Sports Arbitrage Bettor
Final Report

CSEE 4840: Embedded Systems Design

Brennan McManus (bm2530), Shivan Mukherjee (sm5155), Jonathan Nalikka (jmn2193), Chelsea Soemitro (crs2221), Shreya Somayajula (svs2137)

Spring 2024

Contents

1 Overview

The primary purpose of this project is to implement a hardware-accelerated arbitrage detector using a Field-Programmable Gate Array (FPGA). Particularly, our system detects combinations of bets on NBA games that result in guaranteed profit, i.e. are arbitrage opportunities, using real-time data.

1.1 Arbitrage Betting

Arbitrage betting is a strategy where bettors can place multiple bets on the same event to guarantee a profit no matter the result. This takes advantage of different sportsbooks offering different odds on the same event. These odds periodically update before events and while events are underway, which means bettors must act quickly when arbitrage opportunities present themselves. By using custom hardware on the FPGA to execute multiple arbitrage calculations simultaneously, we can achieve significantly lower latency compared to software-based solutions.

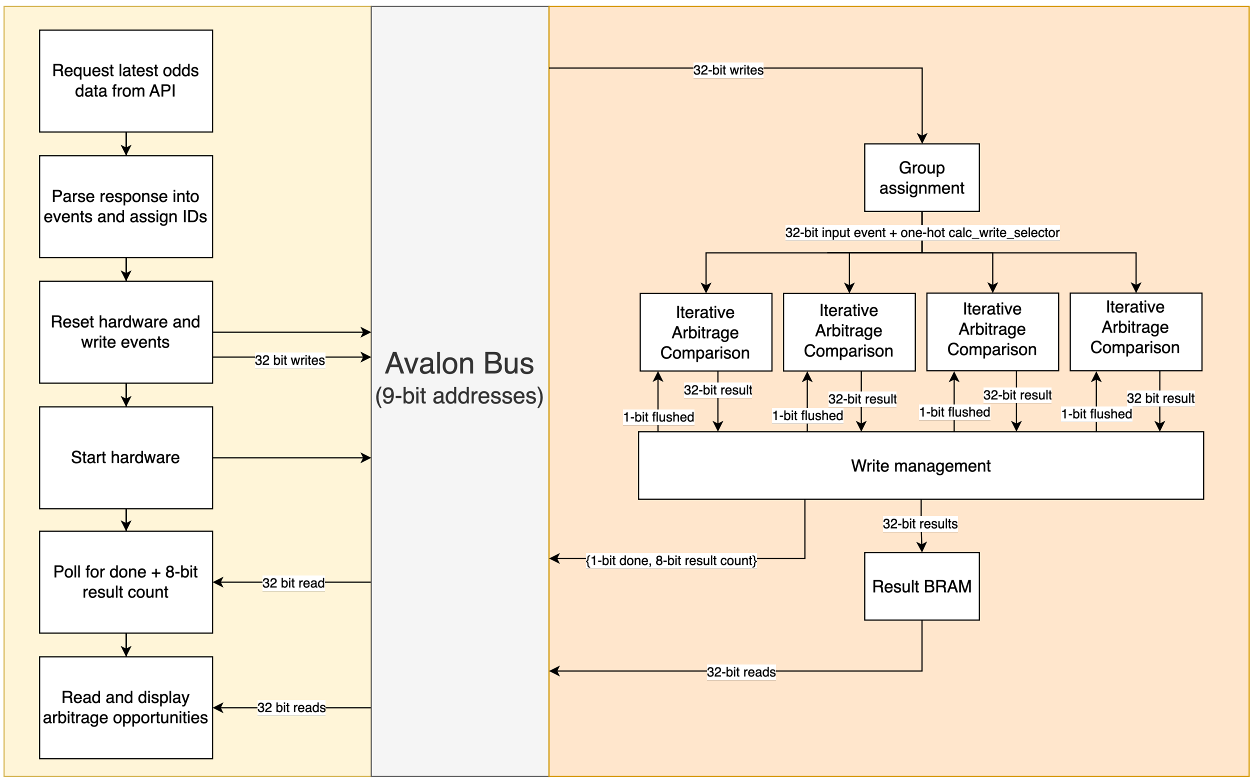
1.2 Terms

Money Line Bets: A money line bet, or head-to-head bet, is a wager on which of the two teams the bettor expects to win, regardless of the margin of victory.
Decimal Odds: Decimal odds are a format of betting odds that represent the total amount one will receive if a bet is successful, including the original stake. They are given as floating point values that range from around 1.001 (representing 1/1000) to 1001 (representing 1000/1).

2 System Block Diagram 🤖

2.1 Overall Structure

Data scraping and parsing/formatting data takes place in software, with the arbitrage calculations occurring in custom hardware. Each betting opportunity (henceforth referred to as as "event") is represented as a 32-bit struct, which our software prepares a buffer of and writes to our hardware. From there our custom hardware groups these structs by game id, and performs a series of arbitrage calculations, comparing each potential pair of them before sending any results back to hardware as 32 bit result structs.



2.2 Workflow Description

Our program begins by using the python requests library to call a betting data API (elaborated in Section 3.1 Data Acquisition) to collect scraped data from various sports betting websites. We organize this raw data into a list of our 32-bit event representation, then send the data to the hardware via calls to ioctl(), defined in our device driver. Our driver supports writes, which reset the hardware, write events, and signal the hardware to begin; and reads, which poll for the hardware finishing then consume the results. Once populated and started, our custom hardware runs a series of calculations in parallel to search for arbitrage opportunities, and makes any results available for the driver to read and copy to userspace. Then, in software, we retrieve and output the arbitrage results.

3 Pre-Processing in Software ☁️

3.1 Data Acquisition

To access sportsbook data, we query an odds API in Python. The website the-odds-api provides historical data and real-time odds data in a JSON format. Particularly, we fetch NBA betting data for head-to-head bets provided by bookies in the United States for the current day. A sample of the data is shown below, with some fields omitted.

GameID	Home Team	Away Team	Bookmaker	Outcome	Price
eef78bee2f630d615486f953ca851264	Cavaliers	Grizzlies	DraftKings	Cavaliers	1.05
eef78bee2f630d615486f953ca851264	Cavaliers	Grizzlies	DraftKings	Grizzlies	12.0

Table 1: Cleveland Cavaliers vs Memphis Grizzlies (Subset of Betting Outcomes)

3.2 Data Parsing

Once the data is fetched, the raw JSON file is transformed to a new csv file with the following standardized format:

| Game ID | Home Team | Away Team | Bookmaker ID | Bookmaker Title | Outcome Name | Outcome Price |

3.2.1 Game and Bookie Encoding

To facilitate communication with the hardware, we maintain a mapping of actual game IDs to simple game IDs in software, with valid game IDs ranging from 0 to 15. Similarly, we map bookmaker names to bookie IDs, where valid bookie IDs also range from 0 to 15.

3.2.2 Conversion to Fixed Point

The odds API provides odds as floating point values. Due to the complexities and performance risks of performing floating-point operations in hardware with the floating point IP, we instead convert these values into a 20-bit fixed-point representation.

We use the first 10 bits for the integer portion and the latter 10 for the decimal portion. This split ensures that we have sufficient range to cover the expected odds values from the API, while still maintaining acceptable precision for the decimal portion. We compared calculations using our fixed-point representation in software against floating-point-only versions, and found that their results always matched and were well within the precision needed for arbitrage comparisons. The C implementation for these conversions are given below: Note that `FIXED_POINT_FRACTIONAL_BITS` is defined as 10, and `fixed_point_t` as a `uint32_t`.

Algorithm 1 double_to_fixed(double input)

```
return (fixed_point_t)(round(input * (1 << FIXED_POINT_FRACTIONAL_BITS)));
```

Algorithm 2 fixed_to_double(fixed_point_t input)

```
return ((double)input / (double)(1 << FIXED_POINT_FRACTIONAL_BITS));
```

4 Software-Hardware Interface 💻

4.1 Struct Representation

Once we acquire the data over the network from the various sports betting sites and process it, we store each event in software in a `arb_event_t` C struct, packed into a bitfield to limit the size of the struct to 32-bits (the maximum hardware supported write size), to simplify later per-write routing in hardware.

```
typedef struct {
    fixed_point_t odds:      20;
    uint32_t game_id:        4;
    uint32_t bookie_id:      4;
    uint32_t outcome:        1;
    uint32_t unused:         3;
} arb_event_t;
```

For hardware to deliver the arbitrage results back to software, we use a 32-bit output vector `result`, that will eventually be written to the result BRAM and mapped back to a C struct in software. When software reads an arbitrage result from the hardware, it is parsed into the following struct, an `arb_result_t`.

```
typedef struct {
    fixed_point_t arb_prob:   20;
    uint32_t game_id:         4;
    uint32_t bookie_id_a:     4;
    uint32_t bookie_id_b:     4;
} arb_result_t;
```

Finally, for hardware to indicate to software that all arbitrage calculations have been completed, as well as report the number of arbitrage opportunities found, we use the following representation:

```
typedef struct {
    uint32_t done:            1;
    uint32_t result_count:    8;
    uint32_t padding:         23;
} arb_read_regs_t;
```

4.2 Avalon Interface Registers

In our system, the software and hardware communicate over the avalon bus with our custom hardware as a memory-mapped peripheral. The outermost hardware takes the avalon-bus-driven signals `read`, `write`, `chipselect`, `address`, and `writedata` as inputs. It interprets these to generate internal control signals, `arb_read`, `arb_write`, `arb_reset`, and `arb_start`, which are used to control the main internal component, `bettor_arb`. When `arb_read` is high, it indicates that there is a arbitrage result to be read. When `arb_write` is high, the current `writedata` is interpreted as a new event in component. When `arb_reset` is high, all of the inner modules counters are cleared back to initial values. When `arb_start` is high, the inner module begins calculation using all of the events that have been written since the last reset. Note read- and writedata are 32 bits wide, and the address is 9 bits wide. The avalon bus address is 1 bit wider than the internal module's read address width of 8 bits. This extra bit is to support ioreads() polling for done, in addition to needing to support addressing up to 255 results. A read from address 0 is used to poll for done and read `resultcount`.

Address	Function
0	ignore writedata, raise arb_reset
1	ignore writedata, raise arb_start
2	write event (in writedata) at address, raise arb_write

Table 2: Write Registers (write is high)

Address	Function
0	read arb_read_regs_t (done struct)
1	read arb_result_t (result struct)
2	read arb_result_t (result struct)
3-255	...
256	read arb_result_t (result struct)

Table 3: Read Registers (read is high)

Note that raising `arb_reset` serves as the `reset` signal for all inner modules, indicating that all wires should be cleared. Raising `arb_start` serves as the start signal for the calculation manager modules (discussed further in 5.2 Comparison Phase), indicating that the arbitrage calculations should begin. Due to the fact that readdata and writedata are 32-bit, the word size for this interface is 4 bytes. The register addresses above use word addressing, not byte addressing.

4.3 Userspace Program

In `src/calc_arb.c`, the user program that initiates the end-to-end pipeline, we read from and write to hardware using `ioctl()`:

```
if (ioctl(fd, CALC_ARB_WRITE_EVENTS, events) == -1) {
    perror("ioctl write");
    return -1;
}

if (ioctl(fd, CALC_ARB_READ_EVENTS, result_buf) == -1) {
    perror("ioctl read");
    return -1;
}
```

4.4 Kernel Device Driver

For the kernel module to communicate with hardware, we implemented functions to handle the `ioctl()` calls from userspace.

