Accelerated Database Processor

Streaming Insertion Sorter Submodule

CSEE 4840 Embedded System Design
Spring/Fall 2014
Supplemental Project Report

Timothy Paine (tkp2108)

Table of Contents

Table of Contents
Abstract
I. Project Overview
    Motivation
    Scope
II. Design
    Hardware
    Tiles
        Partitioner
        Sorter
        FIFO
    Tile Design
    Altera FIFO IP
    Software
        Drivers
            FIFO
            Tile Drivers
III. Details of Design Specification
    Hardware
        Signal Specifications
        Latency 0 vs Latency 1 vs Modified Latency 1
            Latency 0
            Latency 1
            Modified Latency 1
            Control
        Hardware Support for Modified Latency 1
    Sorter Tile
IV. Validation and Testing
    Verilator
    Modelsim/GTKWave
    System Console
    Signal Tap
    FPGA Synthesis
V. Results
    Implementation
        System Console Validation - 10 records sorted in a row
        Device Driver Program
    Design Space Exploration
VI. Future Research
Appendix. Source Code, Tests, Drivers, etc.
        Hardware

    - Sorter
    - Stage
- Software
    - Fifo Driver
- Synthesis
    - Design Compiler
    - IC Compiler
    - Sample Program

Abstract

The Database Processing Unit (DPU) is a database accelerator composed of a heterogenous set of ASIC tiles modeled on SQL functions. For this project, I examine and implement one particular subfunction, *Sort*. I propose an accelerated partition-sorter, leveraging previous work done on hardware accelerated range partitioning, and implement as proof-of-concept an insertion sorter capable of 1-record per cycle throughput.

I. Project Overview

Motivation

Software sorting benefits greatly from scaling up: quicksort on a billion records is not significantly worse than quicksort on a hundred records. However, the opposite is also true: software sorting does not scale down well, meaning a quicksort on a hundred records is not very fast. Something like radix sort also works well, but suffers when the bitwidth increases. So a hardware sorter with size on order of the L2 cache has the potential to seriously outperform software. The conventional hardware approach is a bitonic sort, but as I will show in later sections, even if we perform bitonic sort on fixed size chunks, this sort cannot be truly streaming. I propose a pipelined insertion sort. Because insertion sort is well suited to streaming operations, and because of the pipelined nature of the design, it is energy efficient, highly scalable, and highly performant, with a throughput of one record per cycle, meaning the sort operation is essentially transparent.

Scope

I limit the design to a proof-of-concept level only. I leave optimal pipeline depth, best-area/best-speed/best-energy analysis, and other concerns to a research paper, with a minimal design space exploration given here.

II. Design

Hardware

There are three hardware components to this device. Input/Output fifos use AvalonMM to pull data to/from memory. These are connected via a range-partitioner, and an insertion-sorter, with the insertion sorter implemented here (the range sorter has already been implemented by others).

Tiles

Partitioner

The partitioner breaks up input data into multiple streams based on a range of values. Each stream is connected to a sorter. Thus, at the end of data processing, each output from the partitioner contains a sorted block a data, and the blocks themselves can be concatenated to build the fully sorted set of data.

Sorter

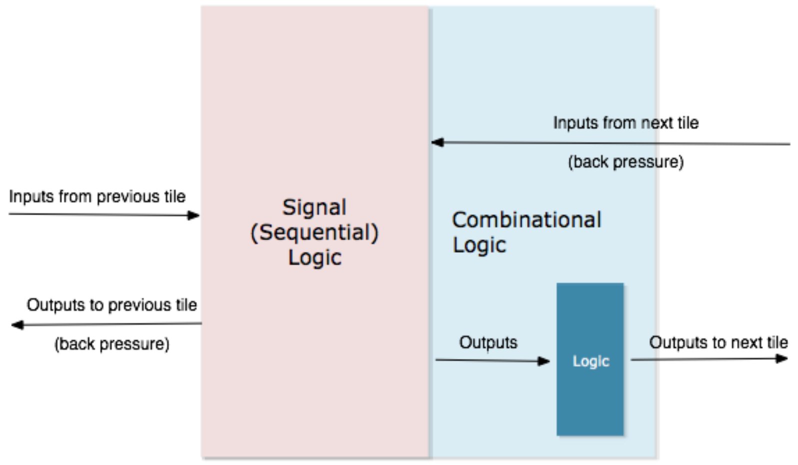
The sorter is a linear pipeline insertion sorter. It is composed of a number stages, each of which has a simple function. Given two inputs, it holds the larger and emits the smaller. Chained together, small values flow forward through the pipeline and larger values are held back. Because each stage knows its position in the pipeline, we are able to configure it in such a way that, once full, pipeline throughput is one piece of data per cycle, meaning the sorting operation is fully-streaming.

FIFO

This is a simple queue provided by Altera IP.

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, as per standard logic design principles.



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.

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.

```
#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 an 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:

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

Each fifo will have a struct associated:

```
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;
```

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.

