Table of Contents

Table of Contents
Abstract
I. Project Overview
   Motivation
   Scope
II. Design
   Hardware
      Relay Station
   Tiles
      Filter
      Joiner
      Aggregator
      ALU
   Hardware in progress
      Window Tile
   Deprecated Hardware
      Splitter
      Merger
      Colselect
      Stitch
      Concatenate
      Streamer
   Tile Design
   Altera FIFO IP
   Design Overview
   Future Design
   Full Abstract System Overview
Software
   Drivers
      FIFO
      Tile Drivers
III. Details of Design Specification
   Hardware
      Signal Specifications
      Hardware Abstraction
      Latency 0 vs Latency 1 vs Modified Latency 1
         Latency 0
         Latency 1
         Modified Latency 1
      Control
      Hardware Support for Modified Latency 1
Pipeline
- Trivial Pipeline (several single module pipelines)
- Single Pipeline

IV. Validation and Testing
- Verilator
- Modelsim
- System Console
- Signal Tap
- FPGA Synthesis
- Sample SQL Queries

V. Results
- Performance Evaluation of the FIFO driver
- Static Query output
  - ALU test
  - Boolgen-Filter-ALU Test

VI. Future Research
- Performance Analysis
- Pipeline Topology
- Heterogenous VS Homogenous design
- ASIC vs FPGA

VII. Miscellaneous
- Roles
  - Tim
  - Andrea
- Advice for future groups
  - Hardware
  - Software

Appendix. Source Code, Tests, Drivers, etc.
- Hardware
- Deprecated (by research team)
- Software

Tiles - Using Professor Edwards' Code
- alu.sv
- colfilter.sv
- boolgen.sv

Tiles with Separate Relay Station
- aggregator.sv
- coljoiner.sv
- relay.sv

Deprecated Tiles
- merger.sv
- splitter.sv
- colselect.sv
concatenate.sv
stitch.sv

Test Benches
alu_test.sv
boolgen_test.sv
relay_test.sv
coljoiner_test.sv
colfilter_test.sv
signal1_test.sv
signal2_test.sv

Testing Scripts
compile_all.sh
vsim_all.sh

Drivers
alu.c
alu.h
filter.c
filter.h
joiner.c
joiner.h
aggregator.c
aggregator.h
fifo#.c
fifo.h

Utility
Makefiles (FIFO as example)
benchmark.c
test-alu.c
test-b-f-a.c
Abstract
For our project, we designed and implemented a database processing unit in hardware. Along with a software based compiler, runtime system, and drivers, we are able to translate database operations to a sequence of hardware instructions which are then processed by our custom logic. The project can be divided into three major components: a software SQL compiler, a runtime system, and a set of ASIC tiles corresponding to several of the major SQL commands (i.e. Aggregation, Join). The compiler processes the sql query and develops a query plan for how to execute it. Then, based on the configuration of the DPU (i.e. topology, number of tiles, etc), and the data itself, the runtime system determines the pipeline configurations necessary to process the query. Query execution configures the pipeline and loads the correct memory addresses for streaming, and a single operation is executed. This repeats until the full query has been executed.

I. Project Overview

Motivation
We are officially 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. Therefore we decided to implement a prototype of an accelerator targeting sql queries.

Scope
We limit our project to the parts that are pertinent for the Embedded Systems class. We have focused on the tiles and drivers, as the compiler and runtime system are beyond the scope of this class.
II. Design

Hardware
Our hardware is based on a set of heterogenous tiles, which loosely maps relational algebra operators, with some extensions to support certain specific SQL operations. We considered exploring the idea of using a mesh of homogenous tiles (similarly to a systolic array). However, for the purposes of this project, we have limited ourselves to just the following set of heterogenous tiles.

Relay Station
Relay stations are critical for working with back pressure. See the Signal Specifications section below for more details. Essentially, relay stations buffer input records in each of the tiles below, and generate or propagate back-pressure as needed to ensure our pipeline is latency insensitive.

Tiles

Filter
The filter is composed of two tiles. A boolgen tile computes whether two records satisfy some operation (i.e. ==, !=, <, >, <=, >=), and generates a single bit output. This output bit is sent to the filter tile, which either passes along the input record, or does not, based on the value of the boolean bit. We restrict the filter to integer values only, for simplicity.

Joiner
The joiner tile performs inner-joins based on a number of comparators (the same set as for the boolgen). It takes a primary record and a foreign record as inputs for comparison, and emits 4 auxiliary record if the join condition is satisfied, shifting the foreign record while holding the primary constant (through back pressure). If it is not satisfied, it increments the primary record. The comparators here are also restricted to integer values.

Aggregator
The aggregator serves to implement the “Group By” clauses in SQL. It outputs either the count, sum, min, max, or average for a given set of records associated with a given group. It also allows us to compute a global count, sum, min, max, or average via a “group-by-all” signal. It takes in group records and data records, and emits the aggregation target as a record, along with the group of the target and the last column. All possible outputs are restricted to integer values.

ALU
The ALU is used to compute basic arithmetic operations. It supports addition, subtraction, multiplication, and division, along with subtract-by-one and one-minus-input operations. There is no support for overflow, so the runtime must handle whether output data is valid or not. We restrict the ALU to integer valued arguments, as with the other tiles.
Hardware in progress

Window Tile
The window tile handles a number of unique and essential functions, such as sorting and partitioning, as well as certain specific operations like block-nested loop-joins.

Deprecated Hardware
Over the course of the semester, we have eliminated certain tiles that were deemed unnecessary, or whose functionality was subsumed by the window tile. These tiles are still observed by our compiler, but we have omitted them from our testing. However, many of them are compatible with our current design, and were developed for this project, so deserve mention here. They may or may not make their way into later design iterations, as many of them are utility tiles and may not be necessary if we decide on a fixed data width design.

Splitter
Splitter tiles form the basis of a partition. Based on the output of some comparison to a value, a record is either emitted, or passed to another splitter. Thus, a chain of N splitters forms N pipelines, each of whose values satisfy the comparison. By splitting and then sorting, we can perform a partition sort. We can also use splitters to insertion sort. The comparisons supported are the same as for the boolgen/filter.

Merger
A merger tile does the opposite of a splitter tile. Based on some comparison, either one record is passed and the other waits, or vice versa. The comparators are {<,>,<=,>=}.

Colselect
Records are made up of columns. The colselect outputs one of the columns.

Stitch
The stitch tile takes in a number of columns, and outputs a single record.

Concatenate
This combines two records into a single record, concatenating the first N/2 bits of each.

Streamer
This is essentially a buffer, implemented as a circular array. We chose to use the Altera FIFO IP instead of these for simplicity.
**Tile Design**

Our tiles can be seen as a sequential logic block, which manages the timing, coupled with a combinational block, which manages the tile logic.

![Diagram of Tile Design](image)

**Altera FIFO IP**

We rely on the Altera IP for our input and output FIFOs. The Altera FIFOs can buffer up to 8192 records, and can cross from Avalon MM to Avalon ST. 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). Writing to a full FIFO or read from an empty one would cause an error.
Design Overview

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

Here is a simple design overview using multiple modules:
Future Design
Here is the future design ideal, which uses a DMA to reduce the hard core overhead:
Full Abstract System Overview
And finally, the full system architecture with software control:
Software

Drivers

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.

```c
#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 latter 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 a argument which points to the user buffer and specified the length of the transfer to perform and whether the stream is ‘done’. The done bit is used to indicate the end of a stream and it is propagated by all tiles until it reaches the output fifos.

The opcode struct is presented next:

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

Each fifo will have a struct associated:

```c
struct fifo_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers can be accessed in memory */

    struct resource status_res; /* register where I can read the status */
    void __iomem *status_virtbase; /* Where this register can be accessed in memory */
}
```

As mentioned earlier Altera FIFOs present two Avalon MM interfaces (one for control and one for data) which might not be mapped adjacently in memory (as shown in the following dts). Therefore two pointers/resource structs are used.

```c
fifo_0: fifo0@0x100000048 {
```
compatible = "ALTR,fifo0";
reg = < 0x00000048 0x00000008
    0x00000020 0x00000020 >;
reg-names = "in", "in_csr";
}; //end fifo@0x100000048 (fifo_0)

The probe function is similar to what done in lab3 with some minor changes to parse correctly the dts. Next we are going to look at the ioctl command to write data to a fifo:

```c
    case FIFO_WRITE_DATA:
        // first check that the fifo is not full
        fill = ioread32(dev.status_virtbase);

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

        //printk("Writer Driver - I received an order for %d writes and I can do %d\n",op->length,
        to_write);
        if ( to_write > 1 ){
            iowrite32( START_PACKET_CHANNEL0, dev.virtbase+4 );
            /* trusting the user buffer to avoid coping that too */
            for ( i = 0 ; i < to_write ; i++ ){
                if ( i == (to_write - 1) ){ /* write the end packet flag before wrting the last int*/
                    iowrite32(DONE_END_PACKET_CHANNEL0, dev.virtbase+4);
                }else{
                    iowrite32(END_PACKET_CHANNEL0, dev.virtbase+4);
                }
            }
            iowrite32( op->buf[i], dev.virtbase);
        }
        }else{
            // SINGLE PACKET CASE OMITTED FOR BREVITY
        }
        }
        /* write back in the op struct how many int were actually sent */
        op->length = to_write;
        break;
```
Notice that no copy is performed of user supplied data structure for performance’s sake. The driver overwrites the user supplied (suggested) length and done field of the 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.

**Tile Drivers**

All tiles have an Avalon MM interface that is used for configuration. This is again done via IOCTL. As an example consider the ALU which allows the user to specify the operation to be performed in the input data. Most tiles have only a few control bits, used for things like setting the operation for arithmetic or comparison operations, setting GROUP_BY_ALL for the aggregator, etc, all of which are bundled into a single byte of control. We use one byte as it is the minimum size synthesizable by the Quartus compiler. As such, we ignore the 4-5 (depending on the tile) most significant bits. One design enhancement we plan on making is allowing all arithmetic or comparison tiles to accept, and potentially emit, immediate values passed as 4 bytes via the Avalon MM interface.
III. Details of Design Specification

Hardware

Signal Specifications

Our design is based on the Avalon ST signal spec, with added support for a modified Latency-1 backpressure and a custom “Done” signal. Backpressure is the most important feature, and to this end, we implemented Relay Station modules as described in the paper below\(^1\), using a variant of Professor Edwards’ own design. These relay stations are embedded in each of our modules, and are essential for buffering data and propagating the back pressure signals during stalls. Their operation follows the following state diagram.

![State Diagram](image)

Latency 1
- Processing→Read into input buffer if valid, write out if buffer holding valid (valid_b)
- ReadAux→Read into input buffer if valid, if valid_b start stalling
- Stalling→buffer full, stall until ready
Latency 0 vs Latency 1 vs Modified Latency 1

Avalon ST defines two separate standards for timing, latency-1 and latency-0. They can be summarized by the following timing diagrams:

**Latency 0**
(from Altera)

*Figure 5–7. Transfer with Backpressure, readyLatency=0*

---

**Latency 1**
(from Altera)

*Figure 5–8. Transfer with Backpressure, readyLatency=1*

---

**Modified Latency 1**

*Figure 5–8. Transfer with Backpressure, readyLatency=1*

---

The difference between latency-1 and our modified latency-1 is the necessity of a second buffer. For latency-1, if ready is deasserted, valid data must still be processed that cycle, which is why data item D4 is processed despite ready being low. We require that ready be high when valid is raised, but do not require data be processed when ready is low, so our spec falls somewhere between latency-0 and latency-1, borrowing the most logical elements from both. In the example above, data item D4 will be sent until valid and ready are both asserted high in the same cycle.

**Control**

For each module, control signals follow the Avalon MM spec, and are controlled via their corresponding driver.
Hardware Support for Modified Latency 1
Altera’s FIFOs only support the latency-0 specification, and after much rewriting and debugging, our tiles have been configured to the modified latency-1 specification illustrated above. Thus, on input and output to a pipeline, the FIFOs require latency-0 modules connected, and our tiles require only modified latency-1 interconnect. To rectify these two, we configure our tiles in Qsys as latency-0. This forces Qsys to insert buffer stages between the Altera FIFOs and our own tiles, without affecting the functionality of our inter-tile communication protocol. The result is a seamless transition between the signal standard of Altera’s IP, and that of our own.
Pipeline
For this project, we looked only at short, trivial pipeline topologies. These consisted of 2-6 inputs, and 1-6 outputs, with only a few stages at most. As we will be doing a full design space exploration of topologies later, we kept to simple pipelines to validate timing and backpressure.

Trivial Pipeline (several single module pipelines)
Each FIFO block here may represent more than one FIFO tile, depending on how many inputs a module needs.
Single Pipeline
We restrict this to 2-input, 1-output and 1-input, 1-output tiles.
IV. Validation and Testing

Validation and testing was done via a set of scripts. We used these for unit and regression tests as we developed each tile, and made modifications to existing ones. These fell into several categories. We tested for compilability using Verilator, viewed waveforms and compared expected output with actual output to test signal timing and backpressure using Modelsim, ran post-synthesis tests to ensure the Modelsim behaviour matched the FPGA behavior using Signal Tap, and finally generated test data to stream through the custom hardware, comparing output with expected output.

**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.

**Modelsim**

We use Modelsim to debug our signal spec and test back pressure. This was essential, as our first version of tiles had back pressure implemented incorrectly. With modelsim, we were able to quickly iterate through code revisions until we achieved the correct behavior.

**System Console**

With system console, we were able to see how the system responded to different signals. We primarily used System Console to load the FIFOs via jtag, in order to verify that no records were dropped. Since the two forms of testing above were done only on our modules, System Console was our first test that involved the Altera IP, including FIFOs and generated modules.

**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.

**FPGA Synthesis**

This was the final step in validating our code. We essentially loaded the tiles into the pipeline, streamed data in, and checked if the output matched the expected operation performed on the input data. Given our limited set of tiles, we are able to execute only a small subset of possible SQL queries, however, this subset will expand as we develop further. Here are a few examples of queries our design can process:

**Sample SQL Queries**

```sql
SELECT col_1 + col_2          //This can be any of {+,−,/,*,col_1,col_2}
FROM table                   //table can have any dimension,
WHERE (col_1+col_2) > 4 && (col_1+col_2) < 20  //but software must adjust for
                                      //only 2 possible data streams

