IC Design HW3

B09901066 謝承修

Note

1. To compile the poker.v, you have to add "-v2005" in your command, like:
   "vcs tb_poker.v poker.v lib.v -full64 -R -debug_access+all +v2k **-v2005**"
2. Github repo: https://github.com/Alanhsiu/IC-Design-HW3 (private until the deadline)
3. Extremely appreciate TA Chen, who helps me a lot!

Circuit diagram

The main idea is as Figure 1. The flowchart outlines a poker hand evaluation module. It starts with five card inputs, splits them into suits and ranks, and then processes them through Flush, Same Rank, and Straight Detectors. Note that the detectors run in parallel. The final stage, *Type Determination*, uses these results to classify the poker hand type, efficiently categorizing each hand in the game. I will go through each detector module and briefly describe how they work. For more detail, you can directly see the comments in poker.v.

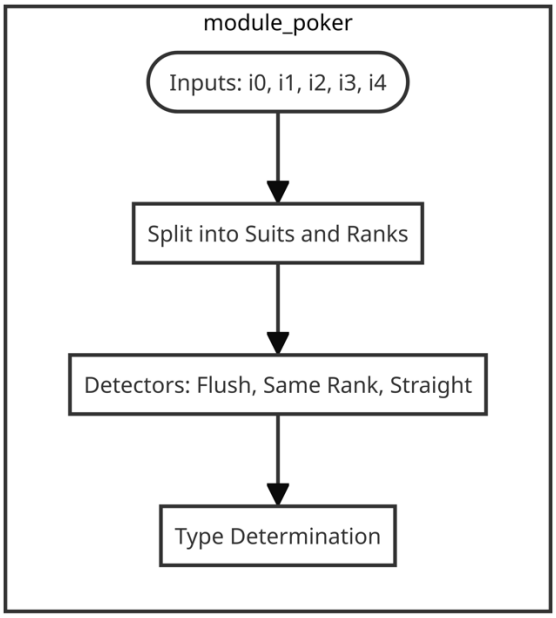

Figure 1: Hierarchy of the poker module.

First, the *flushDetector* module in the diagram processes five suit inputs (suit0 to suit4) to determine if a flush is present. It compares the first and second bits of each suit using EO gates, then uses NOR gates to check if all corresponding bits are the same, resulting in signals *all_same_bit0* and *all_same_bit1*. A Half Adder (HA1) gate combines these signals to output *isFlush* if a flush is detected, while a NAND gate outputs *isNotFlush* if a flush is not present. This module effectively evaluates the uniformity of suit bits to identify a flush in a set of poker cards.

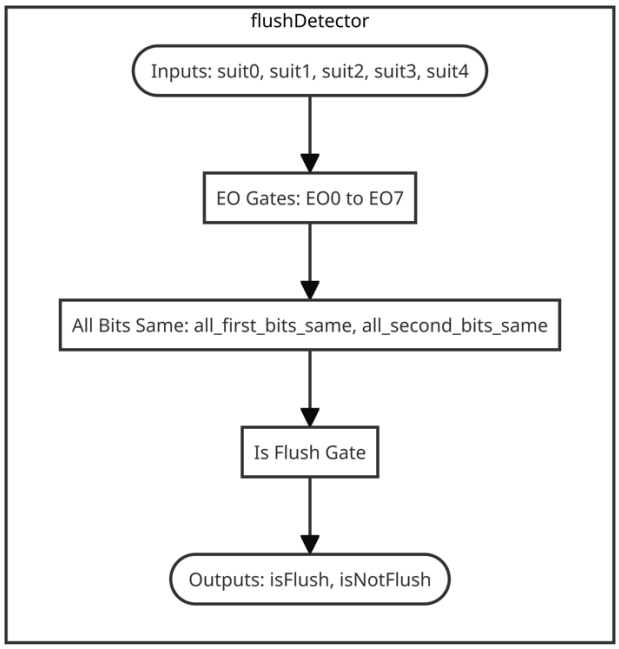

Figure 2: Hierarchy of the flushDetector module.

Second, the *sameRankDetector* module processes five rank inputs (rank0 to rank4) to determine poker hand combinations. It first compares each pair of ranks through sameRankComparator2 gates, generating signals indicating if any two ranks are the same. These signals are then fed into various gates to check for different hand combinations: *checkFourOfAKind* for four of a kind, *checkFullHouse* for a full house, *checkThreeOfAKind* for three of a kind, *checkTwoPairs* for two pairs, and *checkOnlyOnePair* for a single pair. Note that for each type, there are some specific combinations, which help us to simplify the detection process. For example, there are 5 combinations for four of a kind, 10 combinations for a full house, 10 combinations for three of a kind, 15 combinations for two pairs, and 10 combinations for a single pair. The outputs are signals indicating the absence of each specific hand type. This module effectively evaluates the rank similarities to identify possible poker hand combinations.

