Sports Arbitrage Detection Design Document

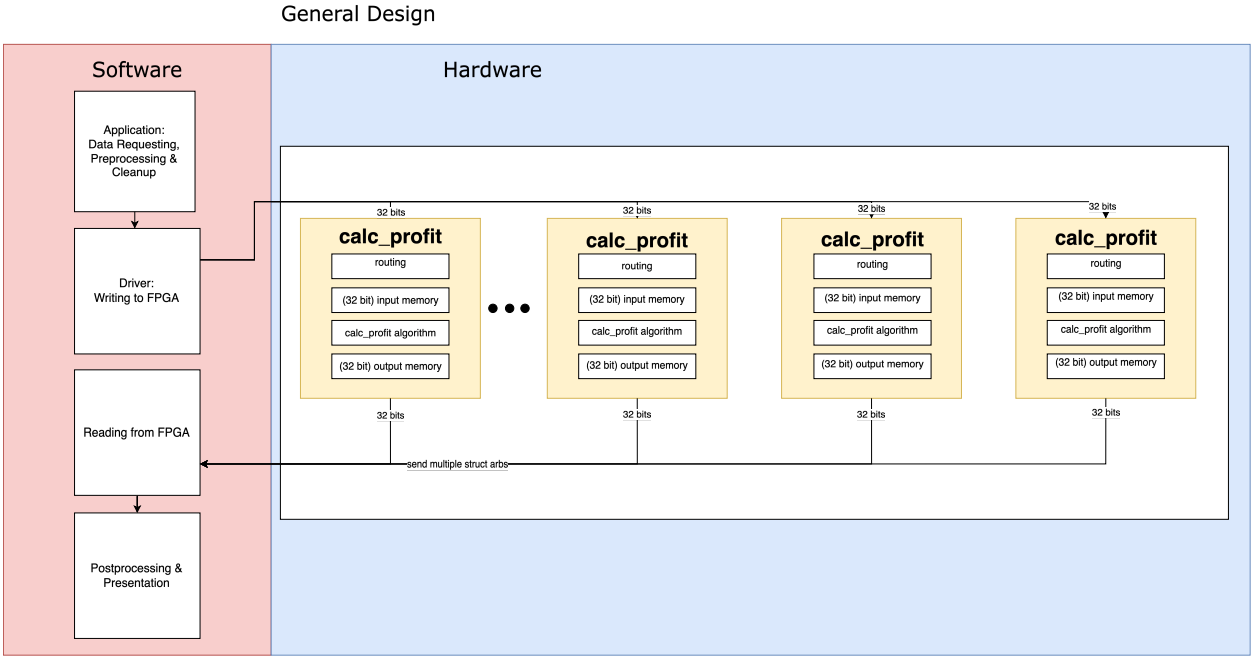
Shivan Mukherjee (sm5155), Shreya Somayajula (svs2137),
Jonathan Nalikka (jmn2193), Brennan McManus (bm2530), Chelsea Soemitro (crs2221)

Spring 2024

1 Introduction 🏀

Welcome to the Sports Betting Arbitrage Detector! 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. Our group aims to create a arbitrage detector accelerated by custom hardware on the FPGA to detect these opportunities and calculate the per-dollar profit from each.

2 System Block Diagram 🤖



The general flow is as follows – in software, we query an API to retrieve data regarding games and all bets associated with the outcomes of those games for a certain moment in time. We handle the pre-processing

and clean-up of data in software (more information in Section 3). Using a device driver, we write the data to the FPGA. In parallel, each module will perform the calculation on a given combination of odds. The software will finally send a request to the hardware to read the outputs of a particular combination.

3 Pre-Processing in Software

3.1 Data Acquisition

To access sportsbook data, we query an odds API in Python. The website prop-odds provides historical data and real-time odds data updated at 60-second intervals in JSON format. As our input to the FPGA, we will provide the games, the props associated with the games, the bookies that have listed those props, and finally the odds associated with each prop. A sample of the data is shown below.

