Risk Management System

based on FIX protocol

CSEE 4840 Final Report

Kaixi Ji (kj2330)

Modi Yan (my2408)

1. Overview

Financial Information eXchange (FIX) protocol is intended for information security transactions in markets. FIX session is used as the standard electronic protocol between the buy side (buyer) and sell side (executor) of the financial markets, as shown in Figure 1.1.

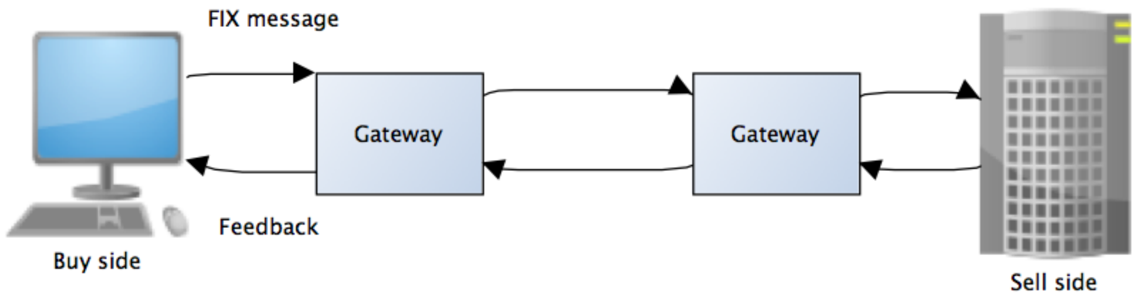


Figure 1.1

But as there are “risks” in the trading market and might cause the buyer a lot of fortune in really short time, we have the desire to check every message that comes to the buyer company’s gateway, check the message with specific rules, and drop the message if it contains any illegal information or certain rules are violated. The refined system is shown in figure 1.2.

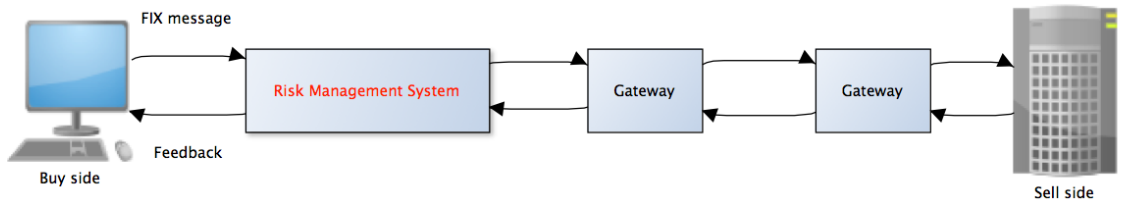


Figure 1.2

So the next question is, what are the rules that we want to apply to the messages? For the first thing, we can consider the situation where the buyer wants to buy 100 cows. But for some reasons, the buyer’s computer system gets busted and keeps sending out the same message time and time again. In this way, the buyer might ends up buying 100,000 cows in a very short time and suffers a lot from the problem. So for this kind of situation, the first rule we wanna check is the maximum number of NewOrderSingle messages the buyer can send in one second.

The second rule sets limit of the maximum amount of contracts a FIX message can contain. For example, if the buyer intends to buy 100,000 cows, he might wants to divide that into 100 messages, each containing 1000 contracts of buying cows, instead of having just one message containing 100,000 contracts.

The third, and the most important rule that we’ve considered so far, is the maximum number of floating contracts in the book for the buyer. Let’s also consider the example of buying cows. If the buyer has sent out a message of buying 1000 cows, for the time being he has 1000 floating contracts for buying cows in the contract book. As this request gets executed, the executor might finds out that there're only 100 cows available for the buyer's buying request, so an execution report of 100 cows bought is sent back to the buyer from executor. In this way, it leaves the buyer with 900 buying contracts floating in the book. This floating contract number intuitively indicates the "risk" the buyer is suffering from. The third rule sets up limit for number of floating contracts for the buyer/target/symbol combination, to have the "risk" contained.

As a matter of fact, there has already been plenty of software applications that does all kinds of risk checking for the messages. So why to implement this on FPGA. The first reason would be the checking acceleration the hardware can bring. This speed merit would be highly important in terms of high frequency trading (HFT). The second reason is that, while the software checking speed varies a lot with input volume, hardware design appears to be more robust and thus more reliable.

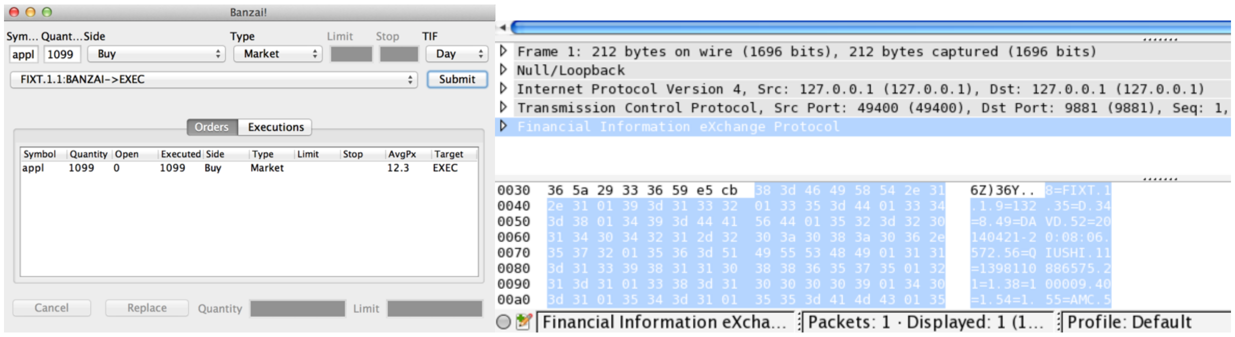
2. Financial Information eXchange (FIX) protocol basics

In FIX encoding, the whole message is presented as a big string, and each character of the string can be recognized as its 1-Byte ASCII code. Each field of message is delimited by the ASCII character 01 <start of header>. Within each field, message is presented in the form of "tag=value". Tag represents the kind of information that is contained in this field, e.g. tag "49" represents "sender company ID". Value contains the actual value of the specific tag, e.g. "49=APPL" says that "sender company ID is APPL". For example, a NewOrderSingle FIX message may look like this:

8=FIXT.1.1|9=73|35=A|34=49|49=BANZAI|52=2014032516:22:19.047|56=EXEC|98=0|108=30|1137=7|10=166|

FIX fields in a message can be divided into three groups: header, body and trailer. For FIXT.1.1, the header contains five mandatory fields: 8(BeginString), 9(BodyLength), 35(MsgType), 49(SenderCompID), and 56(TargetCompID). Body fields vary with message type. Trailer is tagged 10, with value of a three-digit checksum. In practice, we only know for sure that the first field is tagged 8 and the last field is tagged 10 in each FIX message, but order of other fields is unsettled.

In our project, we use quickfix engine to generate FIX message for analysis (Figure 2.1 left), the generated FIX messages can be captured by wireshark (Figure 2.1 right).



appl | 1099 | Buy | Market | Day

FIXT.1.1:BANZAI->EXEC

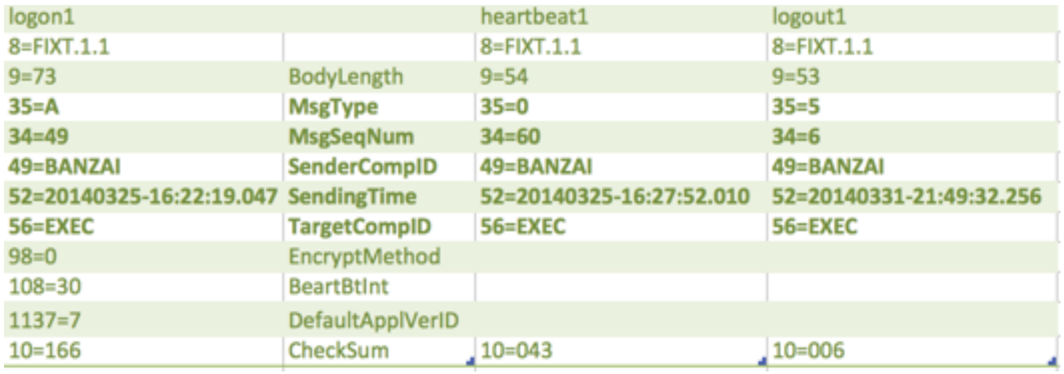
Symbol	Quantity	Open	Executed	Side	Type	Limit	Stop	AvgPx	Target
appl	1099	0	1099	Buy	Market			12.3	EXEC

```
▷ Frame 1: 212 bytes on wire (1696 bits), 212 bytes captured (1696 bits)
▷ Null/Loopback
▷ Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
▷ Transmission Control Protocol, Src Port: 49400 (49400), Dst Port: 9881 (9881), Seq: 1,
▷ Financial Information eXchange Protocol

0030  36 5a 29 33 36 59 e5 cb  38 3d 46 49 58 54 2e 31   6Z)36Y.. 8=FIXT.1
0040  2e 31 01 39 3d 31 33 32  01 33 35 3d 44 01 33 34   .1.9=132 .35=D.34
0050  3d 38 01 34 39 3d 44 41  56 44 01 35 32 3d 32 30   =8.49=DA VD.52=20
0060  31 34 30 34 32 31 2d 32  30 3a 30 38 3a 30 36 2e   140421-2 0:08:06.
0070  35 37 32 01 35 36 3d 51  49 55 53 48 49 01 31 31   572.56=Q IUSHI.11
0080  3d 31 33 39 38 31 31 30  38 38 36 35 37 35 01 32   =1398110 886575.2
0090  31 3d 31 01 33 38 3d 31  30 30 30 30 39 01 34 30   1=1.38=1 00009.40
00a0  3d 31 01 35 34 3d 31 01  35 35 3d 41 4d 43 01 35   =1.54=1. 55=AMC.5

Financial Information eXcha... | Packets: 1 · Displayed: 1 (1... | Profile: Default
```

Figure 2.1

With different message types, the FIX message fields can vary a lot. From the quickfix engine that we have, we are sending/receiving the following three classes of FIX messages:

First class -- logon, logout, and heartbeat messages (Figure 2.2). These messages are sent mutually between the buyer and executor. They normally doesn't contain a lot of information, just the basic header and trailer to indicate link between the two ends.