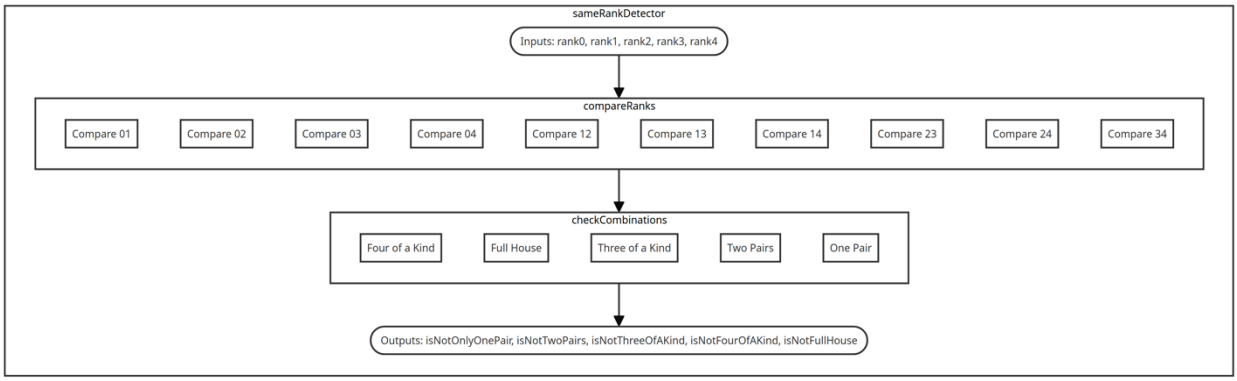

Figure 3: Hierarchy of the sameRankDetector module.

Third, the straightDetector module is designed to evaluate if a set of five poker cards (rank0 to rank4) forms a straight. It begins by assigning the input ranks to an array for easier processing. The module then checks for all possible straight combinations, ranging from A2345 to 10JQKA, using individual *checkStraight* components. If any of these checks pass, indicating a straight, the *isStraight* output is set to true. Conversely, if none of these checks pass, the *isNotStraight* output is set to true, indicating that the hand does not form a straight. This module efficiently evaluates all straight possibilities in a poker hand.

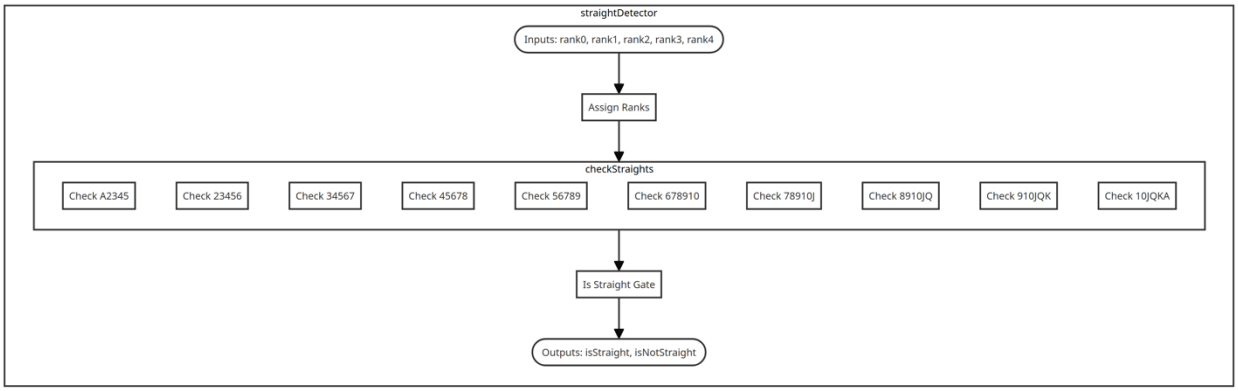

Figure 4: Hierarchy of the straightDetector module.

Finally, we can use the information from the above to determine the type of the poker by using some logic gates to calculate bit by bit, as shown in Figure 5.

```
// type determination
wire notStraight, notFlush;

ND2 flush_gate(notFlush, isFlush, isNotStraight);
ND2 straight_gate(notStraight, isStraight, isNotFlush);

wire temp[2:0];
AN2 temp_gate0(temp[2], isNotFourOfAKind, isNotFullHouse);
AN3 temp_gate2(temp[0], isNotFourOfAKind, isNotThreeOfAKind, isNotOnlyOnePair);

HA1 type_3(.O(type[3]), .A(isFlush), .B(isStraight));
ND3 type_2(type[2], temp[2], notFlush, notStraight);                              // four of a kind, full house, flush,          straight
ND4 type_1(type[1], isNotFourOfAKind, isNotFullHouse, isNotThreeOfAKind, isNotTwoPairs); // four of a kind, full house, three of a kind, two pairs
ND2 type_0(type[0], temp[0], notFlush);                                           // four of a kind, flush,      three of a kind, one pair
```

Figure 5: Type Determination.