```
            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:

```
    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 writing the last int*/

                    if (op->done && to_write == op->length){        /* that was ALSO the last transfer
for this stream ad I wrote it all down*/
                        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.

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.

Most details of the sorter/partitioner are statically configured, and thus control overhead is minimized.

III. Details of Design Specification

Hardware

Signal Specifications

Here are several diagrams which illustrate data flowing through the sorter pipeline.

Key Metrics

Pipeline of length N stages is capable of sorting N+1 items (to sort N items, requires 2N+2 registers).

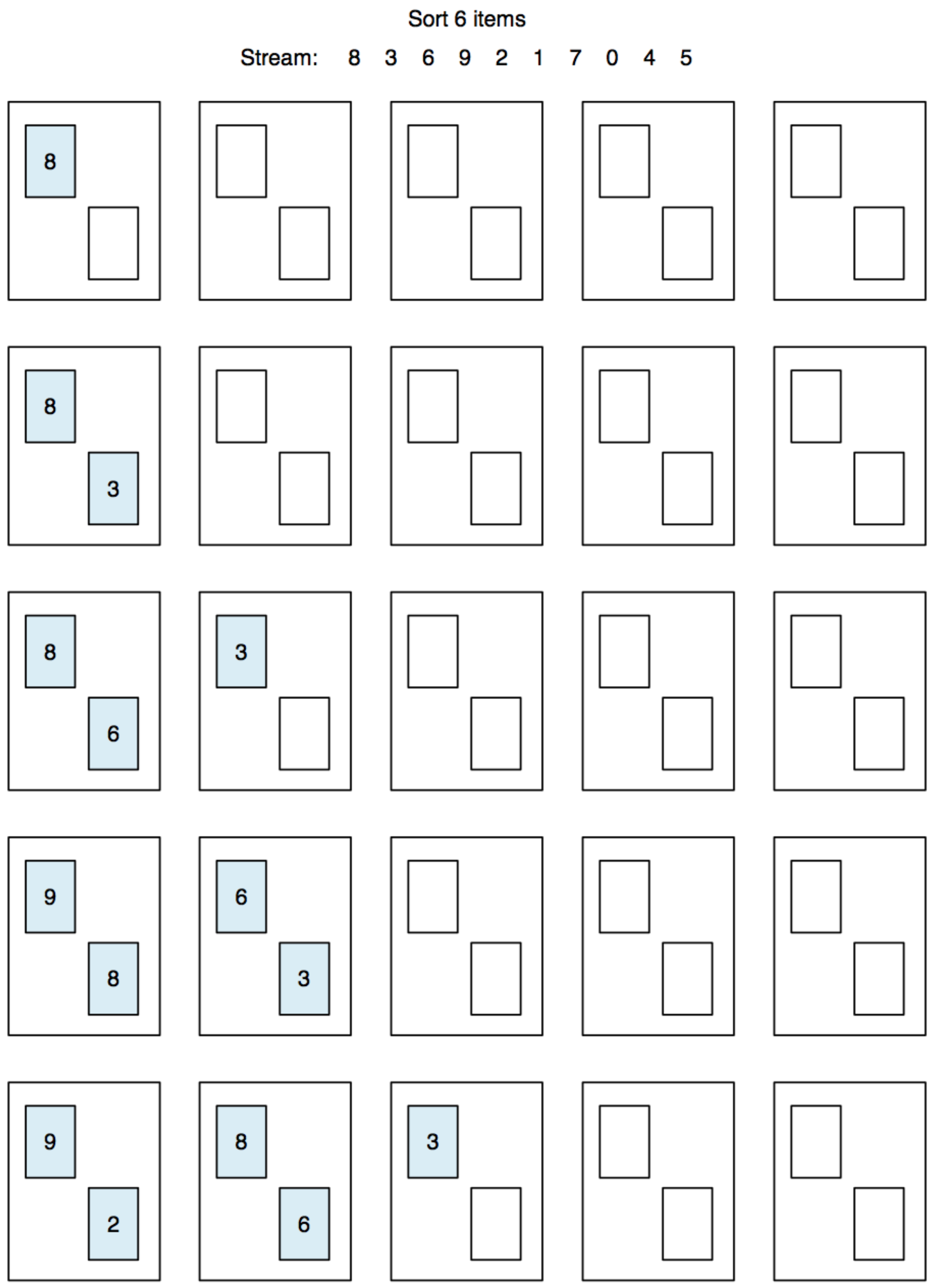
Inflow takes N cycles

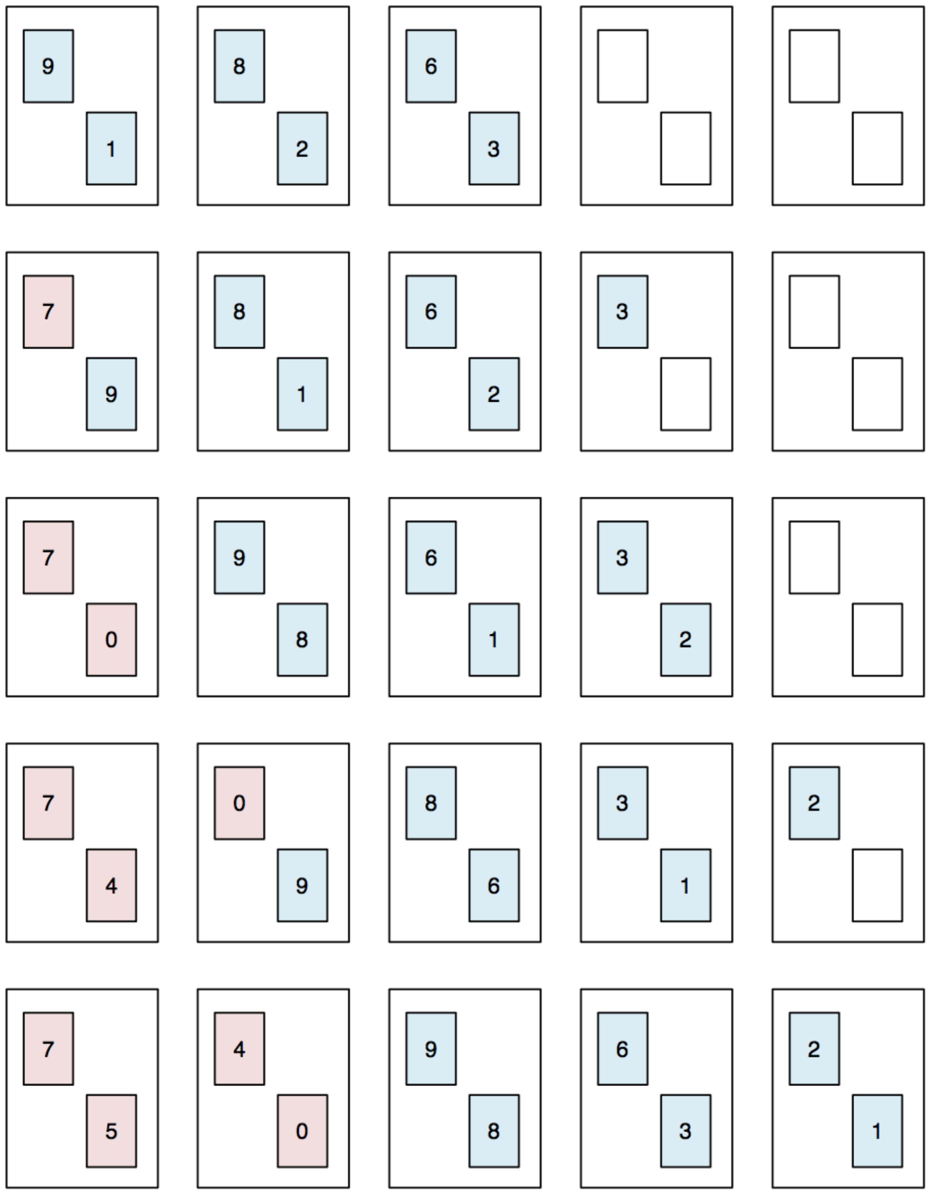
Outflow takes N cycles

Sorting takes N-1 cycles

CPI at full pipeline: 1/cycle

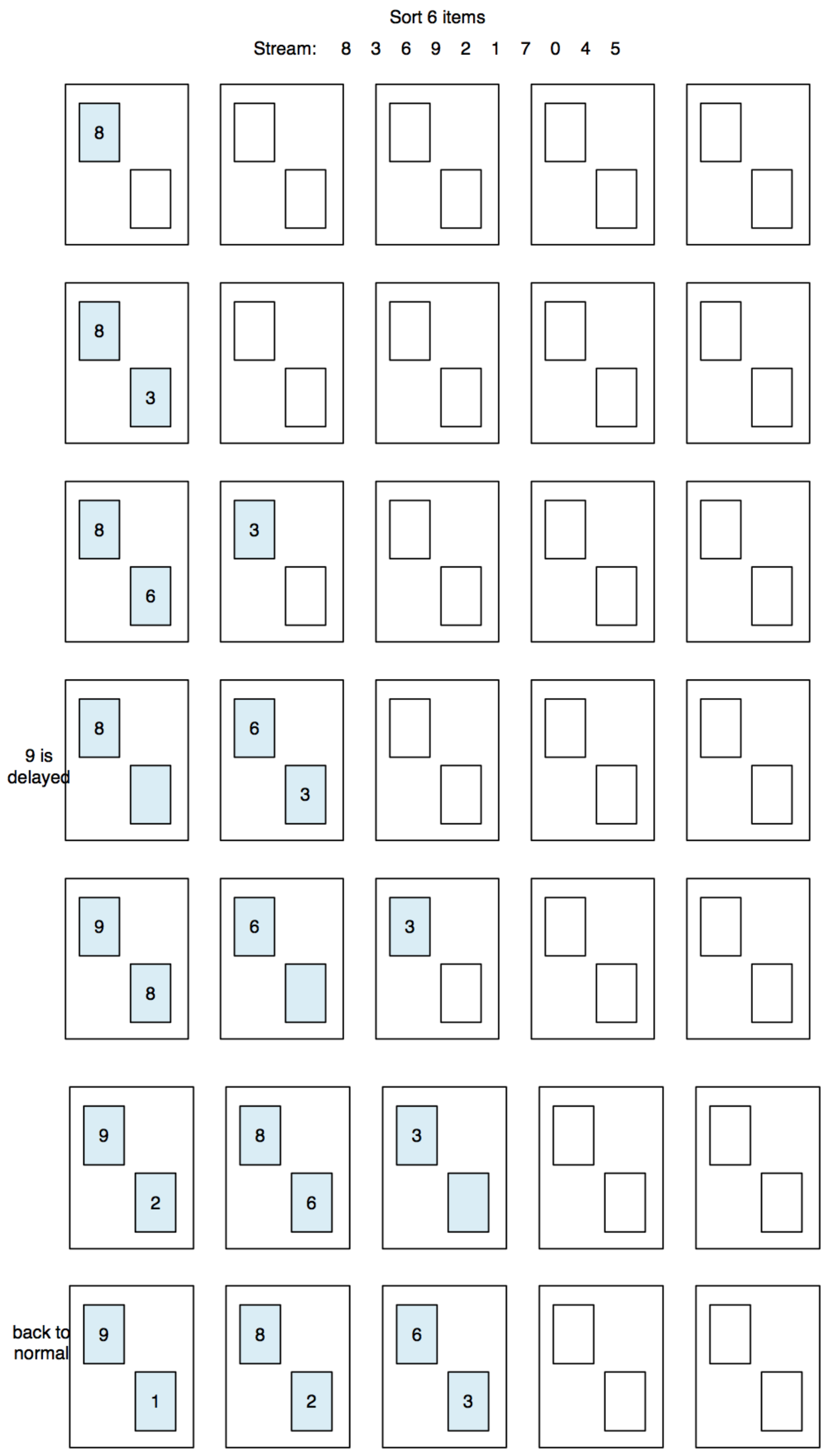
Backpressure: via inflow (output FIFO has stall signal to input FIFO, inputs can be delayed arbitrarily long)





Data is ready for outflow >>>

Additionally, this design can accommodate input bubbles (which allows it to be latency insensitive with respect to writing from the output FIFO)



Input bubble occurs

Pipeline has eliminated bubble

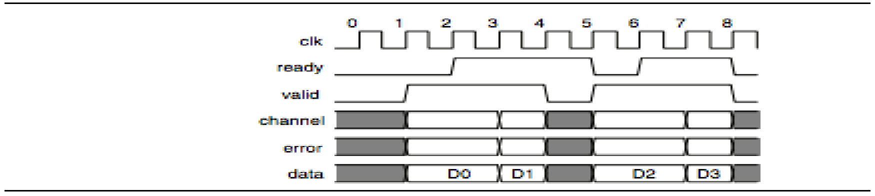
Latency 0 vs Latency 1 vs Modified Latency 1

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)

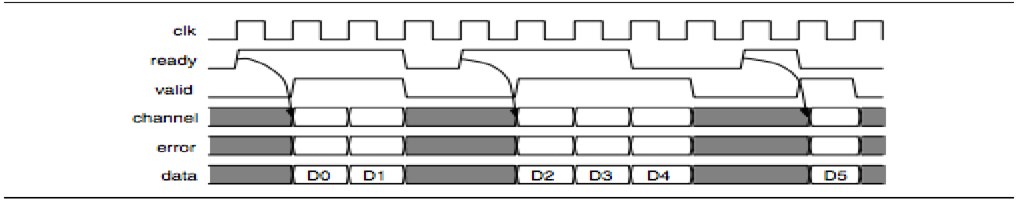
Figure 5–7. Transfer with Backpressure, readyLatency=0



Latency 1

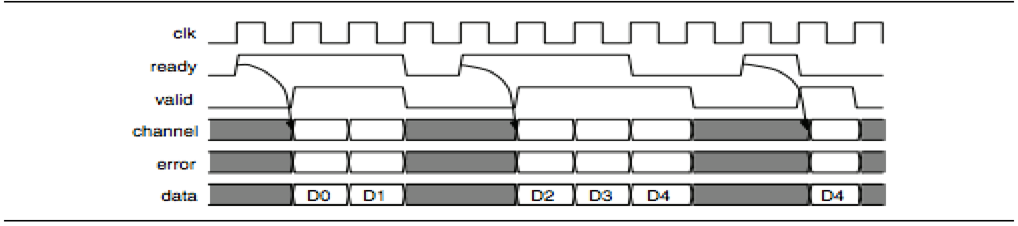
(from Altera)

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



Modified Latency 1

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



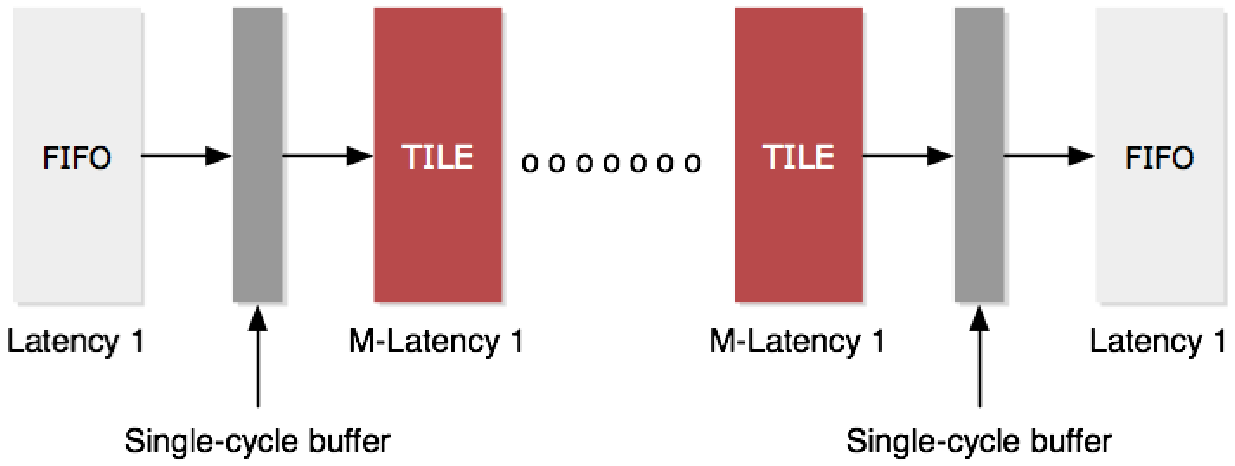
The difference between latency-1 and our modified latency-1 is the necessity of a second buffer. For latency-1, if ready is deasserted, valid data must still be processed that cycle, which is why data item D4 is processed despite ready being low. We require that ready be high when valid is raised, but 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.