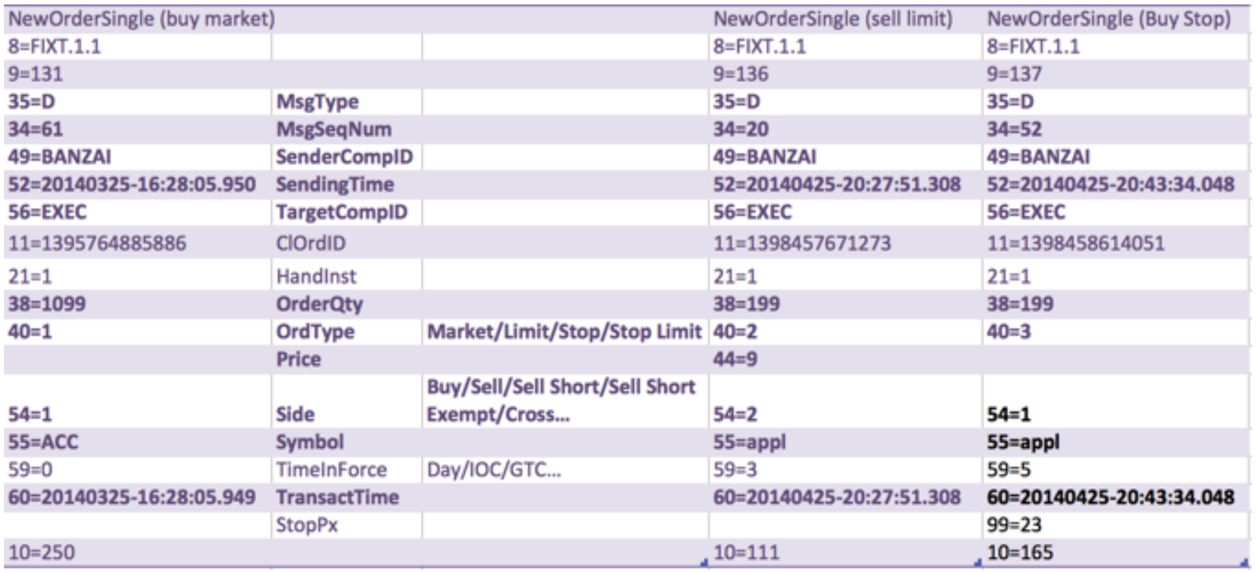


logon1		heartbeat1	logout1
8=FIXT.1.1		8=FIXT.1.1	8=FIXT.1.1
9=73	BodyLength	9=54	9=53
35=A	**MsgType**	**35=0**	**35=5**
34=49	**MsgSeqNum**	**34=60**	**34=6**
49=BANZAI	**SenderCompID**	**49=BANZAI**	**49=BANZAI**
52=20140325-16:22:19.047	**SendingTime**	**52=20140325-16:27:52.010**	**52=20140331-21:49:32.256**
56=EXEC	**TargetCompID**	**56=EXEC**	**56=EXEC**
98=0	EncryptMethod		
108=30	BeartBtInt		
1137=7	DefaultApplVerID		
10=166	CheckSum	10=043	10=006

Figure 2.2

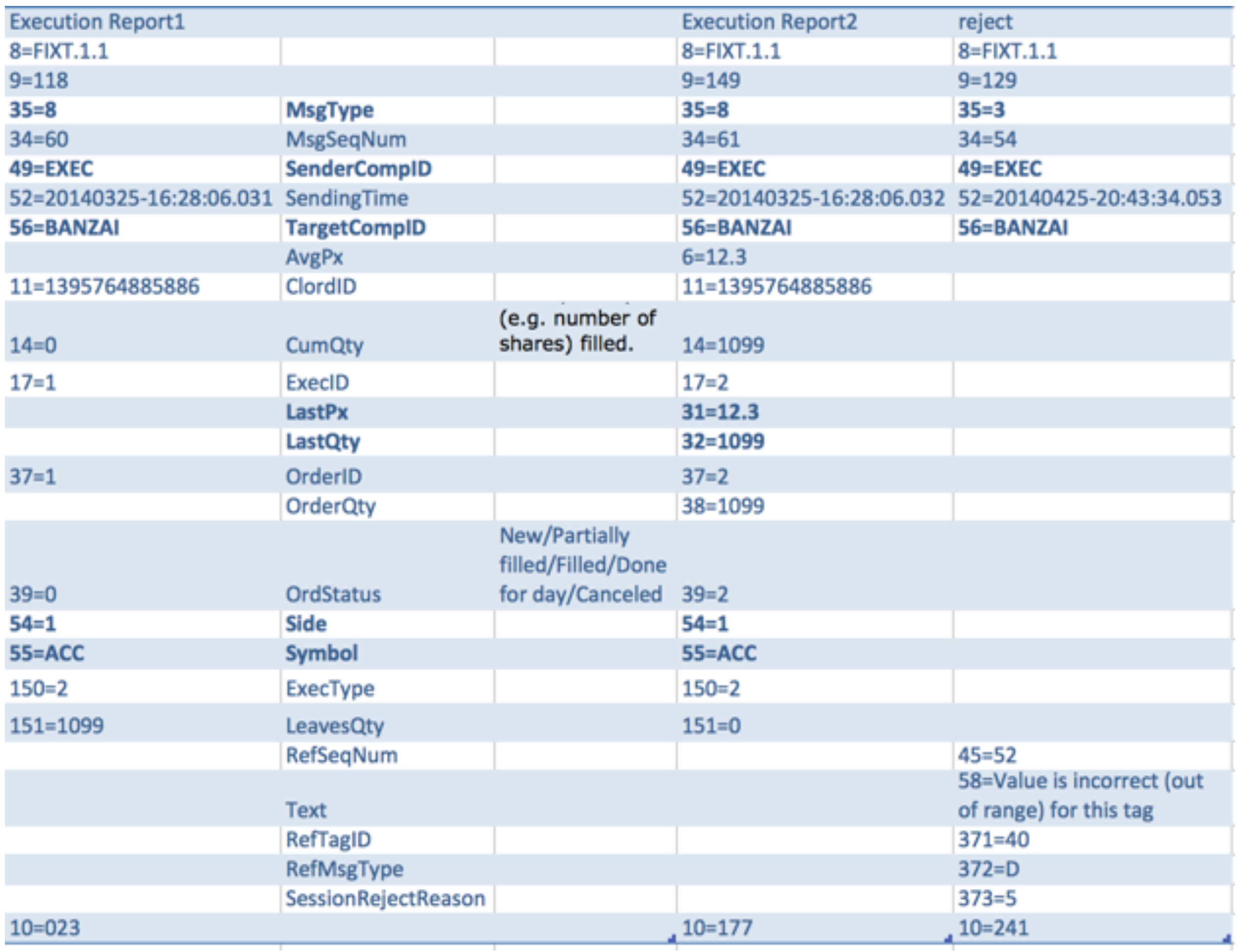
The second class -- NewOrderSingle(FIgure 2.3). This message indicates that the buyer sends out new order to executor to buy/sell something. Actually there're many kinds of FIX messages placing orders, but quickfix is built with this most simple one. NewOrderSingle messages contain many important fields specializing trading condition -- for example, 38(OrderQty), 40(OrdTpye), 54(Side), and 55(Symbol).

Third class -- ExecutionReport, Reject, and other reports (Figure 2.4). These messages go from the executor to buyer, indicating the current execution state of the buyer's previous orders. Here we care most about the 32(LasQty) field, indicating the number of contracts that actually got executed this time.



