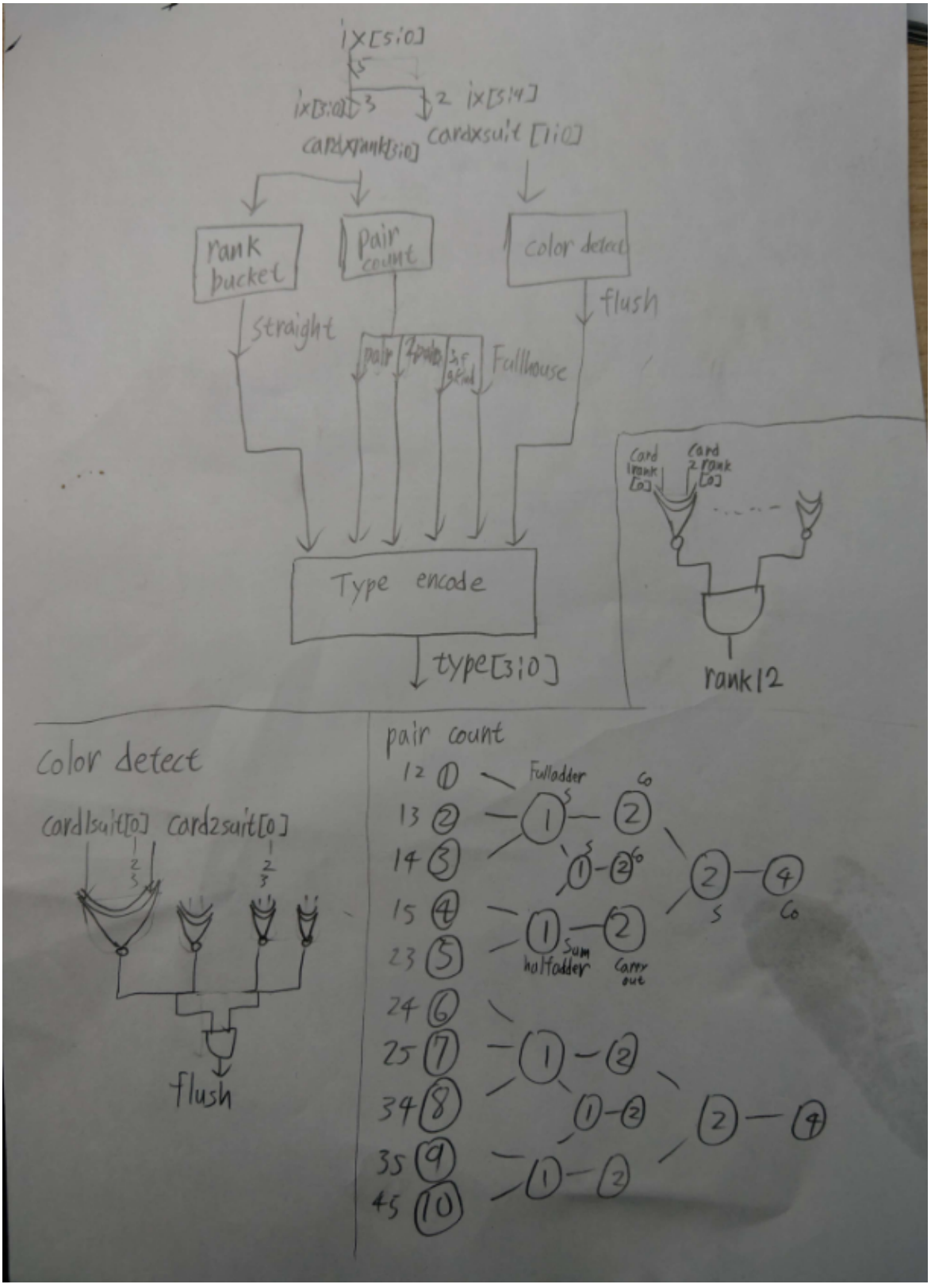
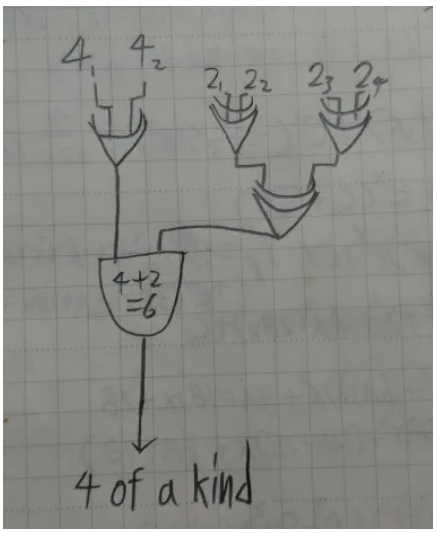
B09901079 IC Design
HW3