On writes, i.e. case `CALC_ARB_WRITE_EVENTS`, the device driver copies the number of events and the events themselves from userspace to the kernel, then calls `write_events()`, which:

- Raises `reset` with `iowrite32()`
- Sends `arb_event_t` structs in a loop with `iowrite32()`

- Raises `start` with `iowrite32()`

On reads, i.e. case `CALC_ARB_READ_EVENTS`, the device driver calls `read_result()` to retrieve the buffer containing the number of arbitrage results and the results themselves, then copies this buffer from the kernel to the userspace. Particularly, `read_result()` does:

- Polls hardware until `done = 1` with `ioread32()`
- Reads `result_count` number of `arb_result_t` structs with `ioread32()`

5 Hardware Design 💿

5.1 Arbitrage Algorithm

We use the following simple formula to determine whether there is an arbitrage opportunity:

$$\frac{1}{a} + \frac{1}{b} < 1$$

where a is the probability of outcome 1 from bookie a, and b is the probability of outcome 2 from bookie b. If this inequality is true, then there is an arbitrage opportunity for this given combination of odds. Otherwise, the pair of is not an arbitrage opportunity.
To avoid the expensive division operation in hardware, we rewrote the formula as such:

$$a + b < ab$$

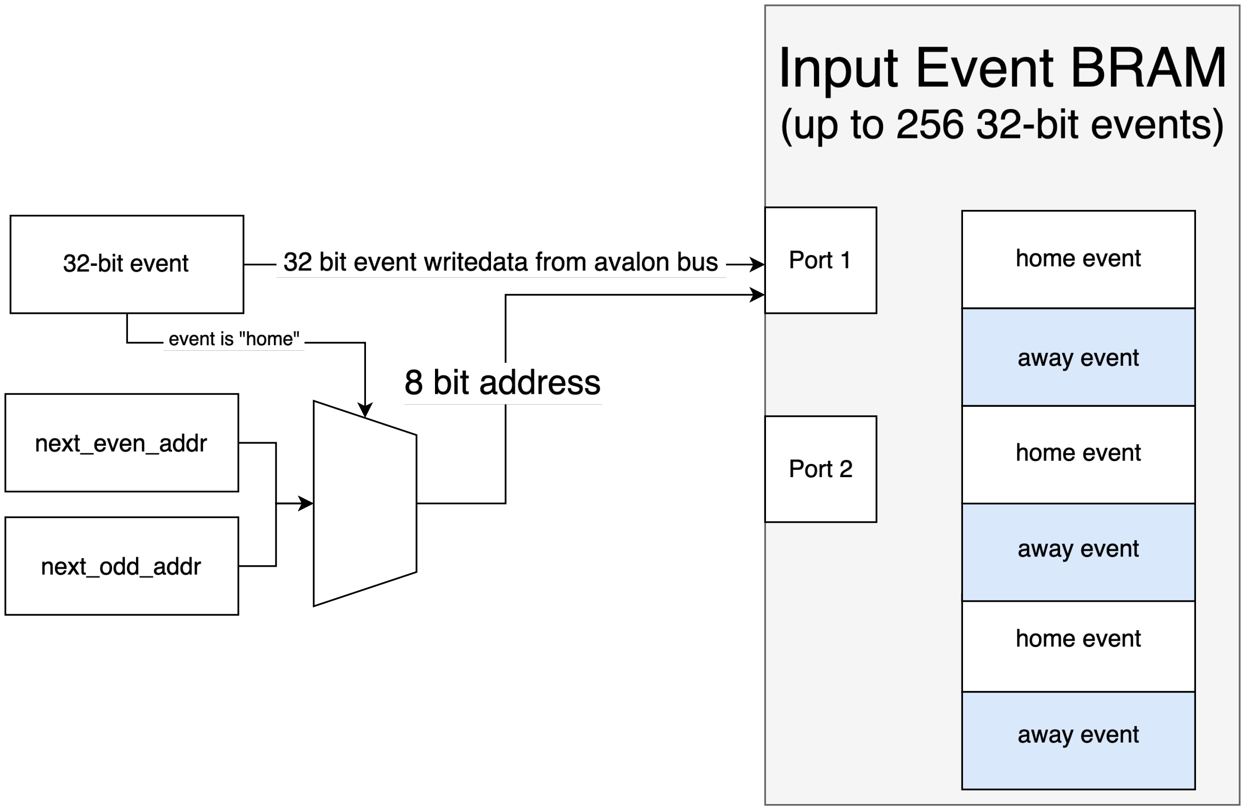
5.2 Grouping Module

Once the events start coming into hardware, we need to send them to their respective calc modules. We separate them by game ID, such that each calc module will deal with all of the events from one particular game ID. To do so, in the grouping module we select the correct module to pipe the `writedata` to using `calc_write_selector`. This is a 16-bit register, assigned as 1/0 depending if write is high, shifted by the game id. For example, if we have just performed an `iowrite32()` for an event with game ID 2, write will be high, and shifted to the left by 2, resulting in the value `0000000000000100`. Each bit in our `calc_write_selector` wire is then associated with a specific calculation module, so if their particular bit is high, they will write whatever is in `writedata` to their BRAM.

5.3 Event Writing Phase

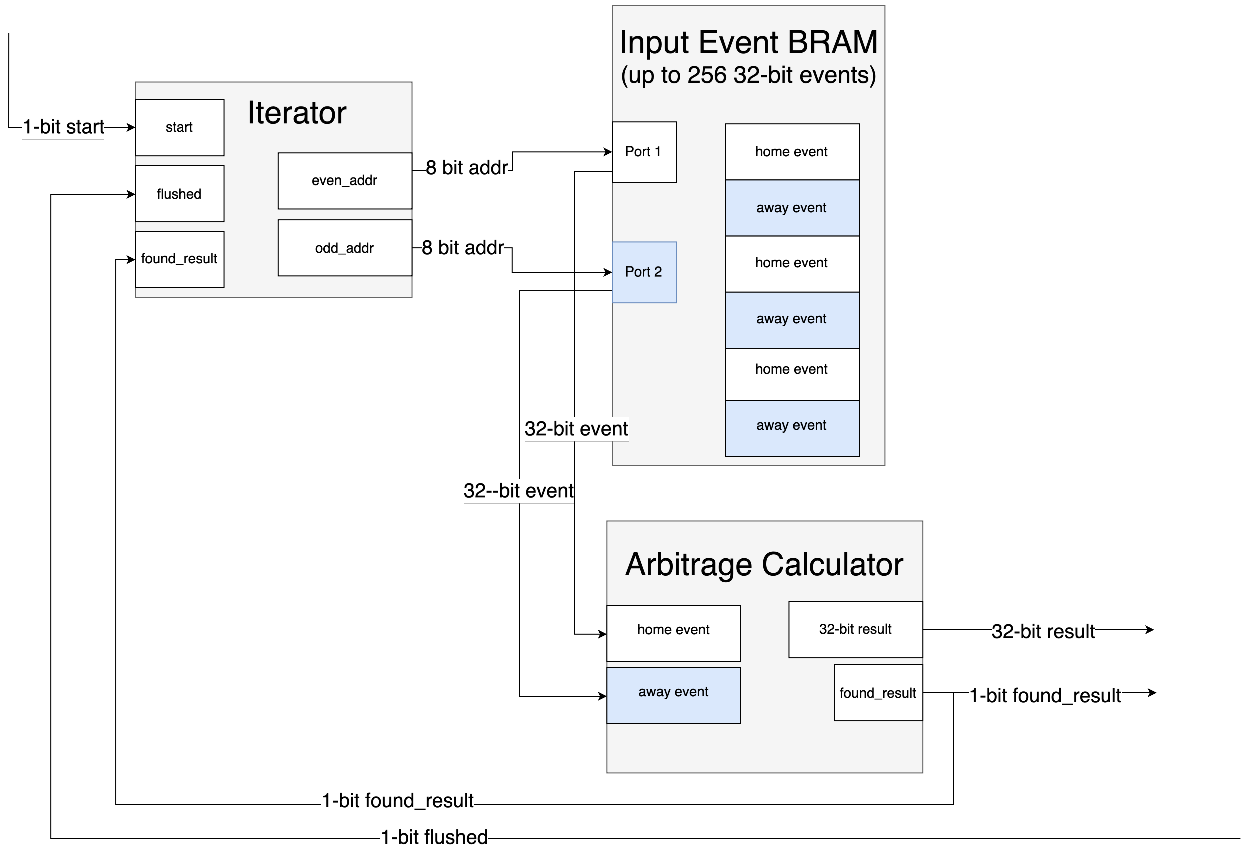
5.3.1 Input Event BRAM Format

Once an individual calculation module sees that its `calc_write_selector` bit is high, it takes the `writedata`, which is a 32-bit event struct. It inspects the bits, which indicates whether the odds are for a home win or an away win. If it is a home win, then it is placed into the BRAM at an even address index. Otherwise, it is placed into an odd address index. We keep track of the next address to place the events in with two registers, `next_even_addr` and `next_odd_addr`.



5.4 Comparison Phase

Next, we actually perform the comparisons. As mentioned above, once all of the events are written to hardware, the software performs another `iowrite32()` to raise the start signal. This start signal will begin the iterator. This iterator module lets the arbitrage calculator know which addresses in memory to look at for the events that they should be comparing at the moment. There are two registers, `even_addr` and `odd_addr`. `even_addr` begins at 0, while `odd_addr` begins at 1. At each clock cycle, these addresses are incremented by 2, until they reach the number of events that we wrote into the BRAM. The even and odd addresses are then used to read from the BRAM, using both ports to read two events simultaneously in the same clock cycle. The arbitrage calculator then takes those two events and performs the calculation.

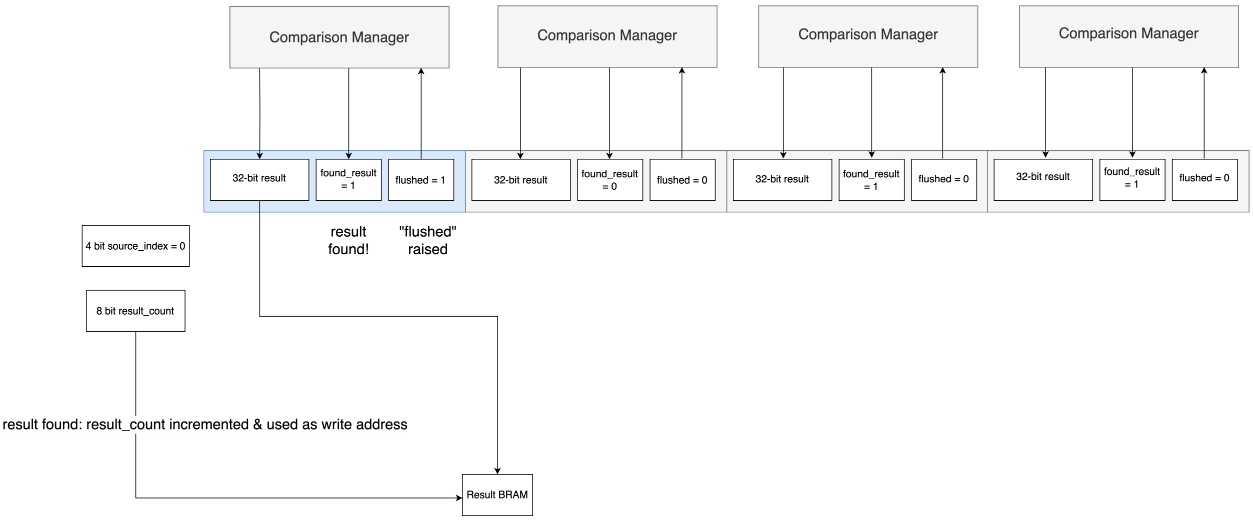


5.5 Parallelization & Synchronization

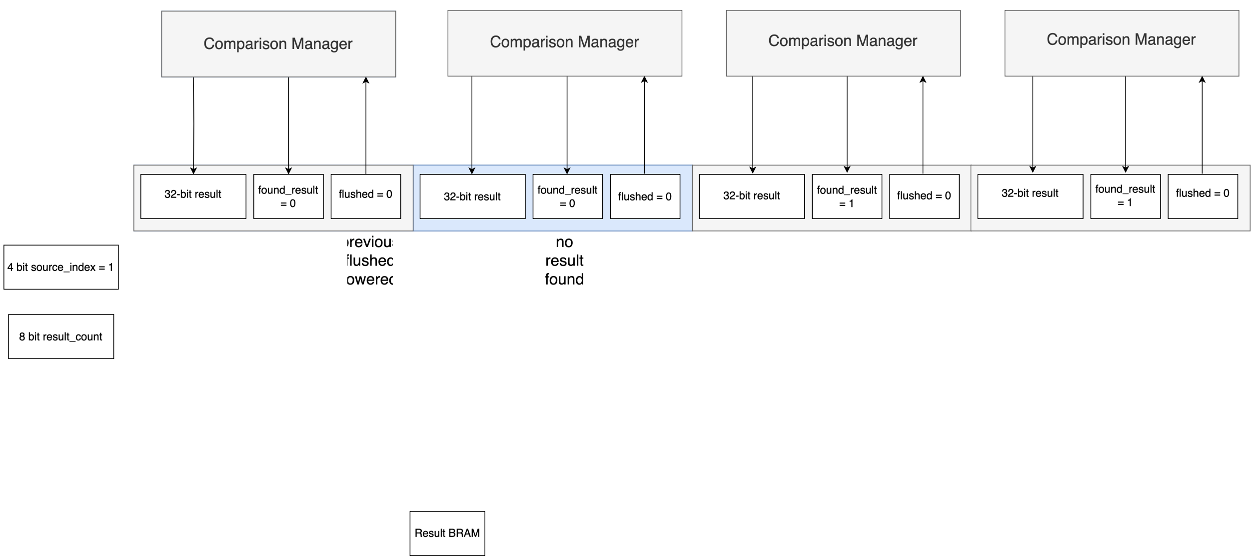
The complexity of the hardware stems from the synchronization of the various aspects of the modules. A large bottleneck comes in the form of the writing to the results BRAM, since we cannot have all calculators writing to the BRAM at the same time. To handle the sychronization, we use both narrow and wide vectors. Narrow vectors are simply registers that want to share the value among all instances of the modules. For example, they all operate on the same clock cycle. We also use wide vectors, such as `calc_write_selector`, where each instance gets a slice of this vector to indicate its action at the time.