NewOrderSingle (buy market)			NewOrderSingle (sell limit)	NewOrderSingle (Buy Stop)
8=FIXT.1.1			8=FIXT.1.1	8=FIXT.1.1
9=131			9=136	9=137
35=D	**MsgType**		**35=D**	**35=D**
34=61	**MsgSeqNum**		**34=20**	**34=52**
49=BANZAI	**SenderCompID**		**49=BANZAI**	**49=BANZAI**
52=20140325-16:28:05.950	**SendingTime**		**52=20140425-20:27:51.308**	**52=20140425-20:43:34.048**
56=EXEC	**TargetCompID**		**56=EXEC**	**56=EXEC**
11=1395764885886	ClOrdID		11=1398457671273	11=1398458614051
21=1	HandInst		21=1	21=1
38=1099	**OrderQty**		**38=199**	**38=199**
40=1	**OrdType**	**Market/Limit/Stop/Stop Limit**	**40=2**	**40=3**
	Price		**44=9**	
54=1	**Side**	**Buy/Sell/Sell Short/Sell Short Exempt/Cross...**	**54=2**	**54=1**
55=ACC	**Symbol**		**55=appl**	**55=appl**
59=0	TimeInForce	Day/IOC/GTC...	59=3	59=5
60=20140325-16:28:05.949	**TransactTime**		**60=20140425-20:27:51.308**	**60=20140425-20:43:34.048**
	StopPx			99=23
10=250			10=111	10=165

Figure 2.3



Execution Report1			Execution Report2	reject
8=FIXT.1.1			8=FIXT.1.1	8=FIXT.1.1
9=118			9=149	9=129
35=8	**MsgType**		**35=8**	**35=3**
34=60	MsgSeqNum		34=61	34=54
49=EXEC	**SenderCompID**		**49=EXEC**	**49=EXEC**
52=20140325-16:28:06.031	SendingTime		52=20140325-16:28:06.032	52=20140425-20:43:34.053
56=BANZAI	**TargetCompID**		**56=BANZAI**	**56=BANZAI**
	AvgPx		6=12.3	
11=1395764885886	ClordID		11=1395764885886	
14=0	CumQty	(e.g. number of shares) filled.	14=1099	
17=1	ExecID		17=2	
	LastPx		**31=12.3**	
	LastQty		**32=1099**	
37=1	OrderID		37=2	
	OrderQty		38=1099	
39=0	OrdStatus	New/Partially filled/Filled/Done for day/Canceled	39=2	
54=1	**Side**		**54=1**	
55=ACC	**Symbol**		**55=ACC**	
150=2	ExecType		150=2	
151=1099	LeavesQty		151=0	
	RefSeqNum			45=52
	Text			58=Value is incorrect (out of range) for this tag
	RefTagID			371=40
	RefMsgType			372=D
	SessionRejectReason			373=5
10=023			10=177	10=241

Figure 2.4

3. Design Flow

The simplified high-level diagram is shown in Figure 3.1.

Uplink is used for messages coming from buyer to executor, and Downlink is used for messages going from executor to the buyer.

Diagram for the Uplink part is in Figure 3.2. The uplink part and the downlink part are almost symmetric except that it is possible for the uplink to drop a message while the downlink part will pass all the messages eventually. More detailed description of each part of the design and difference between uplink and down link will be mentioned latter.

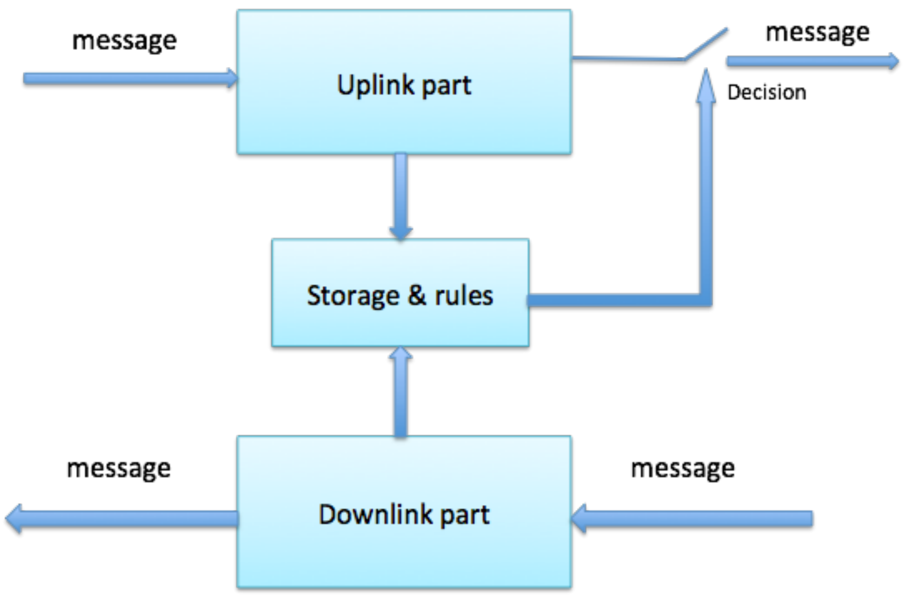


Figure 3.1

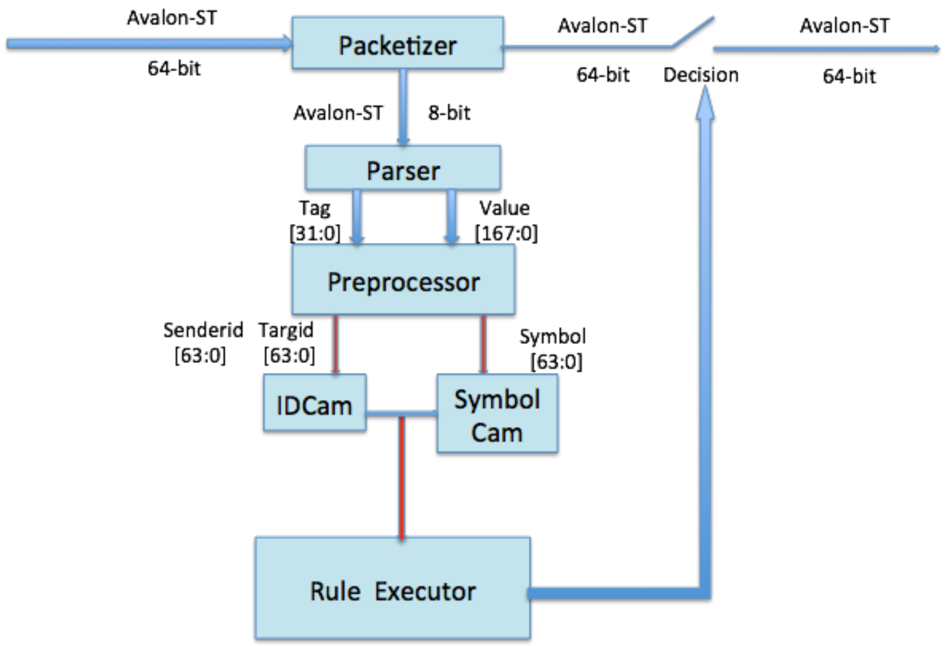


Figure 3.2 Components of the uplink part

4. Message Parsing and Preprocessing

4.1 Packetizer

Uplink packetizer:

The packetizer will buffer the whole message for following analysis and wait for the signal from the controller telling it whether to pass this message.

Since our system only concerns about the FIX message, there are several situations that the packetizer itself can decide to pass the message. Those situations are when the length of the message is shorter than 66 bytes (the length of the header of a TCP/IP message) or when the data following the header are not “8=FIX” which means it is not a FIX message.

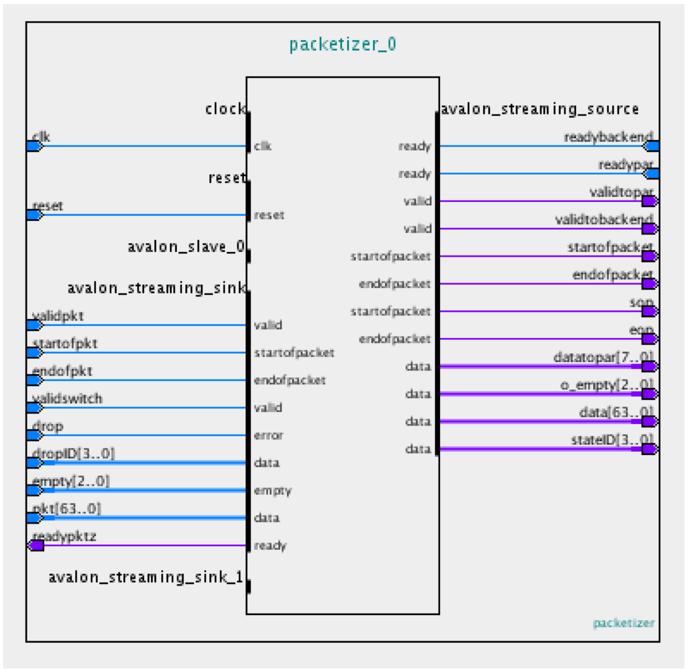


Figure 4.1.1 Block diagram of packetizer

We can notice that the speed of input is 64 bits/clk while the output to parser in 8 bits/clk. This speed sacrifice is for the convenience of parser to build the state machine and to extract the tag and value. But the speed is still fast enough for most cases when the number of messages coming in is not extremely large. This assumption holds especially when the messages are manually sent.

If the incoming message is a FIX message, after the input “endofpkt” is 1 which means the packtizer now buffer the whole message, packtizer starts to output the FIX information (message subtract the header) to the parser. Once the packetizer receives the decision from the controller, it will start to pass the intact message at speed of 64 bits/clk or it immediately drop the message. After processing the message, the packetizer goes back to the original state and becomes ready for the next incoming message.

Downlink packetizer:

It is almost the same as the uplink packetizer. The only difference is that it will pass the message eventually. Once it receives the signal from the following modules telling it that the process is done, it will output the data at the speed of 64 bits/clk.

4.2 Parser

As mentioned earlier, the input for the parser is 8 bits/clk. The reason is that the FIX message has a fixed format "tag=value|tag=value|tag=value|" as the former example

"8=FIXT.1.1|9=73|35=A|34=49|49=BANZAI|52=2014032516:22:19.047|56=EXEC|98 =0|108=30|1137=7|10=166|".

So with one byte input per clock, It becomes easier for the parser the figure out whether the incoming data belongs to the tag field or the value field by detecting the equality sign and the delimiter. The simplified state diagram is shown in the Figure 4.2.1.