Discussion

In this homework, I've tried tremendous ways to detect the straight type and seek for the one which runs fastest, including several methods of sorting like bubble sort and counting sort, and the method of observing pairwise relationship of "rank differ by 1". The former way doesn't work, since it takes too many layers to sort the ranks. The latter one works, since there is an efficient way to check if there are 4 out of 10 pairs are "differ by 1". However, I use none of the method mentioned above, for the reason that is too complex. I choose the brute- force way finally, since it's the most intuitive.

After I complete the part above, the critical path is below 4ns. I then try to replace all of AND/OR gates with NOR/NAND gates, the critical path went below 3ns. The last step, I consider the speed of the signals. Since the *isStraight* is the slowest, I add some gates to compute the rest of the signal in advance, to reduce the fanin of final NAND gates. After all the simplification, the critical path is below 2.5ns, as shown in Figure 6.

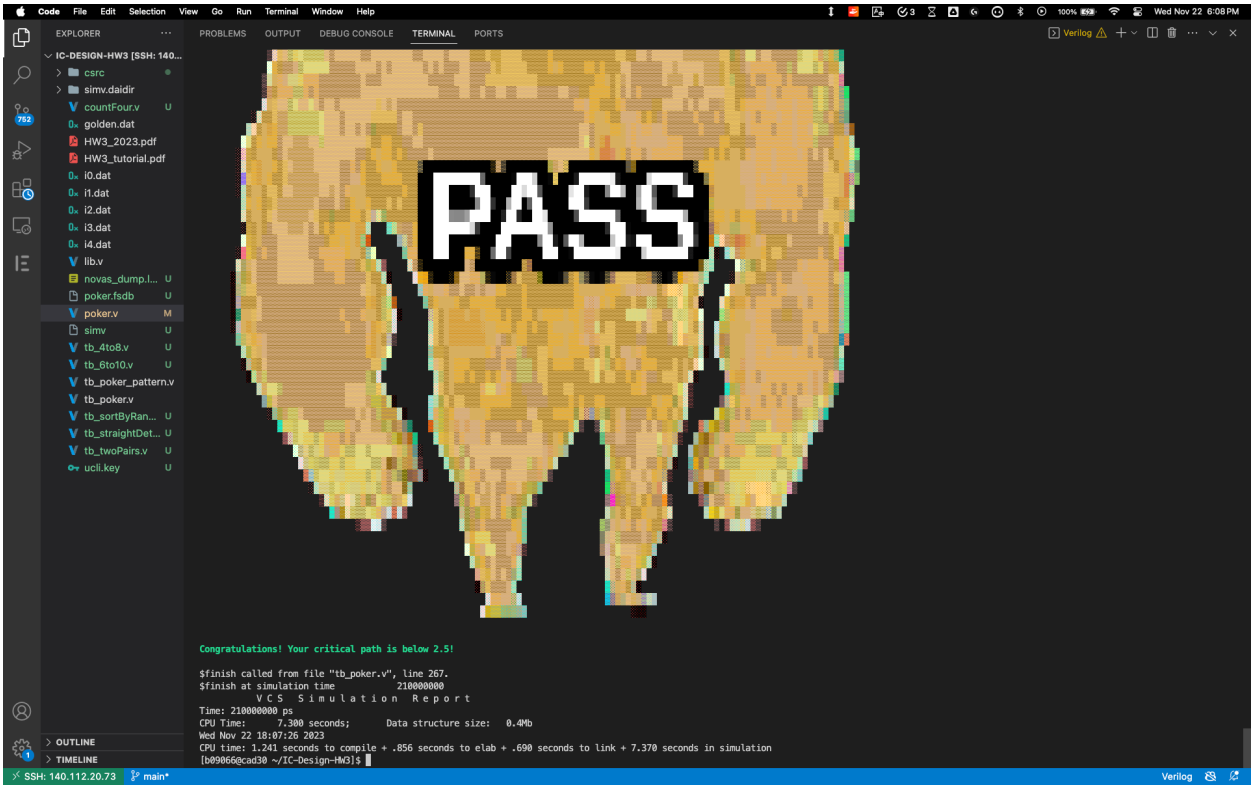

Figure 6: Final Result.