5.5.1 Write Manager

Below are diagrams for the hardware module which handles writes of result structs during our iterative arbitrage comparison. If the comparison manager finds an arbitrage, it raises the `found_result` wire so the write manager can write the 32-bit result struct to BRAM. We use an 8 bit `result_count` wire to address into the BRAM. Once the result has been written to memory we raise the `flushed` wire to indicate to the comparison manager we just wrote a result, so it's ok to move on.



In the case where an arbitrage is not found, `found_result` and `flushed` are set to 0.



6 Resource Utilization 📈

6.1 Timing

Our module's synchronous logic runs using a single shared 50MHz clock, using one of the board's base clocks. Our module is not overclocked, and Quartus timing analysis reports an Fmax of between 61 and 62 MHz, which suggests that we could slightly increase the clock frequency to see further marginal improvements to the speed of our module.

6.2 Board Resources

When instantiated for 16 parallel calculations, the fitting summary in Quartus reports the following resource utilization:

- Logic utilization: $1574/32,070$ (5%)
- Total registers: 1085
- Total pins: 333/457 (73%)
- Total block memory bits $122,880$ (3%)
- Total RAM Blocks (BRAM): 33/397 (8%)
- Total DSP Blocks: 16/87 (18%)

This analysis indicates that a key limiting resource for our system is the number of DSP blocks. The version we tested instantiates 16 individual calculation manager modules, each of which run in parallel and use their own DSP block for arbitrage calculations. This means we can only instantiate as many of these modules as the board has DSP blocks, which suggests a limit of 87. This limit defines how many can be instantied, despite the fact that the board can support upwards of 256 instances based only on BRAM block constraints.

7 Testing

7.1 Simulation

In the process of writing our hardware source code, we created and ran `Verilator` simulations to determine whether our Verilog modules were working as expected. Additionally, we inspected the timing diagrams produced by `gtkwave` to inspect per-cycle behavior. The simulation source code can be found in section 9.7, simulation.

7.2 Experiment Results

To determine both the correctness and efficiency of our end-to-end system, we mimicked our logic in its entirety in a pure Python implementation. This included optimizations we made in our hardware implementation, like only generating combinations between bets that have the potential to generate arbitrage. The Python implementation can be found in section 9.8.2, `python_alg.py`. Upon timing the execution of these programs with the `time` command, we found the following results (on average, over 30 independent runs). Note that before tracking any results, we ran both programs several times such that all file I/O was cached, and thus we were only timing the portion of the program related to arbitrage calculation. Given that the existing arbitrage detectors we found online were all in python we felt that this was a fair point of reference. We found that our device represents a significant improvement in terms of latency, by a wide margin.

Python	**Hardware**
0.220	0.006

Table 4: Time (in seconds)

8 Conclusion 🎀

8.1 Task Distribution

Brennan: Implemented the first simulation in C++ and Verilator, including fixed point conversion. Designed our interfaces, planned out each module, and drew our in-progress mess of a block diagram we referred to as we implemented. Organized simulation work and built a reference template simulator upon which each subsequent simulation was built. Drafted the odds calculation module, the Avalon interface, and portions of the main module and calculation manager. Implemented event grouping, and adapted the modules of our single calc-manager implementation into parallel version using instance arrays (which are really just quite cool). Performed resource analysis, debugged moving from simulation to the board, and ran timing experiments.
Shivan: Did pre-processing steps for input into hardware. Sourced real-time data from API and processed data into csv files. Also worked on Verilator simulation of arbitrage calculation module. Wrote shell scripts for end-to-end simulation and expanded and improved python-only comparison implementation.
Jonathan: Wrote initial software simulation of arbitrage calculation in C, defined functions and macros for initializing event and result structs. Wrote first drafts of ioctl implementation and hardware module which accepted a buffer of event structs and sent a result struct back to software. This initial module was used to scale up calculations in parallel and make changes to fit specifications of the project.
Chelsea: Wrote the write manager and associated verilator simulation. Planned and debugged the communication and timing between the write manager and calculation managers. Worked on debugging hardware module once we moved from simulation to the board, and adapted our initial kernel driver to work with the avalon interface we designed in simulation.
Shreya: Helped write main program for software simulation in C. Contributed to C++ simulations for hardware modules. Helped write Verilog modules needed for single-threaded arbitrage calculations, like `calc_odds.sv` and `iterator.sv`.

8.2 Lessons Learned

As a team, it's hard to articulate everything we've learned. From ideation to actualizing our full hardware implementation, it has been a long road together. Here are just a few of our many lessons:

- Starting small & scaling up makes implementing in hardware much more approachable.
- Simulations are your best friend – be it in Verilator or pure software. The version of our hardware that worked in Verilator simulation needed no changes to work in hardware.
- Avalon bus address math can be trickier than it seems (casting your pointers to the right type makes it easier).
- Ask for help, especially when the previous item is the source of your woes.
- Arbitrage is rare!
- Hardware acceleration is legit.

- Trust your teammates; know when to delegate tasks versus collaborate on a single task side-by-side.
- System Verilog supports a lot of features that would have been a nightmare to work without—including vector-of-vectors inputs and outputs for instance-arrayed modules.
- Nothing ever quite goes to plan. Adjusting on the fly and being flexible is key.

9 Code Listings

9.1 include

9.1.1 arb_buf.h

```c
#ifndef _ARB_BUF_H
#define _ARB_BUF_H

#include "arb.h"

struct event_buf {
    int len;
    arb_event_t events_vec [];
};

struct result_buf {
    int len;
    arb_result_t arbs_vec [];
};

#endif
```

9.1.2 arb_ioctl.h

```
#ifndef _ARB_IOCTL_H
#define _ARB_IOCTL_H

#include <linux/types.h>
#include <linux/ioctl.h>

#include "arb.h"
#include "avalon_interface.h"

#define CALC_ARB_MAGIC 'q'

/* ioctls and their arguments */
#define CALC_ARB_WRITE_EVENTS _IOW(CALC_ARB_MAGIC, 1, arb_event_t)
#define CALC_ARB_READ_RESULTS  _IOR(CALC_ARB_MAGIC, 2, arb_result_t)


#endif
```

9.1.3 arb_parsing.h

```c
#ifndef _ARB_PARSING_H_
#define _ARB_PARSING_H_

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "../include/arb.h"
#include "../include/arb_buf.h"
#include "../include/fixed_point.h"

#define NUM_EVENTS 227

/* Load bets from csv file into temporary struct */
static int parse_csv(const char *filename, struct event_buf *event_buf) {
    FILE* fp;

    // Open csv file for reading
    if ((fp = fopen(filename, "r")) == NULL) {
        perror("fopen()");
        return -1;
    }

    char line[4096];
    int event_idx = 0;

    // Skip header line
    fgets(line, sizeof(line), fp);

    uint32_t game_id;
    char *home;
    uint32_t bookie_id;
    uint32_t outcome;
    char *odds_string;
    double odds;

    // Parse each portion of .csv file
    while (fgets(line, sizeof(line), fp) != NULL) {
        // Parse Game ID, Home Team, Away Team, and Bookie ID
        game_id = atoi(strtok(line, ","));
        home = strtok(NULL, ",");
        strtok(NULL, ",");
        bookie_id = atoi(strtok(NULL, ","));

        // Skip bookie name
        strtok(NULL, ",");

        // Determine outcome based on Outcome Name
        outcome = (strcmp(strtok(NULL, ","), home) == 0) ? 0 : 1;

        // Parse odds
        odds_string = strtok(NULL, ",");
        sscanf(odds_string, "%lf", &odds);


        // Add new event
        arb_event_t new_event = {0};
        init_event(&new_event, double_to_fixed(odds), game_id, bookie_id, outcome);
        event_buf->events_vec[event_idx++] = new_event;
    }

    fclose(fp);
    return event_idx;
}

static struct event_buf *events_from_file(const char *filename)
```

```
{
    struct event_buf *events = (struct event_buf *) malloc(sizeof(struct event_buf) + NUM_EVENTS
    * sizeof(arb_event_t));
    if (events == NULL) {
        perror("malloc");
        exit(1);
    }

    events->len = parse_csv(filename, events);
    if (!events->len) {
        free(events);
        perror("parse csv");
        exit(1);
    }

    printf("%s game odds loaded...\n", filename);

    return events;
}

static void print_arb_results(struct result_buf *arbs)
{
    printf("Arbitrage Opportunities:\n");
    arb_result_t arb;
    double arb_prob;
    for (int i = 0; i < arbs->len; i++) {
        arb = arbs->arbs_vec[i];
        arb_prob = fixed_to_double(arb.arb_prob);
        printf("Game ID: %d, Bookie A: %d, Bookie B: %d, Arbitrage Probability: %f\n", arb.
    game_id, arb.bookie_id_a, arb.bookie_id_b, arb_prob);
    }
}

#endif
```

9.1.4 arb.h

```
#ifndef _ARB_H
#define _ARB_H

#define OC_IS_A(event) (!(event)->outcome)
#define OC_IS_B(event) ((event)->outcome)

typedef struct {
  uint32_t odds:    20; // 19:0
  uint32_t game_id: 4;  // 23:20
  uint32_t bookie_id: 4;  // 27:24
  uint32_t outcome: 1;  // 28
  uint32_t unused:  3;  // 31:29
} arb_event_t;

typedef struct {
  uint32_t arb_prob:    20; // 19:0
  uint32_t game_id:   4;  // 23:20
  uint32_t bookie_id_a: 4;  // 27:24
  uint32_t bookie_id_b: 4;  // 31:28
} arb_result_t;

static void init_event(arb_event_t *event, uint32_t odds, uint32_t game_id, uint32_t bookie_id,
    uint32_t outcome)
{
  event->odds = odds;
  event->game_id = game_id;
  event->bookie_id = bookie_id;
  event->outcome = outcome;
}

static void init_arb(arb_result_t *arb, uint32_t arb_prob, uint32_t game_id, uint32_t bookie_id_a
    , uint32_t bookie_id_b)
{
  arb->arb_prob = arb_prob;
  arb->game_id = game_id;
  arb->bookie_id_a = bookie_id_a;
  arb->bookie_id_b = bookie_id_b;
}

#endif
```

9.1.5 avalon_interface.h

```
#ifndef _ARB_AVALON_INTERFACE_H_
#define _ARB_AVALON_INTERFACE_H_

#define ARB_RESET_ADDR 0
#define ARB_START_ADDR 1
#define ARB_EVENT_WRITE_ADDR 2

#define ARB_READ_REGS_ADDR 0
#define ARB_RESULT_READ_ADDR(x) (x+1)

typedef struct {
  uint32_t done:      1;
  uint32_t result_count:  8;
  uint32_t padding:   23;
} arb_read_regs_t;

#endif
```