SELECT col1 FROM table
```
WHERE col1 < AVERAGE(col1) //Requires 2 passes
// can do {max,min,sum,count,ave}
GROUPBY col2 //Can also group by all (treated as in same group.
//finds table-wide max, min, sum, count, or average

SELECT * //
FROM table1 INNER JOIN table2 //max possible join width is 6-records
ON table1.key == table2.key //can be {==,!==,<=,>=,<,>}

V. 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. You can check this in the fifo_into_fifo_benchmark folder.

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:

<table>
<thead>
<tr>
<th>FIFO Length</th>
<th>Bandwidth (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>6.981309638</td>
</tr>
<tr>
<td>64</td>
<td>8.108659278</td>
</tr>
<tr>
<td>256</td>
<td>8.381337444</td>
</tr>
</tbody>
</table>

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.

Static Query output

For this, we configured our pipeline and streamed data in and out using our software. We did not reconfigure the pipeline, and only a single stream-in,stream-out was done, however, this was done in multithreaded bursts. Two very simple operations were tested, and the results can be found in the Appendix.
VI. Future Research

Performance Analysis
We want to be able to validate our software timing simulator using the FPGA prototype. This involves a number of steps, the first of which is ensuring that our estimated latency and bandwidth numbers are reasonable. Though the SoCkit board is a vastly different platform from our future one, it provides actual numbers, as opposed to simulated ones.

Pipeline Topology
We are going to perform a design space exploration of potential topologies in order to get the max performance per area and power, without sacrificing configurability. This will boil down to a number of tradeoffs, such as size of DPU vs number of data stream passes, flexibility vs energy consumes, etc.

Since we are using a standardized interconnect signal set, we are looking to utilize the CONNECT: Configurable Network Creation Tool to accelerate the creation process. This will be particular useful when we explore homogenous tile designs.

Heterogenous VS Homogenous design
We are exploring the viability of a homogenous data-flow tile design. Each tile could be configured in a number of ways based on the specific operation we are trying to use the hardware for, and control could flow in addition to data. Tiles may come in varying “sizes”, as we may not need certain area and power hungry operations (such as 32-bit division) on every tile. We will be using FPGAs to prototype this design.

ASIC vs FPGA
Our design is targeted toward ASICs not FPGAs, so we want to be able to eliminate Altera IP and SoCkit-specific elements of our code, such as the FIFOs.
VII. Miscellaneous

Roles

Tim
All SystemVerilog code, drivers for joiner, boolgen, aggregator, debugged ALU driver, project report and diagrams

Andrea
Software, drivers, some verilog, project presentation

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. The instructors’ have a wealth of experience with this. 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.

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.

Appendix. Source Code, Tests, Drivers, etc.

File list:
Hardware
alu.sv
colfilter.sv
boolgen.sv
coljoiner.sv
relay.sv
aggregator.sv
boolgen_test.sv
relay_test.sv
coljoiner_test.sv
colfilter_test.sv
signal1_test.sv → to test backpressure
signal2_test.sv → to test backpressure
comile_all.sh
vsim_all.sh

Deprecated (by research team)
merger.sv
splitter.sv
streamer.sv
colselect.sv
concatenate.sv
stitch.sv

Software
alu.c
alu.h
filter.c
filter.h
joiner.c
joiner.h
aggregator.c
aggregator.h
fifo#.c
fifo.h
benchmark.c
test-alu.c
test-b-f-a.c
Various Makefiles
module alu #(  
parameter COLUMN_WIDTH = 32,  
parameter RECORD_WIDTH = 128,  
parameter KEY_WIDTH = 32,  
parameter SORT_WIDTH_LOG2 = 2,  
parameter SPLITTER_COUNT_LOG2 = 3,  
parameter ADDRESS_WIDTH = 32  
)  
(  
input clk,  
input reset,  
/* inputs Avalon ST */  
input [COLUMN_WIDTH-1:0] in0, //input 1  
input [COLUMN_WIDTH-1:0] in1, //input 2  
input valid0,  
input valid1,  
input done0,  
input done1,  
output reg ready0,  
output reg ready1,  

//control in AvalonMM  
input logic [7:0] op,  
input logic write,  
input logic chipselect,  
input logic enable_i,  

/* Output, Avalon ST */  
output reg [COLUMN_WIDTH-1:0] out0, //output 1  
output reg valid,  
input reg ready,  
output reg done  
);  
reg [COLUMN_WIDTH-1:0] buffer0, buffer1; // Input buffers  
reg bufferdone0, bufferdone1;  
wire [COLUMN_WIDTH-1:0] arg0;  
wire argdone0;  
assign arg0 = (~ready0) ? buffer0 : in0; // Core argument from buffer or port  
assign argdone0 = (~ready0) ? bufferdone0 : done0; // Core argument from buffer or port  
wire [COLUMN_WIDTH-1:0] arg1;  
wire argdone1;  
assign arg1 = (~ready1) ? buffer1 : in1;  
assign argdone1 = (~ready1) ? bufferdone1 : done1; // Core argument from buffer or port  
reg [COLUMN_WIDTH-1:0] core0;  
reg coredone0;  
reg coreV; // Comb. outputs and valid sigs.  
wire argV;  
assign argV = (valid0~ready0)&&(valid1~ready1); // Do we have all the inputs?  
logic [2:0] op_b;  
always_comb begin
coreV = 'b0;
core0 = 32'bX;
coredone0 = 'bX;

if ( argV ) begin
  case(op_b)
    3'b000: { core0, coredone0, coreV } = { arg0+arg1, argdone0&&argdone1, 1'b1 }; 
    3'b001: { core0, coredone0, coreV } = { arg0-arg1, argdone0&&argdone1, 1'b1 }; 
    3'b010: { core0, coredone0, coreV } = { arg0*arg1, argdone0&&argdone1, 1'b1 }; 
    3'b011: { core0, coredone0, coreV } = { arg0/arg1, argdone0&&argdone1, 1'b1 }; 
    3'b100: { core0, coredone0, coreV } = { arg1, argdone0&&argdone1, 1'b1 }; 
    3'b101: { core0, coredone0, coreV } = { arg1-arg0, argdone0&&argdone1, 1'b1 }; 
    3'b110: { core0, coredone0, coreV } = { arg1/arg0, argdone0&&argdone1, 1'b1 }; 
    3'b111: { core0, coredone0, coreV } = { arg1/arg0, argdone0&&argdone1, 1'b1 }; 
  endcase
end
end

wire stopped;
assign stopped = ~ready & valid; // Must the output buffer hold its value?

wire sending;
assign sending = (~ready0 & ~ready1); // Are all the input buffers filled?

reg waiting = 1'b0; // Waiting to load an output buffer?
wire willWait;
assign willWait = stopped & (sending ? waiting : coreV);

always_ff @(posedge clk) waiting <= willWait;

always_ff @(posedge clk) begin
  if(reset) begin
    ready0 <= 'b1;
    ready1 <= 'b1;
  end
  if ( argV &! (willWait) ) begin // Are we done with the inputs?
    { buffer0, bufferdone0, ready0 } <= { 32'bX, 1'bX, 1'b1 }; // Empty the buffers
    { buffer1, bufferdone1, ready1 } <= { 32'bX, 1'bX, 1'b1 }; 
  end else begin
    if (ready0) { buffer0, bufferdone0, ready0 } <= { in0, done0, ~valid0 }; // Fill the buffers
    if (ready1) { buffer1, bufferdone1, ready1 } <= { in1, done1, ~valid1 }; 
  end
end

always_ff @(posedge clk) begin
  if(reset) begin
    valid <= 1'b0;
  end
  if ( !stopped ) begin
    if ( !sending || waiting )
      { out0, done, valid } <= { core0, coredone0, coreV };
    else
      { out0, done, valid } <= { 32'bX, 1'bX, 1'b0 }; 
  end
end

always_ff @(posedge clk) begin
  if(reset)
    op_b <= 'b0;
  else if (chipselect && write) begin
    op_b <= op[2:0];
  end
end
module filter #(
  parameter COLUMN_WIDTH = 32,
  parameter RECORD_WIDTH = 128,
  parameter KEY_WIDTH = 32,
  parameter SORT_WIDTH_LOG2 = 2,
  parameter SPLITTER_COUNT_LOG2 = 3,
  parameter ADDRESS_WIDTH = 32
)
(
  input clk,
  input reset,
  /* inputs Avalon ST */
  input [COLUMN_WIDTH-1:0] in0, //input 1
  input valid0,
  input done0,
  output reg ready0,

  //control in AvalonMM
  input logic [7:0] op,
  input logic write,
  input logic chipselect,
  input logic enable_i,

  /* Output, Avalon ST */
  output reg [COLUMN_WIDTH-1:0] out0, //output 1
  output reg valid,
  input reg ready,
  output reg done
);

reg [COLUMN_WIDTH-1:0] buffer0;       // Input buffers
reg bufferdone0;

wire [COLUMN_WIDTH-1:0] arg0;
wire argdone0;
assign arg0 = (~ready0) ? buffer0 : in0; // Core argument from buffer or port
assign argdone0 = (~ready0) ? bufferdone0 : done0; // Core argument from buffer or port

reg [COLUMN_WIDTH-1:0] core0;
reg coredone0;
reg coreV;   // Comb. outputs and valid sigs.
wire argV;
assign argV = (valid0~ready0); // Do we have all the inputs?

logic [2:0] op_b;

always_comb begin
  coreV = 'b0;
  core0 = 32'bX;
  coredone0 = 'bX;

  if ( argV ) begin
    if(in0[0]) { core0, coredone0, coreV } = { arg0, argdone0, 1'b1 }
  end
end

wire stopped;
assign stopped = ~ready & valid; // Must the output buffer hold its value?
wire sending;
assign sending = (~ready0); // Are all the input buffers filled?

reg waiting = 1'b0; // Waiting to load an output buffer?
wire willWait;
assign willWait = stopped & (sending ? waiting : coreV);

always_ff @(posedge clk) waiting <= willWait;

always_ff @(posedge clk) begin
  if(reset) begin
    ready0 <= 'b1;
  end
  if (argV && !(|willWait)) begin // Are we done with the inputs?
    if (ready0) {buffer0, bufferdone0, ready0} <= {32'bX, 1'bX, 1'b1}; // Empty the buffers
  end
end

always_ff @(posedge clk) begin
  if(reset) begin
    valid <= 1'b0;
  end
  if (!stopped) begin
    if (!sending || waiting)
      {out0, done, valid} <= {core0, coredone0, coreV};
    else
      {out0, done, valid} <= {32'bX, 1'bX, 1'b0};
  end
end

always_ff @(posedge clk) begin
  if(reset) begin
    op_b <= 'b0;
  end
  if (chipselect && write) begin
    op_b <= op[2:0];
  end
end

endmodule

boolgen.sv

module boolgen #(parameter COLUMN_WIDTH = 32,
parameter RECORD_WIDTH = 128,
parameter KEY_WIDTH = 32,
parameter SORT_WIDTH_LOG2 = 2,
parameter SPLITTER_COUNT_LOG2 = 3,
parameter ADDRESS_WIDTH = 32)
(
  input clk,
  input reset,
)

/* inputs Avalon ST */
inpu [COLUMN_WIDTH-1:0] in0, //input 1
input [COLUMN_WIDTH-1:0] in1, //input 2
input valid0,
input valid1,
input done0,
input done1,
output reg ready0,
output reg ready1,

//control in AvalonMM
input logic [7:0] op,
input logic write,
input logic chipselect,
input logic enable_i,

/* Output, Avalon ST */
output reg [COLUMN_WIDTH-1:0] out0, //output 1
output reg valid,
input reg ready,
output reg done
);

reg [COLUMN_WIDTH-1:0] buffer0, buffer1; // Input buffers
reg bufferdone0, bufferdone1;

wire [COLUMN_WIDTH-1:0] arg0;
wire argdone0;
assign arg0 = (~ready0) ? buffer0 : in0; // Core argument from buffer or port
assign argdone0 = (~ready0) ? bufferdone0 : done0; // Core argument from buffer or port

wire [COLUMN_WIDTH-1:0] arg1;
wire argdone1;
assign arg1 = (~ready1) ? buffer1 : in1;
assign argdone1 = (~ready1) ? bufferdone1 : done1; // Core argument from buffer or port

reg [COLUMN_WIDTH-1:0] core0;
reg coredone0;
reg coreV; // Comb. outputs and valid sigs.

wire argV;
assign argV = (valid0|~ready0)&(valid1|~ready1); // Do we have all the inputs?

logic [2:0] op_b;

always_comb begin
  coreV = 'b0;
  core0 = 32'bX;
  coredone0 = 'bX;
  if ( argV ) begin
    case(op_b)
      3'b000: { core0, coredone0, coreV } = { 32'b001, argdone0&&argdone1, 1'b1 };
      3'b001: { core0, coredone0, coreV } = { (arg0==arg1), argdone0&&argdone1, 1'b1 };
      3'b010: { core0, coredone0, coreV } = { (arg0==arg1)&'b01, argdone0&&argdone1, 1'b1 };
      3'b011: { core0, coredone0, coreV } = { ((arg0<arg1)||(arg0==arg1)), argdone0&&argdone1, 1'b1 };
      3'b100: { core0, coredone0, coreV } = { ((arg0>arg1)||(arg0==arg1)), argdone0&&argdone1, 1'b1 };
      3'b101: { core0, coredone0, coreV } = { (arg0<arg1), argdone0&&argdone1, 1'b1 };
      3'b110: { core0, coredone0, coreV } = { (arg0>arg1), argdone0&&argdone1, 1'b1 };
      3'b111: { core0, coredone0, coreV } = { 32'b001, argdone0&&argdone1, 1'b1 };
    endcase
  end
end

wire stopped;
assign stopped = ~ready & valid; // Must the output buffer hold its value?
wire sending;
assign sending = (~ready0 & ~ready1); // Are all the input buffers filled?
Tiles with Separate Relay Station

`aggregator.sv`

```vhdl
/* Project: DPU */
module aggregator #(%
    parameter COLUMN_WIDTH = 32,
    parameter RECORD_WIDTH = 128,
    parameter KEY_WIDTH = 32,
    parameter SORT_WIDTH_LOG2 = 2,
    parameter SPLITTER_COUNT_LOG2 = 3,
    parameter ADDRESS_WIDTH = 32
) (%
    //data in AvalonST
    input logic [COLUMN_WIDTH-1:0] group_i,
    input logic [COLUMN_WIDTH-1:0] col_i,
    //control in AvalonMM
```
```verilog
input logic [7:0] op, //op[1:0] is op, op[2] is gb_all
input logic enable_i,
input logic write,
input logic chipselect,

//data out AvalonST
output logic [COLUMN_WIDTH-1:0] group_o,
output logic [COLUMN_WIDTH-1:0] col_o,
output logic [COLUMN_WIDTH-1:0] agg_o,

//control out AvalonMM
//output logic [1:0] op_o,

"write" AvalonST
input logic read_g_i,
input logic read_c_i,
output logic read_c_o,
output logic read_g_o,
output logic read_a_o,

//AvalonST
input logic done_g_i,
input logic done_c_i,
output logic done_g_o,
output logic done_c_o,
output logic done_a_o,

"enable" AvalonST
input logic ready_c_i,
input logic ready_g_i,
input logic ready_a_i,
output logic ready_c_o,
output logic ready_g_o,
output logic ready_a_o,

input logic reset_i, // reset everything to zeros
input logic clk
);

/* hold signals */
logic [COLUMN_WIDTH-1:0] col_hold;
logic [COLUMN_WIDTH-1:0] group_hold;
logic [1:0] op_hold;

/* output */
logic [1:0] op_b;
logic gb_all;

logic[COLUMN_WIDTH-1:0] col; //out from RS1
logic[COLUMN_WIDTH-1:0] group; // out from RS2

logic read_c; //out from RS1
logic read_g; //out from RS2
logic done_c; //out from RS1
logic done_g; //out from RS2

