we used 16 elements fifos. Altera FIFOs provide two Avalon MM interfaces: a status and a data interface. The status interface should be used to first check if the FIFO can accept more data (in case of a write) or contains data (in case of a read). This is necessary since writing to a full FIFO or read from an empty one would cause an error.

Another necessary step is sending done bits at the end of the stream. This requires to add packet information[5]. The fifo will provide two addresses on the data interface, one for the actual data and one for the channel/error/packet bits. We are going to explain this in more details in the next section.

### Design Overview

Here is a simple design overview using an ALU as an example tile:




# IV. Software Design

When designing the software for our cpu system we recognized two possible scenarios.

- Considered every tile as a different device with its own entry in the device tree. Therefore, develop a single driver for each tile each with its own specific functions.
- Consider the dpu, with its set of tiles, as a single device with a single entry in the device tree. All the memory mapped interface will take a contiguous space in memory. The device driver would be agnostic to which tile it is sending data and it is responsibility of the user to specify a valid offset within the memory mapped space of the dpu.

We concluded that the second approach was more reasonable in the long run for two reasons. There is not a *single* dpu as any mix of different tiles can constitute a valid dpu. Moreover the functionalities required for the device driver of a tile are very similar to one another. The cornerstone is the ability to enqueue and dequeue data from the ALTERA FIFOs that feed data to the tiles. It will be up to user code to specify to which fifo read or write.

[5] www.altera.com/literature/ug/ug_embedded_ip.pdf

## Driver

### FIFO

Support for the Altera FIFO is the core of the HW/SW interface for our design. The FIFO driver specifies three types of ioctl commands that can be called by user code.

```
#define FIFO_WRITE_DATA _IOW(FIFO_MAGIC, 1, opcode *)
#define FIFO_READ_DATA  _IOR(FIFO_MAGIC, 2, opcode *)
#define FIFO_READ_STATUS _IOR(FIFO_MAGIC, 3, int*)
```

The last is the simplest one as it returns to user code the number of elements currently stored in the FIFO. This can be used for debugging purposes in user code.
FIFO_WRITE_DATA and FIFO_READ_DATA handles transfers back and forth from the fifos. Both commands take a struct opcode as an argument:

```
typedef struct {
   unsigned short length;
   unsigned char done;
   unsigned char dest;
   int* buf;
} opcode;
```

*buf* points to the user space buffer.
*length* specify the length of the transfer in words: 4bytes, corresponding to the data size for the Avalon link.
*dest* indicates the fifo unique id.
*done* specify whether this chunk of data is at the end of the stream.

In the driver code a list of locations is stored as an array. This is used to translate the dest field specified in the opcode to an actual location in memory. These offsets are obtained from Qsys. Here is the table we used for this project:

```
unsigned int fifo_offset[FIFOS]       = {
                    0x138, 0x130, 0x128,  0x120,  0x110,  0x118,  0x108,  0x100, /*sort*/
                    0x3c0, 0x3d0, 0x3c8, /*boolgen*/
                    0x3e8, 0x3e0, 0x3d8,  /*colfilter*/
                    0x428, 0x420, 0x418, 0x410,  0x408,   0x400,  0x3f8,0x3f0, /*join*/
                    0x3b8 , 0x3b0, 0x3a8, /*alu */
                    0x18 , 0x10, 0x8  /*aggr */
};
```

Therefore, if the user supplied an opcode op = {100,0,2,&buf} to a FIFO_WRITE_DATA _IOW it would correspond to a write of 100 elements from buffer *buf* to the location 0x128 which corresponds to the third input column of the sorter.