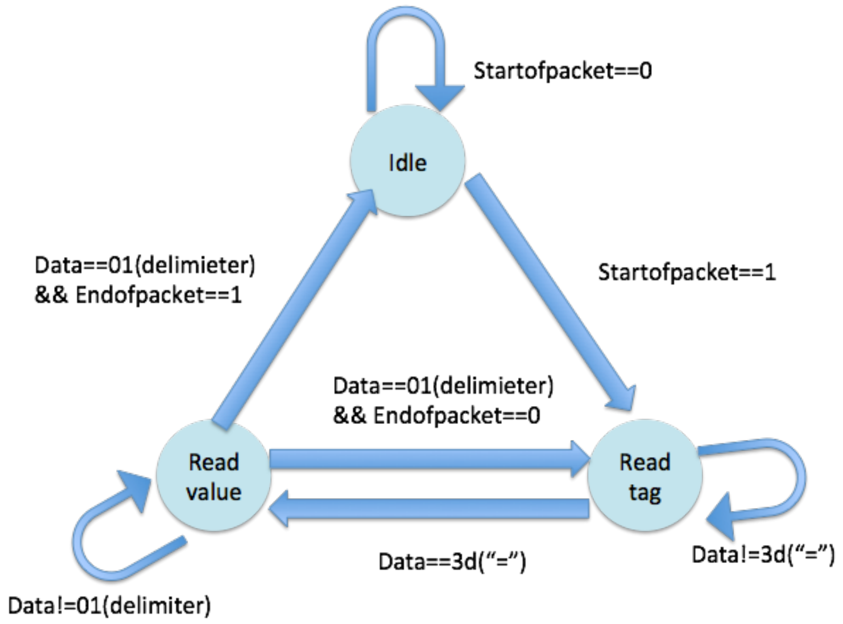


Figure 4.2.1 State diagram of the parser

By detecting the equality sign, the parser can separate the tag and value and will output the tag and value together once the delimiter is detected. The parser is trigger by the signal "startofpkt" and will return to idle as long as it receives the "endofpkt" and the next module is ready to receive the data. The internal counter is reset by the valid signal from controller implying that the process for the current message is done.

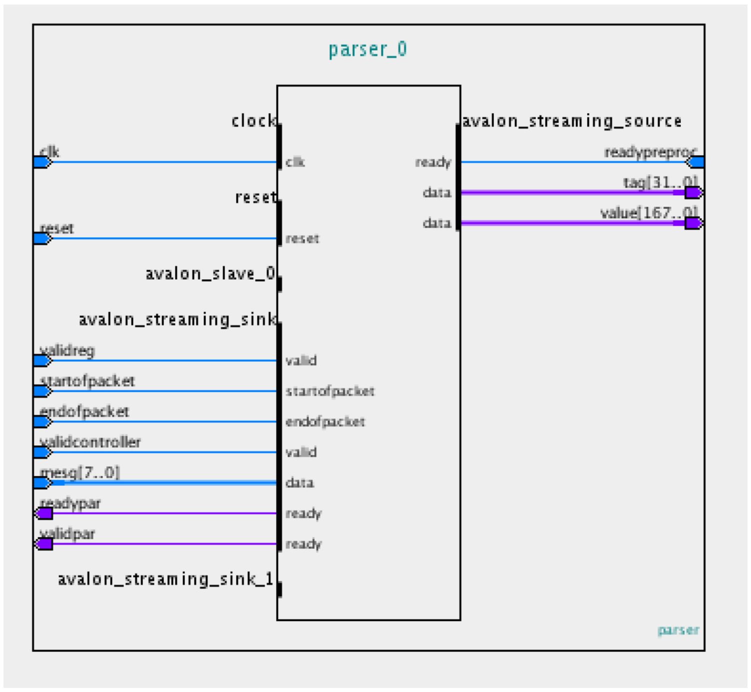


Figure 4.2.2 Block diagram of the parser

4.3 Preprocessor

With all tags and values of the FIX message passed from the parser, the preprocessor is used to select and preprocess with the values that we are interested in. Tag and value pairs should be passed to the preprocessor in separated clock cycles and if the tag matches one of the tags that we are interested in, we store the corresponding value in a register. If the tag talks about information that should be represented by actual numbers instead of strings, e.g. 38(OrderQty), we should parse the value from string to binary number before we store it in the register. The registers are refreshed for every FIX message, when we recognize the tag as 8, meaning the beginning of a message.

Signals coming in to and out from the uplink (from buyer to executor) preprocessor are shown in Figure 4.3.1 left, downlink (from buyer to executor) in figure 4.3.1 right.

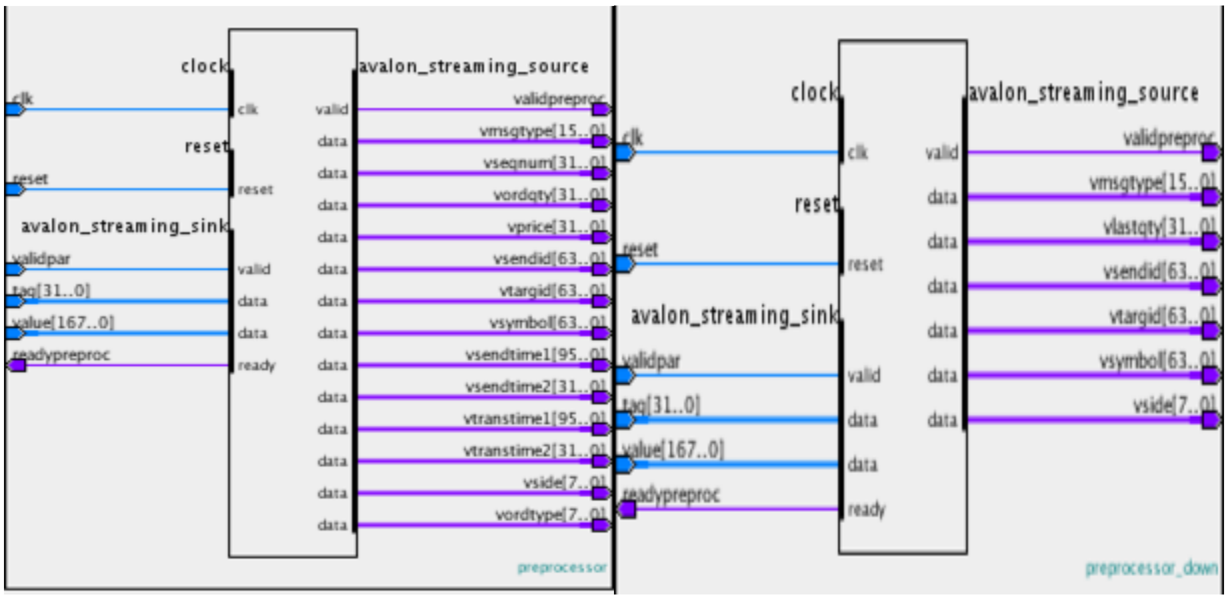
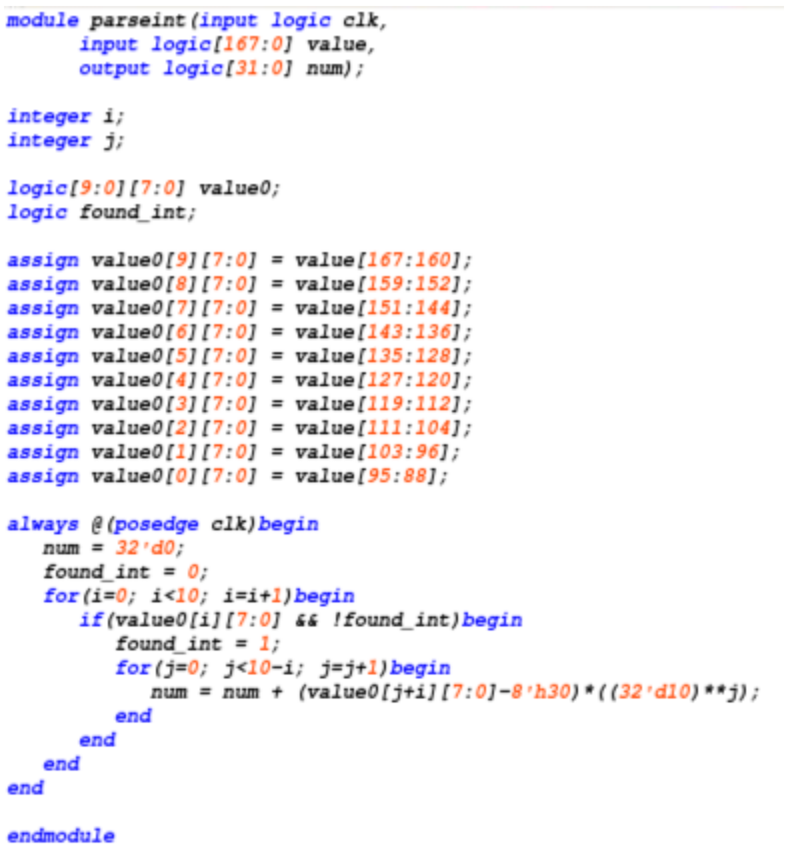


Figure 4.3.1

As our value input is a 168-bit long register, to parse order quantity and sending time values to binary numbers, we must detect effective length of the value string and parse it to binary number at the same time. With code in Figure 4.3.2, we managed to do this in just one clock cycle.



```
module parseint(input logic clk,
      input logic[167:0] value,
      output logic[31:0] num);

integer i;
integer j;

logic[9:0][7:0] value0;
logic found_int;

assign value0[9][7:0] = value[167:160];
assign value0[8][7:0] = value[159:152];
assign value0[7][7:0] = value[151:144];
assign value0[6][7:0] = value[143:136];
assign value0[5][7:0] = value[135:128];
assign value0[4][7:0] = value[127:120];
assign value0[3][7:0] = value[119:112];
assign value0[2][7:0] = value[111:104];
assign value0[1][7:0] = value[103:96];
assign value0[0][7:0] = value[95:88];

always @(posedge clk)begin
   num = 32'd0;
   found_int = 0;
   for(i=0; i<10; i=i+1)begin
      if(value0[i][7:0] && !found_int)begin
         found_int = 1;
         for(j=0; j<10-i; j=j+1)begin
            num = num + (value0[j+i][7:0]-8'h30)*((32'd10)**j);
         end
      end
   end
end

endmodule
```