logic ready_c;/inter RS comms
logic ready_g;/inter RS comms

logic ready_c; //out from RS1
logic ready_g; //out from RS2

assign read_c_o = read_c&read_g;
```
assign read_g_o = read_c && read_g;
assign read_a_o = read_c && read_g && done_c && done_g;
assign done_c_o = done_c && done_g;
assign done_g_o = done_c && done_g;
assign done_a_o = done_c && done_g;
assign ready_c_o = ready_c;
assign ready_g_o = ready_g;
assign ready_c = ready_c_i && ready_g_i && ready_a_i && read_g;
assign ready_g = ready_c_i && ready_g_i && ready_a_i && read_c;

/* aggregation logic (internal)*/
logic [COLUMN_WIDTH-1:0] count;
logic [COLUMN_WIDTH-1:0] max;
logic [COLUMN_WIDTH-1:0] min;
logic [COLUMN_WIDTH-1:0] sum;
logic [COLUMN_WIDTH-1:0] one = 'b00001;

relay col_rs(.col_i(col_i), .read_i(read_c_i), .done_i(done_c_i), .ready_i(ready_c),
.col_o(col), .read_o(read_c), .done_o(done_c), .ready_o(ready_c),
.reset_i(reset_i), .*);
relay group_rs(.col_i(group_i), .read_i(read_g_i), .done_i(done_g_i),.ready_i(ready_g),
.col_o(group), .read_o(read_g), .done_o(done_g),.ready_o(ready_g),
.reset_i(reset_i), .clk(clk), .*);

always_ff @(posedge clk) begin
  if(chipselect && enable_i) begin
    gb_all<=op[2:2];
    op_b<=op[1:0];
  end
  if(reset_i) begin
    col_hold<='b0;
    group_hold<='b0;
    op_hold<='b0;
    agg_o<='b0;
    col_o<='b0;
    group_o<='b0;
    count <= 'b0;
    max <= 'b0;
    min <= 'b1; /* all ones */
    sum <= 'b0;
    gb_all <= 'b0;
    op_b<='b0;
  end
  else if(ready_c_i && ready_g_i && ready_a_i && read_c && read_g) begin
    /* new group */
    if(!(gb_all && (group_hold != group_i)))||((done_c&&done_g)) begin
      col_o <= col_hold;
      group_o <= group_hold;
      /*op_ob <= op_hold;*/
    end
    case(op_hold)
      2'b00: //count
        agg_o <= count;
      2'b01: //max
        agg_o <= max;
      default: //error
2'b10: //min
    agg_o <= min;

2'b11: begin/avg
    if(count>0)
        agg_o <= sum/count;
    end
endcase

    /* new group, update*/
    count <= one;
    max <= col;
    min <= col;
    sum <= col;

    /* fetch new holds */
    op_hold <= op_b;
    col_hold <= col;
    group_hold <= group;
    end

    /* same group as before */
    else begin
        count <= count + one;
        sum <= sum + col;

        if(col > max)
            max <= col;

        if(col < min)
            min <= col;
        end
    end
endmodule

coljoiner.sv

觃

* COLUMN_WIDTH is the width of a column in bits
*/
module coljoiner #(                 
    parameter COLUMN_WIDTH = 32, 
    parameter RECORD_WIDTH = 128, 
    parameter KEY_WIDTH = 32, 
    parameter SORT_WIDTH_LOG2 = 2, 
    parameter SPLITTER_COUNT_LOG2 = 3, 
    parameter ADDRESS_WIDTH = 32
)
(
  //data in AvalonST
  input logic [COLUMN_WIDTH-1:0] primary_key_i,
  input logic [COLUMN_WIDTH-1:0] foreign_key_i,

  input logic [COLUMN_WIDTH-1:0] col_1_i,
  input logic [COLUMN_WIDTH-1:0] col_2_i,
  input logic [COLUMN_WIDTH-1:0] col_3_i,
  input logic [COLUMN_WIDTH-1:0] col_4_i,

  //control AvalonMM
  input logic [7:0] op,
input logic write,
input logic chipselect,
input logic enable_i,

//AvalonST
output logic [COLUMN_WIDTH-1:0] primary_key_o,
output logic [COLUMN_WIDTH-1:0] foreign_key_o,
output logic [COLUMN_WIDTH-1:0] col_1_o,
output logic [COLUMN_WIDTH-1:0] col_2_o,
output logic [COLUMN_WIDTH-1:0] col_3_o,
output logic [COLUMN_WIDTH-1:0] col_4_o,

//AvalonST
output logic shift_primary,
output logic shift_foreign,

//AvalonST
input logic read_p_i,
input logic read_f_i,
input logic read_1_i,
input logic read_2_i,
input logic read_3_i,
input logic read_4_i,

output logic read_p_o,
output logic read_f_o,
output logic read_1_o,
output logic read_2_o,
output logic read_3_o,
output logic read_4_o,

//AvalonST
input logic ready_p_i,
input logic ready_f_i,
input logic ready_1_i,
input logic ready_2_i,
input logic ready_3_i,
input logic ready_4_i,

output logic ready_p_o,
output logic ready_f_o,
output logic ready_1_o,
output logic ready_2_o,
output logic ready_3_o,
output logic ready_4_o,

//AvalonST
input logic done_p_i,
input logic done_f_i,
input logic done_1_i,
input logic done_2_i,
input logic done_3_i,
input logic done_4_i,

output logic done_p_o,
output logic done_f_o,
output logic done_1_o,
output logic done_2_o,
output logic done_3_o,
output logic done_4_o,

/* std signals */
input logic reset_i, // reset everything
input logic clk
logic[COLUMN_WIDTH-1:0] col_1; //out from RS1
logic[COLUMN_WIDTH-1:0] col_2; // out from RS2
logic[COLUMN_WIDTH-1:0] col_3; // out from RS3
logic[COLUMN_WIDTH-1:0] col_4; // out from RS4
logic[COLUMN_WIDTH-1:0] primary_key; // out from RSp
logic[COLUMN_WIDTH-1:0] foreign_key; // out from RSf

logic read_1; //out from RS1
logic read_2; //out from RS2
logic read_3; //out from RS3
logic read_4; //out from RS4
logic read_p; //out from RSp
logic read_f; //out from RSf

logic done_1; //out from RSf
logic done_2; //out from RS1
logic done_3; //out from RS2
logic done_4; //out from RS3
logic done_p; //out from RSp
logic done_f; //out from RSf

logic ready1; //inter RS comms
logic ready2; //inter RS comms
logic ready3; //inter RS comms
logic ready4; //inter RS comms
logic readyp; //inter RS comms
logic readyf; //inter RS comms

logic ready_1; //out from RS1
logic ready_2; //out from RS2
logic ready_3; //out from RS3
logic ready_4; //out from RS4
logic up_primary; //out from RSp
logic up_foreign; //out from RSf

logic [2:0] op_b;

assign read_1_o=read_1&&read_2&&read_3&&read_4&&read_p&&read_f;
assign read_2_o=read_1&&read_2&&read_3&&read_4&&read_p&&read_f;
assign read_3_o=read_1&&read_2&&read_3&&read_4&&read_p&&read_f;
assign read_4_o=read_1&&read_2&&read_3&&read_4&&read_p&&read_f;
assign done_1_o=done_1&&done_2&&done_3&&done_4&&done_p&&done_f;
assign done_2_o=done_1&&done_2&&done_3&&done_4&&done_p&&done_f;
assign done_3_o=done_1&&done_2&&done_3&&done_4&&done_p&&done_f;
assign done_4_o=done_1&&done_2&&done_3&&done_4&&done_p&&done_f;
assign done_p_o=done_1&&done_2&&done_3&&done_4&&done_p&&done_f;
assign done_f_o=done_1&&done_2&&done_3&&done_4&&done_p&&done_f;
assign ready_1_o=ready_1;
assign ready_2_o=ready_2;
assign ready_3_o=ready_3;
assign ready_4_o=ready_4;
assign primary_key_o=primary_key;
assign foreign_key_o=foreign_key;
assign readyp=ready_p_i&&ready_f_i&&ready_1_i&&ready_2_i&&ready_3_i&&ready_4_i&&read_2&&read_3&&read_4&&read_p&&read_f;
assign ready2=ready_p_i&&ready_f_i&&ready_1_i&&ready_2_i&&ready_3_i&&ready_4_i&&read_1&&read_3&&read_4&&read_p&&read_f;
assign ready3=ready_p_i&&ready_f_i&&ready_1_i&&ready_2_i&&ready_3_i&&ready_4_i&&read_2&&read_1&&read_4&&read_p&&read_f;
assign ready4 = ready_p_i && ready_f_i && ready_1_i && ready_2_i && ready_3_i && read_2 && read_3 && read_1 && read_p && read_f;
assign ready_p = ready_p_i && ready_f_i && ready_1_i && ready_2_i && ready_3_i && read_1 && read_2 && read_3 && read_4 && read_f;
assign ready_f = ready_p_i && ready_f_i && ready_1_i && ready_2_i && ready_3_i && read_1 && read_2 && read_3 && read_4 && read_p;
assign shift_primary = up_primary; //(primary_key<foreign_key);
assign shift_foreign = up_foreign; //(primary_key>=foreign_key);
relay in1(.col_i(col_1_i), .read_i(read_1_i), .done_i(done_1_i),.ready_i(ready1),
    .col_o(col_1),.read_o(read_1),.done_o(done_1),.ready_o(ready_1),
    .reset_i(reset_i),.*
);
relay in2(.col_i(col_2_i), .read_i(read_2_i), .done_i(done_2_i),.ready_i(ready2),
    .col_o(col_2),.read_o(read_2),.done_o(done_2),.ready_o(ready_2),
    .reset_i(reset_i),.clk(clk),.*
);
relay in3(.col_i(col_3_i), .read_i(read_3_i), .done_i(done_3_i),.ready_i(ready3),
    .col_o(col_3),.read_o(read_3),.done_o(done_3),.ready_o(ready_3),
    .reset_i(reset_i),.*
);
relay in4(.col_i(col_4_i), .read_i(read_4_i), .done_i(done_4_i),.ready_i(ready4),
    .col_o(col_4),.read_o(read_4),.done_o(done_4),.ready_o(ready_4),
    .reset_i(reset_i),.clk(clk),.*
);
relay inp(.col_i(primary_key_i), .read_i(read_p_i), .done_i(done_p_i),.ready_i(readyp),
    .col_o(primary_key),.read_o(read_p),.done_o(done_p),.ready_o(shift_primary),
    .reset_i(reset_i),.*
);
relay inf(.col_i(foreign_key_i), .read_i(read_f_i), .done_i(done_f_i),.ready_i(readyf),
    .col_o(foreign_key),.read_o(read_f),.done_o(done_f),.ready_o(shift_foreign),
    .reset_i(reset_i),.clk(clk),.*
);

always_ff @(posedge clk) begin
if(chipselect && write) begin
    op_b <= op[2:0];
end
if(reset_i) begin
    op_b <= 'b0;
end
end

always_comb begin
    col_1_o = 'bX;
    col_2_o = 'bX;
    col_3_o = 'bX;
    col_4_o = 'bX;
    up_primary = 'b1;
    up_foreign = 'b1;
    if(enable_i) begin
        if(ready_p_i && ready_f_i && ready_1_i && ready_2_i && ready_3_i && read_4_i && read_2 && read_3 && read_4 && read_p && read_f) begin /* going to give output */
            up_primary = (primary_key < foreign_key) ? 'b1:1'b0;
            up_foreign = (primary_key >= foreign_key) ? 'b1:1'b0;
            case(op_b)
                3'b000: //=
                    if(primary_key==foreign_key) begin
                        col_1_o = col_1;
                        col_2_o = col_2;
                    end
                default: //!
                    if(primary_key!=foreign_key) begin
                        col_1_o = col_2;
                        col_2_o = col_3;
                    end
            endcase
        end
    end
end
col_3_o = col_3;
col_4_o = col_4;
end
3'b001: //!=
if(primary_key!=foreign_key) begin
  col_1_o = col_1;
  col_2_o = col_2;
  col_3_o = col_3;
  col_4_o = col_4;
end
3'b010: //<=
if(primary_key<=foreign_key) begin
  col_1_o = col_1;
  col_2_o = col_2;
  col_3_o = col_3;
  col_4_o = col_4;
end
3'b011: //</
if(primary_key<foreign_key) begin
  col_1_o = col_1;
  col_2_o = col_2;
  col_3_o = col_3;
  col_4_o = col_4;
end
3'b100: //>=
if(primary_key>=foreign_key) begin
  col_1_o = col_1;
  col_2_o = col_2;
  col_3_o = col_3;
  col_4_o = col_4;
end
3'b101: ///</
if(primary_key>foreign_key) begin
  col_1_o = col_1;
  col_2_o = col_2;
  col_3_o = col_3;
  col_4_o = col_4;
end
3'b110: //=
if(primary_key==foreign_key) begin
  col_1_o = col_1;
  col_2_o = col_2;
  col_3_o = col_3;
  col_4_o = col_4;
end
3'b111: //=
if(primary_key==foreign_key) begin
  col_1_o = col_1;
  col_2_o = col_2;
  col_3_o = col_3;
  col_4_o = col_4;
end
case
endcase
end
end
endmodule
/* relay station
 http://www.cs.columbia.edu/~luca/research/rbilsENTCS06.pdf */

/* Working */
module relay #(parameter COLUMN_WIDTH = 32, parameter RECORD_WIDTH = 128, parameter KEY_WIDTH = 32, parameter SORT_WIDTH_LOG2 = 2, parameter SPLITTER_COUNT_LOG2 = 3, parameter ADDRESS_WIDTH = 32 )

 ( //data in AvalonST
   input logic [COLUMN_WIDTH-1:0] col_i,
   //data out AvalonST
   output logic [COLUMN_WIDTH-1:0] col_o,
   //control in AvalonMM
   input logic enable_i,
   //AvalonST
   input logic read_i, output logic read_o,
   //AvalonST
   input logic ready_i, output logic ready_o,
   //AvalonST
   input logic done_i, output logic done_o,
   /* std signals */
   input logic clk,
   input logic reset_i // reset everything );

 logic [COLUMN_WIDTH-1:0] col_ib;
 logic read_ib;
 logic done_ib;

 logic [COLUMN_WIDTH-1:0] col_ob;
 logic ready_ob;
 logic done_ob;
 logic read_ob;

 assign col_o=col_ob;
 assign read_o = read_ob;
 assign ready_o = ready_ob;
 assign done_o = done_ob;

 /* input buffering */
 always_ff @(posedge clk) begin


if(reset_i) begin
  col_ib<=b0;
  read_ib<=1'b0;
  done_ib<=1'b0;

  col_ob<=b0;
  ready_ob<=1'b1;
  done_ob<=1'b0;
end
else if(enable_i) begin
  read_ob<=read_ib;
  done_ob<=done_ib;

  if(ready_i && read_ib) begin /* going to give output */
    ready_ob<=1'b1;
    if(read_i) begin //input 1 valid
      col_ob <= col_ib;
      col_ib<=col_i;
      read_ib<=read_i;
      done_ob<=done_i;
    end
  end
else
  read_ib<=1'b0;
end
else if(ready_i) begin /* waiting on inputs */
  if(read_i) begin //if we need input 1
    col_ib<=col_i;
    read_ib<=read_j;
    done_ob<=done_i;
    ready_ob<=1'b1; //check!
  end
end
else if(read_ib) begin /* buffer */
  col_ob <= col_ib;
  ready_ob<=1'b0;
  if(read_i) begin //if we need input 1
    col_ib<=col_i;
    read_ib<=read_j;
    done_ob<=done_i;
    end
end
else begin
  ready_ob<=1'b1;
  if(read_i) begin //if we need input 1
    col_ib<=col_i;
    read_ib<=read_i;
    done_ob<=done_i;
  end
end
endmodule

Deprecated Tiles

merger.sv

module merger
(
  //data in AvalonST
  input logic [COLUMN_WIDTH-1:0] col_1_i,
input logic [COLUMN_WIDTH-1:0] col_2_i,

//data out AvalonST
output logic [COLUMN_WIDTH-1:0] pass_o,
output logic [COLUMN_WIDTH-1:0] col_o,

//control AvalonMM
input logic [2:0] op,
input logic pass_i,
input logic write,
input logic chipselect,
input logic enable_i,

//AvalonST
input logic read_1_i,
input logic read_2_i,
output logic read_p_o,
output logic read_c_o,

//AvalonST
input logic ready_p_i,
input logic ready_c_i,
output logic ready_1_o,
output logic ready_2_o,

//AvalonST
input logic done_1_i,
input logic done_2_i,
output logic done_o,

/* std signals */
input logic reset_i, // reset everything to zeros
input logic clk
);

logic [COLUMN_WIDTH-1:0] col;
logic [COLUMN_WIDTH-1:0] pass;
logic read_p;
logic read_c;
logic ready_1;
logic ready_2;
logic done;

assign read_c_o = read_c;
assign read_p_o = read_p;
assign ready_1_o = ready_1;
assign ready_2_o = ready_2;
assign done_o = done;

assign pass_o = pass;
assign col_o = col;

always_ff @(posedge clk) begin
if(chipselect&&write) begin
if(reset_i==1'b1) begin
read_c <= 1'b0;
read_p <= 1'b0;
ready_1<=1'b1;
ready_2<=1'b1;
done <= 1'b0;
col <= 'b0;
pass <= 'b0;
end
else if(read_1_i && read_2_i && enable_i && ready_p_i && ready_c_i) begin
done <= done_1_i && done_2_i;
end
case(op)
  3'b000: begin //=
    if(col_1_i==col_2_i) begin
      col <= col_1_i;
      ready_1 <= 1'b1;
      ready_2 <= 1'b0;
      read_c <= 1'b1;
      if(pass_i) begin
        pass<=col_2_i;
        read_p<=1'b1;
      end
    end else begin
      col <= col_2_i;
      ready_2 <= 1'b1;
      ready_1 <= 1'b0;
      read_c <= 1'b1;
      if(pass_i) begin
        pass<=col_1_i;
        read_p<=1'b1;
      end
    end
  end
  3'b001: begin //=/
    if(col_1_i==col_2_i) begin
      col <= col_1_i;
      ready_1 <= 1'b1;
      ready_2 <= 1'b0;
      read_c <= 1'b1;
      if(pass_i) begin
        pass<=col_2_i;
        read_p<=1'b1;
      end
    end else begin
      col <= col_2_i;
      ready_2 <= 1'b1;
      ready_1 <= 1'b0;
      read_c <= 1'b1;
      if(pass_i) begin
        pass<=col_1_i;
        read_p<=1'b1;
      end
    end
  end
  3'b010: begin //</
    if(col_1_i<=col_2_i) begin
      col <= col_1_i;
      ready_1 <= 1'b1;
      ready_2 <= 1'b0;
      read_c <= 1'b1;
      if(pass_i) begin
        pass<=col_2_i;
        read_p<=1'b1;
      end
    end else begin
      col <= col_2_i;
      ready_2 <= 1'b1;
      ready_1 <= 1'b0;
      read_c <= 1'b1;
      if(pass_i) begin
        pass<=col_1_i;
        read_p<=1'b1;
      end
    end
  end
  3'b011: begin //</

if(col_1_i<col_2_i) begin
    col <= col_1_i;
    ready_1 <= 'b1;
    ready_2 <= 'b0;
    read_c <= 'b1;
    if(pass_i) begin
        pass<=col_2_i;
        read_p='b1;
    end
end else begin
    col <= col_2_i;
    ready_2 <= 'b1;
    ready_1 <= 'b0;
    read_c <= 'b1;
    if(pass_i) begin
        pass<=col_1_i;
        read_p='b1;
    end
end

3'b100: begin //=
    if(col_1_i>=col_2_i) begin
        col <= col_1_i;
        ready_1 <= 'b1;
        ready_2 <= 'b0;
        read_c <= 'b1;
        if(pass_i) begin
            pass<=col_2_i;
            read_p='b1;
        end
    end else begin
        col <= col_2_i;
        ready_2 <= 'b1;
        ready_1 <= 'b0;
        read_c <= 'b1;
        if(pass_i) begin
            pass<=col_1_i;
            read_p='b1;
        end
    end
end
3'b101: begin //=
    if(col_1_i>col_2_i) begin
        col <= col_1_i;
        ready_1 <= 'b1;
        ready_2 <= 'b0;
        read_c <= 'b1;
        if(pass_i) begin
            pass<=col_2_i;
            read_p='b1;
        end
    end else begin
        col <= col_2_i;
        ready_2 <= 'b1;
        ready_1 <= 'b0;
        read_c <= 'b1;
        if(pass_i) begin
            pass<=col_1_i;
            read_p='b1;
        end
    end
end
3'b110: begin //pass col_1
    col <= col_1_i;
    ready_1 <= 'b1;

module splitter
(
    //data in AvalonST
    input logic [COLUMN_WIDTH-1:0] col_i,
    input logic [COLUMN_WIDTH-1:0] split,

    //data out AvalonST
    output logic [COLUMN_WIDTH-1:0] pass_o,
    output logic [COLUMN_WIDTH-1:0] col_o,

    //control AvalonMM
    input logic pass_i,
    input logic [2:0] op,
    input logic write,
    input logic enable_i,

    //AvalonST
    input logic read_c_i,
    input logic read_s_i,
    output logic read_p_o,
    output logic read_c_o,

    //AvalonST
    input logic ready_p_i,
    input logic ready_c_i,
    output logic ready_o,

    //AvalonST
    input logic done_c_i,
    input logic done_s_i,
    output logic done_o,

    /* std signals */
    input logic reset_i, // reset everything to zeros
    input logic clk
);

// data in AvalonST
ready_2 <= 1'b0;
read_c <= 1'b1;
endcase
else if(enable_i) begin
    done <= done_1_i && done_2_i;
    ready_1 <= ready_p_i && ready_c_i;
    ready_2 <= ready_p_i && ready_c_i;
    read_p <= 1'b0;
    read_c <= 1'b0;
end
end
endmodule
logic [COLUMN_WIDTH-1:0] col;
logic [COLUMN_WIDTH-1:0] pass;
logic read_p;
logic read_c;
logic ready;
logic done;

assign read_c_o = read_c;
assign read_p_o = read_p;
assign ready_o = ready;
assign done_o = done;

assign pass_o = pass;
assign col_o = col;

always_ff @(posedge clk) begin
if(chipselect && write) begin
if(reset_i==1'b1)begin
read_c <= 1'b0;
read_p <= 1'b0;
ready<=1'b1;
done <= 1'b0;
col <= 'b0;
pass <= 'b0;
end
else if(read_s_i && read_c_i && enable_i && ready_p_i && ready_c_i) begin
done <= done_c_i && done_s_i;
ready <= ready_p_i && ready_c_i;
case(op)
3'b000: begin //=
if(col_i==split) begin
    col <= col_i;
    read_c <= 1'b1;
end
else
    read_c <= 1'b0;
end
3'b001: begin //!=
if(col_i!=split) begin
    col <= col_i;
    read_c <= 1'b1;
end
else begin
    pass <= col_i;
    read_p <= 1'b1;
    read_c <= 1'b0;
end
end
3'b010: begin //<=
if(col_i<=split) begin
    col <= col_i;
    read_c <= 1'b1;
end
else begin
    pass <= col_i;
    read_p <= 1'b1;
    read_c <= 1'b0;
end
end
3'b011: begin //</
if(col_i<split) begin
    col <= col_i;
    read_c <= 1'b1;
end
else begin
    pass <= col_i;
    read_p <= 1'b1;
    read_c <= 1'b0;
end
end
end

else begin
  pass <= col_i;
  read_p <= 1'b1;
  read_c <= 1'b0;
end
end
3'b100: begin //>=
  if(col_i>=split) begin
    col <= col_i;
    read_c <= 1'b1;
  end
  else begin
    pass <= col_i;
    read_p <= 1'b1;
    read_c <= 1'b0;
  end
end
3'b101: begin //</
  if(col_i>split) begin
    col <= col_i;
    read_c <= 1'b1;
  end
  else begin
    pass <= col_i;
    read_p <= 1'b1;
    read_c <= 1'b0;
  end
end
3'b110: begin //pass
  col <= col_i;
  read_c <= 1'b1;
end
3'b111: begin //pass
  col <= col_i;
  read_c <= 1'b1;
end
endcase
if(pass_i) begin
  pass <= col_i;
  read_p <= 1'b1;
end
end
else if(enable_i) begin
  done <= done_s_i && done_c_i;
  ready <= ready_p_i && ready_c_i;
  read_p <= 1'b0;
  read_c <= 1'b0;
end
end
endmodule

colselect.sv

`include "param.sv"
module colselect
(
  //data in AvalonST
  input logic [RECORD_WIDTH-1:0] record_i, // the inbound burst of records
  //control AvalonMM
  input logic [7:0] offset, //how far to shift the column
  //...
input logic write,
input logic chipselect,
input logic enable_i,

//data out AvalonST
output logic [COLUMN_WIDTH-1:0] col_o,

/* std signals */
input logic reset_i, // reset everything
input logic clk,

//AvalonST
input logic read_i,
output logic read_o,
input logic ready_i,
output logic ready_o,
input logic done_i,
output logic done_o
);

logic [RECORD_WIDTH-1:0] col_ob;
logic [RECORD_WIDTH-1:0] record_ib;
logic read_ib;
logic done_ib;
logic ready_ob;
logic done_ob;
logic read_ob;

assign col_o = col_ob[COLUMN_WIDTH-1:0];
assign read_o = read_ob;
assign ready_o = ready_ob;
assign done_o = done_ob;

always_ff @(posedge clk) begin
if(chipselect & write) begin
if(reset_i) begin
record_ib<='b0;
read_ib<='b0;
done_ib<='b0;
end
else if(enable_i) begin
if(ready_i & read_ib) begin /* going to give output */
record_ib<=record_i;
read_ib<=read_i;
done_ib<=done_i;
end
end
else if(ready_i) begin
if(read_ib & read_i) begin
record_ib<=record_i;
read_ib<=read_i;
done_ib<=done_i;
end
end
else if(ready_i) begin /*buffer*/
if(read_ib & read_i) begin
record_ib<=record_i;
read(ib)<=read_i;
done(ib)<=done_i;
end
end
end
end
/* output buffering */
always_ff @(posedge clk) begin
if(chipselect && write) begin
    if(reset_i) begin
        col_ob <= 'b0;
        read_ob <= 1'b0;
        ready_ob <= 1'b1;
        done_ob <= 1'b0;
    end
    else if(enable_i) begin
        if(ready_i && read_ib) begin /* going to give output */
            read_ob<=read_ib; //1
            done_ob<=done_ib;
            col_ob<=record_ib>>(offset);
    end
    else if(ready_i)begin
        read_ob<=read_ib; //0
        done_ob<=done_ib;
    end
    else if(!ready_i) begin
        read_ob<=read_ib;
        done_ob<=done_ib;
    end
end
endmodule

concatenate.sv

`include "param.sv"
module concatenate
(  //AvalonST
    input logic [COLUMN_WIDTH-1:0] col_1_i,
    input logic [COLUMN_WIDTH-1:0] col_2_i,
    //AvalonST
    output logic [COLUMN_WIDTH-1:0] col_o,
    //AvalonST
    input logic read_1_i,
    input logic read_2_i,
input logic read_o,

//AvalonST
input logic ready_i,
output logic ready_1_o,
output logic ready_2_o,

//AvalonST
input logic done_1_i,
input logic done_2_i,
output logic done_o,

//control AvalonMM
input logic enable_i,
input logic write,
input logic chipselect,

/* std signals */
input logic reset_i, // reset everything
input logic clk
);

logic[COLUMN_WIDTH-1:0] col_ob;
logic[COLUMN_WIDTH-1:0] col_1_ib;
logic[COLUMN_WIDTH-1:0] col_2_ib;

logic read_1_ib;
logic read_2_ib;
logic done_1_ib;
logic done_2_ib;

logic ready_1_ob;
logic ready_2_ob;
logic done_ob;
logic read_ob;

assign col_o=col_ob;
assign read_o = read_ob;
assign ready_1_o = ready_1_ob;
assign ready_2_o = ready_2_ob;
assign done_o = done_ob;

/* input buffering */
always_ff @(posedge clk) begin
  if(chipselect & & write) begin
    if(reset_i) begin
      col_1_ib<='b0;
      col_2_ib<='b0;
      read_1_ib<='b0;
      read_2_ib<='b0;
      done_1_ib<='b0;
      done_2_ib<='b0;
    end
    else if(enable_i) begin
      if(ready_i & & read_1_ib & & read_2_ib) begin /* going to give output */
        if(read_1_i) begin //input 1 valid
          col_1_ib<=col_1_i;
          read_1_ib<=read_1_i;
          done_1_ib<=done_1_i;
        end
        if(read_2_i) begin //input 2 valid
          col_2_ib<=col_2_i;
          read_2_ib<=read_2_i;
          done_2_ib<=done_2_i;
        end
      end
    end
  end
end
end
else if(ready_i) begin /* waiting on inputs */
  if(read_1_i & !read_1_ib) begin // if we need input 1
    col_1_ib<=col_1_i;
    read_1_ib<=read_1_i;
    done_1_ib<=done_1_i;
  end
  if(read_2_i & !read_2_ib) begin // if we need input 2
    col_2_ib<=col_2_i;
    read_2_ib<=read_2_i;
    done_2_ib<=done_2_i;
  end
end
else if(!ready_i) begin /* buffer */
  if(read_1_i & !read_1_ib) begin // if we need input 1
    col_1_ib<=col_1_i;
    read_1_ib<=read_1_i;
    done_1_ib<=done_1_i;
  end
  if(read_2_i & !read_2_ib) begin // if we need input 2
    col_2_ib<=col_2_i;
    read_2_ib<=read_2_i;
    done_2_ib<=done_2_i;
  end
end
end
end

/* output buffering */
always_ff @(posedge clk) begin
  if(chipselect & write) begin
    if(reset_i) begin
      col_ob<='b0;
      ready_1_ob<=1'b1;
      ready_2_ob<=1'b1;
      done_ob<=1'b0;
      read_ob<=1'b0;
    end
    else if(enable_i) begin
      if(ready_i & read_1_ib & read_2_ib) begin /* going to give output */
        read_ob<=read_1_ib & read_2_ib; //1
        done_ob<=done_1_ib & done_2_ib;
        col_ob[COLUMN_WIDTH/2-1:0] <= col_1_ib[COLUMN_WIDTH/2-1:0];
        col_ob[COLUMN_WIDTH-1:COLUMN_WIDTH/2] <= col_2_ib[COLUMN_WIDTH/2-1:0];
        ready_1_ob<=ready_i; //1
        ready_2_ob<=ready_i; //1
      end
    else if(ready_i) begin /* waiting on input 1 or 2 */
      read_ob<=read_1_ib & read_2_ib; //0
      done_ob<='b0;
      if(!read_1_ib) begin // if waiting on input 1
        if(ready_i) // going to be buffered
          ready_1_ob<='b0;
        else // still waiting
          ready_1_ob<='b1;
      end
    else
      ready_1_ob<='b1;
    end
    else
      ready_1_ob<='b0;
    if(!read_2_ib) begin // if waiting on input 2
      if(ready_i) // going to be buffered
        ready_2_ob<='b0;
      else // still invalid
        ready_2_ob<=1'b1;
    end
  end
end
```verilog
deready_2_ob<=1'b0;
end
else if(ready_i) begin
read_ob<=read_1_ib&&read_2_ib;
done_ob<=1'b0;
if(!read_1_ib) begin //if waiting on input 1
if(read_1_i) //going to be buffered
ready_1_ob<=1'b0;
else //still waiting
ready_1_ob<=1'b1;
end
else
ready_1_ob<=1'b0;
if(!read_2_ib) begin //if waiting on input 2
if(read_2_i) //going to be buffered
ready_2_ob<=1'b0;
else //still invalid
ready_2_ob<=1'b1;
end
else //already have it
ready_2_ob<=1'b0;
end
endmodule
```