Next we are going to look at the ioctl command to write data to a fifo:

```
    case FIFO_WRITE_DATA:
        // first check that the fifo is not full
        fill = ioread32(stat_addr);

        to_write = MIN(FIFO_SIZE - fill , op->length);

        if ( to_write > 1 ){
            printk("Writer Driver 0 - writing specs %d\n",START_PACKET_CHANNEL0);
            iowrite32( START_PACKET_CHANNEL0, spec_addr );
            /* trusting the user buffer to avoid coping that */
            for ( i = 0 ; i < to_write ; i++ ){
                if ( i == (to_write - 1) ){ /* write the end packet flag before wrting the last int*/

                    if (op->done && to_write == op->length){        /* that was ALSO the last transfer for
this stream ad I wrote it all down*/
                        printk("Writer Driver 0 - writing specs %d\n",DONE_END_PACKET_CHANNEL0);
                        iowrite32(DONE_END_PACKET_CHANNEL0, spec_addr);
                    }else{
                        printk("Writer Driver 0 - writing specs %d\n",END_PACKET_CHANNEL0);
                        iowrite32(END_PACKET_CHANNEL0, spec_addr);
                    }
                }
                printk("Writer Driver 0 - writing %d\n",op->buf[i]);
                iowrite32( op->buf[i], data_addr);
            }
        }else{
            //SINGLE FLIT CASE OMITTED
        }

        /* write back in the op struct how many int were actually sent */
        op->length = to_write;

        break;
```

**FIG9: SIMPLIFIED CODE OF THE IOWRITE OPERATION**

Notice that no copy is performed of user supplied data structure for performance's sake. To communicate the outcome of the write he driver overwrites the user supplied opcode struct. Consider as an example a write request from the user of 100 elements which also happens to be the last one of a stream (user sets the done bit). The driver will check the status interface first and if it can only write 50 it will overwrite the op->length field with 50 and set op->done to 0.

### Test Program

We also included a test program that shows how our system can be used to execute a sample query. For the purposes of this project we added it to show the correct behavior of the whole system. The execution of this program corresponds to the execution of this query:

```
select t1.id, min(t2.id*t2.y)
from   t1, t2
where  t1.id == t2.x and
       t1.id == t2.id
```

All the functionalities in the filter, join and aggregate phase are tested.

# V. Validation and Testing

Validation and testing was done via a set of scripts.
Unit tests were developed for each tile. For these tests we used both modelsim and Altera system console.
The test program introduced in the previous section was instead used as regression testing of the whole system.

During development Verilator was used as a firs tool to check that the verilog was well formed. subsequently signal timing and the correct functioning of backpressure was tested using Modelsim, and comparing the waveforms with an expected output.
During system integration in qsys we used both the system console and SignalTap as a tool to check correctness.
Finally drivers were developed and the whole hw/sw system has been tested end to end.

## Verilator

The verilator testing was mainly to ensure our modules would compile. This allowed us to quickly find and sort out syntax errors, and make sure that small changes did not result in broken code. Unfortunately Verilator does not fully support some of the system verilog features, most notably it has no support for interfaces which we used extensively in the design of the sorter.

## Modelsim

We use Modelsim to debug our signal spec and test back pressure. This was essential to check that our tiles adhere to the Avalon ST ready latency 1 protocol. With modelsim, we were able to quickly iterate through code revisions until we achieved the correct behavior.

After the no errors were evident we produced more strict unit test scripts, this phase was time consuming but fundamental as it made possible to discover unexpected corner cases early on. For each tile we produced a test bench which would drive the tile with constrained random stimuli. The test bench will have to adhere to the ready latency protocol. We also used python scripts to the instantiate and run modelsim simulations:

```
    seed            = random.randint(0,1000000)
    send_threshold  = random.randint(0,9)
    ready_threshold = random.randint(0,9)
    op              = random.randint(0,7)
    constant        = random.randint(0,1000)

    #call modelsim
    cmd  = "vsim -c alu_tb -gSEED={0:<8d}       \
                  -gSEND_THRESHOLD={1}      \
                  -gREADY_THRESHOLD={2}     \
                  -gALU_OP={3}              \
                  -gALU_CONSTANT={4:<4}     \
-do \"run -all\"".format(seed,send_threshold,ready_threshold,op,constant)
```

**SCRIPT TO INITIATE A RANDOM SIMULATION OF THE ALU TILE USING MODELSIM.**

The results produced during these test runs were then analyzed in the same python script to check for functional correctness.