Figure 4.3.2

For uplink (from buyer to executor), we are interested in the following tags:
- 35(MsgType) -- Defines message type ALWAYS THIRD FIELD IN MESSAGE.
- 34(MsgSeqNum) -- Integer message sequence number.
- 49(SendCompID) -- Assigned value used to identify firm sending message.
- 52(SendingTime) -- Time of message transmission (always expressed in UTC (Universal Time Coordinated, also known as "GMT").
- 56(TargetCompID) -- Assigned value used to identify receiving firm.
- 38(OrderQty) -- Quantity ordered. This represents the number of shares for equities or par, face or nominal value for FI instruments.
- 40(OrdType) -- Market/Limit/Stop/Stop Limit, etc
- 54(Side) -- Buy/Sell/Short, etc
- 55(Symbol) -- Ticker symbol. Common, "human understood" representation of the security.
- 60(TransactTime) -- Timestamp when the business transaction represented by the message occurred.

For downlink (from executor to buyer), we are interested in the following tags:
- 35(MsgType)
- 49(SenderCompID)
- 56(TargetCompID)
- 54(Side)
- 55(Symbol)
- 32(LastQty) -- Quantity (e.g. shares) bought/sold on this (last) fill.

4.4 Cam

The structure and functionality of the CAM vary depending on what kinds of rules that we are implementing.

We write codes for two kinds of IDCAM. For the rule that the buyer cannot send out transaction messages without sending logon message previously, the IDCAM works as a cache providing the information whether this ID combination is a “hit” or a “miss” and giving the address if it is a “hit”. If the incoming FIX message is a logon message, we will insert a new combination of two IDs (buyer and exchange). Otherwise, we can only search in the IDCAM and cannot modify it. So if there is a miss, an error signal will be return and the message will be dropped.

For the other rule that we already predefine the combinations of ID that we permit, the CAM is not extendable. All possible combinations that we permit are already in the IDCAM, if it is a miss for one combination, the message will be dropped. If it hits in the IDCAM, the IDCAM will return the address of that combination of ID.

When use the latter rule in our demonstration. The codes for these two rules are in the appendix. The structure of the latter IDCAM is as following:

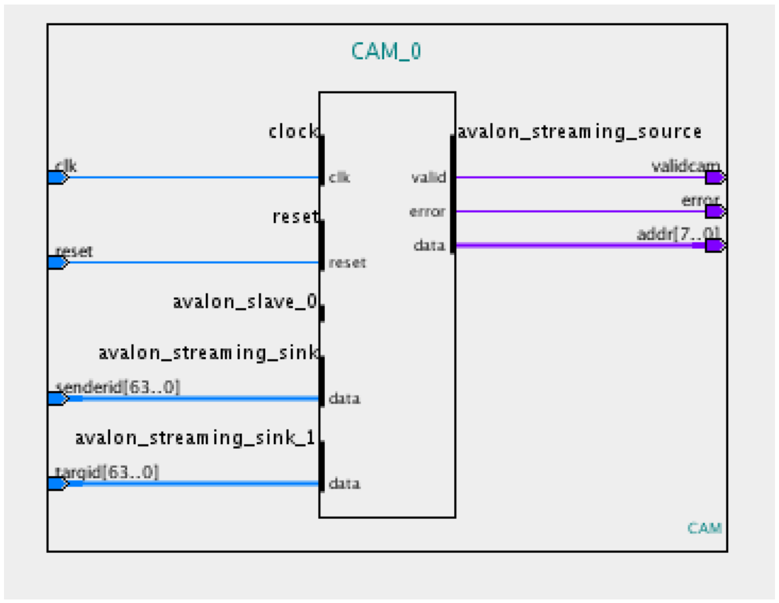


Figure 4.1 Block diagram of IDCAM

The symbol CAM works in a similar way as IDCAM except that it is used to retrun the address of a particular symbol. The inputs/outputs of it is shown in Figure 4.2.

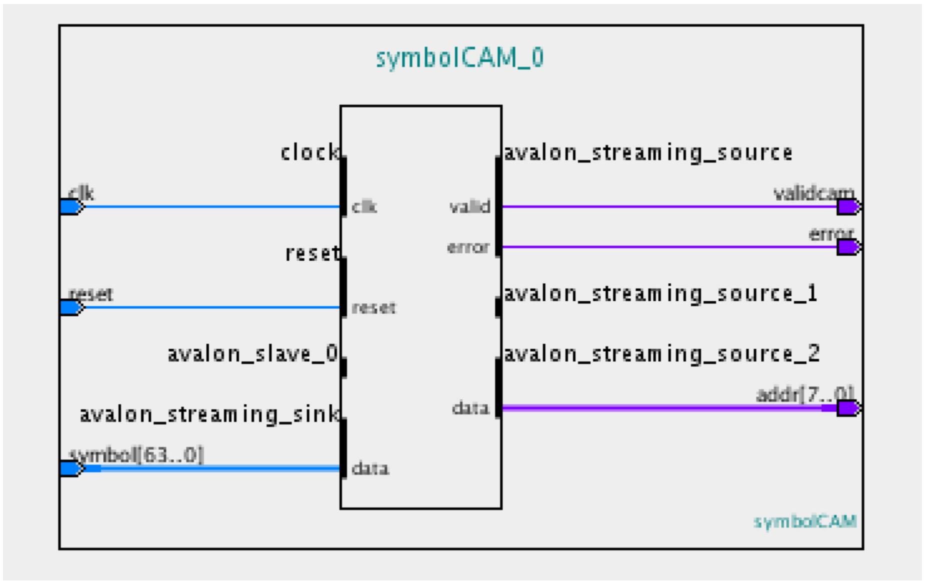


Figure 4.2 Block diagram of symbolCAM

5. Applying Rules and Decision Making

With all the interested FIX message values parsed out and {sender company ID, target company ID} and symbol hashed into address, we can finally get into the phase of applying rules to the message contents and make the final decision of whether to drop the message or not.

5.1 Maximum number of NewOrderSingle messages in one second

The first rule we want to apply is the maximum number of NewOrderSingle messages that can be sent out from the system in one second. For this purpose, we set up a time counter that adds one to itself by 1 at every posedge of clock starting from reset. When the counter adds up to 200,000,000, indicating the end of one second for a 200MHz clock, it is set to 0 again for the next second. We also have a # counter that adds to itself by 1 every time a NewOrderSingle (35=D) message comes. This counter is set to 0 at the end of each second together with the time counter.

The rule is violated whenever the # counter exceeds the limit that we set up. For every FIX message that goes into the system, if the rule is obeyed, we get valid signal and no error signal from this module, and if the rule is violated, we get valid signal and error signal from the module to the final controller.

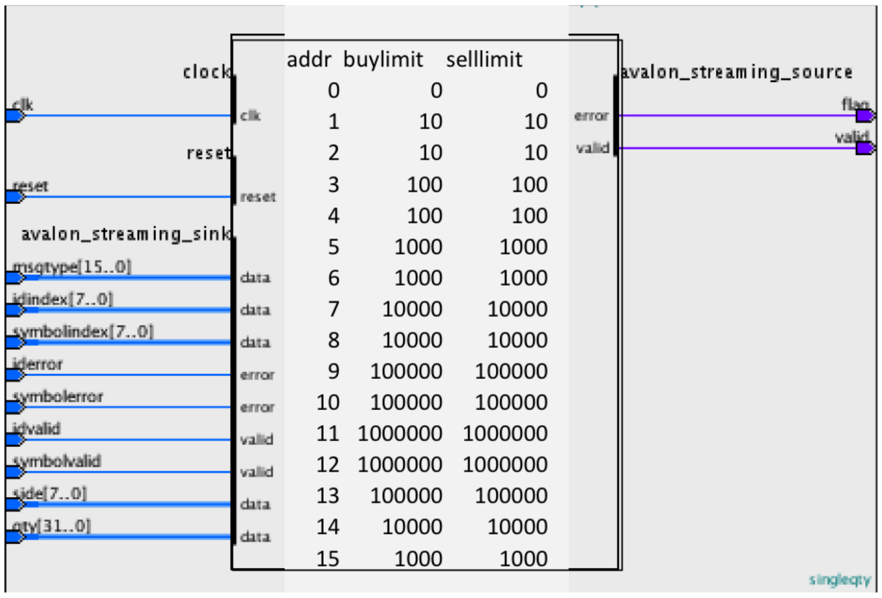
The output valid and error signals are refreshed every time a new FIX message goes in. Indicated by message type input from preprocessor being 16’d0.

5.2 Maximum number of contracts a NewOrderSingle message could contain

The second rule to apply and check is the maximum number of contracts a NewOrderSingle message of a certain {sender company ID, target company ID} combination could contain. Basic idea here is setting up rules for different people buying different things.