9.1.6 fixed_point.h

```
#ifndef _FIXED_POINT_H
#define _FIXED_POINT_H

#include <math.h>
#include <stdint.h>
#include <stdlib.h>

#define FIXED_POINT_MAX 1024
#define FIXED_POINT_FRACTIONAL_BITS 10

#define IS_ARB(a, b)((((a) + (b)) << FIXED_POINT_FRACTIONAL_BITS) < ((a) * (b)))
#define PROFIT(a, b, i)((((a) * (b) * (i)) / ((a) + (b))) - (i))

typedef uint32_t fixed_point_t;

static inline fixed_point_t double_to_fixed(double input)
{
  return (fixed_point_t)(round(input * (1 << FIXED_POINT_FRACTIONAL_BITS)));
}

static inline double fixed_to_double(fixed_point_t input)
{
    return ((double)input / (double)(1 << FIXED_POINT_FRACTIONAL_BITS));
}

#endif
```

9.2 scrape

9.2.1 scrape.py

```python
import requests
import json
import os
import sys
import csv

# Function to transform and write data to a CSV file
def write_to_csv(transformed_data, file_path):
    with open(file_path, mode='w', newline='') as file:
        writer = csv.writer(file)

        # Write the header row
        writer.writerow(['Game ID', 'Home Team', 'Away Team', 'Bookmaker ID', 'Bookmaker Title', 'Outcome Name', 'Outcome Price'])

        for game in transformed_data:
            game_id = game['id']
            home_team = game['home_team']
            away_team = game['away_team']

            for bookmaker in game['bookmakers']:
                bookmaker_id = bookmaker['key']
                bookmaker_title = bookmaker.get('title', 'N/A')  # Using .get() to handle missing fields gracefully

                for market in bookmaker['markets']:
                    for outcome in market['outcomes']:
                        outcome_name = outcome['name']
                        outcome_price = outcome['price']

                        # Write a row for each outcome
                        writer.writerow([game_id, home_team, away_team, bookmaker_id, bookmaker_title, outcome_name, outcome_price])

def transform_data(data):
    # Dictionary to hold the new integer IDs for bookmakers
    bookie_id_map = {}
    bookie_counter = 0

    for game_index, game in enumerate(data):
        # Redefine the game ID
        game['id'] = game_index

        # Remove unwanted fields
        game.pop('sport_key', None)
        game.pop('sport_title', None)
        game.pop('commence_time', None)

        for bookmaker in game['bookmakers']:
            # Check if bookmaker's key is already mapped, otherwise add to the map
            if bookmaker['key'] not in bookie_id_map:
                bookie_id_map[bookmaker['key']] = bookie_counter
                bookie_counter += 1

            # Redefine the bookmaker's key to the new ID
            bookmaker['key'] = bookie_id_map[bookmaker['key']]

            # Remove unwanted fields
            bookmaker.pop('last_update', None)

            for market in bookmaker['markets']:
                market.pop('key', None)
                market.pop('last_update', None)

```

```python
    return data

API_KEY = '01a2f84bb8770bcc6ea4494a2d85b5f2'

SPORT = 'basketball_nba'  # use the sport_key from the /sports endpoint below, or use 'upcoming' to see the next 8 games across all sports
REGIONS = 'us'  # uk | us | eu | au. Multiple can be specified if comma delimited
MARKETS = 'h2h'  # h2h | spreads | totals. Multiple can be specified if comma delimited
ODDS_FORMAT = 'decimal'  # decimal | american
DATE_FORMAT = 'iso'  # iso | unix

if __name__ == "__main__":
    odds_response = requests.get(
        'https://api.the-odds-api.com/v4/sports/{}/odds'.format(SPORT),
        params={
            'api_key': API_KEY,
            'regions': REGIONS,
            'markets': MARKETS,
            'oddsFormat': ODDS_FORMAT,
            'dateFormat': DATE_FORMAT,
        }
    )

    if odds_response.status_code != 200:
        print('Failed to get odds: status_code {}, response body {}'.format(odds_response.status_code, odds_response.text))
    else:
        odds_json = odds_response.json()
        transformed_json = transform_data(odds_json)

        csv_file = 'nba_game_odds.csv'
        write_to_csv(transformed_json, os.path.join(sys.path[0], csv_file))

        print('Remaining requests', odds_response.headers['x-requests-remaining'])
        print('Used requests', odds_response.headers['x-requests-used'])
```

9.3 src

9.3.1 calc_arb.c

```c
#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdint.h>
#include <unistd.h>

#include "../include/arb.h"
#include "../include/arb_ioctl.h"
#include "../include/fixed_point.h"
#include "../include/arb_parsing.h"

void print_arb_event(arb_event_t event)
{
  printf("Outcome: %c\n", OC_IS_A(&event) ? 'a' : 'b');
  printf("Game ID: %d\n", event.game_id);
  printf("Bookie ID: %d\n", event.bookie_id);
  printf("Odds: %f\n", fixed_to_double(event.odds));
        printf("Unused: %d\n", event.unused);
}

void print_arb_result(arb_result_t result)
{
  printf("Game ID: %d\n", result.game_id);
  printf("Bookie ID a: %d\n", result.bookie_id_a);
  printf("Bookie ID b: %d\n", result.bookie_id_b);
  printf("Odds: %f\n", fixed_to_double(result.arb_prob));
}

int main(int argc, char **argv)
{
  if (argc != 2) {
    fprintf(stderr, "usage: %s <events-file>\n", argv[0]);
    exit(1);
  }

  int fd;
    struct result_buf *result_buf = malloc(sizeof(*result_buf) + 256 * sizeof(arb_result_t));

  static const char filename[] = "/dev/calc_arb";
  char *events_file = argv[1];
  struct event_buf *events = events_from_file(events_file);

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

  //call to ioctl
  if (ioctl(fd, CALC_ARB_WRITE_EVENTS, events) == -1){
    perror("ioctl write");
    return -1;
  }

  if (ioctl(fd, CALC_ARB_READ_RESULTS, result_buf) == -1){
    perror("ioctl read");
    return -1;
  }

    printf("Calculations complete...\n");
    print_arb_results(result_buf);

  free(result_buf);
```

```
  free(events);

  return 0;
}
```

9.3.2 main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "../include/arb.h"
#include "../include/arb_buf.h"
#include "../include/arb_parsing.h"

static struct arb_result_buf *calc_profits(struct arb_event_buf *events)
{
    int i, j;
    int arb_buf_len = 0;
    struct arb_result_buf *arbs;
    arb_event_t *event_a, *event_b;

    arbs = alloc_result_buf(events->len * events->len);

    for(i = 0, event_a = events->events_vec; i < events->len; i++, event_a++) {
        if(OC_IS_B(event_a))
            continue;

        for(j = 0, event_b = events->events_vec; j < events->len; j++, event_b++) {
            if(OC_IS_A(event_b))
                continue;

            if(event_b->game_id != event_a->game_id)
                continue;

            if(event_a->bookie_id == event_b->bookie_id)
                continue;

            if(IS_ARB(event_a->odds, event_b->odds)) {
                init_arb(arbs->arbs_vec + arb_buf_len++, event_a->odds + event_b->odds, event_a->game_id, event_a->bookie_id, event_b->bookie_id);
                double profit = PROFIT(fixed_to_double(event_a->odds), fixed_to_double(event_b->odds), 100);
                printf("Profit: %f\n", profit);
            }
        }
    }

    arbs->len = arb_buf_len;

    return arbs;

}

int main(int argc, char **argv) {
  if (argc != 2) {
    fprintf(stderr, "usage: %s <events-file>\n", argv[0]);
    exit(1);
  }

  char *events_file = argv[1];
  struct event_buf *events = events_from_file(events_file);

    struct arb_result_buf *arbs = calc_profits(events);
    if (arbs == NULL) {
        perror("calc_profits");
        return 1;
    }

    printf("Calculations complete...\n");

    print_arb_results(arbs);

```

```
    return 0;
}
```

9.4 kernel_mod

9.4.1 calc_arb.c

```c
/* * Device driver for the sports arbitrage calculator
 *
 * A Platform device implemented using the misc subsystem
 *
 * Jonathan Nalikka
 * 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 calc_arb_ball.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree calc_arb.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 <linux/slab.h>
#include <linux/printk.h>

#include "../include/arb_ioctl.h"
#include "../include/arb_buf.h"
#include "../include/avalon_interface.h"

#define DRIVER_NAME "calc_arb"

/*
 * macros to get bytes 0->3 of struct arb
 */

#define BYTE_X(word, x) ((char)(((word) >> (8 * x)) & 0xff))
#define EVENT_STRUCT_BYTE_X(event, x) (BYTE_X(*(uint32_t *)(event), x))

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


void print_result(arb_result_t *result)
{
    pr_info("Game ID: %d, Bookie A: %d, Bookie B: %d, Arbitrage Probability: %d\n", result->
    game_id, result->bookie_id_a, result->bookie_id_b, result->arb_prob);
}

```

```
void print_arb_event(arb_event_t event)
{
  pr_info("Outcome: %c, Game ID: %d, Bookie ID: %d, Odds: %x, Unused: %d\n",
        OC_IS_A(&event) ? 'a' : 'b', event.game_id, event.bookie_id, event.odds, event.unused);
}

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_events(struct event_buf *buf)
{
    int i;

  // (1) send reset signal
  iowrite32(0, ((uint32_t *)dev.virtbase) + ARB_RESET_ADDR);

  // (2) write events
    for(i = 0; i < buf->len; i++) {
        iowrite32(*(uint32_t *)((buf->events_vec) + i),
        ((uint32_t *)dev.virtbase) + ARB_EVENT_WRITE_ADDR);
    }

  // (3) raise start
    iowrite32(0, ((uint32_t *)dev.virtbase) + ARB_START_ADDR);
}

static struct result_buf *read_result(void)
{
  arb_read_regs_t read_regs;
  int i;
  struct result_buf *results_buf;
  uint32_t readdata;

  // (1) poll for done
  i = 1000;
  while(i--) {
    readdata = ioread32(((uint32_t *)dev.virtbase) + ARB_READ_REGS_ADDR);
    read_regs = *((arb_read_regs_t *) &readdata);

    if (read_regs.done)
      break;
  }

    results_buf = kmalloc(sizeof(int) + read_regs.result_count * sizeof(arb_result_t), GFP_KERNEL
    );
  results_buf->len = read_regs.result_count;

  // (2) read results structs
  for (i=0; i < results_buf->len; i++) {
    uint32_t readdata = ioread32(((uint32_t *)dev.virtbase) + ARB_RESULT_READ_ADDR(i));
    results_buf->arbs_vec[i] = *((arb_result_t *) &readdata);
  }

  return results_buf;
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long calc_arb_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    struct event_buf *buf;
  struct result_buf *res_buf;
    int len;
    size_t size;
```

```
  switch (cmd) {
      case CALC_ARB_WRITE_EVENTS:
        if (copy_from_user(&(len), (int *) arg,
               sizeof(int)))
          return -EACCES;

      size = sizeof(int) + len * sizeof(arb_event_t);
      buf = kmalloc(size, GFP_KERNEL);

      if (copy_from_user(buf, (struct event_buf *) arg,
               size))
          return -EACCES;

        write_events(buf);
        kfree(buf);
        break;

      case CALC_ARB_READ_RESULTS:
          res_buf = read_result();
          size = sizeof(int) + res_buf->len * sizeof(arb_result_t);
        if (copy_to_user((struct result_buf *) arg, res_buf,
             size))
          return -EACCES;
      kfree(res_buf);
        break;
      default:
        return -EINVAL;
  }

  return 0;
}

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

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

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

  /* Register ourselves as a misc device: creates /dev/calc_arb */
  ret = misc_register(&calc_arb_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;
  }
  //initial write
  //iowrite32(0xffffffff, dev.virtbase);
  return 0;

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

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

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id calc_arb_of_match[] = {
  { .compatible = "bettor,calc_arb-3.0" },
  {},
};
MODULE_DEVICE_TABLE(of, calc_arb_of_match);
#endif

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

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

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

module_init(calc_arb_init);
module_exit(calc_arb_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Jonathan Nalikka, Columbia University");
MODULE_DESCRIPTION("sports arbitrage calculator driver");
```

9.5 hardware

9.5.1 arb_avalon_interface.sv