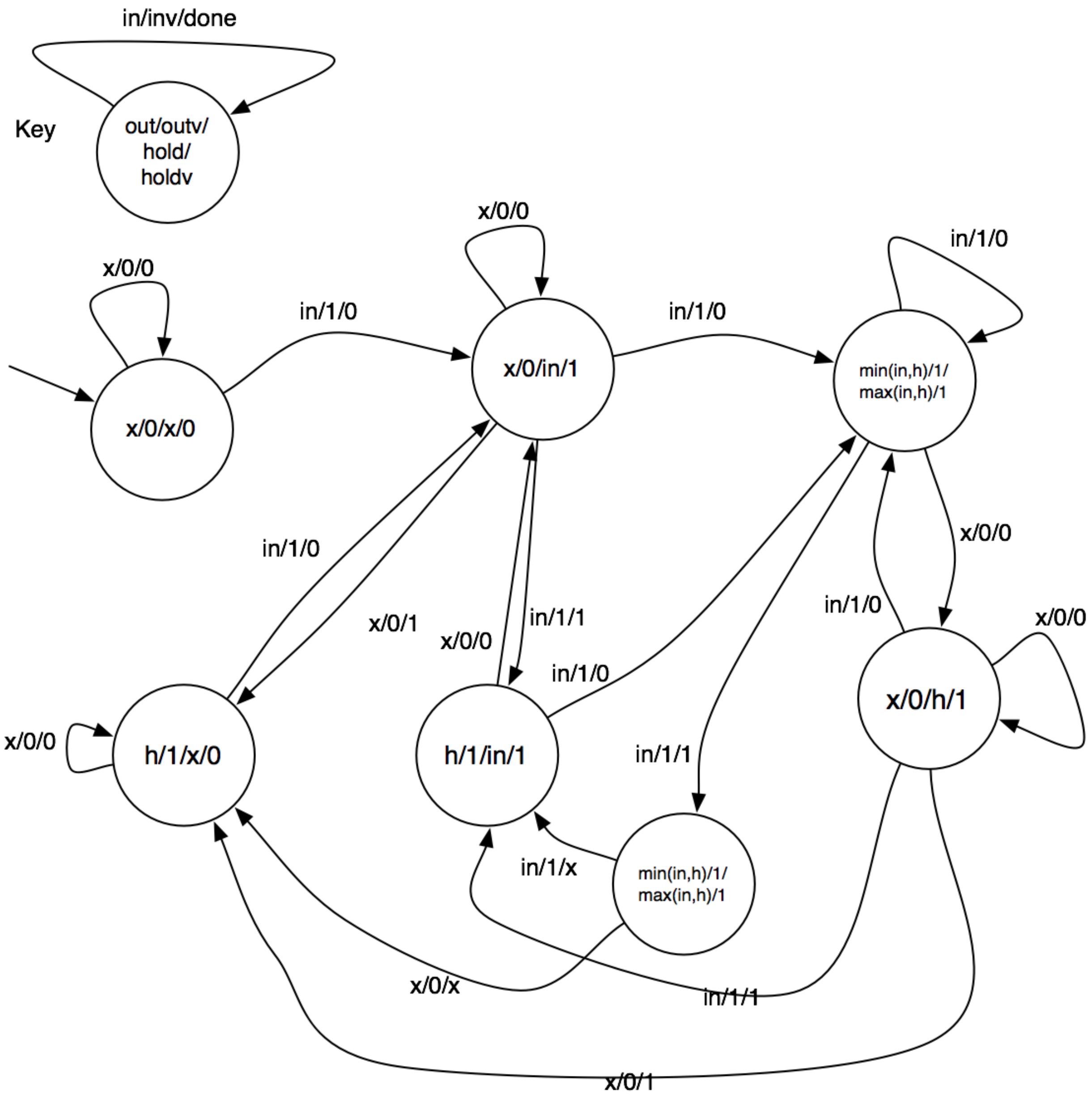
Hardware Support for Modified Latency 1

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. Because the insertion sorter is bounded on either end by standard tiles or FIFOs, only sorts a fixed block at a time, and can tolerate arbitrary input delays, it need not support backpressure, as tiles downstream of the pipeline can stall tiles upstream of the pipeline directly. This eliminates the need for a costly third register in each sorter stage.



Sorter Tile

The sorter tile adheres (with some modification) to the following state machine, which encapsulates the behavior given in the signal diagram above. Essentially, the Sorter tile consists of a sequence of stages, each of which contains two registers. Incoming records are compared to “hold” records, and the smaller of the two is output. Once the appropriate number of records has been seen, the records flush through the pipeline and the next set of records comes in. Thus each tile consists of a comparator and two registers, with some small amount of logic to handle counting incoming records, and processing and propagating done signals (in instances where the number of records to sort is less than the max able to sort).



in/inv/done
Key
out/outv/ hold/ holdv
x/0/0
x/0/0
in/1/0
x/0/x/0
x/0/in/1
in/1/0
in/1/0
min(in,h)/1/ max(in,h)/1
in/1/0
x/0/1
x/0/0
in/1/1
in/1/0
x/0/0
in/1/0
x/0/0
x/0/0
h/1/x/0
h/1/in/1
in/1/1
x/0/h/1
in/1/x
min(in,h)/1/ max(in,h)/1
x/0/x
in/1/1
x/0/1

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/GTKWave

We used Modelsim and GTKWave to debug our signal spec and test corner cases. 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:

V. Results

Implementation

The sorter is implemented on the Altera Cyclone V FPGA with ARM HPS. The design I picked can sort 10 records at a time of bitwidth 32 bits. This design was picked because it is relatively compact and easy to debug. Random sets of 10 records are sent to an input fifo, and sorted sets of 10 are streamed out. First I validated the design with system console. For this project, I only worry about sorting the keys; payload is exported and not used for anything.

System Console Validation - 10 records sorted in a row