Bookie Key	**Market Key**	**Timestamp**	**Handicap**	**Odds**
fanduel	spread	2024-03-28 22:34:19	-3.5	-110
fanduel	spread	2024-03-28 22:41:02	-1.5	-110
fanduel	spread	2024-03-28 23:13:19	-1.0	-108

Table 1: Oklahoma City Thunder Spread Betting Outcomes

3.2 Pre-Processing

For any event, defined as a head to head matchup based on an outcome, we query our API for the odds data. The data is then processed as follows:

All odds are converted from American odds to an implied probability to normalize the range of values. American odds use a baseline value of $100. For favorites you risk the money to win $100 while for underdogs, you risk $100 to win the amount. The algorithm for the conversion is described in the Algorithms section.

For a given event, we extract the *bookie_id*, the *game_id*, and odds for both outcomes of the prop (*prob_a* and *prob_b*). These *prob_a* and *prob_b* values, after processing, will be the implied probability values divided by 100.

3.3 Communication to Hardware

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 `event` 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 {
    uint64_t      game_id:    5;
    uint64_t      bookee_id:  3;
    uint64_t      prob_a:    13;
    uint64_t      prob_b:    13;
} event;
```

From there, we store all of the `event` structs in a `struct event_buf`:

```
struct event_buf {
    int len;
    event events[];
};
```

To write to hardware, we perform a single `iowrite` to write the `len` variable. Then, we loop through the `events` array, writing each `event` to hardware.

The interface for the hardware device is:

```
input logic write, chipselect, clk, reset,
input logic start,
input logic waddr,
input logic writedata[31:0],
input logic raddr[18:0],
output logic readdata[31:0],
output logic done
```

`writedata` and `readdata` facilitates the communication between hardware and software. We set `start` to 1 when all of the entries have been written from software to hardware, and `done` to 1 when all of the modules have completed their calculations.

The interface for each of the individual `calc_profit` modules are as follows:

```
input logic write, clk, reset,
input logic start,
input logic waddr,
input logic writedata[31:0],
input logic raddr[18:0],
output logic readdata[31:0],
output logic done
```

When the entry with the corresponding bookie has been written to `writedata` (which is determined through the routing algorithm in 4.2.1), the module will save the corresponding value. Once `start` has pulsed and the corresponding combination is requested to be read through `raddr`, the module writes to the output memory. When the software reads from the hardware, it is parsed into the following C struct:

```
typedef struct {
    uint64_t      game_id:     5;
    uint64_t      bookee_id_a: 3;
    uint64_t      bookee_id_b: 3;
    uint64_t      arb_prob:   21;
} arb;
```

4 Algorithms 💻

4.1 Software

4.1.1 Conversion to Fixed-Point

The values that will be operated on in hardware are the odds for each outcome given by the betting websites. Given the complexities of floating point operations in hardware, we will convert the floating point values to fixed-point values for faster operations.

First, the odds will be converted from American Odds, which has no upper bound on its value, to an equivalent implied probability. The implied probability is within the range $[0, 100]$. Then, this probability will be converted from a floating point to fixed point. To ensure we have sufficient capacity for the odds while maintaining precision, each odd will be represented as an 13-bit fixed point value, in which the first singular bits is used for the whole number and the latter 12 for the decimal:

This conversion step will be implemented in software. The pseudocode for the algorithm is given below.

Algorithm 1 convertToFixed(odds)

1: BITS $\leftarrow 3$
2: prob $\leftarrow 100/(odds + 100) * 100$
3: **return** $(fixedpoint)(round(prob * (1 << BITS)))$

4.2 Hardware

4.2.1 Pairwise Odds Combinations

Each profit calculation module in the FPGA calculates arbitrage on pair of odds from different bet provider sources (bookies). In order to ensure that each module performs a unique calculation from the set of bookie combinations, and that each are able to do so in parallel without contending the same shared memory, each module is responsible for storing its own inputs when the relevant information is written from software. In order do to this, they each contain routing logic. This routing logic examines the writedata, and use the game id and bookie ids to determine if the write in question contains information that the module should store for its pairwise calculation.

Each module has their own index and each bookie is assigned their own id in the range [0,N) for N bookies. For a single game and single bet with odds from multiple bookies, the formula is the following: Due

Algorithm 2 routeinputs

1: **if** $writedata.bookieId = index/N$ **then**
2: $\quad chanceOfA \leftarrow writedata.chanceOfA$
3: **end if**
4:
5: **if** $writedata.bookieId = index\%N$ **then**
6: $\quad chanceOfB \leftarrow writedata.chanceOfB$
7: **end if**
8:

to the cost of modulo and division for arbitrary denominators, we will round the numbers (and associated number of profit calculation modules) up to the nearest power of 2. This trades off some wasted memory to ensure that the routing remains fast.

4.2.2 Identification of Arbitrage Opportunities

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 website a, and b is the probability of outcome 2 from website b. If this inequality is true, then there is an arbitrage opportunity for this given combination of odds. This algorithm will be implemented in hardware, considering all combinations between the betting websites listed in the previous section.

To calculate the profit, we use the following formula:

Profit = (Investment / Arbitrage %) – Investment

The pseudocode for the algorithm is given below.

Algorithm 3 calcodds