```
module arb_avalon_interface(
  /* verilator lint_off UNUSEDSIGNAL */
  input logic     clk,
  input logic     reset,
  /* verilator lint_on UNUSEDSIGNAL */
  input logic     read,
  input logic     write,
  input logic     chipselect,

  // one wider than inner module to allow dummy 0th address for reading registers
  input logic [8:0] address,
  input logic [31:0]  writedata,

  output logic [31:0] readdata
);

  logic arb_start;
  logic arb_write;
  logic arb_read;
  logic [31:0] arb_writedata;
  logic [7:0] arb_readaddr;
  logic arb_done;
  logic [7:0] arb_resultcount;
  logic [31:0] arb_readdata;
  logic arb_reset;

  assign arb_read       = read && chipselect && address != 0;
  assign arb_write      = write && chipselect && address != 0;
  assign arb_writedata    = writedata; // contents of writedata are ignored when writing to 0th
    address
  assign arb_readaddr     = address[7:0] - 8'h1; // reading from this module over avalon from
    address + 1 reads inner module at 0

  always_comb begin
    if (chipselect && read) begin
      if (arb_read)
        readdata = arb_readdata;
      else
        readdata = {23'h0, arb_resultcount, arb_done}; //backwards bc endianness
    end else begin
      readdata = 0;
    end

    if (reset) begin
      arb_reset = 1;
      arb_start = 0;
    end else if (chipselect && write && address == 0) begin
      if (writedata == 0) begin
        arb_start = 0;
        arb_reset = 1;
      end else begin
        arb_reset = 0;
        arb_start = 1;
      end
    end else begin
      arb_start = 0;
      arb_reset = 0;
    end
  end

  bettor_arb #(.MAX_GAME_ID(15), .GAME_ID_WIDTH(4)) mainModule(
    .clk(clk),
    .reset(arb_reset),
    .start(arb_start),
```

```
    .write(arb_write),
    .writedata(arb_writedata),
    .readaddress(arb_readaddr),
    .readdata(arb_readdata),
    .done(arb_done),
    .resultcount(arb_resultcount)
  );
endmodule
```

9.5.2 bettor_arb.sv

```
module bettor_arb #(parameter MAX_GAME_ID = 1'h1, GAME_ID_WIDTH = 1'h1)
(
  input logic     clk,
  input logic     reset,
  input logic     start,
  input logic     write,
  input logic [31:0]  writedata,
  input logic [7:0] readaddress,

  output logic    done,
  output logic [7:0]  resultcount,
  output logic [31:0] readdata
);

  logic [MAX_GAME_ID:0] flushed;
  logic [MAX_GAME_ID:0] found_result;
  logic [MAX_GAME_ID:0] calc_done;
  logic [MAX_GAME_ID:0] calc_write_selector;
  logic [31:0] result [MAX_GAME_ID:0];
  logic [7:0] num_arb;

  // instance array of cal mans share an array of equal length
  // each bit gets assigned to different instance
  // by bitshifting write by the game id, one bit in the selector array
  // is raised: game id 0 is assigned to calc_man[0], game id 6 is assigned
  // to calc_man[6], etc
  assign calc_write_selector = {{MAX_GAME_ID{1'b0}},write} << writedata[23:20];

  // instance array of calc managers
  calc_manager calc_man[MAX_GAME_ID: 0] (
    .clk(clk), // duplicated
    .reset(reset), // duplicated
    .start(start), // duplicated
    .flushed(flushed), // split across instances
    .write(calc_write_selector), // split across instances
    .writedata(writedata),  // duplicated
    .found_result(found_result), // split across instances
    .done(calc_done), // split across instances
    .result(result) // split across instances
  );

  logic [31:0] result_writedata;
  write_manager #(.MAX_GAME_ID(MAX_GAME_ID), .GAME_ID_WIDTH(GAME_ID_WIDTH)) write_man(
    .clk(clk),
    .reset(reset),
    .result_vec(result),
    .found_result_vec(found_result),
    .flushed_vec(flushed),
    .write(write1),
    .write_counter(num_arb),
    .writedata(result_writedata)
  );

  logic write1;
  /* verilator lint_off UNDRIVEN */
  logic write2;
  logic [31:0] writedata2;
  /* verilator lint_on UNDRIVEN */

  /* verilator lint_off UNUSEDSIGNAL */
  logic [31:0] readdata1;
  /* verilator lint_on UNUSEDSIGNAL */

  assign resultcount = num_arb;

  // Port 1 for writing, Port 2 for reading
```

```
  twoport_memory result_mem(
    .clk(clk),
    .a1(resultcount),
    .a2(readaddress),
    .write1(write1),
    .write2(write2),
    .writedata1(result_writedata),
    .writedata2(writedata2),
    .readdata1(readdata1),
    .readdata2(readdata)
  );

  // Introduce one cycle delay to account for writing time
  always_ff @(posedge clk) begin
    done <= calc_done == {MAX_GAME_ID + 1{1'b1}};
  end

endmodule
```

9.5.3 calc_manager.sv

```
module calc_manager(
  input logic     clk,
  input logic     reset,
  input logic     start,
  input logic     flushed,
  /* verilator lint_off UNUSEDSIGNAL */
  input logic     write,
  input logic [31:0]  writedata,
  /* verilator lint_on UNUSEDSIGNAL */

  output logic    found_result,
  output logic    done,
  /* verilator lint_off UNDRIVEN */
  output logic [31:0] result
  /* verilator lint_on UNDRIVEN */
);

  /* verilator lint_off UNDRIVEN */
  logic [7:0]   e_mem_a1, e_mem_a2;
  logic     e_mem_write1, e_mem_write2;
  logic [31:0]  e_mem_writedata1, e_mem_writedata2;
  /* verilator lint_on UNDRIVEN */

  logic [31:0]  e_mem_readdata1, e_mem_readdata2;

  twoport_memory events_mem(
    .clk(clk),
    .a1(e_mem_a1),
    .write1(e_mem_write1),
    .writedata1(e_mem_writedata1),
    .readdata1(e_mem_readdata1),
    .a2(e_mem_a2),
    .write2(e_mem_write2),
    .writedata2(e_mem_writedata2),
    .readdata2(e_mem_readdata2)
  );


  logic [7:0] even_index;
  /* verilator lint_off UNUSED */
  logic [7:0] odd_index;
  logic   running;
  /* verilator lint_on UNUSED */

  // events iter should probably just take number of odd and even events
  iterator events_iter(
    .clk(clk),
    .reset(reset),
    .start(start),
    .found_result(found_result),
    .flushed(flushed),
    .even_index_end(next_home_event_index), // after writes stop, these will each be 2 past the
    last indices written to
    .odd_index_end(next_away_event_index),
    .even_index(even_index),
    .odd_index(odd_index),
    .running(running),
    .done(done)
  );

  logic [31:0]  arb_event_a, arb_event_b;
  logic [19:0]  arb_prob;
  logic     calc_odds_found_result;

  assign found_result = calc_odds_found_result && running;
  calc_odds arb_odds_calc(
```

```
    .a(arb_event_a),
    .b(arb_event_b),
    .found_result(calc_odds_found_result),
    .arb_prob(arb_prob)
  );

  logic     event_is_away;
  logic [7:0] next_home_event_index;
  logic [7:0] next_away_event_index;

  initial begin
    next_home_event_index = 0;
    next_away_event_index = 1;
  end

  assign event_is_away  = writedata[28];
  assign e_mem_writedata1 = writedata;
  assign e_mem_write1   = write;
  assign e_mem_writedata2 = 0;
  assign e_mem_write2   = 0;
  assign e_mem_a2 = odd_index; // port2 always used for reading from odd index

  /* Populate e_mem_a1 with correct idx to write into */
  /* When writes are finished, port1 is switched to reading port */
  always_comb begin
    if (write) begin
      if (event_is_away) e_mem_a1 = next_away_event_index;
      else e_mem_a1 = next_home_event_index;
    end else begin
      e_mem_a1 = even_index;
    end
  end

  // Update indices used for writing
  // Both blocks together: memory[next_index += 2] = writedata;
  always_ff @(posedge clk) begin
    if (reset) begin
      next_home_event_index <= 0;
      next_away_event_index <= 1;
    end
    if (write) begin
      if (event_is_away) next_away_event_index <= next_away_event_index + 2;
      else next_home_event_index <= next_home_event_index + 2;
    end
  end

  // Populate arb_event_a and arb_event_b with correct events for calc_odds to use
  assign arb_event_a = e_mem_readdata1;
  assign arb_event_b = e_mem_readdata2;

  // combinational logic for populating $result
  // TODO would it be crazy to define a module that takes an arb_event_t
  // and exposes the relevant fields?
  assign result = {
    arb_event_b[27:24], // 4 bits of bookie_id_b
    arb_event_a[27:24], // 4 bits of bookie_id_a
    arb_event_a[23:20], // 4 bits of game_id
    arb_prob // 20 bits of arb_prob
  };


endmodule
```

9.5.4 calc_odds.sv

```
// CSEE 4840 Bettor Sports Arbitrage Calc Odds Module

module calc_odds(
  /* verilator lint_off UNUSED */
  input logic [31:0]  a,
  input logic [31:0]  b,
  /* verilator lint_on UNUSED */
  output logic found_result,
  output logic [19:0] arb_prob
);
  logic [19:0] a20, b20;
  logic [39:0] ab;
  logic [39:0] aplusb;


    assign a20 = a[19:0];
    assign b20 = b[19:0];

  assign ab = a20 * b20;
  assign aplusb = {10'b0, a20 + b20, 10'b0};

  always_comb begin
    if (aplusb < ab) begin
      found_result = 1;
      arb_prob = aplusb[29:10];
    end else begin
      found_result = 0;
      arb_prob = 0;
    end
  end

endmodule
```

9.5.5 iterator.sv

```
// CSEE 4840 Bettor Sports Arbitrage Iterator Module

module iterator(
  input logic clk,
    input logic reset,
    input logic start,
    input logic found_result,
    input logic flushed,
    input logic [7:0] even_index_end,
    input logic [7:0] odd_index_end,

  output logic [7:0] even_index,
    output logic [7:0] odd_index,
    output logic running,
  output logic done
);

    logic wait_one_cycle;
    initial begin
        even_index = 0;
        odd_index = 1;
        running = 0;
        wait_one_cycle = 0;
    end

    assign done = even_index + 2 > even_index_end;

  always_ff@(posedge clk) begin
    if (reset) begin
            running <= 0;
    end

        if (start) begin
            running <= 1;
            even_index <= 0;
      odd_index <= 1;
            wait_one_cycle <= 0;
        end

        wait_one_cycle <= !wait_one_cycle;

    if(running && !reset && !done && !wait_one_cycle) begin
            if (!found_result || flushed) begin
                if(odd_index + 2 >= odd_index_end) begin
                    odd_index <= 1;
                    even_index <= even_index + 2;
                    if (even_index + 2 == even_index_end) running <= 0;
                end else begin
                    odd_index <= odd_index + 2;
                end
            end
    end
    end
endmodule
```

9.5.6 twoport_memory.sv

```
module twoport_memory(
    input logic         clk,
    input logic [7:0]   a1, a2,
    input logic         write1, write2,
    input logic [31:0]  writedata1, writedata2,
    output logic [31:0] readdata1, readdata2
);

    logic [31:0] mem [255:0];

    always_ff @(posedge clk) begin
        if (write1) begin
            mem[a1] <= writedata1;
            readdata1 <= writedata1;
        end else readdata1 <= mem[a1];
    end

    always_ff @(posedge clk) begin
        if (write2) begin
            mem[a2] <= writedata2;
            readdata2 <= writedata2;
        end else readdata2 <= mem[a2];
    end
endmodule
```

9.5.7 write_manager.sv