To do this, we store the limit for every combination in its corresponding address in a “RAM” (actually registers) at system setup time. We have one “RAM” for buying limits and another for selling limits, but the two limits are symmetrical. Every time a FIX message comes in, with message type of NewOrderSingle, side buy or sell (54=1 or 54=2), valid and not error ID combination address and symbol address, we compare the order quantity (tag 38) of this message with the limit that we setup. If the order quantity exceeds the limit in the corresponding address, the valid and error signal are sent to the controller. If the order quantity doesn’t exceeds the limit, only valid signal for the controller. Figure 5.2.1 represents this module.

Also, the output valid and error signals are refreshed every time a new FIX message goes in. Indicated by message type input from preprocessor being 16’d0.



addr	buylimit	selllimit
0	0	0
1	10	10
2	10	10
3	100	100
4	100	100
5	1000	1000
6	1000	1000
7	10000	10000
8	10000	10000
9	100000	100000
10	100000	100000
11	1000000	1000000
12	1000000	1000000
13	100000	100000
14	10000	10000
15	1000	1000

Figure 5.2.1

5.3 Maximum number of floating contracts

The third rule to apply is the maximum number of floating contracts of a {sender company ID, target company ID} combination.

To do this, we first also need a “RAM” storing the buying limit and selling limit of every combination. The buying and selling limits for the same combination is still symmetrical. Yet, we need another “RAM” storing the actual floating contract number of each combination.

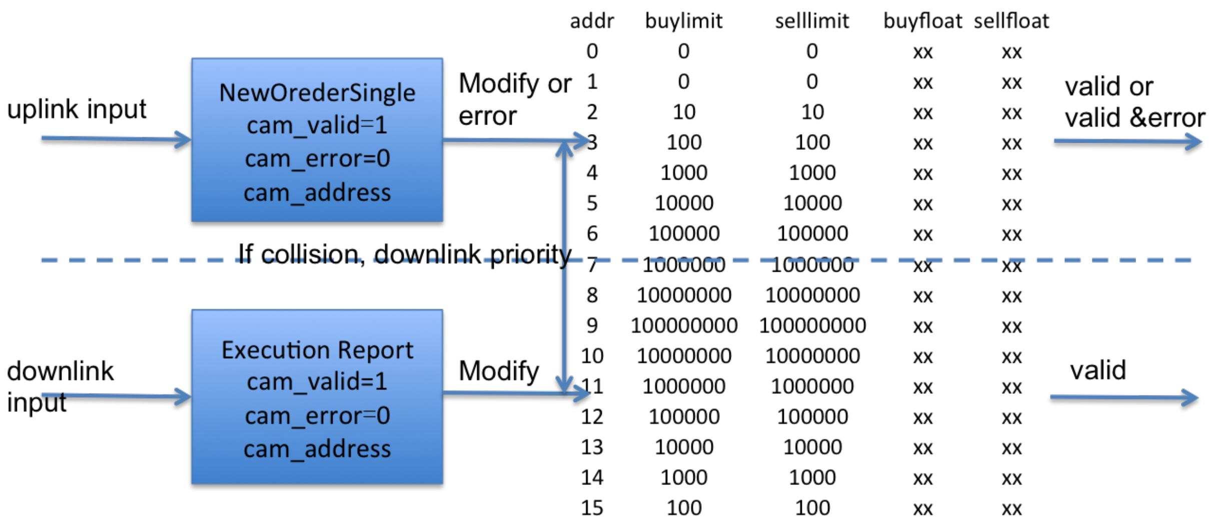
Every time an uplink FIX message comes in, with message type of NewOrderSingle, side buy or sell, valid and not error ID combination address and symbol address, we add the order quantity of this message with the floating quantity that has been stored, and compare this new floating quantity with the floating quantity limit stored in the corresponding address. If the new floating quantity exceeds the limit, floating quantity in the “RAM” stays the same, the valid and error signal are sent to the controller. If the new floating quantity doesn’t exceed the limit, this new quantity is stored in the place of the corresponding old floating quantity, only valid signal is sent to controller this time.

Every time an downlink FIX message comes in, with message type of Execution Report (35=8), side buy or sell, valid and not error ID combination address and symbol address, we subtract the floating quantity of the corresponding address with the filled contract quantity (tag 32), and store the new floating quantity in the place of the old floating quantity. This way, next time the uplink message accesses the module, it will see the modified floating quantity. When this is done, the module would send valid signal to the downlink controller.

As the uplink and downlink messages could both access the same address of the “RAM”, there’s chance that they access the same address in the same clock cycle, thus collision happens. To prevent this collision, we simply let the downlink message get the priority to access the

"RAM" before the uplink message, so that the uplink message is judged by the newest floating contract number.

Also, the uplink/downlink output valid and error signals are refreshed every time a new FIX message goes in, indicated by message type input from uplink/downlink preprocessor being 16'd0. The module is shown in Figure 5.3.1.



addr	buylimit	selllimit	buyfloat	sellfloat
0	0	0	xx	xx
1	0	0	xx	xx
2	10	10	xx	xx
3	100	100	xx	xx
4	1000	1000	xx	xx
5	10000	10000	xx	xx
6	100000	100000	xx	xx
7	1000000	1000000	xx	xx
8	10000000	10000000	xx	xx
9	100000000	100000000	xx	xx
10	10000000	10000000	xx	xx
11	1000000	1000000	xx	xx
12	100000	100000	xx	xx
13	10000	10000	xx	xx
14	1000	1000	xx	xx
15	100	100	xx	xx

Figure 5.3.1

5.4 Decision Controller

Decision Controller receives input from the two cams and the three rule-checking modules. Every time the message type value from preprocessor turns 0, the decision controller knows that a new FIX message has come in, and begin waiting for results from the previous modules. There 5 situations that we could encounter here:

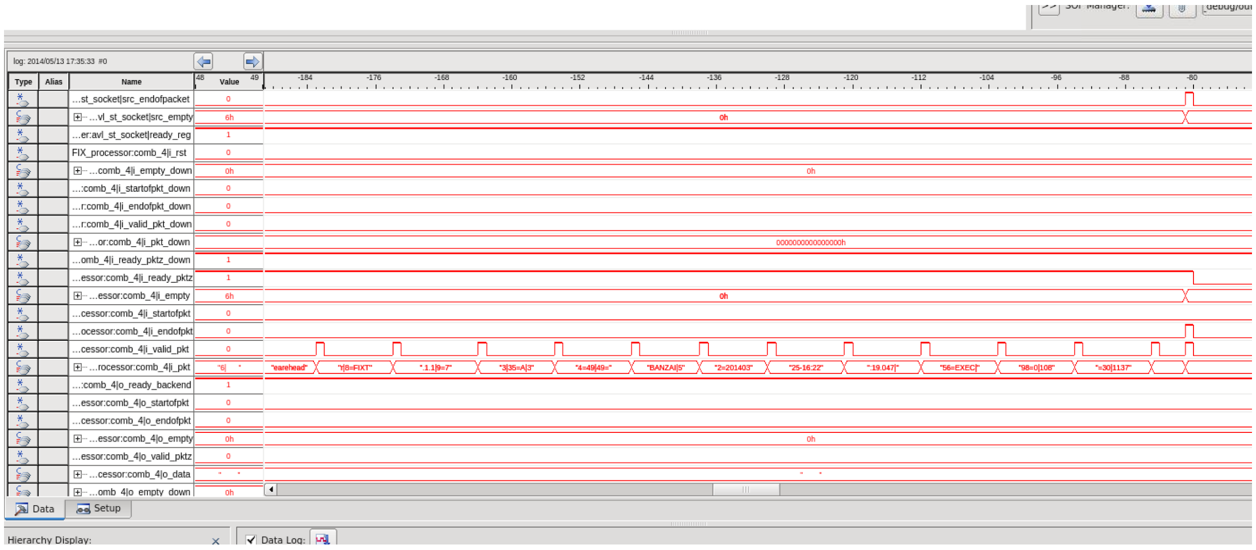
- The FIX message is simply not a NewOrderSingle message, and the ID combination and symbol are both legal, so the controller gives valid and no error signal to the packetizer.
- The FIX message is not a NewOrderSingle message, but the ID combination or symbol is illegal, so the controller gives valid and error signal to the packetizer.
- The FIX message is a NewOrderSingle message, and the ID combination and symbol are both legal, all the three criterions are met. So the controller gives valid and no error signal to the packetizer.
- The FIX message is a NewOrderSingle message, but the ID combination or symbol is illegal, so the controller gives valid and error signal to the packetizer.
- The FIX message is a NewOrderSingle message, and the ID combination and symbol are both legal, but one or more of the three criterions are not met. So the controller gives valid and error signal to the packetizer.

6. Testing and Results

Before implementing the whole project on FPGA, we first did the Modelsim Simulation, testbench will be shown in Appendix.

To test the whole project in FPGA, we use Avalon MM-ST to read binary data from .txt document and stream in data to our system.