```
% master_write_32 $m 0x40 4
% master_write_32 $m 0x40 3
% master_write_32 $m 0x40 5
% master_write_32 $m 0x40 6
% master_write_32 $m 0x40 2
% master_write_32 $m 0x40 3
% master_write_32 $m 0x40 6
% master_write_32 $m 0x40 6
% master_write_32 $m 0x40 8
% master_write_32 $m 0x40 9
% master_write_32 $m 0x40 3
% master_write_32 $m 0x40 1
%
% master_read_32 $m 0x48 1
0x00000002
% master_read_32 $m 0x48 1
0x00000003
% master_read_32 $m 0x48 1
0x00000003
% master_read_32 $m 0x48 1
0x00000004
% master_read_32 $m 0x48 1
0x00000005
% master_read_32 $m 0x48 1
0x00000006
% master_read_32 $m 0x48 1
0x00000006
% master_read_32 $m 0x48 1
0x00000006
% master_read_32 $m 0x48 1
0x00000008
% master_read_32 $m 0x48 1
0x00000009
```

```
//Here, 3 and 1 are in the pipeline, so I need to write a few more to get output (total must be
10)
% master_write_32 $m 0x40 9
% master_write_32 $m 0x40 5
% master_write_32 $m 0x40 7
% master_write_32 $m 0x40 4
% master_write_32 $m 0x40 2
% master_write_32 $m 0x40 6
% master_write_32 $m 0x40 8
% master_write_32 $m 0x40 0
% master_write_32 $m 0x40 9
0x00000000
% master_read_32 $m 0x48 1
0x00000001
% master_read_32 $m 0x48 1
0x00000002
% master_read_32 $m 0x48 1
0x00000003
% master_read_32 $m 0x48 1
0x00000004
% master_read_32 $m 0x48 1
0x00000005
% master_read_32 $m 0x48 1
0x00000006
% master_read_32 $m 0x48 1
0x00000007
% master_read_32 $m 0x48 1
0x00000008
% master_read_32 $m 0x48 1
0x00000009
```

Device Driver Program

```
FIFO Userspace program started
Thread structs allocated




Out 1:  43   86   21   1    49   82   87   36   49   47   42   19
        88   77   93   37   40   3    83   80   46   78   63   28
        85   8    35   61   90   36   4    16   36   16   9    45
        42   79   98   37   13   17   89   90   84   63   15   50
        99   32