```
module write_manager #(parameter MAX_GAME_ID = 1'h1, GAME_ID_WIDTH = 1'h1)
(
  input logic clk,
  input logic reset,
  input logic [31:0] result_vec [MAX_GAME_ID:0],
  input logic [MAX_GAME_ID:0] found_result_vec,
  output logic [MAX_GAME_ID:0] flushed_vec,
  output logic write,
  output logic [7:0] write_counter,
  output logic [31:0] writedata
);

  logic [GAME_ID_WIDTH - 1:0] source_index;

  initial begin
    write_counter = 8'h0;
    source_index  = 0;
    flushed_vec   = 0;
  end

  always_ff @(posedge clk) begin
    if (reset) begin
      write_counter <= 0;
      flushed_vec   <= 0;
      source_index  <= 0;
    end

    if (source_index == MAX_GAME_ID) begin
      source_index <= 0;
    end else begin
      source_index <= source_index + 1;
    end

    if (found_result_vec[source_index]) begin
      flushed_vec[source_index] <= 1'h1;
    end else begin
      flushed_vec[source_index] <= 1'h0;
    end

    if (found_result_vec[source_index] && !flushed_vec[source_index]) begin
      write_counter <= write_counter + 1'h1;
    end
  end

  always_comb begin
    if (found_result_vec[source_index]) begin
      writedata = result_vec[source_index];
      write = 1'h1;
    end else begin
      writedata = 32'h0;
      write = 1'h0;
    end
  end
endmodule
```

9.6 calc_arb.sh

```
echo "+++++ PULLING EVENTS FROM WEBSITES ++++++"
python3 scrape/scrape.py

echo "+++++ CALCULATING ARBITRAGE OPPORTUNITIES ++++++"
./src/calc_arb ./scrape/nba_game_odds.csv
```

9.7 simulation

9.7.1 arb_avalon_interface.cpp

```cpp
#include <iostream>
#include "Varb_avalon_interface.h"
#include <verilated.h>
#include <verilated_vcd_c.h>

#include "../include/arb.h"
#include "../include/arb_buf.h"
#include "../include/arb_parsing.h"
#include "../include/fixed_point.h"
#include "../include/avalon_interface.h"

#define CLOCK_CYCLE 20

void print_result(arb_result_t *result) {
  uint32_t masked_arb_prob = result->arb_prob & 0xFFFFF;
    double arb_prob = fixed_to_double(masked_arb_prob);

    printf("Game ID: %d, Bookie A: %d, Bookie B: %d, Arbitrage Probability: %f\n", result->
    game_id, result->bookie_id_a, result->bookie_id_b, arb_prob);
}

static void update_clk(Varb_avalon_interface& arbSim, VerilatedVcdC& tfp,
  int time)
{
  arbSim.clk = (time % 20) >= 10;
  arbSim.eval();
  tfp.dump( time );
}

static void avalon_clear_registers(Varb_avalon_interface& arbSim)
{
  arbSim.reset     = 0;
  arbSim.read      = 0;
  arbSim.write     = 0;
  arbSim.chipselect = 0;
  arbSim.address    = 0;
  arbSim.writedata  = 0;
}

static void avalon_write(Varb_avalon_interface& arbSim, uint32_t data,
  uint16_t addr)
{
  arbSim.reset     = 0;
  arbSim.read      = 0;
  arbSim.write     = 1;
  arbSim.chipselect = 1;
  arbSim.address    = addr;
  arbSim.writedata  = data;
}

static void avalon_request_read(Varb_avalon_interface& arbSim, uint16_t addr)
{
  arbSim.reset     = 0;
  arbSim.read      = 1;
  arbSim.write     = 0;
  arbSim.chipselect = 1;
  arbSim.address    = addr;
  arbSim.writedata  = 0;
}

static uint32_t avalon_finish_read(Varb_avalon_interface& arbSim)
{
  uint32_t res = arbSim.readdata;
  avalon_clear_registers(arbSim);
```

```
  return res;
}

void run_dummy_avalon_inputs(Varb_avalon_interface& arbSim, VerilatedVcdC& tfp)
{
  int time = 0;
  uint32_t res = 0;

  avalon_clear_registers(arbSim);

  for ( ; time < 500; time += 10) {
    // writing an event
    if (time == 110)
      avalon_write(arbSim, 0xffff, ARB_EVENT_WRITE_ADDR); // all dummy values for now
    if (time == 130)
      avalon_clear_registers(arbSim);

    // asserting start
    if (time == 210)
      avalon_write(arbSim, 0, ARB_START_ADDR);
    if (time == 230)
      avalon_clear_registers(arbSim);

    // checking done + count
    if (time == 310)
      avalon_request_read(arbSim, ARB_READ_REGS_ADDR);
    if (time == 330)
      res = avalon_finish_read(arbSim);

    // reading 0th result
    if (time == 410)
      avalon_request_read(arbSim, ARB_RESULT_READ_ADDR(0));
    if (time == 430)
      res = avalon_finish_read(arbSim);

    update_clk(arbSim, tfp, time);
  }
  update_clk(arbSim, tfp, time);
}

void simulate_actual_events(Varb_avalon_interface& arbSim, VerilatedVcdC& tfp)
{

  int time = 0;
  uint32_t writedata =0;
  uint32_t readdata = 0;
  int event_index = 0;
  struct event_buf *events = events_from_file("../scrape/nba_game_odds_template.csv");
  printf("processing %d events\n", events->len);
  arb_event_t *event = NULL;
  int reset_time = 45;
  int clear_time = -1;
  int write_time = 115;
  int next_polling_ts = -1;
  int complete_poll_ts =-1;
  int clear_start_ts = -1;
  int start_read_ts = -1;
  int finish_read_ts = -1;
  int end_ts = -1;
  uint8_t result_count = 0;
  arb_read_regs_t read_regs = {0};
  arb_result_t *results = NULL;
  uint32_t res = 0;
  uint8_t read_index = 0;

  for ( ; time < 800000; time += 5) {
    // resetting hardware
    if (time == reset_time) {
```

```
      avalon_write(arbSim, 0, ARB_RESET_ADDR);
      clear_time = time + CLOCK_CYCLE;
    }

    // writing an event
    if (time == write_time && event_index < events->len) {
      event = &events->events_vec[event_index++];
      writedata = *((uint32_t *)(event));
      avalon_write(arbSim, writedata, ARB_EVENT_WRITE_ADDR);
      clear_time = time + CLOCK_CYCLE;
      write_time += 5 * CLOCK_CYCLE;
    }
    if (time == clear_time) {
      avalon_clear_registers(arbSim);
    }

    // asserting start
    if (time == write_time && event_index == events->len) {
      avalon_write(arbSim, 0, ARB_START_ADDR);
      clear_start_ts = time + CLOCK_CYCLE;
      next_polling_ts = time + 5 * CLOCK_CYCLE;
      clear_time = -1;
    }
    if (time == clear_start_ts) {
      avalon_clear_registers(arbSim);
    }

    // polling for done
    if (time == next_polling_ts) {
      avalon_request_read(arbSim, ARB_READ_REGS_ADDR);
      complete_poll_ts = time + 2 * CLOCK_CYCLE;
    }
    if (time == complete_poll_ts) {
      readdata = avalon_finish_read(arbSim);
      read_regs = *((arb_read_regs_t *) &readdata);
      result_count = read_regs.result_count;
      if (read_regs.done) {
        printf("Done! result count: %d\n", result_count);
        start_read_ts = time + 5 * CLOCK_CYCLE;
        results = (arb_result_t *) malloc(result_count * sizeof(*results));
      } else {
        next_polling_ts += 5 * CLOCK_CYCLE;
      }
    }

    // reading events
    if (time == start_read_ts) {
      avalon_request_read(arbSim, ARB_RESULT_READ_ADDR(read_index));
      finish_read_ts = time + 2 * CLOCK_CYCLE;
    }
    if (time == finish_read_ts) {
      res = avalon_finish_read(arbSim);
      results[read_index] = *((arb_result_t *)&res);
      read_index++;
      if (read_index < result_count)
        start_read_ts = time + 5 * CLOCK_CYCLE;
      else
        end_ts = time + 5 * CLOCK_CYCLE;
    }

    if (time == end_ts) {
      break;
    }

    update_clk(arbSim, tfp, time);
  }
  update_clk(arbSim, tfp, time);

```

```
  for (int i = 0; i < result_count; i++) {
    printf("%d: ", i);
    print_result(&results[i]);
  }

  free(events);
  free(results);
}

int main(int argc, const char **argv, const char **env) {
  Verilated::commandArgs(argc, argv);
  Varb_avalon_interface arbSim;
  Verilated::traceEverOn(true);
  VerilatedVcdC tfp;
  arbSim.trace(&tfp, 99);
  tfp.open("arb_avalon_interface.vcd");

  // run_dummy_avalon_inputs(arbSim, tfp);
  simulate_actual_events(arbSim, tfp);


  tfp.close();
  arbSim.final();

  return 0;
}
```

9.7.2 calc_manager.cpp

```
#include <iostream>
#include "Vcalc_manager.h"
#include <verilated.h>
#include <verilated_vcd_c.h>

#include "../include/arb.h"
#include "../include/arb_buf.h"
#include "../include/arb_parsing.h"

#define MAX_ENTRIES 256
#define CLOCK_CYCLE 20

void print_result(arb_result_t *result) {
  uint32_t masked_arb_prob = result->arb_prob & 0xFFFFF;
    double arb_prob = fixed_to_double(masked_arb_prob);

    printf("Game ID: %d, Bookie A: %d, Bookie B: %d, Arbitrage Probability: %f\n", result->
    game_id, result->bookie_id_a, result->bookie_id_b, arb_prob);
}

void print_event(arb_event_t *event) {
  printf("odds: %d, bookie_id: %d, game_id: %d, outcome: %d\n", event->odds, event->bookie_id,
    event->game_id, event->outcome);
}

static void update_clk(Vcalc_manager& calc_man_sim, VerilatedVcdC& tfp,
  int time)
{
  calc_man_sim.clk = (time % 20) >= 10;
  calc_man_sim.eval();
  tfp.dump( time );
}

static void clear_all_regs(Vcalc_manager& calc_man_sim)
{
  calc_man_sim.reset    = 0;
  calc_man_sim.start    = 0;
  calc_man_sim.flushed  = 0;
  calc_man_sim.write    = 0;
  calc_man_sim.writedata  = 0;
}

static void write_one_event(Vcalc_manager& calc_man_sim, arb_event_t *event)
{
  calc_man_sim.write    = 1;
  calc_man_sim.writedata  = *((uint32_t *) event);
}

static void clear_event_writing_regs(Vcalc_manager& calc_man_sim)
{
  calc_man_sim.write    = 0;
  calc_man_sim.writedata  = 0;
}

static void raise_reset(Vcalc_manager& calc_man_sim)
{
  calc_man_sim.reset = 1;
}

static void lower_reset(Vcalc_manager& calc_man_sim)
{
  calc_man_sim.reset = 0;
}

static void raise_start(Vcalc_manager& calc_man_sim)
{
```

```
  calc_man_sim.start = 1;
}

static void lower_start(Vcalc_manager& calc_man_sim)
{
  calc_man_sim.start = 0;
}

static void raise_flushed(Vcalc_manager& calc_man_sim)
{
  calc_man_sim.flushed = 1;
}

static void lower_flushed(Vcalc_manager& calc_man_sim)
{
  calc_man_sim.flushed = 0;
}

static int read_found_result(Vcalc_manager& calc_man_sim)
{
  return calc_man_sim.found_result;
}

static uint32_t read_result(Vcalc_manager& calc_man_sim)
{
  return calc_man_sim.result;
}

void simulate_input_events(Vcalc_manager& calc_man_sim, VerilatedVcdC& tfp)
{
  int time = 0;
  int write_index = 0, read_index = 0;
  int write_ts = 40, write_clear_ts = 60;
  int compare_ts = -1;
  int found_result;
  arb_result_t *result;
  arb_event_t *write_event, *read_event;
  struct event_buf *events = events_from_file("../scrape/example_with_arbitrage.csv");
  // struct arb_event_buf *events = events_from_file("../scrape/example_without_arbitrage.csv");
  uint8_t event_count = std::min(MAX_ENTRIES, events->len);
  printf("event_count: %d\n", event_count);
  int reset_time = -1;
  int raise_flush_ts = -1, lower_flush_ts = -1;
  int waiting_for_flush = false;

  for ( ; time < 300000; time += 10) {

    if (time > 460 && calc_man_sim.done) {
      printf("DONE!\n");
      break;
    }
    // (1) Write Phase
    // (a) write next event to memory
    if (time == write_ts && write_index < event_count) {
      write_event = &events->events_vec[write_index];
      write_one_event(calc_man_sim, write_event);
      write_ts += 100;
      write_index++;
    }