- Testcase 1: Incoming message is not FIX message, so the message passes the system without going into the parser.
- Testcase 2: Incoming FIX message has prohibited ID combination, so the message is dropped whether or not the three criterions are met. Test result for this case is shown in Figure 6.1 and Figure 6.2.
- Testcase 3: Incoming FIX message has legal ID combination and symbol, but order quantity exceeds the limit, so the packet is dropped.
- Testcase 4: The first FIX message satisfies all the rules at the beginning and it can get through. But after keeping sending the same message for several times, the number of floating contracts exceeds the limit and the same message will be dropped time. If we receive the execution report in downlink showing that some amount of contracts got filled, the floating number will decrease and the same message can go through the next time it is sent. Test result for this case is shown in Figure 6.3 through Figure 6.6.



log: 2014/05/13 17:35:33 #0

Type	Alias	Name	Value
		...st_socket|src_endofpacket	0
		...vl_st_socket|src_empty	6h
		...er:avl_st_socket|ready_reg	1
		FIX_processor:comb_4|i_rst	0
		...comb_4|i_empty_down	0h
		...:comb_4|i_startofpkt_down	0
		...r:comb_4|i_endofpkt_down	0
		...r:comb_4|i_valid_pkt_down	0
		...or:comb_4|i_pkt_down	
		...omb_4|i_ready_pktz_down	1
		...essor:comb_4|i_ready_pktz	1
		...essor:comb_4|i_empty	6h
		...cessor:comb_4|i_startofpkt	0
		...ocessor:comb_4|i_endofpkt	0
		...cessor:comb_4|i_valid_pkt	0
		...rocessor:comb_4|i_pkt	"6| "
		...:comb_4|o_ready_backend	1
		...essor:comb_4|o_startofpkt	0
		...cessor:comb_4|o_endofpkt	0
		...essor:comb_4|o_empty	0h
		...essor:comb_4|o_valid_pktz	0
		...cessor:comb_4|o_data	" "
		...omb_4|o_empty_down	0h

Figure 6.1 Testcase 2: Input to packetizer

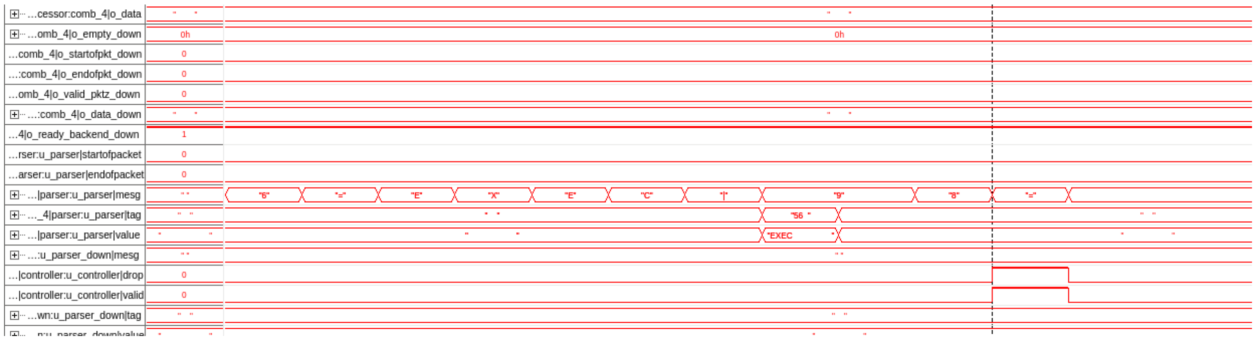


Figure 6.2 Testcase 2: Decision Output

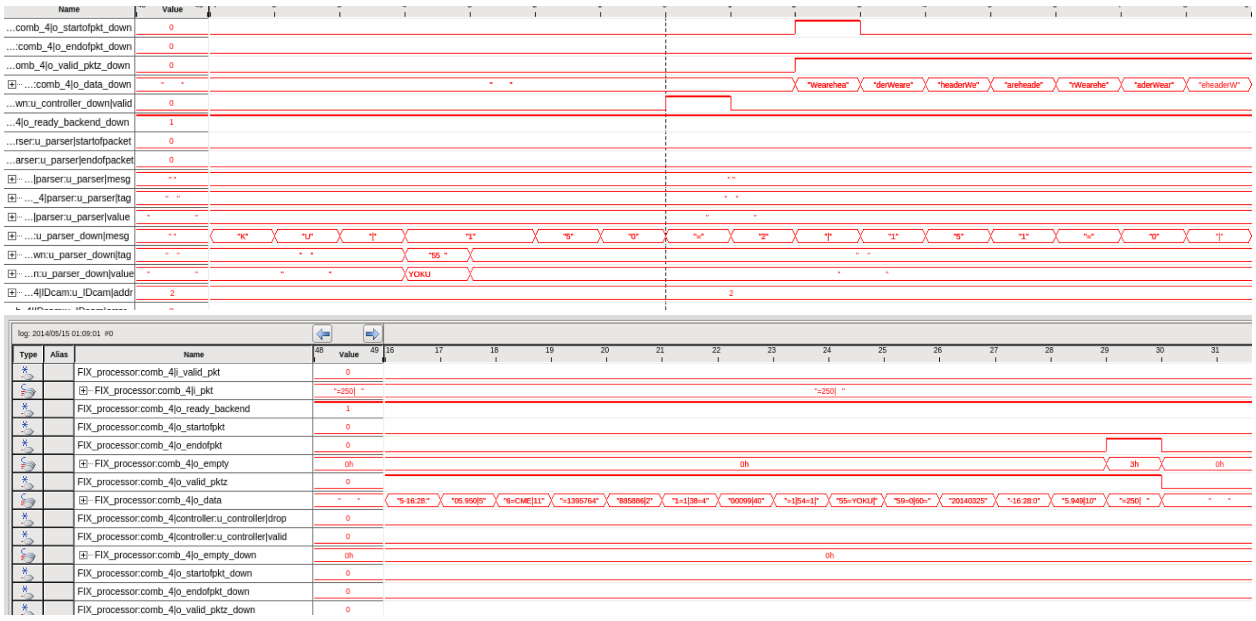

Name
Value
…comb_4|o_startofpkt_down
…:comb_4|o_endofpkt_down
…omb_4|o_valid_pktz_down
…:comb_4|o_data_down
…wn:u_controller_down|valid
…4|o_ready_backend_down
…rser:u_parser|startofpacket
…arser:u_parser|endofpacket
…|parser:u_parser|mesg
…_4|parser:u_parser|tag
…|parser:u_parser|value
…:u_parser_down|mesg
…wn:u_parser_down|tag
…n:u_parser_down|value
…4|IDcam:u_IDcam|addr
log: 2014/05/15 01:09:01 #0
Type
Alias
Name
Value
FIX_processor:comb_4|i_valid_pkt
FIX_processor:comb_4|i_pkt
FIX_processor:comb_4|o_ready_backend
FIX_processor:comb_4|o_startofpkt
FIX_processor:comb_4|o_endofpkt
FIX_processor:comb_4|o_empty
FIX_processor:comb_4|o_valid_pktz
FIX_processor:comb_4|o_data
FIX_processor:comb_4|controller:u_controller|drop
FIX_processor:comb_4|controller:u_controller|valid
FIX_processor:comb_4|o_empty_down
FIX_processor:comb_4|o_startofpkt_down
FIX_processor:comb_4|o_endofpkt_down
FIX_processor:comb_4|o_valid_pktz_down

Figure 6.3 Testcase 4: Message first got passed through the system

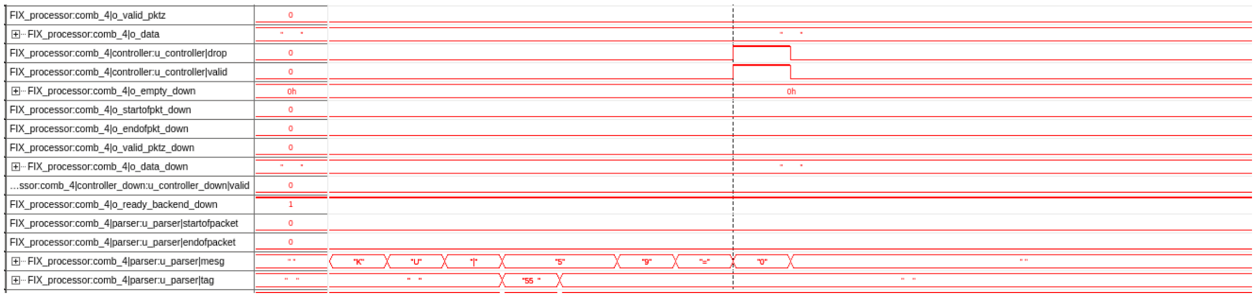

FIX_processor:comb_4|o_valid_pktz
FIX_processor:comb_4|o_data
FIX_processor:comb_4|controller:u_controller|drop
FIX_processor:comb_4|controller:u_controller|valid
FIX_processor:comb_4|o_empty_down
FIX_processor:comb_4|o_startofpkt_down
FIX_processor:comb_4|o_endofpkt_down
FIX_processor:comb_4|o_valid_pktz_down
FIX_processor:comb_4|o_data_down
…ssor:comb_4|controller_down:u_controller_down|valid
FIX_processor:comb_4|o_ready_backend_down
FIX_processor:comb_4|parser:u_parser|startofpacket
FIX_processor:comb_4|parser:u_parser|endofpacket
FIX_processor:comb_4|parser:u_parser|mesg
FIX_processor:comb_4|parser:u_parser|tag

Figure 6.4 Testcase 4: After it got through twice, the message is dropped when we try to send it out for the third time. We can see that the valid and drop signal from the controller are both high and there is nothing going out from the packetizer to the backend.