## System Console

With system console, we were able to see test whether our tiles matched correctly with the rest of the system. System Console was our first test that involved the Altera IP, including FIFOs and generated modules. We found that integration of different IPs in Qsys is very error prone. Therefore we produced for each tile a script to test whether data is correctly sent and received. These tcl scripts do not use random stimuli to check functional correctness as this is assumed from the previous phase of testing in modelsim.

## Signal Tap

We used Signal Tap to make sure the simulated behavior matched the expected behavior. It allowed us to look into the registers on the FPGA and ensure that all of our tiles were functioning properly.

### Software Testing

This was the final step in validating our code. After thawing a functional driver for the system we started devising a test program that would correspond to a SQL query. Therefore validating that every tile was working correctly and more importantly that they were able collectively to perform a meaningful computation.

# VI. Results

## Performance Evaluation of the FIFO driver

Performance of streaming data in and out of the Altera FIFOs is crucial for our project. We therefore implemented a fifo testbench by connecting two fifos together and having the CPU continuously write in one fifo and read from the other.

An initial implementation of the driver would transfer only a word (32 bits) at a time and had a transfer rate of 2MB/s (2MB/s in and 2MB/s out). The implementation presented beforehand can transfer data 4 times that fast. We also tested the impact of the length of the fifo and these are the results:

| FIFO Length | Bandwidth (MB/s) |
|---|---|
| 16 | 6.981309638 |
| 64 | 8.108659278 |
| 256 | 8.381337444 |

Increasing the FIFO size helps amortizing some fixed cost of the transfers (e.g. traps to perform the ioctl) however the bandwidth remains very limited.

## Resource Utilization

This is the fitter summary

```
; Logic utilization (in ALMs)     ; 29,194 / 41,910 ( 70 % )                ;
; Total registers                 ; 33799                                    ;
; Total pins                      ; 289 / 499 ( 58 % )                       ;
; Total virtual pins              ; 0                                        ;
; Total block memory bits         ; 15,184 / 5,662,720 ( < 1 % )             ;
; Total DSP Blocks                ; 2 / 112 ( 2 % )                          ;
; Total HSSI RX PCSs              ; 0 / 9 ( 0 % )                            ;
; Total HSSI PMA RX Deserializers ; 0 / 9 ( 0 % )                            ;
; Total HSSI TX PCSs              ; 0 / 9 ( 0 % )                            ;
; Total HSSI TX Channels          ; 0 / 9 ( 0 % )                            ;
; Total PLLs                      ; 0 / 15 ( 0 % )                           ;
; Total DLLs                      ; 1 / 4 ( 25 % )                           ;
+---------------------------------+-----------------------------------------+
```

We can see how the design uses mostly LUTs and has very little use for BRAM. Only two DSPs are used to perform multiplication and division.
This is consistent with the fact that memory is only used in the Altera FIFO which are present in very small number: there are 28 fifos instantiated, each with 16 entries of 32 bits which corresponds to 14,000 bits.
Buffers contained in the tiles uses LUTs instead and can not utilize BRAM.