Data: 1
Data: 21
```

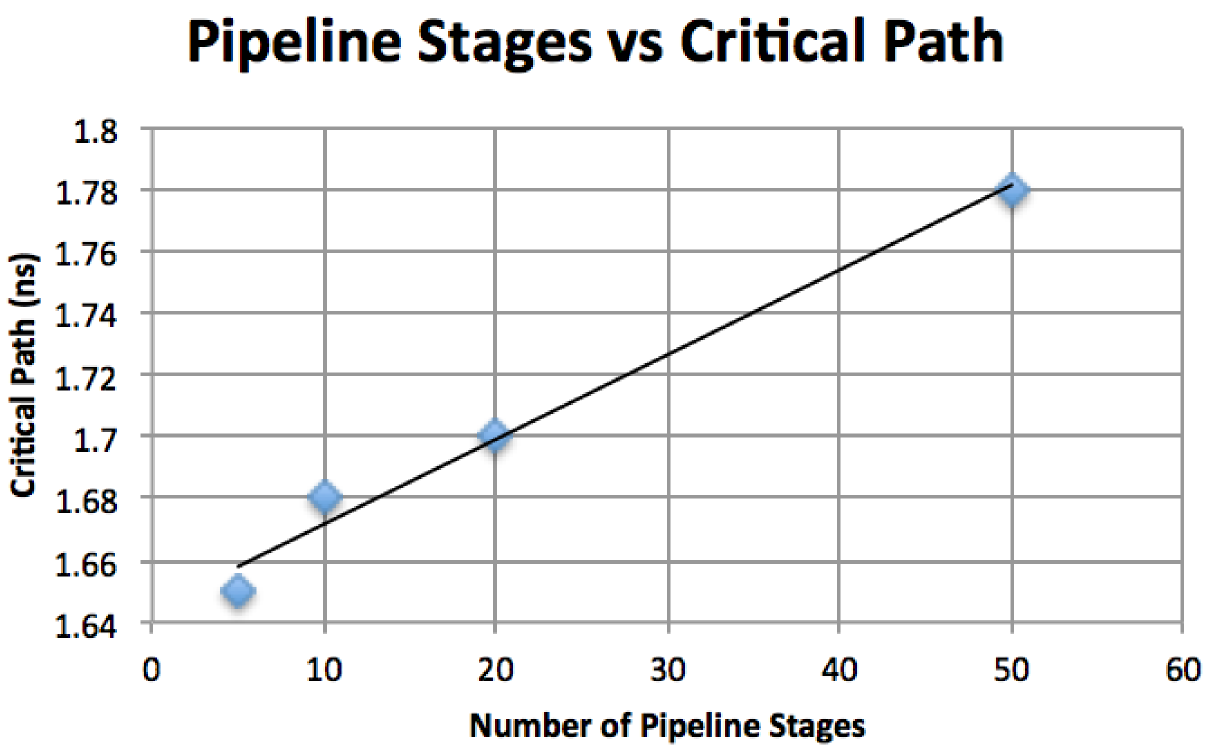
```
Data: 36
Data: 43
Data: 47
Data: 49
Data: 49
Data: 82
Data: 86
Data: 87
Data: 3
Data: 19
Data: 37
Data: 40
Data: 42
Data: 77
Data: 80
Data: 83
Data: 88
Data: 93
Data: 8
Data: 28
Data: 35
Data: 36
Data: 46
Data: 61
Data: 63
Data: 78
Data: 85
Data: 90
Data: 4
Data: 9
Data: 16
Data: 16
Data: 36
Data: 37
Data: 42
Data: 45
Data: 79
Data: 98
Data: 13
Data: 15
Data: 17
Data: 32
Data: 50
Data: 63
Data: 84
Data: 89
Data: 90
```

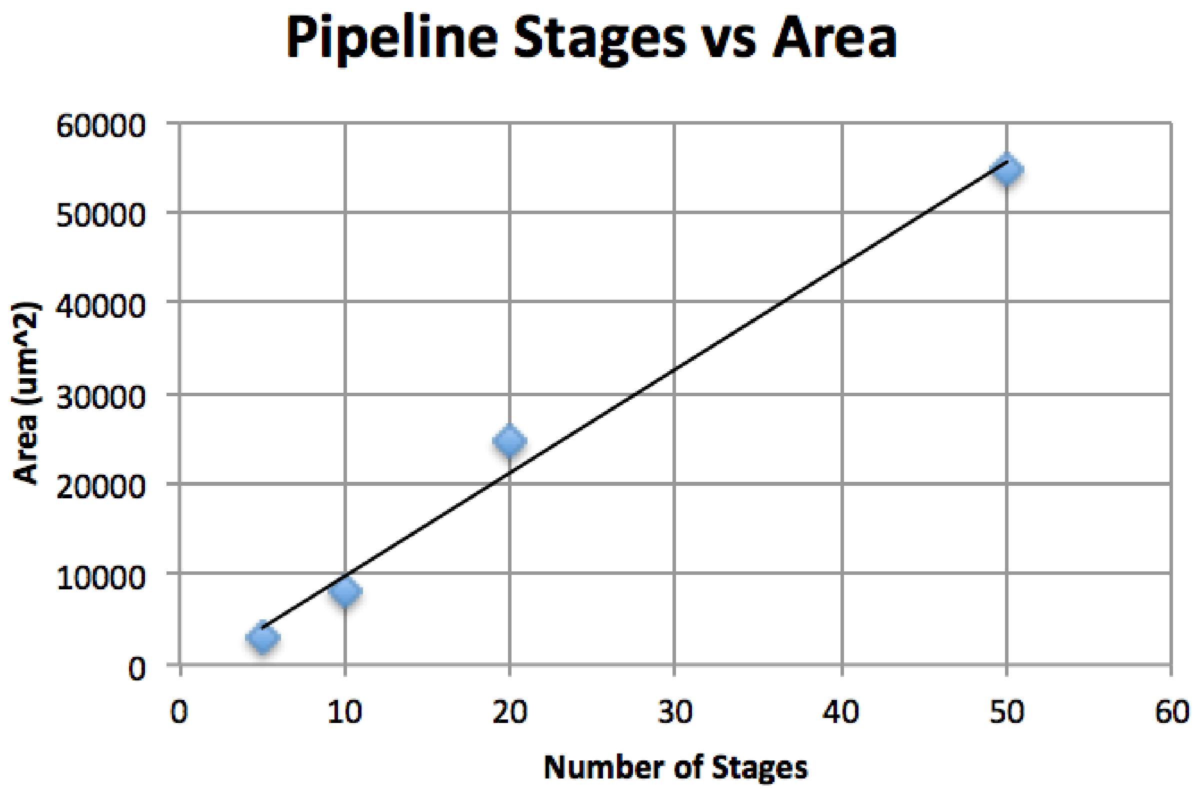
```
Data: 99
TIME TAKEN 1684
FIFO Userspace program terminating: 0
```

Design Space Exploration

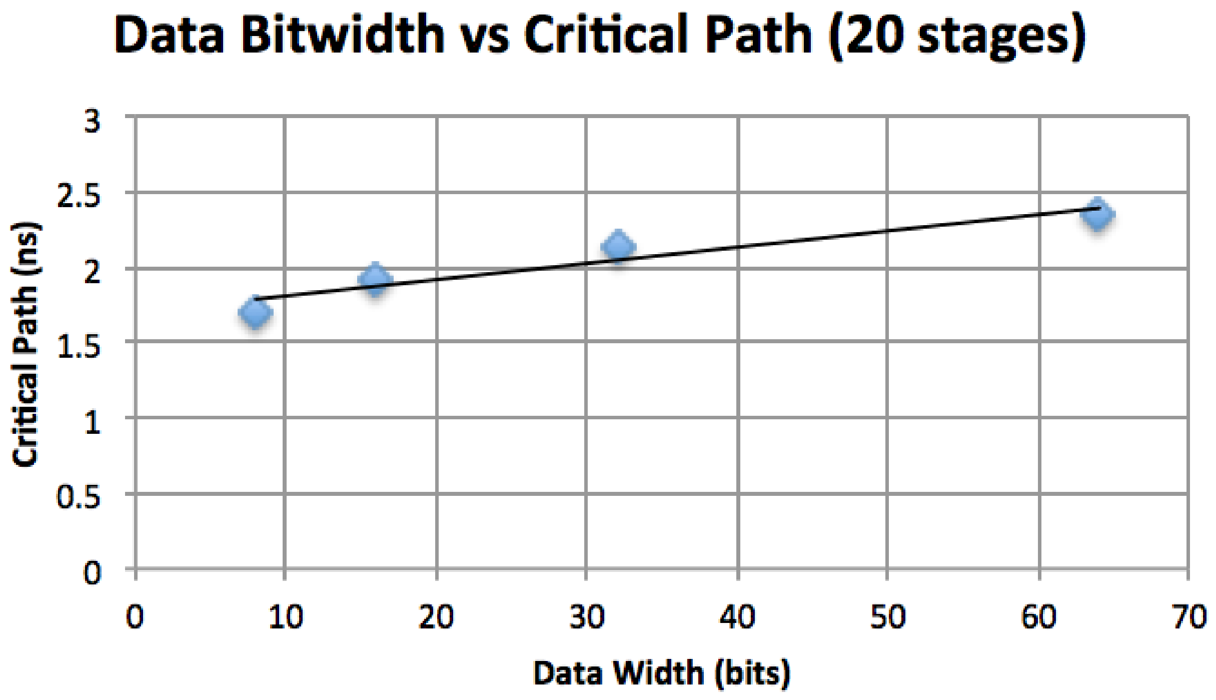
Since we can achieve 1 record per cycle sorting throughput, the ideal design is one that minimizes clock frequency, but is still large enough to do a meaningful amount of work. As such, we want to have some sense of the Area/Power/Timing tradeoff between different design points. To this end, the Synopsys Design and IC compiler tools were used to get implementation numbers at a 32nm process.

Number stages	Sortable depth	Comparator Bitwidth	Critical Path (ns)	Area (um^2)
4	5	8	1.65	3099.9
9	10	8	1.68	8298.3
19	20	8	1.7	18987
49	50	8	1.78	54567.5

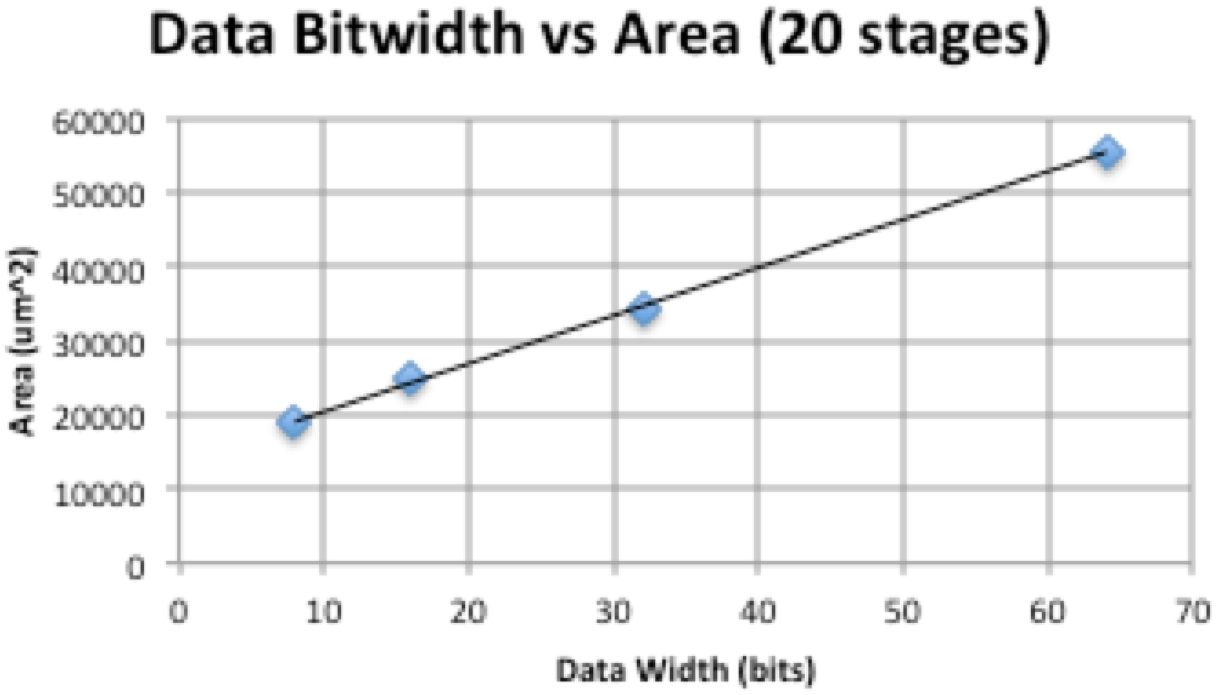

Pipeline Stages vs Critical Path
Critical Path (ns)
1.8
1.78
1.76
1.74
1.72
1.7
1.68
1.66
1.64
0
10
20
30
40
50
60
Number of Pipeline Stages


Pipeline Stages vs Area
Area (um^2)
60000
50000
40000
30000
20000
10000
0
0
10
20
30
40
50
60
Number of Stages

Number stages	Sortable depth	Comparator Bitwidth	Critical Path (ns)	Area (um^2)
19	20	8	1.7	18987
19	20	16	1.92	24692.5
19	20	32	2.14	34334
19	20	64	2.35	55473.1



Data Bitwidth vs Critical Path (20 stages)
Critical Path (ns)
3
2.5
2
1.5
1
0.5
0
0
10
20
30
40
50
60
70
Data Width (bits)



Data Bitwidth vs Area (20 stages)
Area (um^2)
60000
50000
40000
30000
20000
10000
0
0
10
20
30
40
50
60
70
Data Width (bits)

VI. Future Research

For 8-bit comparisons, we are able to sort blocks of 50 elements at or above 500MHz. For 32-bit comparisons, we are able to sort blocks of 20 elements at or above 450MHz. This is promising for the cache resident sorting workloads we are targeting, and there is future work to be done in determining the best configuration (pipeline depth, bitwidth, number of pipelines, etc). This work is outside the scope of this report.

Appendix. Source Code, Tests, Drivers, etc.

File list:

Hardware

Sorter