    // (b) clear registers
    if (time == write_clear_ts) {
      clear_event_writing_regs(calc_man_sim);
      write_clear_ts += 100;
    }

    // Once all events written, raise start
    if (time == write_ts && write_index >= event_count) {
```

```
      printf("<<< all events written >>>\n");
      raise_start(calc_man_sim);
      reset_time = time + CLOCK_CYCLE;
      compare_ts = time + 2 * CLOCK_CYCLE;
    }

    if (time == reset_time) {
      lower_start(calc_man_sim);
    }

    // (2) Comparison phase
    if (time == compare_ts && !waiting_for_flush) {
      found_result = read_found_result(calc_man_sim);
      // uint32_t event_a = read_event_a(calc_man_sim);
      // uint32_t event_b = read_event_b(calc_man_sim);

      // printf("eventA: ");
      // print_event((arb_event_t *) &event_a);
      // printf("eventB: ");
      // print_event((arb_event_t *) &event_b);

      if (found_result) {
        printf("found result!\n");
        uint32_t res = read_result(calc_man_sim);
        raise_flush_ts = time + CLOCK_CYCLE;
        lower_flush_ts = time + 2 * CLOCK_CYCLE;
        result = (arb_result_t *) &res;
        print_result(result);
        waiting_for_flush = true;
      } else {
        compare_ts += CLOCK_CYCLE;
      }
    }

    if (time == raise_flush_ts) {
      raise_flushed(calc_man_sim);
    }

    if (time == lower_flush_ts) {
      lower_flushed(calc_man_sim);
      waiting_for_flush = false;
      compare_ts = time + CLOCK_CYCLE;
    }

    update_clk(calc_man_sim, tfp, time);
  }
  update_clk(calc_man_sim, tfp, time);

  free(events);
}

int main(int argc, const char **argv, const char **env) {
  Verilated::commandArgs(argc, argv);
  Vcalc_manager calc_man_sim;
  Verilated::traceEverOn(true);
  VerilatedVcdC tfp;
  calc_man_sim.trace(&tfp, 99);
  tfp.open("calc_manager.vcd");

  simulate_input_events(calc_man_sim, tfp);

  tfp.close();
  calc_man_sim.final();

  return 0;
}
```

9.7.3 calc_odds.cpp

```
#include <iostream>
#include "Vcalc_odds.h"
#include <verilated.h>
#include <verilated_vcd_c.h>
#include <cassert>

#include "../include/arb.h"
#include "../include/fixed_point.h"


int main(int argc, const char ** argv, const char ** env) {
  Verilated::commandArgs(argc, argv);

  Vcalc_odds calc_oddsSim;

  assert(sizeof(arb_event_t) * 8 == 32);

  arb_event_t events[6] = {{0}};
  std::string str_odds[6];

  // 50 50
  events[0].odds = double_to_fixed(2.0);
  events[0].outcome = 0;
  events[0].bookie_id = 0;
  str_odds[0] = "50";

  events[1].odds = double_to_fixed(2.0);
  events[1].outcome = 1;
  events[1].bookie_id = 0;
  str_odds[1] = "50";

  // 75 25
  events[2].odds = double_to_fixed(1.33);
  events[2].outcome = 0;
  events[2].bookie_id = 1;
  str_odds[2] = "75";

  events[3].odds = double_to_fixed(4.0);
  events[3].outcome = 1;
  events[3].bookie_id = 1;
  str_odds[3] = "25";

  // 60 40
  events[4].odds = double_to_fixed(1.67);
  events[4].outcome = 0;
  events[4].bookie_id = 2;
  str_odds[4] = "60";

  events[5].odds = double_to_fixed(2.5);
  events[5].outcome = 1;
  events[5].bookie_id = 2;
  str_odds[5] = "40";

  size_t event_count = sizeof(events)/sizeof(events[0]);
  for (int i = 0; i < event_count; i += 2) {
    for (int j = 1; j < event_count; j += 2) {
      if (events[i].outcome == events[j].outcome)
        continue;
      if (events[i].bookie_id == events[j].bookie_id)
        continue;

      std::cout << "Testing " << str_odds[i] << " and " << str_odds[j] << "\n";
      double odds_a = fixed_to_double(events[i].odds);
      double odds_b = fixed_to_double(events[j].odds);

      bool swIsOpp = odds_a + odds_b < odds_a * odds_b;
```

```
      auto swArbProb = swIsOpp ? odds_a + odds_b : 0;

      std::cout << "Software Results:\n";
      std::cout << "\tarbitrage: " << (swIsOpp ? "yes" : "no") << "\n";
      std::cout << "\tarbitrage odds: " << swArbProb << "\n";

      calc_oddsSim.a = events[i].odds;
      calc_oddsSim.b = events[j].odds;
      calc_oddsSim.eval();

            bool hwIsArb = calc_oddsSim.found_result;

      // Splice out first 12 bits
      uint32_t maskedArbProb = calc_oddsSim.arb_prob & 0xFFFFF;
      double hwArbProb = fixed_to_double(maskedArbProb);

      // auto hwArbProb = fixed_to_double(calc_oddsSim.arb_prob);

      std::cout << "Hardware Results:\n";
      std::cout << "\tarbitrage: " << (hwIsArb ? "yes" : "no") << "\n";
      std::cout << "\tarbitrage odds: " << hwArbProb << "\n";
      std::cout << "\n";

      assert(hwIsArb == swIsOpp);
    }
  }

  std::cout << "SUCCESS\n";
  calc_oddsSim.final();
}
```

9.7.4 iterator.cpp

```
#include <iostream>
#include "Viterator.h"
#include <verilated.h>
#include <verilated_vcd_c.h>
#include <cassert>

static void update_clk(Viterator& iteratorSim, VerilatedVcdC& tfp, int time)
{
  iteratorSim.clk = (time % 20) >= 10;
  iteratorSim.eval();
  tfp.dump( time );
}


static void iterator_clear_registers(Viterator& iteratorSim, VerilatedVcdC& tfp) {
    iteratorSim.reset = 0;
    iteratorSim.found_result = 0;
    iteratorSim.flushed = 0;
    iteratorSim.start = 0;
}

static void iterator_reset(Viterator& iteratorSim, VerilatedVcdC& tfp,
    uint8_t even_index_end, uint8_t odd_index_end) {
    iteratorSim.reset = 1;
    iteratorSim.found_result = 0;
    iteratorSim.flushed = 0;
    iteratorSim.even_index_end = even_index_end;
    iteratorSim.odd_index_end = odd_index_end;
}

static void iterator_assert_start(Viterator& iteratorSim, VerilatedVcdC& tfp)
{
    iteratorSim.start = 1;
}

static void iterator_found_result_high(Viterator& iteratorSim, VerilatedVcdC& tfp)
{
    iteratorSim.found_result = 1;
}

static void iterator_found_result_low(Viterator& iteratorSim, VerilatedVcdC& tfp)
{
    iteratorSim.found_result = 0;
}

static void iterator_flushed_high(Viterator& iteratorSim, VerilatedVcdC& tfp)
{
    iteratorSim.flushed = 1;
}

static void iterator_flushed_low(Viterator& iteratorSim, VerilatedVcdC& tfp)
{
    iteratorSim.flushed = 0;
}

static uint8_t iterator_read_even(Viterator& iteratorSim, VerilatedVcdC& tfp) {
    uint8_t res = iteratorSim.even_index;
    return res;
}

static uint8_t iterator_read_dd(Viterator& iteratorSim, VerilatedVcdC& tfp) {
    uint8_t res = iteratorSim.odd_index;
    return res;
}

void simulate_iterator(Viterator& iteratorSim, VerilatedVcdC& tfp)
```

```
{
    int time = 0;
    uint8_t even;
    uint8_t odd;

    for ( ; time < 300000; time += 10) {
        /* Pulse reset signal and set num_events */
        if (time == 100)
            iterator_reset(iteratorSim, tfp, 16, 13);
        if (time == 120)
            iterator_clear_registers(iteratorSim, tfp);

        /* Pulse start signal */
        if (time == 140)
            iterator_assert_start(iteratorSim, tfp);
        if (time == 160)
            iterator_clear_registers(iteratorSim, tfp);

        /* Read i and j */
        if (time == 240) {
            even = iterator_read_even(iteratorSim, tfp);
            odd = iterator_read_dd(iteratorSim, tfp);
            printf("-- TIME 240 --:");
            printf("even = %u, odd = %u\n", even, odd);
        }

        if (time == 260) {
            iterator_found_result_high(iteratorSim, tfp);
            even = iterator_read_even(iteratorSim, tfp);
            odd = iterator_read_dd(iteratorSim, tfp);
            printf("-- TIME 260 --:");
            printf("even: %u, odd: %u\n", even, odd);
        }

        if (time == 280) {
            iterator_flushed_high(iteratorSim, tfp);
            even = iterator_read_even(iteratorSim, tfp);
            odd = iterator_read_dd(iteratorSim, tfp);
            printf("-- TIME 280 --:");
            printf("even: %u, odd: %u\n", even, odd);
        }

        if (time == 300) {
            iterator_found_result_low(iteratorSim, tfp);
            even = iterator_read_even(iteratorSim, tfp);
            odd = iterator_read_dd(iteratorSim, tfp);
            printf("-- TIME 300 --:");
            printf("even: %u, odd: %u\n", even, odd);
        }

        if (time == 320) {
            iterator_flushed_low(iteratorSim, tfp);
            even = iterator_read_even(iteratorSim, tfp);
            odd = iterator_read_dd(iteratorSim, tfp);
            printf("-- TIME 320 --:");
            printf("even: %u, odd: %u\n", even, odd);
        }

        if (time == 340) {
            even = iterator_read_even(iteratorSim, tfp);
            odd = iterator_read_dd(iteratorSim, tfp);
            printf("-- TIME 340 --:");
            printf("even: %u, odd: %u\n", even, odd);
        }

        update_clk(iteratorSim, tfp, time);
    }

```

```
    update_clk(iteratorSim, tfp, time);
}

int main(int argc, const char ** argv, const char ** env) {
    Verilated::commandArgs(argc, argv);
    Viterator iteratorSim;
    Verilated::traceEverOn(true);
    VerilatedVcdC tfp;
    iteratorSim.trace(&tfp, 99);
    tfp.open("iterator.vcd");

    simulate_iterator(iteratorSim, tfp);

    tfp.close();
    iteratorSim.final();

    return 0;
}
```

9.7.5 twoport_memory.cpp

```
#include <iostream>
#include "Vtwoport_memory.h"
#include <verilated.h>
#include <verilated_vcd_c.h>

#include "../include/arb.h"
#include "../include/arb_buf.h"
#include "../include/arb_parsing.h"

#define MAX_ENTRIES 256

static void update_clk(Vtwoport_memory& mem_sim, VerilatedVcdC& tfp,
  int time)
{
  mem_sim.clk = (time % 20) >= 10;
  mem_sim.eval();
  tfp.dump( time );
}

static void clear_registers_for_port(Vtwoport_memory& mem_sim,
  VerilatedVcdC& tfp, int port=1)
{
  if (port == 1) {
    mem_sim.write1    = 0;
    mem_sim.writedata1  = 0;
  } else {
    mem_sim.write2    = 0;
    mem_sim.writedata2  = 0;
  }
}

static void mem_write(Vtwoport_memory& mem_sim, VerilatedVcdC& tfp,
  uint32_t data, uint16_t addr, int port=1)
{
  if (port == 1) {
    mem_sim.a1      = addr;
    mem_sim.write1    = 1;
    mem_sim.writedata1  = data;
  } else {
    mem_sim.a2      = addr;
    mem_sim.write2    = 1;
    mem_sim.writedata2  = data;
  }
}

static void mem_request_read(Vtwoport_memory& mem_sim, VerilatedVcdC& tfp,
  uint16_t addr, int port=1)
{
  if (port == 1) {
    mem_sim.a1      = addr;
    mem_sim.write1    = 0;
    mem_sim.writedata1  = 0;
  } else {
    mem_sim.a2      = addr;
    mem_sim.write2    = 0;
    mem_sim.writedata2  = 0;
  }
}

static uint32_t mem_finish_read(Vtwoport_memory& mem_sim, VerilatedVcdC& tfp,
  int port=1)
{
  uint32_t res;
  if (port == 1) {
    res = mem_sim.readdata1;
  } else {
```