```
`include "param.sv"
module stitch
//data in AvalonST
(input logic [COLUMN_WIDTH-1:0] col_1_i,
input logic [COLUMN_WIDTH-1:0] col_2_i,
input logic [COLUMN_WIDTH-1:0] col_3_i,
input logic [COLUMN_WIDTH-1:0] col_4_i,
//data out AvalonST
(output logic [RECORD_WIDTH-1:0] rec_o,
//AvalonST
(input logic done_1_i,
input logic done_2_i,
input logic done_3_i,
input logic done_4_i,
output logic done_o,
//AvalonST
(input logic read_1_i,
input logic read_2_i,
input logic read_3_i,
input logic read_4_i,
output logic read_o,
//AvalonST
(input logic ready_i,
output logic ready_1_o,
output logic ready_2_o,
output logic ready_3_o,
output logic ready_4_o,
```
/* std signals */
input logic reset_i, // reset everything
input logic clk
);

logic[RECORD_WIDTH-1:0] rec_ob;
logic[COLUMN_WIDTH-1:0] col_1_ib;
logic[COLUMN_WIDTH-1:0] col_2_ib;
logic[COLUMN_WIDTH-1:0] col_3_ib;
logic[COLUMN_WIDTH-1:0] col_4_ib;

logic read_1_ib;
logic read_2_ib;
logic read_3_ib;
logic read_4_ib;
logic done_1_ib;
logic done_2_ib;
logic done_3_iv;
logic done_4_ib;

logic ready_1_ob;
logic ready_2_ob;
logic ready_3_ob;
logic ready_4_ob;
logic done_ob;
logic read_ob;

assign rec_o=rec_ob;
assign read_o = read_ob;
assign ready_1_o = ready_1_ob;
assign ready_2_o = ready_2_ob;
assign ready_3_o = ready_3_ob;
assign ready_4_o = ready_4_ob;
assign done_o = done_ob;

input logic enable_i,
input logic write,
input logic chipselect,

always_ff @(posedge clk) begin
if(chipselect && write) begin
if(reset_i begin
    col_1_ib<='b0;
    col_2_ib<='b0;
    col_3_ib<='b0;
    col_4_ib<='b0;
    read_1_ib<=1'b0;
    read_2_ib<=1'b0;
    read_3_ib<=1'b0;
    read_4_ib<=1'b0;
    done_1_ib<=1'b0;
    done_2_ib<=1'b0;
    done_3_ib<=1'b0;
    done_4_ib<=1'b0;
end
else if(enable_i begin
    if(ready_i && read_1_i && read_2_i && read_3_i && read_4_i) begin /* going to give output */
    if(read_1_i begin
        col_1_ib<=col_1_i;
        read_1_ob<=read_1_i;
        done_1_ob<=done_1_i;
    end
end
end
end
if(read_2_i) begin
    col_2_ib<=col_2_i;
    read_2_ib<=read_2_i;
    done_2_ib<=done_2_i;
end
if(read_3_i) begin
    col_3_ib<=col_3_i;
    read_3_ib<=read_3_i;
    done_3_ib<=done_3_i;
end
if(read_4_i) begin
    col_4_ib<=col_4_i;
    read_4_ib<=read_4_i;
    done_4_ib<=done_4_i;
end
end
else if(ready_i) begin /*buffer other*/
if(!read_1_ib) begin
    if(read_1_i) begin
        col_1_ib<=col_1_i;
        read_1_ib<=read_1_i;
        done_1_ib<=done_1_i;
    end
end
if(!read_2_ib) begin
    if(read_2_i) begin
        col_2_ib<=col_2_i;
        read_2_ib<=read_2_i;
        done_2_ib<=done_2_i;
    end
end
if(!read_3_ib) begin
    if(read_3_i) begin
        col_3_ib<=col_3_i;
        read_3_ib<=read_3_i;
        done_3_ib<=done_3_i;
    end
end
if(!read_4_ib) begin
    if(read_4_i) begin
        col_4_ib<=col_4_i;
        read_4_ib<=read_4_i;
        done_4_ib<=done_4_i;
    end
end
end
else if(ready_i) begin /*buffer other*/
if(!read_1_ib) begin
    if(read_1_i) begin
        col_1_ib<=col_1_i;
        read_1_ib<=read_1_i;
        done_1_ib<=done_1_i;
    end
end
if(!read_2_ib) begin
    if(read_2_i) begin
        col_2_ib<=col_2_i;
        read_2_ib<=read_2_i;
        done_2_ib<=done_2_i;
    end
end
if(!read_3_ib) begin
    if(read_3_i) begin
        col_3_ib<=col_3_i;
        read_3_ib<=read_3_i;
        done_3_ib<=done_3_i;
    end
end
if(!read_4_ib) begin
    if(read_4_i) begin
        col_4_ib<=col_4_i;
        read_4_ib<=read_4_i;
        done_4_ib<=done_4_i;
    end
end
end
if(read_3_i) begin
    if(read_3_i) begin
        col_3_ib<=col_3_i;
    end
end
if(read_4_i) begin
    if(read_4_i) begin
        col_4_ib<=col_4_i;
    end
end
if(read_1_i) begin
    if(read_1_i) begin
        col_1_ib<=col_1_i;
    end
end
if(read_2_i) begin
    if(read_2_i) begin
        col_2_ib<=col_2_i;
    end
end
if(read_3_i) begin
    if(read_3_i) begin
        col_3_ib<=col_3_i;
    end
end
if(read_4_i) begin
    if(read_4_i) begin
        col_4_ib<=col_4_i;
    end
end
read_3_ib<=read_3_i;
done_3_ib<=done_3_i;
end
end
if(!read_4_ib) begin
if(read_4_i) begin
col_4_ib<=col_4_i;
read_4_ib<=read_4_i;
done_4_ib<=done_4_i;
end
end
end
end
end
end
end
end
/* output buffering */
always_ff @(posedge clk) begin
if(chipselect && write) begin
if(reset_i) begin
rec_ob<='b0;
ready_1_ob<=1'b1;
ready_2_ob<=1'b1;
ready_3_ob<=1'b1;
ready_4_ob<=1'b1;
doneto_ob<=1'b0;
read_ob<=1'b0;
end
else if(enable_i) begin
if(ready_i && read_1_ib && read_2_ib && read_3_ib && read_4_ib) begin /* going to give output */
read_ob<=read_1_ib && read_2_ib && read_3_ib && read_4_ib; //1
doneto_ob<=done_1_ib && done_2_ib && done_3_ib && done_4_ib;
ready_1_ob<=ready_i; //1
ready_2_ob<=ready_i; //1
ready_3_ob<=ready_i; //1
ready_4_ob<=ready_i; //1
rec_ob[COLUMN_WIDTH-1:0] <= col_1_ib;
rec_ob[2*COLUMN_WIDTH-1:COLUMN_WIDTH] <= col_2_ib;
rec_ob[3*COLUMN_WIDTH-1:2*COLUMN_WIDTH] <= col_3_ib;
rec_ob[4*COLUMN_WIDTH-1:3*COLUMN_WIDTH] <= col_4_ib;
end
else if(ready_i) begin /*buffer*/
read_ob<=read_1_ib&&read_2_ib&&read_3_ib&&read_4_ib; //0
doneto_ob<=1'b0;
if(read_1_i) begin
if(read_1_i)
ready_1_ob<=1'b0;
else
ready_1_ob<=1'b1;
end
else
ready_1_ob<=1'b0;
if(read_2_i) begin
if(read_2_i)
ready_2_ob<=1'b0;
else
ready_2_ob<=1'b1;
end
else
ready_2_ob<=1'b0;
if(read_3_i) begin
if(read_3_i)
ready_3_ob<=1'b0;
else
ready_3_ob<=1'b1;
end
else
ready_3_ob<=1'b0;
if(read_4_i) begin
if(read_4_i)
ready_4_ob<=1'b0;
else
ready_4_ob<=1'b1;
end
else
ready_4_ob<=1'b0;
end
else
  ready_3_ob<=1'b0;
if(read_4_ib) begin
  if(read_4_i)
    ready_4_ob<=1'b0;
elself
    ready_4_ob<=1'b1;
else
  ready_4_ob<=1'b0;
end
else if(!ready_i) begin /*buffer*/
  read_ob<=read_1_ib&&read_2_ib&&read_3_ib&&read_4_ib;
  done_ob<=done_1_ib&&done_2_ib&&done_3_ib&&done_4_ib;
  if(read_1_ib) begin
    if(read_1_i)
      ready_1_ob<=1'b0;
elself
      ready_1_ob<=1'b1;
  end
  else
    ready_1_ob<=1'b0;
  if(read_2_ib) begin
    if(read_2_i)
      ready_2_ob<=1'b0;
elself
      ready_2_ob<=1'b1;
  end
  else
    ready_2_ob<=1'b0;
  if(read_3_ib) begin
    if(read_3_i)
      ready_3_ob<=1'b0;
elself
      ready_3_ob<=1'b1;
  end
  else
    ready_3_ob<=1'b0;
  if(read_4_ib) begin
    if(read_4_i)
      ready_4_ob<=1'b0;
elself
      ready_4_ob<=1'b1;
  end
else
  ready_4_ob<=1'b0;
end
end
endmodule

Test Benches
alu_test.sv

/*
 * COLUMN_WIDTH is the width of a record in bits
 */
timescale 1 ns / 1 ps
#include "param.sv"
module alu_test ();
declare clk_per 100
define Max_Sim_Time 1000000000 // stop simulation after this time (ns)

//data in AvalonST
logic [COLUMN_WIDTH-1:0] col_1_i;
logic [COLUMN_WIDTH-1:0] col_2_i;

//control in AvalonMM
logic [7:0] op;
logic write;
logic chipselect;
logic enable_i;

//data out AvalonST
logic [COLUMN_WIDTH-1:0] col_o;

//AvalonST
logic read_1_i;
logic read_2_i;
logic read_o;

//AvalonST
logic ready_i;
logic ready_1_o;
logic ready_2_o;

//AvalonST
logic done_1_i;
logic done_2_i;
logic done_o;

/* std signals */
logic clk;
logic reset_i; // reset everything to zeros

alu alu_inst
(.);

always #(`clk_per/2) clk = ~clk;

initial begin
// Initialize TB signals
clk = 1'b0;
reset_i = 1'b1;
write = 1'b1;
chipselect = 1'b1;
enable_i = 1'b1;
#100;
reset_i = 1'b0;
#100;
col_1_i = 32'b00000000000000000000000000000001;
col_2_i = 32'b00000000000000000000000000000001;
op = 8'b000000010;
ready_i = 1'b1;
read_1_i = 1'b1;
read_2_i = 1'b1;
done_1_i = 1'b0;
done_2_i = 1'b0;
#100;
#100;
boolgen_test.sv

/*
 * COLUMN_WIDTH is the width of a record in bits
 */
`timescale 1 ns / 1 ps
#include "param.sv"
module boolgen_test();
`define clk_per 100
`define Max_Sim_Time 1000000000     // stop simulation after this time (ns)

//data in AvalonST
logic [COLUMN_WIDTH-1:0] col_1_i;
logic [COLUMN_WIDTH-1:0] col_2_i;

//control in AvalonMM
logic [7:0] op;
logic write;
logic chipselect;
logic enable_i;

//data out AvalonST
logic [COLUMN_WIDTH-1:0] bool_o;

//AvalonST
logic read_1_i;
logic read_2_i;
logic read_o;

//AvalonST
logic ready_i;
logic ready_1_o;
logic ready_2_o;

//AvalonST
logic done_1_i;
logic done_2_i;
logic done_o;

/* std signals */
logic clk;
logic reset_i; // reset everything to zeros

boolgen boolgen_inst
(.*);

always #(`clk_per/2) clk = ~clk;
initial begin
  // Initialize TB signals
  clk = 1'b0;
  reset_i = 1'b1;
  write = 1'b1;
  chipselect = 1'b1;
  enable_i = 1'b1;
  ready_i = 1'b1;
  #100;
  reset_i = 1'b0;
  #100;
  op = 8'b00000001;
  read_1_i = 1'b1;
  read_2_i = 1'b1;
  done_1_i = 1'b0;
  done_2_i = 1'b0;
  col_1_i = 32'b00000000000000000000000000000001;
  col_2_i = 32'b00000000000000000000000000000001;
  #100;
  #100;
  col_1_i = 32'b00000000000000000000000000000000;
  col_2_i = 32'b00000000000000000000000000000001;
  #100;
  #100;
  col_1_i = 32'b00000000000000000000000000000000;
  col_2_i = 32'b00000000000000000000000000000001;
  #100;
  #100;
  end

pet
pet

relay_test.sv

/*
 * COLUMN_WIDTH is the width of a record in bits
 */
timescale 1 ns / 1 ps
#include "param.sv"
module relay_test ();
define clk_per 100
define Max_Sim_Time 1000000000 // stop simulation after this time (ns)
// data in AvalonST
logic [COLUMN_WIDTH-1:0] col_i;
// control in AvalonMM
logic write;
logic chipselect;
logic enable_i;

//data out AvalonST
logic [COLUMN_WIDTH-1:0] col_o;

//AvalonST
logic read_i;
logic read_o;

//AvalonST
logic ready_i;
logic ready_o;

//AvalonST
logic done_i;
logic done_o;

/* std signals */
logic clk;
logic reset_i;// reset everything to zeros

relay relay_inst
(.*);

always #(`clk_per/2) clk = ~clk;

initial begin
// Initialize TB signals
clk = 1'b0;
reset_i = 1'b1;
write = 1'b1;
chipselect = 1'b1;
enable_i = 1'b1;
#100;
$display("nReset asserted at time = %t", $realtime);
reset_i = 1'b0;
#100;
col_i = 32'b00000000000000000000000000000001;
ready_i = 1'b1;
read_i = 1'b1;
done_i = 1'b0;
#100;
col_i = 32'b00000000000000000000000000000010;
ready_i = 1'b1;
#100;
col_i = 32'b00000000000000000000000000000011;
ready_i = 1'b0;
#100;
col_i = 32'b00000000000000000000000000000000;
ready_i = 1'b0;
#100;
end

////////////////////////////////////////////////////////////////////////////////////
// Always End at Max_Sim_Time
////////////////////////////////////////////////////////////////////////////////////

initial begin

coljoiner_test.sv

 /*
 * COLUMN_WIDTH is the width of a record in bits
 */
`timescale 1 ns / 1 ps
`include "param.sv"
module coljoiner_test ();
`define clk_per 100
`define Max_Sim_Time 1000000000 // stop simulation after this time (ns)

//data in AvalonST
input logic [COLUMN_WIDTH-1:0] primary_key_i;
input logic [COLUMN_WIDTH-1:0] foreign_key_i;
input logic [COLUMN_WIDTH-1:0] col_1_i;
input logic [COLUMN_WIDTH-1:0] col_2_i;
input logic [COLUMN_WIDTH-1:0] col_3_i;
input logic [COLUMN_WIDTH-1:0] col_4_i;

//control AvalonMM
input logic [7:0] op;
input logic write;
input logic chipselect;
input logic enable_i;

//AvalonST
output logic [COLUMN_WIDTH-1:0] primary_key_o;
output logic [COLUMN_WIDTH-1:0] foreign_key_o;
output logic [COLUMN_WIDTH-1:0] col_1_o;
output logic [COLUMN_WIDTH-1:0] col_2_o;
output logic [COLUMN_WIDTH-1:0] col_3_o;
output logic [COLUMN_WIDTH-1:0] col_4_o;

//AvalonST
output logic shift_primary;
output logic shift_foreign;

//AvalonST
input logic read_p_i;
input logic read_f_i;
input logic read_1_i;
input logic read_2_i;
input logic read_3_i;
input logic read_4_i;

output logic read_p_o;
output logic read_f_o;
output logic read_1_o;
output logic read_2_o;
output logic read_3_o;
output logic read_4_o;

//AvalonST
input logic ready_p_i;
input logic ready_f_i;
input logic ready_1_i;
input logic ready_2_i;
input logic ready_3_i;
input logic ready_4_i;

output logic ready_p_o;
output logic ready_f_o;
output logic ready_1_o;
output logic ready_2_o;
output logic ready_3_o;
output logic ready_4_o;

// AvalonST
input logic done_p_i;
input logic done_f_i;
input logic done_1_i;
input logic done_2_i;
input logic done_3_i;
input logic done_4_i;

output logic done_p_o;
output logic done_f_o;
output logic done_1_o;
output logic done_2_o;
output logic done_3_o;
output logic done_4_o;

/* std signals */
input logic reset_i; // reset everything to zeros
input logic clk;

coljoiner coljoiner_inst (.*);

always #(`clk_per/2) clk = ~clk;

initial begin
  // Initialize TB signals
  clk = 1’b0;
  reset_i = 1’b1;
  write = 1’b1;
  chipselect = 1’b1;
  enable_i = 1’b1;
  #100;
  reset_i = 1’b0;
  #100;