```
`timescale 1ns/1ns

module sorter #(
		parameter NO_SORT=10, //need 9 stages to sort 10 items
		parameter KEY_WIDTH = 32,
		parameter PAY_WIDTH = 8
)
(
		input logic clk,
		input logic rst, en,
		input logic[KEY_WIDTH-1:0] key_i,
		input logic [PAY_WIDTH-1:0] pay_i,
		input logic vld_i, dne_i,
		output logic[KEY_WIDTH-1:0] key_o,
		output logic [PAY_WIDTH-1:0] pay_o,
		output logic vld_o, dne_o
);


logic [KEY_WIDTH-1:0] keys [NO_SORT-1-1-1:0];
logic [PAY_WIDTH-1:0] pays [NO_SORT-1-1-1:0];
logic vlds [NO_SORT-1-1-1:0];
logic dnes [NO_SORT-1-1-1:0];

/* manuallu connect first */

stage  #(.NO_SORT(NO_SORT), .KEY_WIDTH(KEY_WIDTH), .PAY_WIDTH(PAY_WIDTH))
	first_stage_inst
	 (.key_i(key_i), .pay_i(pay_i),
		.vld_i(vld_i), .dne_i(dne_i),
		.key_o(keys[0]), .pay_o(pays[0]),
		.vld_o(vlds[0]), .dne_o(dnes[0]),
		.en(en),.clk(clk),.rst(rst)
		);

genvar iter;
generate
for(iter = 1; iter<NO_SORT-1-1;iter++) begin : GENER
stage #(.NO_SORT(NO_SORT), .KEY_WIDTH(KEY_WIDTH), .PAY_WIDTH(PAY_WIDTH))
	stage_inst
	 (.key_i(keys[iter-1]), .pay_i(pays[iter-1]),
		.vld_i(vlds[iter-1]), .dne_i(dnes[iter-1]),
		.key_o(keys[iter]), .pay_o(pays[iter]),
		.vld_o(vlds[iter]), .dne_o(dnes[iter]),
		.en(en),.clk(clk),.rst(rst)
		);
end
endgenerate

/* manually connect last */
stage #(.NO_SORT(NO_SORT), .KEY_WIDTH(KEY_WIDTH), .PAY_WIDTH(PAY_WIDTH))
	last_stage_inst
	 (.key_i(keys[NO_SORT-1-1-1]), .pay_i(pays[NO_SORT-1-1-1]),
        .vld_i(vlds[NO_SORT-1-1-1]), .dne_i(dnes[NO_SORT-1-1-1]),
        .key_o(key_o), .pay_o(pay_o),
        .vld_o(vld_o), .dne_o(dne_o),
        .en(en),.clk(clk),.rst(rst)
        );


endmodule
```

Stage

```
`timescale 1ns/1ns

module stage #(
  parameter NO_SORT=10,
  parameter KEY_WIDTH = 32,
  parameter PAY_WIDTH = 8
)
(
  input logic clk,
  input logic rst,
  input logic en,
  input logic [KEY_WIDTH-1:0] key_i,
  input logic [PAY_WIDTH-1:0] pay_i,
  input logic vld_i,

  input logic dne_i, //into front


  output logic [KEY_WIDTH-1:0] key_o,
  output logic [PAY_WIDTH-1:0] pay_o,
  output logic vld_o,
  output logic dne_o