```
arbprob ← 0
if a + b < a * b then
    isopp ← 1
    arbprob ← (a * b)/(a + b)
else
    arbprob ← 0
end if
return arbprob
```

In order to calculate per-dollar profit, we unfortunately still need to perform a division. However, this calculation only needs to take place when there is an arbitrage opportunity. In the case where division becomes too expensive, we will modify the algorithm to simply indicate 1 or 0 in the case of an arbitrage opportunity, and leave out the calculation of the profits.

5 Resource Budgets 💵

For our project we decided to use the in-fabric BRAM. We plan to route writes from software to the memory blocks contained within each profit calculation module to parallelize the process without serializing reads and writes to shared memory.

5.1 Upper Limit Memory Constraints

Parameter	**Value**
Total Block RAM in DE1-SOC	397 10kB blocks
Total Memory	3970kB (approximately 3.88MB)
Total Memory in Bytes	3970kB $\times$ 1024 = 4065280 bytes
Module Size	64 bits (8 byte)
Max Modules in Parallel	$\frac{4065280 \text{ bytes}}{8 \text{ bytes/module}} = 508160$

Table 2: Block RAM Memory Constraints

5.2 Memory Usage

In the initial version of our Arbitrage Detector, our goal is to highlight arbitrage opportunities across various moneyline bets for several NBA games. This requires a dedicated module for each comparison made between bookmakers.

The estimated resource consumption is as follows:

- Number of bookies: 6
- Number of NBA games: 10

For 6 distinct bookies, the combinations without repetition are calculated as $C(6,2) = \frac{6!}{2!(6-2)!} = 15$. Each of the 15 combinations is multiplied by 2 to account for comparisons made for both winning and losing outcomes, as sportsbook odds may not sum to 100%.

Consequently, the total number of modules required is:

$$30 \text{ comparisons} \times 10 \text{ NBA Games} = 300 \text{ modules} \tag{1}$$

Given that 300 is significantly less than 508,160, the resource utilization is within the constraints of the FPGA. Nonetheless, as the project evolves, we plan to enhance the sportsbook portfolio by incorporating multiple props for each game, further increasing the total module count.

With the addition of game props, the estimated module requirement grows by a scaling factor:

$$30 \text{ Comparisons} \times 10 \text{ NBA Games} \times 30 \text{ Props} = 9000 \text{ Modules} \tag{2}$$

Inclusion of additional sports leagues and player props will continue to increase the number of necessary modules, while remaining within the memory constraints, indicating our system's capability to scale accordingly.

6 Timeline ⌚

We'll divide the project into 5 major milestones, with a timeline for each. We will first implement a prototype of our system in C, including each of the key algorithmic steps that will occur in software (next week). Once we have this working in software, we will implement a single arbitrage detection module on the board, and verify its behavior using historical data (in the next 2 weeks). Over the following two weeks, we will then expand this into a profit-calculation module for one input. Next, we will build the parallel structure of multiple modules, and simulate the routing logic with verilator to verify that each input is written into each module. We will then scale the system up, and test on larger batches of historical data. We will then incorporate calls to the actual API into the software application instead of pre-stored historical data, and begin profiling the performance of our system compared to identical workloads performed entirely in software. Finally, we will expand the number of bets to support multiple games, each of which contain multiple possible bets, each of which have multiple bookies.