  primary_key_i = 32’b00000000000000000000000000000001;
  foreign_key_i = 32’b00000000000000000000000000000001;
  col_1_i = 32’b00000000000000000000000000000001;
  col_2_i = 32’b00000000000000000000000000000001;
  col_3_i = 32’b00000000000000000000000000000001;
  col_4_i = 32’b00000000000000000000000000000001;
  op = 8’b00000000;
  ready_p_i = 1’b1;
  ready_f_i = 1’b1;
  ready_1_i = 1’b1;
  ready_2_i = 1’b1;
  ready_3_i = 1’b1;
  ready_4_i = 1’b1;
  read_p_i = 1’b1;
  read_f_i = 1’b1;
  read_1_i = 1’b1;
  read_2_i = 1’b1;
  read_3_i = 1’b1;
  read_4_i = 1’b1;
  done_p_i = 1’b0;
done_f_i = 1'b0;
done_1_i = 1'b0;
done_2_i = 1'b0;
done_3_i = 1'b0;
done_4_i = 1'b0;
end

///////////////////////////////////////////////////////////////////////////////
// Always End at Max_Sim_Time
///////////////////////////////////////////////////////////////////////////////

initial
begin
$monitor("%d %d %d %d", primary_key_i, foreign_key_i, primary_key_o, foreign_key_o);
#(`Max_Sim_Time);
$stop;
end
endmodule

colfilter_test.sv

/*
* COLUMN_WIDTH is the width of a record in bits
*/
`timescale 1 ns / 1 ps
`include "param.sv"
module colfilter_test();
`define clk_per 100
`define Max_Sim_Time 1000000000
module colfilter_test();
`define Max_Sim_Time 1000000000
//data in AvalonST
logic [COLUMN_WIDTH-1:0] col_1_i;
logic [COLUMN_WIDTH-1:0] col_2_i;
//control in AvalonMM
logic [2:0] op;
logic write;
logic chipselect;
logic enable_i;
//data out AvalonST
logic [COLUMN_WIDTH-1:0] col_o;
//AvalonST
logic read_1_i;
logic read_2_i;
logic read_o;
//AvalonST
logic ready_i;
logic ready_1_o;
logic ready_2_o;
logic ready_o;
//AvalonST
logic done_1_i;
logic done_2_i;
logic done_o;
/* std signals */
logic clk;
logic reset_i;// reset everything to zeros

logic [COLUMN_WIDTH-1:0] bool;
logic done;
logic ready;
logic read;

boolgen boolgen_inst(.bool_o(bool), .done_o(done), .read_o(read), .ready_i(ready),.*);
colfilter colfilter_inst(.bool_i(bool), .read_b_i(read), .done_b_i(done), .ready_b_o(ready),.ready_1_o(ready_o),.*);

always #(`clk_per/2) clk = ~clk;

initial begin
  // Initialize TB signals
  clk = 1'b0;
  reset_i = 1'b1;
  write = 1'b1;
  chipselect = 1'b1;
  enable_i = 1'b1;
  done_1_i = 1'b0;
  done_2_i = 1'b0;
  ready_i = 1'b1;
  col_1_i = 32'b00000000000000000000000000000000;
  col_2_i = 32'b00000000000000000000000000000000;
  op = 3'b100;
  //#50;
  #100;
  reset_i = 1'b0;
  #100;
  #100;
  read_1_i = 1'b1;
  read_2_i = 1'b1;
  #100;
  #100;
  if(ready_1_o==1'b1)
    col_1_i = col_1_i + 32'b00000000000000000000000000000001;
  #100;
  if(ready_1_o==1'b1)
    col_1_i = col_1_i + 32'b00000000000000000000000000000001;
  #100;
  if(ready_1_o==1'b1)
    col_1_i = col_1_i + 32'b00000000000000000000000000000001;
  #100;
  if(ready_1_o==1'b1)
    col_1_i = col_1_i + 32'b00000000000000000000000000000001;
  #100;
  if(ready_1_o==1'b1)
    col_1_i = col_1_i + 32'b00000000000000000000000000000001;
  #100;
  if(ready_1_o==1'b1)
    col_1_i = col_1_i + 32'b00000000000000000000000000000001;
  #100;
  if(ready_1_o==1'b1)
    col_1_i = col_1_i + 32'b00000000000000000000000000000001;
  #100;
  if(ready_1_o==1'b1)
    col_1_i = col_1_i + 32'b00000000000000000000000000000001;
  #100;
  if(ready_1_o==1'b1)
    col_1_i = col_1_i + 32'b00000000000000000000000000000001;
  #100;
  if(ready_1_o==1'b1)
    col_1_i = col_1_i + 32'b00000000000000000000000000000001;
  #100;
  if(ready_1_o==1'b1)
    col_1_i = col_1_i + 32'b00000000000000000000000000000001;
  #100;
  if(ready_1_o==1'b1)
    col_1_i = col_1_i + 32'b00000000000000000000000000000001;
  #100;
  if(ready_1_o==1'b1)
    col_1_i = col_1_i + 32'b00000000000000000000000000000001;
  #100;
  if(ready_1_o==1'b1)
    col_1_i = col_1_i + 32'b00000000000000000000000000000001;
if(ready_1_o==1'b1)
  col_1_i = col_1_i + 32'b00000000000000000000000000000001;
end

 cider

 //////////////////////////////////////////////////////////////////////////
// Always End at Max_Sim_Time
 //////////////////////////////////////////////////////////////////////////

initial
begin
$monitor("%d %d", col_1_i, col_o);
#(`Max_Sim_Time);
$stop;
end
endmodule

 signal1_test.sv

 /*
  * COLUMN_WIDTH is the width of a record in bits
  */
`timescale 1 ns / 1 ps
`include "param.sv"
module signal1_test();
`define clk_per 100
`define Max_Sim_Time 1000000000     // stop simulation after this time (ns)

 //_data in AvalonST
logic [COLUMN_WIDTH-1:0] col_1_i;
logic [COLUMN_WIDTH-1:0] col_2_i;

 //_control in AvalonMM
logic [7:0] op;
logic write;
logic chipselect;
logic enable_i;

 //_data out AvalonST
logic [COLUMN_WIDTH-1:0] col_o;

 //_AvalonST
logic read_1_i;
logic read_2_i;
logic read_o;

 //_AvalonST
logic ready_i;
logic ready_1_o;
logic ready_2_o_1;
logic ready_2_o_2;
AvalonST
logic done_1_i;
logic done_2_i;
logic done_o;

/* std signals */
logic clk;
logic reset_i;// reset everything to zeros
logic [COLUMN_WIDTH-1:0] col_1_1;
logic done_1_1;
logic ready_1_1;
logic read_1_1;

alu alu_inst1(.col_o(col_1_1), .read_o(read_1_1), .done_o(done_1_1), .ready_i(ready_1_1), .ready_2_o(ready_2_o_1),.*);
alu alu_inst3(.col_1_i(col_1_1), .read_1_i(read_1_1), .done_1_i(done_1_1), .ready_1_o(ready_1_1), .ready_2_o(ready_2_o_2), .col_o(col_o),.*);

always #(`clk_per/2) clk = ~clk;

initial begin
   // Initialize TB signals
   clk = 1'b0;
   reset_i = 1'b1;
   write = 1'b1;
   chipselect = 1'b1;
   enable_i = 1'b1;
   done_1_i = 1'b0;
   done_2_i = 1'b0;
   ready_i = 1'b1;
   col_1_i = 32'b00000000000000000000000000000001;
   col_2_i = 32'b00000000000000000000000000000001;
   op = 8'b00000000;

   //#50;
   #100;
   #100;
   #100;
   read_1_i = 1'b1;
   read_2_i = 1'b1;
   #100;
   #100;
   if(ready_1_o==1'b1)
      col_1_i = col_1_i + 32'b00000000000000000000000000000001;
   #100;
   if(ready_1_o==1'b1)
      col_1_i = col_1_i + 32'b00000000000000000000000000000001;
   #100;
   if(ready_1_o==1'b1)
      col_1_i = col_1_i + 32'b00000000000000000000000000000001;
   #100;
   if(ready_1_o==1'b1)
      col_1_i = col_1_i + 32'b00000000000000000000000000000001;
   #100;
   if(ready_1_o==1'b1)
      col_1_i = col_1_i + 32'b00000000000000000000000000000001;
   #100;
   if(ready_1_o==1'b1)
      col_1_i = col_1_i + 32'b00000000000000000000000000000001;
   #100;
   if(ready_1_o==1'b1)
      col_1_i = col_1_i + 32'b00000000000000000000000000000001;
   #100;
   if(ready_1_o==1'b1)
      col_1_i = col_1_i + 32'b00000000000000000000000000000001;
   #100;
   if(ready_1_o==1'b1)
      col_1_i = col_1_i + 32'b00000000000000000000000000000001;
   #100;
   if(ready_1_o==1'b1)
      col_1_i = col_1_i + 32'b00000000000000000000000000000001;
   #100;
   if(ready_1_o==1'b1)
      col_1_i = col_1_i + 32'b00000000000000000000000000000001;
   #100;
   if(ready_1_o==1'b1)
      col_1_i = col_1_i + 32'b00000000000000000000000000000001;
   #100;
   if(ready_1_o==1'b1)
      col_1_i = col_1_i + 32'b00000000000000000000000000000001;
   #100;
   if(ready_1_o==1'b1)
      col_1_i = col_1_i + 32'b00000000000000000000000000000001;
   #100;
   if(ready_1_o==1'b1)
      col_1_i = col_1_i + 32'b00000000000000000000000000000001;
   #100;
   if(ready_1_o==1'b1)
      col_1_i = col_1_i + 32'b00000000000000000000000000000001;
   #100;
}
if(ready_1_o==1'b1)
  col_1_i = col_1_i + 32'b00000000000000000000000000000001;
#100;
if(ready_1_o==1'b1)
  col_1_i = col_1_i + 32'b00000000000000000000000000000001;
#100;
if(ready_1_o==1'b1)
  col_1_i = col_1_i + 32'b00000000000000000000000000000001;
#100;
#100;
#100;
#100;
#100;
#100;
#100;
#100;
#100;
end

///////////////////////////////////////////////////////////////////////////////
// Always End at Max_Sim_Time
///////////////////////////////////////////////////////////////////////////////

initial
begin
$monitor("%d %d", col_1_i, col_o);
#('Max_Sim_Time);
$stop;
end
endmodule

signal2_test.sv

//-- COLUMN_WIDTH is the width of a record in bits

module signal2_test();

#define clk_per 100
#define Max_Sim_Time 1000000000  // stop simulation after this time (ns)

//data in AvalonST
logic [COLUMN_WIDTH-1:0] col_1_i;
logic [COLUMN_WIDTH-1:0] col_2_i;

//control in AvalonMM
logic [7:0] op;
logic write;
logic chipselect;
logic enable_i;

//data out AvalonST
logic [COLUMN_WIDTH-1:0] col_o;

//AvalonST
logic read_1_i;
logic read_2_i;
logic read_o;

//AvalonST
logic ready_i;
logic ready_1_o;
logic ready_2_o_1;
logic ready_2_o_2;
logic ready_2_o_3;

//AvalonST
logic done_1_i;
logic done_2_i;
logic done_o;

/* std signals */
logic clk;
logic reset_i; // reset everything to zeros

logic [COLUMN_WIDTH-1:0] col_1_1;
logic [COLUMN_WIDTH-1:0] col_1_2;
logic done_1_1;
logic done_1_2;
logic ready_1_1;
logic ready_1_2;
logic read_1_1;
logic read_1_2;
alu alu_inst1(.col_o(col_1_1), .read_o(read_1_1), .done_o(done_1_1),
.ready_i(ready_1_2),.ready_2_o(ready_2_o_1),.ready_1_o(ready_1_o),.*);
alu alu_inst2(.col_1_i(col_1_1), .read_1_i(read_1_1), .done_1_i(done_1_1), .ready_i(ready_1_1), .ready_1_o(ready_1_2),
.col_o(col_1_2), .read_o(read_1_2), .done_o(done_1_2), .ready_2_o(ready_2_o_2),.*);
alu alu_inst3(.col_1_i(col_1_2), .read_1_i(read_1_2),
.done_1_i(done_1_2), .ready_1_o(ready_1_1), .ready_2_o(ready_2_o_3),*);
always #(`clk_per/2) clk = ~clk;

initial begin
  // Initialize TB signals
  clk = 1'b0;
  reset_i = 1'b1;  
  write = 1'b1;  
  chipselect = 1'b1;
  enable_i = 1'b1;
  done_1_i = 1'b0;
  done_2_i = 1'b0;
  ready_i = 1'b1;
  col_1_i = 32'b00000000000000000000000000000001;
  col_2_i = 32'b00000000000000000000000000000001;
  op = 8'b00000000;
  //#50;
  #100;
  reset_i = 1'b0;
  #100;
  #100;
  read_1_i = 1'b1;
  read_2_i = 1'b1;
  #100;
  if(ready_1_o==1'b1)
    col_1_i = col_1_i + 32'b00000000000000000000000000000001;
  #100;
  if(ready_1_o==1'b1)
    col_1_i = col_1_i + 32'b00000000000000000000000000000001;
  #100;
  if(ready_1_o==1'b1)
    col_1_i = col_1_i + 32'b00000000000000000000000000000001;
#100;
if(ready_1_o==1'b1)
  col_1_i = col_1_i + 32'b00000000000000000000000000000001;
#100;
if(ready_1_o==1'b1)
  col_1_i = col_1_i + 32'b00000000000000000000000000000001;
#100;
if(ready_1_o==1'b1)
  col_1_i = col_1_i + 32'b00000000000000000000000000000001;
#100;
if(ready_1_o==1'b1)
  col_1_i = col_1_i + 32'b00000000000000000000000000000001;
#100;
if(ready_1_o==1'b1)
  col_1_i = col_1_i + 32'b00000000000000000000000000000001;
#100;
if(ready_1_o==1'b1)
  col_1_i = col_1_i + 32'b00000000000000000000000000000001;
#100;
end

/////////////////////////////////////////////////////////////////////////////
// Always End at Max_Sim_Time
/////////////////////////////////////////////////////////////////////////////

initial
begin
$monitor("%d %d", col_1_i, col_o);
#(Max_Sim_Time);
$stop;
end
endmodule

Testing Scripts

compile_all.sh

#!/bin/bash
touch compilation.log
echo -e "Compilation history \n" > compilation.log
echo "" > compilation.log
cd ../../standard/
for module in aggregator alu boolgen coljoiner ; do
  echo -e $module > ../testers/scripts/compilation.log
  ../testers/scripts/modules/$module/verilator.sh > ../testers/scripts/compilation.log 2>&1
  echo "" >> ../testers/scripts/compilation.log
done
echo -e "Compilation complete" >> ../testers/scripts/compilation.log
cd ../testers/scripts
diff ../compilation.log ../modules/success_verilator.log

vsim_all.sh

#!/bin/bash
touch vsim.log
echo -e "Vsim history \n" > vsim.log
echo -e "" >> vsim.log
test=""test"

cd modules/
for module in boolgen signal1 signal2 colfilter coljoiner; do
echo -e "$module" >> ../vsim.log
cd $module
/opt/altera/13.1/modelsim_ase/bin/vlib ./work >> ../../vsim.log 2>&1
/opt/altera/13.1/modelsim_ase/bin/vlog *.sv >> ../../vsim.log 2>&1

./opt/altera/13.1/modelsim_ase/bin/vsim -c work.$module$test -do 'run 10us;quit' >> ../../vsim.log 2>&1
rm -rf work
cd ..
echo -e "" >> ../vsim.log
done
echo -e "Modelsim complete" >> ../vsim.log
cd ../
diff ../vsim.log ../modules/success_vsim.log

Drivers

alu.c

/*
 * Device driver for the DPU ALU
 * 
 * A Platform device implemented using the misc subsystem
 * 
 * Andrea Lottarini and Tim Paine
 * Columbia University
 * 
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 * drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 * 
 * "make" to build
 * insmod alu.ko
 * 
 * Check code style with
 * checkpatch.pl --file --no-tree alu.c
 */

#include <linux/module.h>
#include <linux/init.h>
```c
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "alu.h"