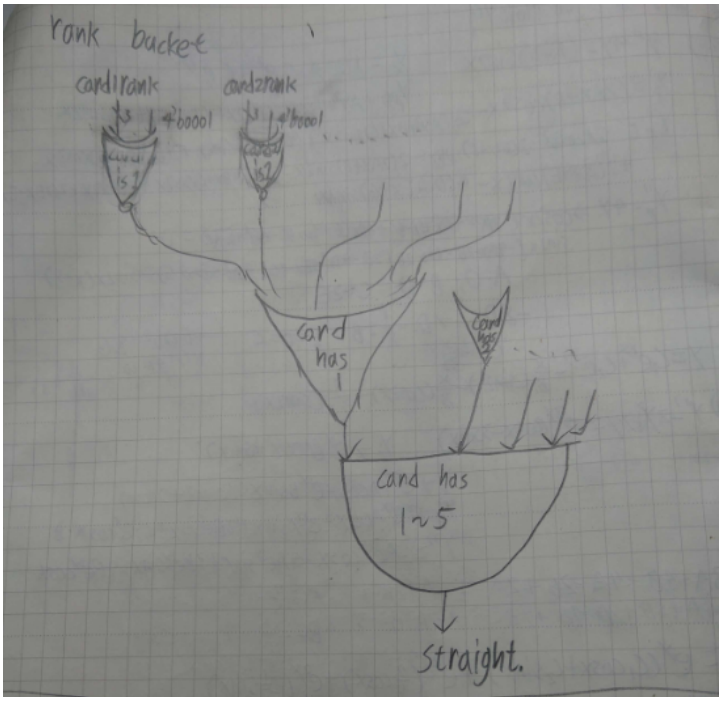
Part 1:Circuit Diagram



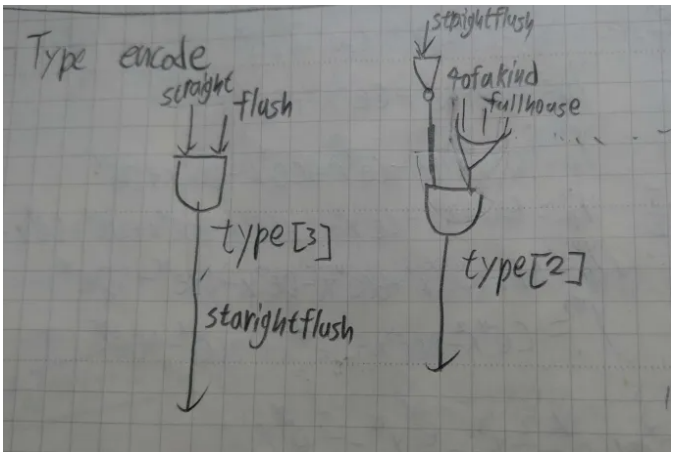
Picture 1:Full diagram, rough sketch to each sub-module: Pair Count, Color Detect



Picture 2:Pair count after adder phase



Picture 3: Rank Bucket



Picture 4:Encoder

Description&Discussion:

The circuit detects 3 different card attributes, “sequence”, “pair counts”, “suit(color)”. To make things easier, I denote ix[5:4] as CardXSuit[1:0], and ix[3:0] as CardXRank[3:0]

Start with the easiest one: Suit. By using the module EN(XNOR), we can practically create a logic operation that outputs 1 when both inputs are the same and vice versa*. Since we can only do it one bit by one bit, I use 2 XNOR gates to compare CardXSuit[0] and [1]. Then connect the outputs of XNOR gate to inputs of AND gate to detect if they truly have the same suit with both bits matching.Since the only types considering suits are flush related, so I use a larger AND gate to output “Flush” if card 1&2,2&3,3&4,4&5 all have the same suit, thus card 1~5 are all in the same suit.

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

*XNOR gate truth table

Then Pair Count, I found that all pair related types have different numbers of matching pairs so it’s a valid strategy to count how many matching pairs exist in the sequence. (For example, AAAQQ with each denoted as 12345, has pairing of 12,13,23 and 45, a total of 4 pairs.)