);

  integer count;
  logic [KEY_WIDTH-1:0] key;
  logic [PAY_WIDTH-1:0] pay;
  logic holding;
  logic done;

  /* pass the done reg to out */
  always @(posedge clk) begin
    if(rst) begin
      dne_o <= 1'b0;
    end else if(en) begin
      dne_o <= done;
    end
  end

  /* handle count and done logic */
  always @(posedge clk) begin
    if(rst) begin
      count = 0;
      done = 0;
    end else if(en) begin
      if(count == NO_SORT || dne_i==1'b1) begin //last record in series
        done = 1'b1;
        if(vld_i) begin
         count = 1'b1;
```

```
                        end else begin
            count = 1'b0;
                        end
        end else if(vld_i) begin
            count = count + 1'b1; //increment count
                        done = 1'b0;
                        //if(count + 1'b1 == NO_SORT) begin
                        //            done <= 1'b1;
            //end else begin
                        //            done <= 1'b0; //make sure done is low
                        //end
        end
    end else begin
        /* do nothing */
    end
end


always_ff @(posedge clk) begin
    if(rst) begin
        key <= 'b0;
        pay <= 'b0;
        holding <= 1'b0;
        key_o <= 'b0;
        pay_o <= 'b0;
        vld_o <= 1'b0;
    end if(en) begin
        if(done) begin /* pass hold to out */
            if(holding) begin
                key_o <= key;
                pay_o <= pay;
                vld_o <= 1'b1;
            end else begin
                /* last record was emitted last cycle */
                vld_o <=1'b0;
            end
            if(vld_i) begin /* start next block */
                key<=key_i;
                pay<=pay_i;
                holding<=1'b1;
            end else begin /* bubble */
                holding<=1'b0;
            end
        end else if(vld_i) begin /* still processing */
            if(holding) begin
                vld_o <= 1'b1;
                if(key<key_i) begin //emit hold, store in */
                    key_o<=key;
                    pay_o<=pay;
                    key<=key_i;
                    pay<=pay_i;
                    holding<=1'b1;
                end else begin /* emit in, store hold */
                    key_o<=key_i;
                    pay_o<=pay_o;
                    /* holding stays */
                    holding<=1'b1;
                end
            end else begin /* not holding, so store (bubble) */
                key<=key_i;
                pay<=pay_i;
                holding<=1'b1;
                vld_o <= 1'b0;
            end
        end else begin
```

```
                /* do nothing */
                        vld_o<=1'b0;
            end
        end else begin
            /* do nothing */
        end
    end

endmodule
```

Software

Fifo Driver

```
#ifndef _FIFO_DATA_H
#define _FIFO_DATA_H

#include <linux/ioctl.h>

#define FIFO_MAGIC 'q'

#define SINGLE_PACKET_CHANNEL0         0b00000000000000000000000000000011
#define START_PACKET_CHANNEL0          0b00000000000000000000000000000001
#define END_PACKET_CHANNEL0            0b00000000000000000000000000000010
#define DONE_END_PACKET_CHANNEL0       0b00000000000000010000000000000010
#define DONE_SINGLE_PACKET_CHANNEL0    0b00000000000000010000000000000011
#define DONE_MASK                      0b00000000000000010000000000000000

#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*)
#endif




/*
 * Device driver for the Altera FIFO
 *
 * A Platform device implemented using the misc subsystem
 *
 * 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 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*/

										if (op->done && to_write == op->length){				/* that was
ALSO the last transfer for this stream ad I wrote it all down*/
												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 coping 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;
                        }
                }else{
                        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\n");
	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\n");
}
```

```
module_init(fifo_init);
module_exit(fifo_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Tim Paine, Columbia University");
MODULE_DESCRIPTION("Altera FIFO driver");
```

Synthesis

Design Compiler

```
## Adapted from: http://www.csl.cornell.edu/courses/ece5950/handouts/ece5950-tut2-dc.pdf
## Invoke with: dc_shell-xg-t -64bit

## NOTE: Topographical mode seems to disable wire load estimates (resulting in undefined
## interconnect areas).  Not using Topographical mode causes several commands below to fail.
## -topographical_mode

# Setup the environment: point to your Verilog source directory, create a
# work directory for the tool, and point to the Synopsys process libraries.

set LIB_PATH  "/proj/arcade/synopsys/SAED32_EDK/lib/stdcell_hvt/db_nldm"
set CELL      "saed32hvt_tt1p05v25c"
set TECH      "/proj/arcade/synopsys/SAED32_EDK/tech/milkyway/saed32nm_1p9m_mw.tf"
set MW_REF_LIB "/proj/arcade/synopsys/SAED32_EDK/lib/stdcell_hvt/milkyway/saed32nm_hvt_1p9m"
set MAX_TLUPLUS "/proj/arcade/synopsys/SAED32_EDK/tech/star_rcxt/saed32nm_1p9m_Cmax.tluplus"
set MIN_TLUPLUS "/proj/arcade/synopsys/SAED32_EDK/tech/star_rcxt/saed32nm_1p9m_Cmin.tluplus"
set TECH2ITF_MAP "/proj/arcade/synopsys/SAED32_EDK/tech/milkyway/saed32nm_tf_itf_tluplus.map"

set DESIGN_NAME sorter
set RTL_PATH    "./rtl"
set RTL_SRC     "${RTL_PATH}/${DESIGN_NAME}.sv ./rtl/stage.sv"
set MW_LIB_NAME "${DESIGN_NAME}_LIB"
set CLK_NAME      "clk"
set CLK_NS "2"

###############################################################################
## BE CAREFUL WITH FLOW MODIFICATIONS BEYOND THIS POINT
###############################################################################

set search_path ". ${RTL_PATH} ${LIB_PATH}"
set target_library "${CELL}.db"
set synthetic_library "dw_foundation.sldb"
set link_library "* $target_library $synthetic_library"
set alib_library_analysis_path "./alib"
set mw_logic1_net "VDD"
set mw_logic0_net "VSS"

# Create Milkyway library
create_mw_lib -technology ${TECH} -mw_reference_library ${MW_REF_LIB} "${MW_LIB_NAME}"
open_mw_lib "${MW_LIB_NAME}"
check_library

# NOTE: Fails (harmlessly) outside of topo mode
set_tlu_plus_files -max_tluplus ${MAX_TLUPLUS} -min_tluplus ${MIN_TLUPLUS} -tech2itf_map ${TECH2ITF_MAP}
check_tlu_plus_files

define_design_lib WORK -path "./work"

# Our environment should be setup, now load your Verilog design into DC with
# the analyze, elaborate, link, and check design commands. Executing these
# commands will result in a great deal of log output as the tool elaborates
# Verilog constructs and starts to infer some high-level components.
analyze -format sverilog "${RTL_SRC}"

# During elaboration DC will report all state inferences. This is a good way
# to verify that latches and flip-flops are not being accidentally inferred.
elaborate "${DESIGN_NAME}"
link

# The check_design command checks that the design is consistent. You will not
# be able to synthesize your design until you eliminate all ERRORS. Many
# WARNINGS are not an issue, but it is still useful to skim through this output.
check_design

# Set the constraints. Before we can synthesize our design, we must specify
# some constraints like the target clock period. The following command tells
# the tool that the pin named clk is the clock and that the desired clock
# period is 1 nanosecond.
create_clock ${CLK_NAME} -name ideal_clock -period ${CLK_NS}