If we analyze more in details the Logic utilization we have:

```
; Logic utilization (ALMs needed / total ALMs on device)     ; 29,194 / 41,910     ; 70 %  ;
; ALMs needed [=A-B+C]                                       ; 29,194              ;      ;
;     [A] ALMs used in final placement [=a+b+c+d]            ; 29,883 / 41,910     ; 71 %  ;
;         [a] ALMs used for LUT logic and registers          ; 13,269              ;      ;
;         [b] ALMs used for LUT logic                        ; 14,336              ;      ;
;         [c] ALMs used for registers                        ; 2,278               ;      ;
;         [d] ALMs used for memory (up to half of total ALMs) ; 0                  ;      ;
;     [B] Estimate of ALMs recoverable by dense packing      ; 1,506 / 41,910      ; 4 %   ;
;     [C] Estimate of ALMs unavailable [=a+b+c+d]            ; 817 / 41,910        ; 2 %   ;
```

We can see how a considerable portion of ALMs are used as registers, which is expected.
Notice also that by synthesizing the sorter tile with a block size of 32 entries and four columns input/output the logic utilization is already 50%.

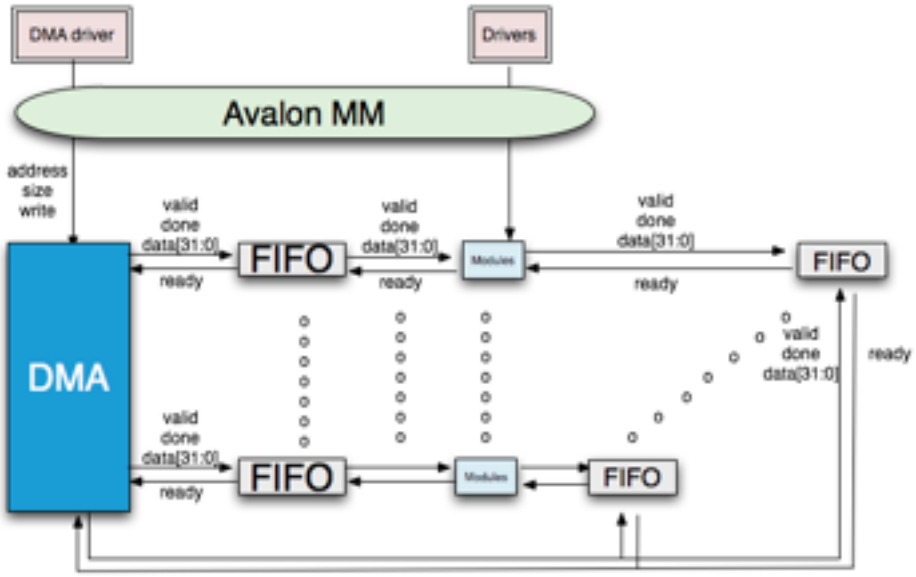
# VII. Conclusions and Future Work

We are quite satisfied with our current design as it proves that execution of sql queries in hardware is feasible and a limited set of hardware modules is sufficient to do so. However we recognize that there are ample possibilities to extend our work to make it better in terms of performance and programmability.

## Future HW Design

From the analysis of the area utilization it seems clear that all the tiles (except for the sorter) occupy a small area, it would be interesting to try to merge all this into a single reconfigurable entity. We could pay a small penalty in terms of area overhead but we could gain in terms of programmability by having a simpler design and adapt much better to different query workloads. Since every tile will be reconfigurable to emulate the others we will not incur in cases where there are too many or too few tiles of a specific type for the workload at hand.

We were not able to use DMA. Having to transfer data from the processor is clearly inefficient. Here is the future design ideal, which uses a DMA to reduce this overhead:

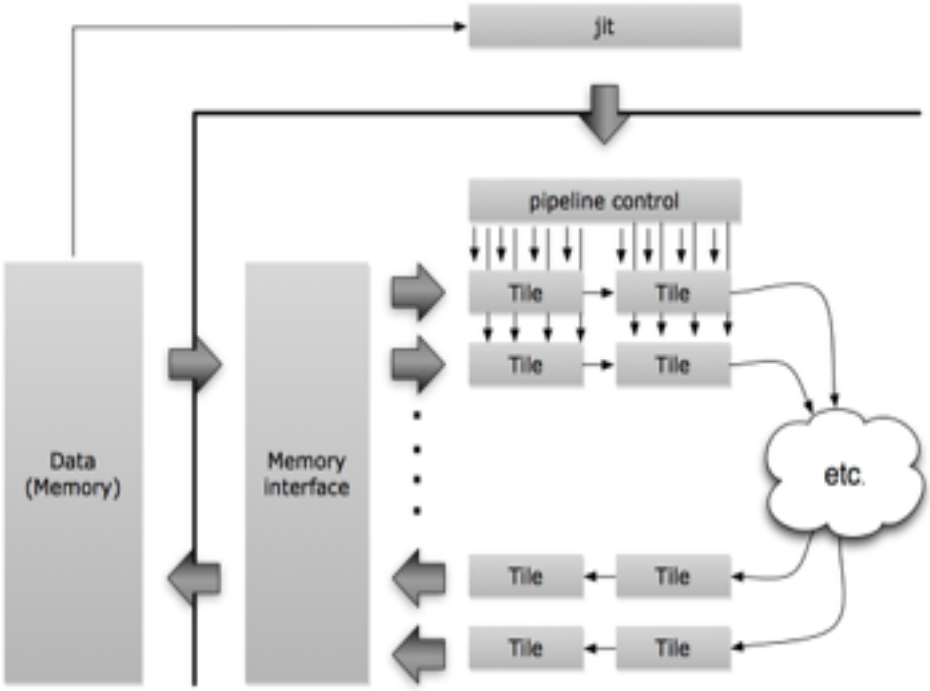



Moreover, we did not explore the implications of arranging the processing tiles in different topologies where they communicate directly between themselves. This should produce some interesting tradeoffs, most notably it is not clear if this interconnection network should be all to all and how much FPGA resources it will need.
Since we are using a standardized interface to communicate between tiles, we think the CONNECT: Configurable Network Creation Tool[6] could be used without having to write an ad-hoc network.

[6] http://users.ece.cmu.edu/~mpapamic/connect/

## Full Abstract System Overview

We should not expect that programmers will write programs like our test program. A full system should behave exactly like a DBMS henceforth sql queries should be translated into executable query plans.
The full system architecture with software control should rely on a JIT software that can execute these query plans .




## Performance Analysis

We think it would be interesting to compare the performance of our system with a standard DBMS. For the purpose of this analysis it would also be interesting to understand where the gains are instead of solely comparing the performance numbers. We could be better (or worse) than a standard system in many aspects, e.g., producing query plan, reducing power by eliminating the need for caches, increasing performance by computing some of the tile operations faster, etc.

# VIII. Miscellaneous

## Advice for future groups

### Hardware

Testing suits proved invaluable for catching bugs. Don't waste time synthesizing and then trying to debug, use unit and regression tests for verilator, modelsim, whatever you need to catch bugs early and often. Altera's IP documentation can be spotty, so look for examples wherever possible. Timing is critical, and is a common problem in digital hardware design, so utilize the resources available to you and your experience with drawing out and implementing (potentially) complex state machines. Code modularity is also critical. Implement common code as a submodule, so it can be fixed in one place, as opposed to having to modify every bit of code to debug.

### Qsys and Quartus

Qsys offers very limited support for SystemVerilog, to a smaller extent this is true for Quartus. On the other hand, simulation tools like modelsim support pretty much everything; if you don't test your design in qwys early on you might discover at a later time that the features you are used are not sysnthesizable. Most notably qsys does not support interfaces while quartus does. We found interfaces to be invaluable since they simplify the design of modules and reduce the possibility of errors. We ended writing code to auto generate wrappers for our modules and wished to know this in advance. Moreover, if not strictly necessary for your design, we suggest that you use qsys as little as possible, it makes the whole flow extremely cumbersome and error prone.

### Software

Writing drivers for embedded devices is tough. Start by a simple, inefficient version and make you way up to more complicated stuff. The very few examples you'll find online can be very valuable.