Four of a Kind: 6 pairs.
Full House: 4 pairs.
Three of a Kind: 3 pairs.
Two Pairs: 2 pairs.
Pair: 1 pair.

By using the same way I detect suit (XNOR gate to detect each matching bit, then AND gate to check if they all match), I can detect if Card X has the same rank as Card Y. Doing each possible pair (orderless) of 5 cards I get a total of 10 different possible combinations.
Then I use a combination of Half Adder and Full Adder to add up the matching pairs to decrease the complicity of possible combinations. With 2 layers of adder, I have two bits representing “4(pairs)”, four bits representing “2” and two bits representing “1”. Then design a sub-sub-module for each type such as: Four of a Kind has 6 pairs, could be 4+2, 4+1+1, 2+2+2…. It’s trivial in difficulty to design these logical combinations so I don’t show them here, though there’s one interesting catch that is I use two XOR gates with four “2” as input, then use another XOR gate to ensure output is true if only one “2” is true. While from a simple logic point of view, it could be three out of four “2” is true, but in that case “4” must be false and I already designed a special case for triple “2”.

The last detection module is Rank Bucket. Why I call it this way is because I detect if each card X rank is K(a number) and put them into bucket K. The basic idea of detecting if card X rank is K is similar to that of detecting if card X suit is the same as card Y suit, just that the other input of the XNOR gate is constant. Doing 5 times for all 5 cards and using an OR gate to output if there exists any card that's rank K. After doing all 5 cards with all 13 rank buckets, we then use an array of AND gates to detect if they're straight. The idea is simple: If Bucket x to bucket x+4 are all full, and there exist only 5 cards, then it must mean each bucket has exactly one card inside, thus being straight.

Now we have all sufficient indicator wires we need that are:

Straight
Flush
Four of a Kind
Full House
Three of a Kind
Two Pairs
Pair

Note that we don't need StraightFlush (yet) and High Hand as Straight&Flush=StraightFlush and if they're all false, it's High Hand.

To build the encoder, I connect logic operations using indicators above to type[3:0]. For example wire StraightFlush is made by Straight&Flush using an AND gate, then directly connects it to type*[3]. So if StraightFlush is true, it would be "1000", it would be "0000" otherwise. Using the same idea, we can encode each type such as: Flush, just connect the wire Flush to type*[2] and type*[0] so if it's true, it would be "0101", vice versa. Finally use a large OR gate to connect all type encoding wires(type*1, type*2….) together to the output. Since false indicators will have "0000" in their responding wire, only the indicator whose state is true will determine the result.
Do note that since StraightFlush is using a combination of existing indicator wires (not base indicator) so it's possible to conflict with existing encoding such as Flush. Flush will encode "0101" and StraightFlush encode "1000", if using simple OR gate it will become "1101", thus I specifically designed a "NotStraightFlush" signal with an inverter, and connect it to an AND gate with other inputs being original indicator ones (type*), so type[2:0] can only be 1 if StraightFlush is false, in other words, StraightFlush will suppress type[2:0] to be all 0 if it's true.

Further discuss:

It seems like a huge waste to only use the bucket system to detect straight, as it should be a good tool to detect pairs or even suits if modified a bit. But due to the time limit I decided to leave it as it is. I was originally going to sort the cards out but I find it to be really difficult with the limit of only using existing gates, but the bucket method seems to be applicable and decently fast despite the sheer size of the layout since they're all parallelly computing.

Though my score is mere 5ns, I still find some interesting parts of the idea of critical path, that is if you lay things out, it will be quick. In other words one can trade space and power for faster computing speed, a trade off. An iteration method will be energy efficient and compact but it just won't be as fast as parallel computing all the things at once.

Another thing I found in this extremely limiting level of design is that everything is specialized, case by case. With a lower level of computer design, one can go more specialized than higher level design which has to rely on IDE and compiler. What I mean is for example, it's possible to further optimize the pair counting by a pentagram diagram with the idea that roughly goes as "if these 3 cards are pairing, then the other 3(overlapping) must be pairing in a specific way such that I don't even need to count all 3 pairs I have", these kind of oddly specific cases. But I think it's the part of lower level design that is to find these optimizations that's so specialized that it's hard to replicate in higher level coding.