# The compile_ultra command begins the actuall synthesis process that
# transforms your design into a gate-level netlist. The -no_autoungroup
# flag is specied in order to preserve the hierarchy during synthesis.
compile_ultra -gate_clock -no_autoungroup
# The compile_ultra command may take a while!

change_names -rules verilog -hierarchy

# Write output files
write -format ddc -hierarchy -output ${DESIGN_NAME}.mapped.ddc
write -f verilog -hierarchy -output ${DESIGN_NAME}.mapped.v
write_sdf ${DESIGN_NAME}.mapped.sdf
write_sdc -nosplit ${DESIGN_NAME}.mapped.sdc
write_milkyway -overwrite -output "${DESIGN_NAME}_DCT"

# Report results
report_timing -path full -delay min -max_paths 10 -nworst 2 > ${DESIGN_NAME}.dc.holdtiming
report_timing -path full -delay max -max_paths 10 -nworst 2 > ${DESIGN_NAME}.dc.setuptiming
report_area -hierarchy > ${DESIGN_NAME}.dc.area
report_power -hier -hier_level 2 > ${DESIGN_NAME}.dc.power
report_resources > ${DESIGN_NAME}.dc.resources
report_constraint -verbose > ${DESIGN_NAME}.dc.constraint

quit
```

IC Compiler

```
## Adapted from: http://www.csl.cornell.edu/courses/ece5950/handouts/ece5950-tut3-icc.pdf
## Invoke with: icc_shell -64bit

# Setup the environment: point to your Verilog source directory, create a
# work directory for the tool, and point to the Synopsys process libraries.

set LIB_PATH  "/proj/arcade/synopsys/SAED32_EDK/lib/stdcell_hvt/db_nldm"
set CELL      "saed32hvt_tt1p05v25c"
set TECH      "/proj/arcade/synopsys/SAED32_EDK/tech/milkyway/saed32nm_1p9m_mw.tf"
set MW_REF_LIB "/proj/arcade/synopsys/SAED32_EDK/lib/stdcell_hvt/milkyway/saed32nm_hvt_1p9m"
set MAX_TLUPLUS "/proj/arcade/synopsys/SAED32_EDK/tech/star_rcxt/saed32nm_1p9m_Cmax.tluplus"
set MIN_TLUPLUS "/proj/arcade/synopsys/SAED32_EDK/tech/star_rcxt/saed32nm_1p9m_Cmin.tluplus"
set TECH2ITF_MAP "/proj/arcade/synopsys/SAED32_EDK/tech/milkyway/saed32nm_tf_itf_tluplus.map"
set DESIGN_NAME sorter
set MW_LIB_NAME "${DESIGN_NAME}_LIB"

################################################################################
## BE CAREFUL WITH FLOW MODIFICATIONS BEYOND THIS POINT
################################################################################

# Execute some commands to setup your environment.
set search_path "${LIB_PATH}"
set target_library "${CELL}.db"
set link_library "* ${CELL}.db dw_foundation.sldb"

# Open Milkyway database.
open_mw_lib ${MW_LIB_NAME}

import_designs "./${DESIGN_NAME}.mapped.ddc" -format "ddc" -top "${DESIGN_NAME}" -cel "${DESIGN_NAME}"

set_tlu_plus_files -max_tluplus ${MAX_TLUPLUS} -min_tluplus ${MIN_TLUPLUS} -tech2itf_map ${TECH2ITF_MAP}

derive_pg_connection \
    -power_net "VDD" \
    -power_pin "VDD" \
    -ground_net "VSS" \
    -ground_pin "VSS" \
    -create_ports "top"

# Make an initial floorplan and synthesize power rails. At this point, you
# can see the estimated voltage drops on the power rails.
# The numbers in the right column of the GUI are specied in mW.
initialize_floorplan \
    -control_type "aspect_ratio" -core_aspect_ratio "1" \
    -core_utilization "0.7" -row_core_ratio "1" \
    -left_io2core "30" -bottom_io2core "30" -right_io2core "30" -top_io2core "30" \
    -start_first_row
create_fp_placement
synthesize_fp_rail \
    -power_budget "1000" -voltage_supply "1.2" -target_voltage_drop "250" \
    -output_dir "./pna_output" -nets "VDD VSS" -create_virtual_rails "M1" \
    -synthesize_power_plan -synthesize_power_pads -use_strap_ends_as_pads
# If you have met your power budget, go ahead and commit the power plan.
commit_fp_rail

# Some options to make routing go faster
# zroute is now the default...
set_route_mode_options -zroute true
set_host_options -max_cores 16
## FIXME: need to figure out how to get route_opt commands below to run only one iteration
#route_zrt_auto -max_detail_route_iterations 1

# Perform clock tree synthesis. To look at the generated clock tree choose
# Clock > Color By Clock Trees. Hit Reload, and then hit OK on the popup window.
# Now you will be able to see the synthesized clock tree.
clock_opt -only_cts -no_clock_route
route_zrt_group -all_clock_nets -reuse_existing_global_route true -max_detail_route_iterations 1

# Route the remaining nets
# NB: Set some flags to keep iterations down while routing
set_route_opt_strategy -xtalk_reduction_loops 1 -search_repair_loops 1  -eco_route_search_repair_loops 1
route_opt -initial_route_only
route_opt -skip_initial_route -effort low

# Add filler cells.
insert_stdcell_filler \
    -cell_with_metal "SHFILL1 SHFILL2 SHFILL3" \
    -connect_to_power "VDD" -connect_to_ground "VSS"
```

*/

```
# FIXME: try without any of the flags
# FIXME: generated an error / rv of 0
route_opt -incremental -size_only

# Your design is now on silicon! Generate the post place and route netlist,
# the constraint file, and parasitics files to generate power estimates.
change_names -rules verilog -hierarchy
write_verilog "${DESIGN_NAME}.output.v"
write_sdf "${DESIGN_NAME}.output.sdf"
write_sdc "${DESIGN_NAME}.output.sdc"
extract_rc -coupling_cap
write_parasitics -format SBPF -output "${DESIGN_NAME}.output.sbpf"
write_stream "${DESIGN_NAME}.output.gds2"

# Write reports
report_timing -path full -delay min -max_paths 10 -nworst 2 > ${DESIGN_NAME}.icc.holdtiming
report_timing -path full -delay max -max_paths 10 -nworst 2 > ${DESIGN_NAME}.icc.setuptiming
report_area -hierarchy > ${DESIGN_NAME}.icc.area
report_power -hier -hier_level 2 > ${DESIGN_NAME}.icc.power
#report_resources > ${DESIGN_NAME}.icc.resources
report_constraint -verbose > ${DESIGN_NAME}.icc.constraint
report_reference -nosplit -hierarchy > ${DESIGN_NAME}.icc.ref

# Save and close library.
save_mw_cel
close_mw_cel

quit
```

Sample Program

```
/*
 * Userspace program that communicates with the led_vga device driver
 * primarily through ioctls
 *
 * Stephen A. Edwards
 * Columbia University
 */

#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"

#define NTHREADS 2
#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;

                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*/
                //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 (!op.length){
                    //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;
        int *data_in0, * data_out;

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

        struct timeval start, end;

        /* get the size of the data to transfer from the user */
        if (argc != 2){
            printf ("Usage: %s n\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));

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

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

        printf("FIFO Userspace program started\n");

        printf("Thread structs allocated\n\n\n");

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




        gettimeofday(&start, NULL);
        pthread_create(&(threads[0]), NULL,worker,&write_job0);
        pthread_create(&(threads[1]), NULL,worker,&read_job);
        //printf("Threads started!\n");

        for ( i = 0 ; i < NTHREADS ; i++ ){
                pthread_join(threads[i],NULL);
        }
        gettimeofday(&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 */

        printf("FIFO Userspace program terminating: %d\n",i);
        return 0;
}
```