```
    res = mem_sim.readdata2;
  }
  clear_registers_for_port(mem_sim, tfp, port);
  return res;
}

void run_dummy_reads_and_writes(Vtwoport_memory& mem_sim, VerilatedVcdC& tfp)
{
  int time = 0;
  uint32_t res1 = 0, res2 = 0;

  clear_registers_for_port(mem_sim, tfp, 1);
  clear_registers_for_port(mem_sim, tfp, 2);

  for ( ; time < 500; time += 10) {
    // simultaneous writes
    if (time == 120) {
      mem_write(mem_sim, tfp, 0xffff, 1, 1);
      mem_write(mem_sim, tfp, 0xf0f0, 0, 2);
    }
    if (time == 140) {
      clear_registers_for_port(mem_sim, tfp, 1);
      clear_registers_for_port(mem_sim, tfp, 2);
    }

    // simultaneous reads
    if (time == 220) {
      mem_request_read(mem_sim, tfp, 0, 1);
      mem_request_read(mem_sim, tfp, 1, 2);
    }
    if (time == 240) {
      res1 = mem_finish_read(mem_sim, tfp, 1);
      res2 = mem_finish_read(mem_sim, tfp, 2);
    }

    update_clk(mem_sim, tfp, time);
  }
  update_clk(mem_sim, tfp, time);
}

void simulate_actual_reads_and_writes(Vtwoport_memory& mem_sim, VerilatedVcdC& tfp)
{
  int time = 0;
  uint32_t writedata, readdata = 0;
  int write_index = 0, read_index = 0;
  int write_ts = 40, write_clear_ts = 60;
  int read_ts = 140, read_finish_ts = 160;
  arb_event_t *write_event, *read_event;
  struct event_buf *events = events_from_file("../scrape/nba_game_odds_template.csv");
  uint8_t event_count = std::min(MAX_ENTRIES, events->len);

  for ( ; time < 300000 && read_index < event_count; time += 10) {
    // writing an event
    if (time == write_ts && write_index < event_count) {
      write_event = &events->events_vec[write_index];
      writedata = *((uint32_t *)(write_event));
      mem_write(mem_sim, tfp, writedata, write_index); // default to port 1
      write_ts += 100;
      write_index++;
    }
    if (time == write_clear_ts) {
      clear_registers_for_port(mem_sim, tfp);
      write_clear_ts += 100;
    }

    // read from port 2 while we keep writing to port 1
    if (time == read_ts && read_index < event_count) {
      mem_request_read(mem_sim, tfp, read_index, 2);
```

```
      read_ts += 100;
    }

    // check that what we read matches what we wrote in
    if (time == read_finish_ts) {
      readdata = mem_finish_read(mem_sim, tfp, 2);
      read_event = &events->events_vec[read_index];
      if(memcmp(&readdata, read_event, sizeof(readdata))) {
        std::cerr << "FAILURE\n";
        std::cerr << "\tEvent read at " << read_index << " doesn't match ";
        std::cerr << "event written from " << read_index << "\n";
        std::cerr << "\treaddata: " << readdata << "\n";
        std::cerr << "\toriginal: " << *((uint32_t *)read_event) << "\n";
        exit(EXIT_FAILURE);
      }
      read_index++;
      read_finish_ts += 100;
    }
    update_clk(mem_sim, tfp, time);
  }
  update_clk(mem_sim, tfp, time);

  free(events);
  std::cout << "SUCCESS!\n";
}

int main(int argc, const char **argv, const char **env) {
  Verilated::commandArgs(argc, argv);
  Vtwoport_memory mem_sim;
  Verilated::traceEverOn(true);
  VerilatedVcdC tfp;
  mem_sim.trace(&tfp, 99);
  tfp.open("twoport_memory.vcd");

  // run_dummy_reads_and_writes(mem_sim, tfp);
  simulate_actual_reads_and_writes(mem_sim, tfp);

  tfp.close();
  mem_sim.final();

  return 0;
}
```

9.7.6 write_manager.cpp

```
#include <iostream>
#include "Vwrite_manager.h"
#include <verilated.h>
#include <verilated_vcd_c.h>
#include <stdio.h>

#include "../include/arb.h"
#include "../include/arb_buf.h"
#include "../include/arb_parsing.h"


static void update_clk(Vwrite_manager& writeSim, VerilatedVcdC& tfp,
  int time)
{
  writeSim.clk = (time % 20) >= 10;
  writeSim.eval();
  tfp.dump( time );
}


void print_event(arb_event_t *event) {
  printf("odds: %d, bookie_id: %d, game_id: %d, outcome: %d\n", event->odds, event->bookie_id,
    event->game_id, event->outcome);
}


static void send_result(Vwrite_manager & writeSim, uint32_t data)
{
  uint8_t found_result_vec = 1;
  memcpy(&writeSim.found_result_vec, &found_result_vec,
    sizeof(writeSim.found_result_vec));
  memcpy(&writeSim.result_vec, &data, sizeof(writeSim.result_vec));
}


static void reset_manager(Vwrite_manager& writeSim)
{
  writeSim.reset = 1;
  memset(&writeSim.found_result_vec, 0, sizeof(writeSim.found_result_vec));
  memset(&writeSim.result_vec, 0, sizeof(writeSim.result_vec));
}


static void clear_registers(Vwrite_manager& writeSim)
{
  writeSim.reset = 0;
  memset(&writeSim.found_result_vec, 0, sizeof(writeSim.found_result_vec));
  memset(&writeSim.result_vec, 0, sizeof(writeSim.result_vec));
}

static uint32_t get_writedata(Vwrite_manager& writeSim)
{
  uint32_t res = writeSim.writedata;
  return res;
}


static void lower_foundresult(Vwrite_manager& writeSim)
{
  writeSim.found_result_vec = 0;
}

static int get_writecounter(Vwrite_manager& writeSim)
{
  return writeSim.write_counter;
}
```

```


void simulate_writes(Vwrite_manager& writeSim, VerilatedVcdC& tfp)
{
  int time = 0;
  uint32_t result;

  uint32_t writedata, readdata = 0;
  int event_index = 0;
  struct event_buf *events = events_from_file("../scrape/nba_game_odds_template.csv");
  arb_event_t *event;
  arb_event_t *write_event;
  int write_time = 120;
  int flushed_time = 140;
  int clear_time = 140;
  char wrote_start = 0;

  for ( ; time < 300000; time += 10) {
    if (time == 50)
      reset_manager(writeSim);
    else if (time == 70)
      clear_registers(writeSim);

    // writing an event
    if (time == write_time && event_index < events->len) {
      printf("Sending Event %d:\n", event_index);

      event = &events->events_vec[event_index++];
      result = *((uint32_t *)(event));
      send_result(writeSim, result);
      write_time += 100;

      print_event(event);
    } else if (time == flushed_time) {
      uint32_t res = get_writedata(writeSim);
      write_event = (arb_event_t *) &res;
      flushed_time += 100;

      printf("Written Event: \n");
      print_event(write_event);
    } else if (time + 60 == write_time) {
      lower_foundresult(writeSim);
    }

    if (time == write_time && event_index == events->len)
      break;

    update_clk(writeSim, tfp, time);
  }
  update_clk(writeSim, tfp, time);

  printf("Number of events written: %d\n", get_writecounter(writeSim));
}

int main(int argc, const char **argv, const char **env) {
  Verilated::commandArgs(argc, argv);
  Vwrite_manager writeSim;
  Verilated::traceEverOn(true);
  VerilatedVcdC tfp;
  writeSim.trace(&tfp, 99);
  tfp.open("write_manager.vcd");

  simulate_writes(writeSim, tfp);

  tfp.close();
  writeSim.final();

  return 0;
```

```
}
```

9.8 python_sim

9.8.1 python_alg.py

```
import time
import csv
import sys

class arb_event:
    def __init__(self, odds, game_id, bookie_id, outcome):
        self.odds = odds
        self.game_id = game_id
        self.bookie_id = bookie_id
        self.outcome = outcome

    def print_arb_event(self):
        print("Outcome: %c" % (self.outcome))
        print("Game ID: %d" % self.game_id)
        print("Bookie ID: %d" % self.bookie_id)
        print("Odds: %f" % self.odds)

class arb_result:
    def __init__(self, arb_prob, game_id, bookie_id_a, bookie_id_b):
        self.arb_prob = arb_prob
        self.game_id = game_id
        self.bookie_id_a = bookie_id_a
        self.bookie_id_b = bookie_id_b

    def print_arb_event(self):
        print("Game ID: %d" % self.game_id)
        print("Bookie ID a: %d" % self.bookie_id_a)
        print("Bookie ID b: %d" % self.bookie_id_b)
        print("Odds: %f" % self.arb_prob)

class event_buf:
    def __init__(self, buf):
        self.events_vec = buf

    def append_event(self, odds, game_id, bookie_id, outcome):
        self.events_vec.append(arb_event(odds, game_id, bookie_id, outcome))

    def print_arb_events(self):
        for event in self.events_vec:
            event.print_arb_event()

class result_buf:
    def __init__(self, buf):
        self.arbs_vec = buf

    def append_result(self, arb_prob, game_id, bookie_id_a, bookie_id_b):
        self.arbs_vec.append(arb_result(arb_prob, game_id, bookie_id_a, bookie_id_b))

    def print_arb_results(self):
        print("Arbitrage Opportunities:\n")
        for result in self.arbs_vec:
            print("Game ID: " + str(result.game_id) + ", Bookie Home: " + str(result.bookie_id_a)
     + ", Bookie Away: " + str(result.bookie_id_b) + ", Arbitrage Probability: " + str(result.
    arb_prob))

def calc_arb(events_a: event_buf, events_b: event_buf) -> result_buf:
    results = result_buf([])
    for event_a in events_a.events_vec:
        for event_b in events_b.events_vec:
            if(event_a.game_id != event_b.game_id):
                continue

            if((1 / event_a.odds) + (1 / event_b.odds) < 1):
                results.append_result(event_a.odds + event_b.odds, event_a.game_id, event_a.
```

```python
    bookie_id, event_b.bookie_id)

    return results


def read_csv(file_path):
    events_a = event_buf([])
    events_b = event_buf([])

    with open(file_path, mode='r') as csvfile:
        reader = csv.DictReader(csvfile)
        for row in reader:
            game_id = int(row['Game ID'])
            bookie_id = int(row['Bookmaker ID'])
            odds = float(row['Outcome Price'])
            home_team = row['Home Team']
            away_team = row['Away Team']
            outcome_name = row['Outcome Name']

            if outcome_name == home_team:
                outcome = 'a'
                events_a.append_event(odds, game_id, bookie_id, outcome)
            elif outcome_name == away_team:
                outcome = 'b'
                events_b.append_event(odds, game_id, bookie_id, outcome)

    return events_a, events_b


def main(argv):
    filename = 'scrape/nba_game_odds_template.csv'
    if len(argv) == 2:
        filename = argv[1]
    # start_time = time.time()

    # populating events of head to head matchup

    # a = home, b = away
    # events_a, events_b = read_csv('scrape/example_with_arbitrage.csv')
    events_a, events_b = read_csv(filename)

    # print("Outcome a events:")
    # events_a.print_arb_events()
    # print("Outcome b events:")
    # events_b.print_arb_events()

    results = calc_arb(events_a, events_b)

    results.print_arb_results()

    # end_time = time.time()
    # elapsed_time = end_time - start_time
    # print(f"Elapsed time: {elapsed_time:.5f} seconds")

if __name__ == "__main__":
    main(sys.argv)
```

9.8.2 full_python_sim.py

```
import time
import subprocess
from tqdm import tqdm


while True:
    start_time = time.time()


    subprocess.run(["python", "scrape/scrape.py"])
    subprocess.run(["python", "python_sim/python_alg.py"])

    end_time = time.time()
    iteration_duration = end_time - start_time


    print(f"Iteration took {iteration_duration:.5f} seconds.")


    for _ in tqdm(range(30), desc="Wait", unit="s"):
        time.sleep(1)
```