#define DRIVER_NAME "alu"

/*
 * Information about our device
 */
struct alu_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers can be accessed in memory */

    unsigned char op;
}
dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_digit(unsigned char op)
{
    iowrite8((u8) op, dev.virtbase);
    dev.op = op;
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long alu_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    alu_arg_t alua;

    switch (cmd) {
    case ALU_WRITE_OP:
        pr_info("Here2\n");
        if (copy_from_user(&alua, (alu_arg_t *) arg, sizeof(alu_arg_t)))
            return -EACCES;
        write_digit(alua.op);
        break;
    case ALU_READ_OP:
        pr_info("Here3\n");
        if (copy_to_user((unsigned char *)arg, &dev.op), sizeof(unsigned char))
            return -EACCES;
        break;
    default:
        return -EINVAL;
    }

    return 0;
}
```
/* The operations our device knows how to do */
static const struct file_operations alu_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = alu_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice alu_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = DRIVER_NAME,
    .fops = &alu_fops,
};

/* Initialization code: get resources (registers) and display
   a welcome message */
static int __init alu_probe(struct platform_device *pdev)
{
    int ret;

    /* Register ourselves as a misc device: creates /dev/alu */
    ret = misc_register(&alu_misc_device);

    /* Get the address of our registers from the device tree */
    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret) {
        ret = -ENOENT;
        goto out_deregister;
    }

    /* Make sure we can use these registers */
    if (request_mem_region(dev.res.start, resource_size(&dev.res), DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }

    /* Arrange access to our registers */
    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_mem_region;
    }

    /* Clean-up code: release resources */
    return 0;

out_release_mem_region:
    release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
    misc_deregister(&alu_misc_device);
    return ret;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id alu_of_match[] = {
    {.compatible = "altr,alu"},
};
MODULE_DEVICE_TABLE(of, alu_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver alu_driver = {
    .driver = {
        .name = DRIVER_NAME,
        .owner = THIS_MODULE,
        .of_match_table = of_match_ptr(alu_of_match),
    },
    .remove = __exit_p(alu_remove),
};

/* Called when the module is loaded: set things up */
static int __init alu_init(void)
{
    pr_info(DRIVER_NAME " : init
");
    return platform_driver_probe(&alu_driver, alu_probe);
}

/* Called when the module is unloaded: release resources */
static void __exit alu_exit(void)
{
    platform_driver_unregister(&alu_driver);
    pr_info(DRIVER_NAME " : exit
");
}

module_init(alu_init);
module_exit(alu_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Andrea Lottarini and Tim Paine, Columbia University");
MODULE_DESCRIPTION("Driver for DPU ALU");

alu.h

#define _ALU_H

#include <linux/ioctl.h>

typedef struct {
    unsigned char op; /* 0, 1 */
} alu_arg_t;

#define ALU_MAGIC 'q'

/* ioctls and their arguments */
#define ALU_WRITE_OP _IOW(ALU_MAGIC, 1, alu_arg_t *)
#define ALU_READ_OP _IOWR(ALU_MAGIC, 2, alu_arg_t *)

#endif
/*
 * Device driver for the DPU filter
 *
 * A Platform device implemented using the misc subsystem
 *
 * Andrea Lottarini and Tim Paine
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 * drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod filter.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree filter.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/io.h>
#include <linux/platform_device.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "filter.h"

#define DRIVER_NAME "filter"

/*
 * Information about our device
 */
struct filter_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers can be accessed in memory */
    unsigned char op;
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_digit(unsigned char op) {
    iowrite8((u8) op, dev.virtbase);
    dev.op = op;
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long filter_ioctl(struct file *f, unsigned int cmd, unsigned long arg) {
    filter_arg_t filtera;
    switch (cmd) {
    case filter_WRITE_OP:
        pr_info("Here2\n");
        if (copy_from_user(&filtera, (filter_arg_t *)arg,
                          sizeof(filter_arg_t)))
            return -EACCES;
        write_digit(filtera.op);
        break;
    case filter_READ_OP:
        pr_info("Here3\n");
        if (copy_to_user((unsigned char *)arg, &dev.op),
            sizeof(unsigned char)))
            return -EACCES;
        break;
    default:
        return -EINVAL;
    }
    return 0;
}

static const struct file_operations filter_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = filter_ioctl,
};

static struct miscdevice filter_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = DRIVER_NAME,
    .fops = &filter_fops,
};

__init int filter_probe(struct platform_device *pdev) {
    int ret;
    /* Register ourselves as a misc device: creates /dev/filter */
    ret = misc_register(&filter_misc_device);
    /* Get the address of our registers from the device tree */
    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret)
        goto out_deregister;
    /* Make sure we can use these registers */
    if (request_mem_region(dev.res.start, resource_size(&dev.res),
                            DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }
    /* Arrange access to our registers */
dev.virtbase = of_iomap(pdev->dev.of_node, 0);
if (dev.virtbase == NULL) {
    ret = -ENOMEM;
    goto out_release_mem_region;
}

return 0;

out_release_mem_region:
    release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
    misc_deregister(&filter_misc_device);
    return ret;
}

/* Clean-up code: release resources */
static int filter_remove(struct platform_device *pdev)
{
    iounmap(dev.virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    misc_deregister(&filter_misc_device);
    return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id filter_of_match[] = {
    { .compatible = "altr,filter" },
    {});
MODULE_DEVICE_TABLE(of, filter_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver filter_driver = {
    .driver = {
        .name = DRIVER_NAME,
        .owner = THIS_MODULE,
        .of_match_table = of_match_ptr(filter_of_match),
    },
    .remove = __exit_p(filter_remove),
};

/* Called when the module is loaded: set things up */
static int __init filter_init(void)
{
    pr_info(DRIVER_NAME " : init\n");
    return platform_driver_probe(&filter_driver, filter_probe);
}

/* Called when the module is unloaded: release resources */
static void __exit filter_exit(void)
{
    platform_driver_unregister(&filter_driver);
    pr_info(DRIVER_NAME " : exit\n");
}

module_init(filter_init);
module_exit(filter_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Tim Paine, Columbia University");
MODULE_DESCRIPTION("Driver for DPU filter");
filter.h

```c
#ifndef _FILTER_H
#define _FILTER_H

#include <linux/ioctl.h>

typedef struct {
    unsigned char op; /* 0, 1 */
} filter_arg_t;

#define FILTER_MAGIC 'q'

/* ioctls and their arguments */
#define FILTER_WRITE_OP _IOW(FILTER_MAGIC, 1, filter_arg_t *)
#define FILTER_READ_OP _IOWR(FILTER_MAGIC, 2, filter_arg_t *)

#endif
```

joiner.c

```c
/*
 * Device driver for the DPU joiner
 *
 * A Platform device implemented using the misc subsystem
 *
 * Andrea Lottarini and Tim Paine
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod joiner.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree joiner.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "joiner.h"

#define DRIVER_NAME "joiner"

/*
 * Information about our device
 */
```
struct joiner_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers can be accessed in memory */
    unsigned char op;
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_digit(unsigned char op)
{
    iowrite8((u8) op, dev.virtbase);
    dev.op = op;
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long joiner_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    joiner_arg_t joinera;
    switch (cmd) {
    case joiner_WRITE_OP:
        pr_info("Here2\n");
        if (copy_from_user(&joinera, (joiner_arg_t *) arg,
                        sizeof(joiner_arg_t)))
            return -EACCES;
        write_digit(joinera.op);
        break;
    case joiner_READ_OP:
        pr_info("Here3\n");
        if (copy_to_user((unsigned char *)arg, &dev.op),
           sizeof(unsigned char))
            return -EACCES;
        break;
    default:
        return -EINVAL;
    }
    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations joiner_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = joiner_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice joiner_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = DRIVER_NAME,
    .fops = &joiner_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init joiner_probe(struct platform_device *pdev)
{
    int ret;

    /* Register ourselves as a misc device: creates /dev/joiner */
    ret = misc_register(&joiner_misc_device);

    /* Get the address of our registers from the device tree */
    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret) {
        ret = -ENOENT;
        goto out_deregister;
    }

    /* Make sure we can use these registers */
    if (request_mem_region(dev.res.start, resource_size(&dev.res),
                           DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }

    /* Arrange access to our registers */
    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_mem_region;
    }

    return 0;
}

out_release_mem_region:
    release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
    misc_deregister(&joiner_misc_device);
    return ret;
}

/* Clean-up code: release resources */
static int joiner_remove(struct platform_device *pdev)
{
    iounmap(dev.virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    misc_deregister(&joiner_misc_device);
    return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id joiner_of_match[] = {
    { .compatible = "altr,joiner" },
};
MODULE_DEVICE_TABLE(of, joiner_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver joiner_driver = {
    .driver = {
        .name = DRIVER_NAME,
        .owner = THIS_MODULE,
        .of_match_table = of_match_ptr(joiner_of_match),
    },
    .remove = __exit_p(joiner_remove),
};
/* Called when the module is loaded: set things up */
static int __init joiner_init(void)
{
    pr_info(DRIVER_NAME " : init\n");
    return platform_driver_probe(&joiner_driver, joiner_probe);
}

/* Called when the module is unloaded: release resources */
static void __exit joiner_exit(void)
{
    platform_driver_unregister(&joiner_driver);
    pr_info(DRIVER_NAME " : exit\n");
}

module_init(joiner_init);
module_exit(joiner_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Tim Paine, Columbia University");
MODULE_DESCRIPTION("Driver for DPU joiner");

joiner.h

#ifndef _JOINER_H
#define _JOINER_H
#include <linux/ioctl.h>

typedef struct {
    unsigned char op; /* 0, 1*/
} joiner_arg_t;

#define JOINER_MAGIC 'q'

/* ioctls and their arguments */
#define JOINER_WRITE_OP _IOW(JOINER_MAGIC, 1, joiner_arg_t *)
#define JOINER_READ_OP _IOWR(JOINER_MAGIC, 2, joiner_arg_t *)

#endif

aggregator.c

/*
 * Device driver for the DPU aggregator
 * 
 * A Platform device implemented using the misc subsystem
 * 
 * Andrea Lottarini and Tim Paine
 * Columbia University
 * 
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 * drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 * 
 * "make" to build
 * insmod aggregator.ko
 */
* Check code style with
* checkpatch.pl --file --no-tree aggregator.c
*/

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/of.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "aggregator.h"
#define DRIVER_NAME "aggregator"

/*
 * Information about our device
 */
struct aggregator_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers can be accessed in memory */
    unsigned char op;
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_digit(unsigned char op)
{
    iowrite8((u8) op, dev.virtbase);
    dev.op = op;
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long aggregator_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    aggregator_arg_t aggregatora;

    switch (cmd) {
    case aggregator_WRITE_OP:
        pr_info("Here2\n");
        if (copy_from_user(&aggregatora, (aggregator_arg_t *) arg,
            sizeof(aggregator_arg_t)))
            return -EACCES;
        write_digit(aggregatora.op);
        break;
    case aggregator_READ_OP:
        pr_info("Here3\n");
        if (copy_to_user((unsigned char *)arg, &(dev.op),
            sizeof(unsigned char)))
            return -EACCES;
        break;
    }
default:    return -EINVAL;
}
    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations aggregator_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = aggregator_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice aggregator_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = DRIVER_NAME,
    .fops = &aggregator_fops,
};

/* Initialization code: get resources (registers) and display
 * a welcome message */
static int __init aggregator_probe(struct platform_device *pdev)
{
    int ret;
    /* Register ourselves as a misc device: creates /dev/aggregator */
    ret = misc_register(&aggregator_misc_device);
    /* Get the address of our registers from the device tree */
    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret) {
        ret = -ENOENT;
        goto out_deregister;
    }
    /* Make sure we can use these registers */
    if (request_mem_region(dev.res.start, resource_size(&dev.res),
        DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }
    /* Arrange access to our registers */
    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_mem_region;
    }
    out_release_mem_region:
    release_mem_region(dev.res.start, resource_size(&dev.res));
    out_deregister:
    misc_deregister(&aggregator_misc_device);
    return ret;
}

/* Clean-up code: release resources */
static int aggregator_remove(struct platform_device *pdev)
{
    iounmap(dev.virtbase);
release_mem_region(dev.res.start, resource_size(&dev.res));
misc_deregister(&aggregator_misc_device);
return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id aggregator_of_match[] = {
   { .compatible = "altr,aggregator" },
   {};
}
MODULE_DEVICE_TABLE(of, aggregator_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver aggregator_driver = {
   .driver = {
      .name = DRIVER_NAME,
      .owner = THIS_MODULE,
      .of_match_table = of_match_ptr(aggregator_of_match),
   },
   .remove = __exit_p(aggregator_remove),
};

/* Called when the module is loaded: set things up */
static int __init aggregator_init(void)
{
    pr_info(DRIVER_NAME "\n");
    return platform_driver_probe(&aggregator_driver, aggregator_probe);
}

/* Called when the module is unloaded: release resources */
static void __exit aggregator_exit(void)
{
    platform_driver_unregister(&aggregator_driver);
    pr_info(DRIVER_NAME "\n");
}

module_init(aggregator_init);
module_exit(aggregator_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Tim Paine, Columbia University");
MODULE_DESCRIPTION("Driver for DPU aggregator");

aggregator.h

#ifndef _AGGREGATOR_H
#define _AGGREGATOR_H

#include <linux/ioctl.h>

typedef struct {
   unsigned char op; /* 0, 1*/
} aggregator_arg_t;

#define AGGREGATOR_MAGIC 'q'

/* ioctls and their arguments */
#define AGGREGATOR_WRITE_OP _IOW(AGGREGATOR_MAGIC, 1, aggregator_arg_t *)
#define AGGREGATOR_READ_OP _IOWR(AGGREGATOR_MAGIC, 2, aggregator_arg_t *)
fifo.c

/*
 * Device driver for the Altera FIFO
 *
 * A Platform device implemented using the misc subsystem
 *
 * Andrea Lottarini
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 * drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod fifo0.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree fifo_data0.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "fifo.h"

#define DRIVER_NAME "fifo0"

/*
 * Information about our device
 */
struct fifo_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers can be accessed in memory */

    struct resource status_res; /* register where I can read the status */
    void __iomem *status_virtbase; /* Where this register can be accessed in memory */
} dev;

/*
 * Handle ioctl() calls from userspace:
 * This believes whatever the user passes without checking it
 */
static long fifo_ioctl(struct file *f, unsigned int cmd, unsigned long arg) {

int i, specs, fill, to_write, to_read;
opcode* op = (opcode*) arg;

switch (cmd) {
  case FIFO_WRITE_DATA:
    // first check that the fifo is not full
    fill = ioread32(dev.status_virtbase);
    to_write = MIN(FIFO_SIZE - fill, op->length);
    //printk("Writer Driver - I received an order for %d writes and I can do %d\n", op->length, to_write);
    if (to_write > 1) {
      iowrite32(START_PACKET_CHANNEL0, dev.virtbase+4);
      /* trusting the user buffer to avoid copying that too */
      for (i = 0; i < to_write; i++) {
        if (i == (to_write - 1)) { /* write the end packet flag before writing the last int*/
          if (op->done && to_write == op->length){/* that was
            iowrite32(DONE_END_PACKET_CHANNEL0, dev.virtbase+4);
          } else {
            iowrite32(END_PACKET_CHANNEL0, dev.virtbase+4);
          }
        }
      }
      iowrite32(op->buf[i], dev.virtbase);
    }
    else {
      if (to_write > 0) {
        if (op->done && to_write == op->length) {
          iowrite32(DONE_SINGLE_PACKET_CHANNEL0, dev.virtbase+4);
        } else {
          iowrite32(SINGLE_PACKET_CHANNEL0, dev.virtbase+4);
        }
      }
      iowrite32(op->buf[0], dev.virtbase);
    }
    /* write back in the op struct how many int were actually sent */
    op->length = to_write;
    break;
  case FIFO_READ_DATA:
    fill = ioread32(dev.status_virtbase);
    to_read = MIN(fill, op->length);
    //printk("Reader Driver - I received an order for %d reads but there are %d in the fifo\n", op->length, fill);
    if (fill > 0) {
      /* trusting the user buffer to avoid copying that too */
      for (i = 0; i < to_read; i++) {
        op->buf[i] = ioread32(dev.virtbase);
      }
      /* write back in the op struct how many int were actually read */
      op->length = to_read;
      /* check if it was the last one */
      specs = ioread32(dev.virtbase+4);
      //printk("Reader Driver - these are the specs: %d\n", specs);
      if (specs & DONE_MASK){
        op->done = 1;
      } else {
        op->done = 0;
      }
  }
if (op->length = 0) {
    op->done = 0;
}
break;

case FIFO_READ_STATUS:
    fill = ioread32(dev.status_virtbase);
    if (copy_to_user((int*)arg, &fill, sizeof(int)) )
        return -EACCES;
    break;

default:
    return -EINVAL;
}
return 0;

/* The operations our device knows how to do */
static const struct file_operations fifo_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = fifo_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice fifo_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = DRIVER_NAME,
    .fops = &fifo_fops,
};

/* Initialization code: get resources (registers) and display * a welcome message */
static int __init fifo_probe(struct platform_device *pdev)
{
    int ret;

    pr_info(DRIVER_NAME " : probe\n");

    /* Register ourselves as a misc device: creates /dev/fifo */
    ret = misc_register(&fifo_misc_device);

    /* Get the address of data registers from the device tree */
    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret) {
        ret = -ENOENT;
        goto out_deregister;
    }

    printk("DEVICE DATA START %x END %x \n", dev.res.start, dev.res.end);

    /* Make sure we can use these registers */
    if (request_mem_region(dev.res.start, resource_size(&dev.res),
        DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }

    /* Get the address of status registers from the device tree */
    ret = of_address_to_resource(pdev->dev.of_node, 1, &dev.status_res);
if (ret) {
    ret = -ENOENT;
    goto out_deregister;
}

printk("DEVICE STATUS START %x END %x \n", dev.status_res.start, dev.status_res.end);

/* Make sure we can use these registers */
if (request_mem_region(dev.status_res.start, resource_size(&dev.status_res), DRIVER_NAME) == NULL) {
    ret = -EBUSY;
    goto out_deregister;
}

/* Arrange access to data registers */
dev.virtbase = of_iomap(pdev->dev.of_node, 0);
if (dev.virtbase == NULL) {
    ret = -ENOMEM;
    goto out_release_mem_region2;
}

printk("VIRTUAL ADDRESS OF DATA FIFO: %x \n",(unsigned int) dev.virtbase);

/* Arrange access to status registers */
dev.status_virtbase = of_iomap(pdev->dev.of_node, 1);
if (dev.status_virtbase == NULL) {
    ret = -ENOMEM;
    goto out_release_mem_region1;
}

printk("VIRTUAL ADDRESS OF STATUS FIFO: %x \n",(unsigned int) dev.status_virtbase);

return 0;

out_release_mem_region1:
    release_mem_region(dev.res.start, resource_size(&dev.res));
out_release_mem_region2:
    release_mem_region(dev.status_res.start, resource_size(&dev.status_res));
out_deregister:
    misc_deregister(&fifo_misc_device);
    return ret;
}

/* Clean-up code: release resources */
static int fifo_remove(struct platform_device *pdev)
{
    iounmap(dev.virtbase);
    iounmap(dev.status_virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    release_mem_region(dev.status_res.start, resource_size(&dev.status_res));
    misc_deregister(&fifo_misc_device);
    return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id fifo_of_match[] = {
    { .compatible = "altr,fifo0" },
    { },
};
MODULE_DEVICE_TABLE(of, fifo_of_match);
#endif

/* Information for registering ourselves as a *platform* driver */
static struct platform_driver fifo_driver = {

.driver = {
    .name = DRIVER_NAME,
    .owner = THIS_MODULE,
    .of_match_table = of_match_ptr(fifo_of_match),
},
.remove = __exit_p(fifo_remove),
};

/* Called when the module is loaded: set things up */
static int __init fifo_init(void)
{
    pr_info(DRIVER_NAME " : init
";
    return platform_driver_probe(&fifo_driver, fifo_probe);
}

/* Called when the module is unloaded: release resources */
static void __exit fifo_exit(void)
{
    platform_driver_unregister(&fifo_driver);
    pr_info(DRIVER_NAME " : exit
";
}

module_init(fifo_init);
module_exit(fifo_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Andrea Lottarini, Columbia University");
MODULE_DESCRIPTION("Altera FIFO driver");

fifo.h

#ifndef _FIFO_DATA_H
#define _FIFO_DATA_H

#include <linux/ioctl.h>

#define FIFO_MAGIC 'q'

#define SINGLE_PACKET_CHANNEL 0b00000000000000000000000000000011
#define START_PACKET_CHANNEL 0b00000000000000000000000000000001
#define END_PACKET_CHANNEL 0b00000000000000000000000000000010
#define DONE_END_PACKET_CHANNEL 0b00000000000000000000000000000010
#define DONE_SINGLE_PACKET_CHANNEL 0b00000000000000000000000000000011
#define DONE_MASK 0b00000000000000000000000000000000

#define FIFO_SIZE 16

#define IS_DONE(A) ((A) & DONE_MASK)

#define MIN(A,B) ((A) < (B) ? (A) : (B))

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

/* ioctls and their arguments */
#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*)
Utility

Makefiles (FIFO as example)

```makefile
ifeq ($(KERNELRELEASE),)

# KERNELRELEASE defined: we are being compiled as part of the Kernel
obj-m := fifo0.o fifo1.o fifo2.o

else # We are being compiled as a module: use the Kernel build system

#if we are cross compiling from the host use its kernel and don’t compile the test program
ifeq ($(CROSS_COMPILE),)
KERNEL_SOURCE := /usr/src/linux
default: module
else
KERNEL_SOURCE := /export/board02/root/usr/src/linux_for_host
default: module
endif

PWD := $(shell pwd)

module:
  $(MAKE) -C $(KERNEL_SOURCE) SUBDIRS=$(PWD) modules

clean:
  $(MAKE) -C $(KERNEL_SOURCE) SUBDIRS=$(PWD) clean
  $(RM) *~

endif
```

benchmark.c

```c
#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include "fifo.h"

#define NTHREADS 2

#define IOCTL(A,B,C)\
  if(ioctl(A, B, &C)) perror("ioctl(A) failed"); -1;

typedef enum {READ, WRITE} transfer_type;

typedef struct {
    char* fifo_name;  
    transfer_type t;  
    /* if t is WRITE then DATA has been allocated and there are LENGTH elemets to transfer
```
otherwise DATA is a buffer of length LENGTH (you should check that is is enough to contain the data)
*/  
int * data;
int length;
}
job;

void * worker (void* arg){
  int to_write, length;
  job * j = (job*) arg;
  int fifo_fd;
  opcode op;

  if ( (fifo_fd = open(j->fifo_name, O_RDWR)) == -1) {
    fprintf(stderr, "could not open %s
", j->fifo_name);
    return (void*) -1;
  }

  if (j->t == READ){
    print("Read thread started
");
    length = 0;
    do{
      /* create a struct for the job */
      op = (opcode) {FIFO_SIZE, 0, &(j->data[length]);
      /* tell the driver to copy stuff */
      if (ioctl(fifo_fd, FIFO_READ_DATA, &op)) {
        perror("ioctl(FIFO_READ_DATA) failed");
        return (void*) -1;
      }
      #ifdef DEBUG
      for ( i = length ; i < length + op.length ; i++ ){
        printf("Data: %d
",j->data[i]);
      }
      #endif
      /* adjust pointer in the read buffer */
      length += op.length;
      if (op.length == 0){
        #ifdef DEBUG
        printf("I've read %d elements - I am going to yield\n",op.length);
        #endif
        pthread_yield();
      }
    }while(!op.done); /* until you see the done signal reported back by ioctl*/
    print("READER is done!
");
  }else{  /*write task*/
    print("Writer thread started\n");
    length = 0;
    while (length < j->length){
      /* see if we are at the end of the stream */
      to_write = MIN( FIFO_SIZE , j->length - length );
      op = (opcode) (to_write, 0, &(j->data[length]));
      if( length + to_write == j->length ){
        printf("WRITER is sending done\n");
        op.done = 1;
      }
    }
#ifdef DEBUG
printf("Writer - I am going to ship %d element\n", to_write);
#endif

/* tell the driver to copy stuff */
if (ioctl(fifo_fd, FIFO_WRITE_DATA, &op)) {
    perror("ioctl(FIFO_WRITE_DATA) failed");
    return (void*) -1;
}

/* adjust index in the write buffer */
length += op.length;
if (!op.length){
    #ifdef DEBUG
    printf("I've wrote %d elements - I am going to yield\n", op.length);
    #endif
    pthread_yield();
}
}

printf("WRITER is done!\n");

int main(int argc, char * argv[])
{
    int i, length;
    int * data_in,* data_out;

    char filename0[] = "/dev/fifo0";
    char filename1[] = "/dev/fifo1";
    struct timeval start, end;

    /* get the size of the data to transfer from the user */
    if (argc != 2){
        printf("Usage: %s %d
where n is size of data transferred\n", argv[0]);
        exit(1);
    }
    length = atoi(argv[1]);

    if (length < 1){
        printf("The data should have positive size (%d < 1)\n", length);
        exit(1);
    }

data_out = (int*) malloc(length*sizeof(int));
data_in = (int*) malloc(length*sizeof(int));

    srand(time(NULL));
    for (i = 0 ; i < length ; i++ ){
        data_in[i] = rand();
    }

    pthread_t threads[NTHREADS];
    job write_job = {filename0 , WRITE, data_in, length};
    job read_job = {filename1 , READ, data_out, length};

    printf("FIFO Userspace program started\n");
    printf("Thread structs allocated\n");
    gettimeofday(&start, NULL);
    pthread_create(&threads[0], NULL, worker, &write_job);
    pthread_create(&threads[1], NULL, worker, &read_job);

    //printf("Threads started!\n");
    for ( i = 0 ; i < NTHREADS ; i++ ){
```c
#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>
#include "../fifo_driver/fifo.h"
#include "../alu­sw/alu.h"

#define NTHREADS 3
#define FIFO_SIZE 16

typedef enum {READ, WRITE} transfer_type;

typedef struct {
    char* fifo_name;
    transfer_type t;
    /* if t is WRITE then DATA has been allocated and there are LENGTH elements to transfer
    otherwise DATA is a buffer of length LENGTH (you should check that is is enough to contain the data)
    */
    int * data;
    int length;
} job;

void * worker (void* arg){
    int to_write, all, length, i;
    job * j = (job*) arg;
    int fifo_fd;
    opcode op;

    if ( (fifo_fd = open(j->fifo_name, O_RDWR)) == -1) {
        fprintf(stderr, "could not open %s\n", j->fifo_name);
        return (void*) -1;
    }

    if (j->t == READ){
        //print("Read thread started\n");
        length = 0;
        all = 0;
    }
```
do{
/* create a struct for the job */
op = (opcode) {FIFO_SIZE, 0, &(j->data[length])};
/* tell the driver to copy stuff */
if (ioctl(fifo_fd, FIFO_READ_DATA, &op)) {
    perror("ioctl(FIFO_READ_DATA) failed");
    return (void*) -1;
}
for ( i = length ; i < length + op.length ; i++ ){
    printf("Data: %d\n",j->data[i]);
    all++;
}
/* adjust pointer in the read buffer */
length += op.length;
if (op.length == 0){
    //printf("I've read %d elements - I am going to yield\n",op.length);
    pthread_yield();
}
}while(!op.done&&(all<j->length)); /* until you see the done signal reported back by ioctl*/
//printf("READER is done!\n");
}else{ /*write task*/
/*write task*/
//printf("Writer thread started\n");
length = 0;
while (length < j->length){
    /* see if we are at the end of the stream */
to_write = MIN( FIFO_SIZE , j->length - length );
op = (opcode) {to_write, 0, &(j->data[length])};
if (length + to_write == j->length ){
    //printf("WRITER is sending done!\n");
    op.done = 1;
}
    //printf("Writer - I am going to ship %d element\n",to_write);
    /* tell the driver to copy stuff */
    if (ioctl(fifo_fd, FIFO_WRITE_DATA, &op)) {
        perror("ioctl(FIFO_WRITE_DATA) failed");
        return (void*) -1;
    }
    /* adjust index in the write buffer */
    length += op.length;
    if (length==0){
        //printf("I've wrote %d elements - I am going to yield\n",op.length);
        pthread_yield();
    }
    //printf("WRITER is done!\n");
}
return 0;
}
int main(int argc, char * argv[]){
    int i,length,alu_fd;
    int * data_in0, * data_in1,* data_out , *golden;

    char filename0[] = "/dev/fifo0";
    char filename1[] = "/dev/fifo1";
    char filename2[] = "/dev/fifo2";
char alu[] = "/dev/alu";

struct timeval start, end;

/* get the size of the data to transfer from the user */
if (argc != 2){
    printf("Usage: %s n
where n is size of data transferred\n", argv[0]);
    exit(1);
} else{
    length = atoi(argv[1]);
}

if (length < 1){
    printf("The data should have positive size (%d < 1)\n", length);
    exit(1);
}

data_out = (int*) malloc(length*sizeof(int));
data_in0 = (int*) malloc(length*sizeof(int));
data_in1 = (int*) malloc(length*sizeof(int));
golden = (int*) malloc(length*sizeof(int));

srand(time(NULL));
for (i = 0 ; i < length ; i++ ){
    data_in0[i] = rand()%100;
    data_in1[i] = rand()%100;
    golden[i] = data_in0[i] + data_in1[i];
}

pthread_t threads[NTHREADS];
job write_job0 = {filename0 , WRITE, data_in0, length};
job write_job1 = {filename1 , WRITE, data_in1, length};
job read_job = {filename2 , READ, data_out, length};

printf("FIFO Userspace program started\n");
printf("Thread structs allocated\n\n");

if ( (alu_fd = open(alu, O_RDWR)) == -1) {
    fprintf(stderr, "could not open %s\n", alu);
    return -1;
}

alu_arg_t alua;
alua.op = 0;
if (ioctl(alu_fd, ALU_WRITE_OP, &alua ) ) {
    perror("ioctl(ALU_WRITE_OP) failed");
    return -1;
}

printf("\nOut 1:\t");
for (i = 0 ; i < length ; i++ ){
    printf("%d	", data_in0[i]);
}

printf("\nOut 2:\t");
for (i = 0 ; i < length ; i++ ){
    printf("%d	", data_in1[i]);
}

printf("Out:\t");
for (i = 0 ; i < length ; i++ ){
    printf("%d	", golden[i]);
}

printf("\n\n");

gettimeofday(&start, NULL);
```c
#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>

#include "./fifo_driver/fifo.h"
#include "./alu-sw/alu.h"
#include "./filter/filter.h"

#define NTHREADS 4
#define FIFO_SIZE 16

typedef enum {READ, WRITE} transfer_type;

typedef struct {
    char* fifo_name;
    transfer_type t;
    /* if t is WRITE then DATA has been allocated and there are LENGTH elemets to transfer
     * otherwise DATA is a buffer of length LENGTH (you should check that is is enough to contain the data)
     */
    int * data;
    int length;
} job;

void * worker (void* arg){
    int to_write, all, length, i;
    job * j = (job*) arg;
    int fifo_fd;
    opcode op;

    if ( (fifo_fd = open(j->fifo_name, O_RDWR)) == -1) {
```
fprintf(stderr, "could not open %s\n", j->fifo_name);
return (void*) -1;
}

if (j->t == READ){
  //printf("Read thread started\n");
  length = 0;
  all = 0;
  int max;
  max = 0;
  do{
      /* create a struct for the job */
      op = (opcode) {FIFO_SIZE, 0, &j->data[length]};
      /* tell the driver to copy stuff */
      if (ioctl(fifo_fd, FIFO_READ_DATA, &op)) {
          perror("ioctl(FIFO_READ_DATA) failed");
          return (void*) -1;
      }
      for (i = length ; i < length + op.length ; i++) {
          printf("Data: %d\n", j->data[i]);
          all++;
      }
      /* adjust pointer in the read buffer */
      length += op.length;
      if (op.length == 0){
          //printf("I've read %d elements - I am going to yield\n", op.length);
          if (max > 100)
              break;
          max++;
          pthread_yield();
      }
  }while(!op.done && (all < (j->length)));
  printf("READER is done!\n");
} else{ /*write task*/
  //printf("Writer thread started\n");
  length = 0;
  while (length < j->length){
      /* see if we are at the end of the stream */
      to_write = MIN(FIFO_SIZE , j->length - length);
      op = (opcode) {to_write, 0, &j->data[length]};
      if (length + to_write == j->length ){
          //printf("WRITER is sending done!\n");
          op.done = 1;
      }
      //printf("Writer - I am going to ship %d elements\n", to_write);
      /* tell the driver to copy stuff */
      if (ioctl(fifo_fd, FIFO_WRITE_DATA, &op)) {
          perror("ioctl(FIFO_WRITE_DATA) failed");
          return (void*) -1;
      }
      /* adjust index in the write buffer */
      length += op.length;
      if (op.length){
          //printf("I've wrote %d elements - I am going to yield\n", op.length);
          pthread_yield();
      }
  }
  //printf("WRITER is done!\n");
```c
int main(int argc, char * argv[])
{
    int i,length,alu_fd,filter0_fd, total;
    int * data_in0, * data_in1, *data_in3, * data_out , *golden;

    char filename0[] = "/dev/fifo0";
    char filename1[] = "/dev/fifo1";
    char filename2[] = "/dev/fifo2";
    char filename3[] = "/dev/fifo3";
    char alu[] = "/dev/alu";
    char filter0[] = "/dev/filter0";

    struct timeval start, end;

    /* get the size of the data to transfer from the user */
    if (argc != 2){
        printf ("Usage: %s n
   where n is size of data transferred\n",argv[0]);
        exit(1);
    }
    length = atoi(argv[1]);
    if (length < 1){
        printf("The data should have positive size (%d < 1)\n",length);
        exit(1);
    }

    data_out = (int*) malloc(length*sizeof(int));
data_in0 = (int*) malloc(length*sizeof(int));
data_in1 = (int*) malloc(length*sizeof(int));
data_in3 = (int*) malloc(length*sizeof(int));
golden = (int*) malloc(length*sizeof(int));

    total =0;

    srand(time(NULL));
    for (i = 0 ; i < length ; i++){
        data_in0[i] = rand()%100;
        data_in1[i] = 50;
    }

    for (i = 0 ; i < length ; i++){
        if(data_in0[i]<data_in1[i]){[49]
            data_in3[total] = 49;
            golden[i] = data_in3[total]+1;
            total++;
        } else {
            golden[i] = -1;
        }
    }

    pthread_t threads[NTHREADS];
    job write_job0 = {filename0 , WRITE , data_in0, length};
    job write_job1 = {filename1 , WRITE , data_in1, length};
    job write_job3 = {filename3 , WRITE , data_in3, total};
    job read_job = {filename2 , READ , data_out, total};

    printf("Userspace program started\n");

    printf("Thread structs allocated\n");

    return 0;
}
```
if ( (alu_fd = open(alu, O_RDWR)) == -1) {
    fprintf(stderr, "could not open %s\n", alu);
    return -1;
}

alu_arg_t alua;
alua.op = 0;
if (ioctl(alu_fd, ALU_WRITE_OP, &alua )) {
    perror("ioctl(ALU_WRITE_OP) failed");
    return -1;
}
printf("Alu op written\n");

if ( (filter0_fd = open(filter0, O_RDWR)) == -1) {
    fprintf(stderr, "could not open %s\n", filter0);
    return -1;
}

filter_arg_t fa0;
fa0.op = 5;
if (ioctl(filter0_fd, FILTER_WRITE_OP, &fa0 ) ) {
    perror("ioctl(FILTER_WRITE_OP) failed");
    return -1;
}
printf("Filter0 op written\n");

printf("Out 1:\t");
for (i = 0 ; i < length ; i++ )
    printf("%d\t",data_in0[i]);
printf("\nOut 2:\t");
for (i = 0 ; i < total ; i++ )
    printf("%d\t",data_in3[i]);
printf("\nOut:\t");
for (i = 0 ; i < length ; i++ )
    printf("%d\t",golden[i]);
printf("\n\n");

gmtimeofday(&start, NULL);
pthread_create(&(threads[0]), NULL, worker, &write_job0);
pthread_create(&(threads[1]), NULL, worker, &write_job1);
pthread_create(&(threads[3]), NULL, worker, &write_job3);
pthread_create(&(threads[2]), NULL, worker, &read_job);
//printf("Threads started!\n");
for ( i = 0 ; i < NTHREADS ; i++ )
    pthread_join(threads[i],NULL);

gmtimeofday(&end, NULL);
printf("TIME TAKEN %ld\n", ((end.tv_sec * 1000000 + end.tv_usec) - (start.tv_sec * 1000000 + start.tv_usec)));

/*check that the stuff received is the same as the one sent*/
i = memcmp(golden, data_out, length*sizeof(int));
printf("Userspace program terminating: %d\n",i);
return 0;