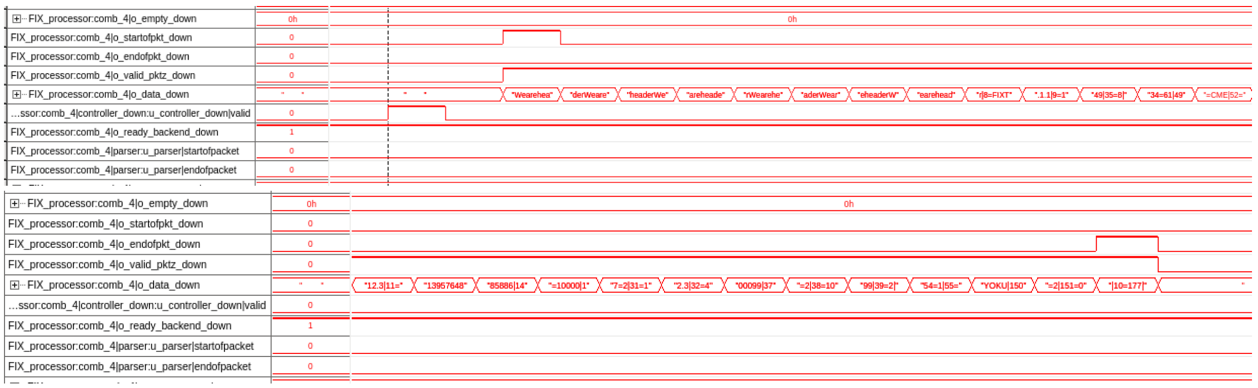

FIX_processor:comb_4|o_empty_down
FIX_processor:comb_4|o_startofpkt_down
FIX_processor:comb_4|o_endofpkt_down
FIX_processor:comb_4|o_valid_pktz_down
FIX_processor:comb_4|o_data_down
...ssor:comb_4|controller_down:u_controller_down|valid
FIX_processor:comb_4|o_ready_backend_down
FIX_processor:comb_4|parser:u_parser|startofpacket
FIX_processor:comb_4|parser:u_parser|endofpacket
"Wearehea"
"derWeare"
"headerWe"
"areheade"
"rWearehe"
"aderWear"
"eheaderW"
"earehead"
"r|8=FIXT"
".1.1|9=1"
"49|35=8|"
"34=61|49"
"=CME|52="
"12.3|11="
"13957648"
"85886|14"
"=10000|1"
"7=2|31=1"
"2.3|32=4"
"00099|37"
"=2|38=10"
"99|39=2|"
"54=1|55="
"YOKU|150"
"=2|151=0"
"|10=177|"

Figure 6.5 Testcase 4: Execution report shows that some contracts got filled

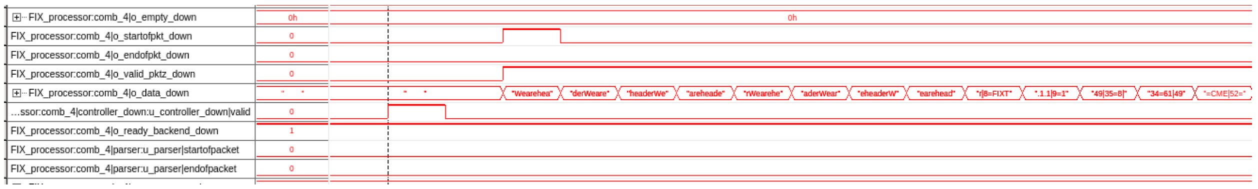

FIX_processor:comb_4|o_empty_down
FIX_processor:comb_4|o_startofpkt_down
FIX_processor:comb_4|o_endofpkt_down
FIX_processor:comb_4|o_valid_pktz_down
FIX_processor:comb_4|o_data_down
...ssor:comb_4|controller_down:u_controller_down|valid
FIX_processor:comb_4|o_ready_backend_down
FIX_processor:comb_4|parser:u_parser|startofpacket
FIX_processor:comb_4|parser:u_parser|endofpacket
"Wearehea"
"derWeare"
"headerWe"
"areheade"
"rWearehe"
"aderWear"
"eheaderW"
"earehead"
"r|8=FIXT"
".1.1|9=1"
"49|35=8|"
"34=61|49"
"=CME|52="

Figure 6.6 Testcase 4: Then the previous rejected message can pass the system again since the number of floating transaction is lower than the limit

7. Experiences

To do this project, we first learned the basic concepts of the FIX protocol and then designed the structure of the parsing and processing system. As there's not much previous work to refer to, the design part is fairly important. Then we gradually designed and verified functionality of each component. Different from other projects, our whole project is based on hardware and SystemVerilog.

There're still many parts of the projects that can be improved. For example, we are not able to send packets continuously in high speed in hardware, and that limits our text for the first criterion. Also, we know that output to signaltap is not that user-friendly, wish that we could output it through software if time allows.

Our project is extendable because every module is independent and can be reused for different cases. For now we are just caring about "Side" of buy and sell, but other kinds like short and short exempt can also be considered. Also, there are many other rules that can be implemented as long as we add more modules to deal with other tags and values coming from the preprocessor.

8. Thanks

Thanks to Prof. Stephen Edwards for the great lectures and encouragement for the whole semester.
Thanks to Prof. David Lariviere for offering the wonderful project ideas and guidance all the way through the project.

Thanks to TA Qiushi Ding for the Avalon MM-ST input adjustment and helping us building the structure of the project.

Reference

[1] *Financial Information eXchange*,
http://en.wikipedia.org/wiki/Financial_information_exchange;
[2] *Avalon Interface Specifications*,
http://www.altera.com/literature/manual/mnl_avalon_spec.pdf.

Appendix: Original Codes

Modelsim testbench:

```
`timescale 1ns/1ps

module tb_FIX_processor();


logic      i_clk;
logic      i_rst;
logic      i_validpkt;
logic[2:0] i_empty;
logic      i_startofpkt;
logic      i_endofpkt;
logic[63:0]     i_pkt;
logic      i_ready_backend;
logic      i_validpkt_down;
logic[2:0] i_empty_down;
logic      i_startofpkt_down;
logic      i_endofpkt_down;
logic[63:0]     i_pkt_down;
logic      i_ready_backend_down;

logic      o_ready_pktz;
logic      o_valid_pktz;
logic[2:0] o_empty;
logic      o_startofpkt;
logic      o_endofpkt;
logic[63:0]     o_data;
logic      o_ready_pktz_down;
logic      o_valid_pktz_down;
logic[2:0] o_empty_down;
logic      o_startofpkt_down;
logic      o_endofpkt_down;
logic[63:0]     o_data_down;

integer counter4=0;
integer counter5=0;
integer counter6=0;
integer counter7=0;
integer counter8=0;
integer counter9=0;
integer m0,m1,m2,m3,m4,r0,r1,r2,r3,r4;
integer i,j,k,m,n,q;

//logic [7:0] message0 [0:167];
logic [7:0] message1 [0:223];
//logic [7:0] message2 [0:223];
logic [7:0] message8 [0:207];
logic [7:0] message3 [0:239];

//logic [63:0] message4 [0:20];
logic [63:0] message5 [0:27];
//logic [63:0] message6 [0:27];
logic [63:0] message7 [0:29];
logic [63:0] message9 [0:25];


FIX_processor FIX_processor_1(.i_clk(i_clk), .i_rst(i_rst),
.i_valid_pkt(i_validpkt), .i_startofpkt(i_startofpkt),
.i_endofpkt(i_endofpkt), .i_pkt(i_pkt),
.i_ready_backend(i_ready_backend), .i_empty(i_empty),
.i_valid_pkt_down(i_validpkt_down),
.i_startofpkt_down(i_startofpkt_down),
.i_endofpkt_down(i_endofpkt_down), .i_pkt_down(i_pkt_down),
.i_ready_backend_down(i_ready_backend_down),
.i_empty_down(i_empty_down), .o_ready_pktz(o_ready_pktz),
.o_valid_pktz(o_valid_pktz), .o_startofpkt(startofpkt),
.o_endofpkt(o_endofpkt), .o_data(o_data), .o_empty(o_empty),
.o_ready_pktz_down(o_ready_pktz_down),
.o_valid_pktz_down(o_valid_pktz_down),
.o_startofpkt_down(o_startofpkt_down),
.o_endofpkt_down(o_endofpkt_down), .o_data_down(o_data_down),
.o_empty_down(o_empty_down));

    initial
    begin
      i_clk = 0;
      i_rst = 0;
      i_validpkt = 0;
      i_startofpkt = 0;
      i_endofpkt = 0;
      i_pkt=64'd0;
      i_validpkt_down = 0;
      i_startofpkt_down = 0;
      i_endofpkt_down = 0;
      i_pkt_down=64'd0;
      i_empty=3'd0;
```

```
	i_empty_down=3'd0;
	counter4=0;
	counter5=0;
	counter6=0;

	/*m0 = $fopen("nrs0.txt","r");
	for (i = 0 ; i < 168; i++) begin
		r0=$fscanf(m0,"%c",message0[i]);
	end*/
	m1 = $fopen("nrs1.txt","r");
	for (j = 0 ; j < 224; j++) begin
		r1=$fscanf(m1,"%c",message1[j]);
	end

	/*m2 = $fopen("nrs2.txt","r");
	for (k = 0 ; k < 224; k++) begin
		r2=$fscanf(m2,"%c",message2[k]);
	end*/

	m3 = $fopen("execreport1.txt","r");
	for (k = 0 ; k < 240; k++) begin
		r3=$fscanf(m3,"%c",message3[k]);
	end

	m4 = $fopen("execreport0.txt","r");
	for (k = 0 ; k < 209; k++) begin
		r4=$fscanf(m4,"%c",message8[k]);
	end

	/*for (m = 0; m < 168; m=m+8) begin
		message4[m/8] =
{message0[m],message0[m+1],message0[m+2],message0[m+3],message0[m+4],m
essage0[m+5],message0[m+6],message0[m+7]};
	end*/
	for (n = 0; n < 224; n=n+8) begin
		message5[n/8] =
{message1[n],message1[n+1],message1[n+2],message1[n+3],message1[n+4],m
essage1[n+5],message1[n+6],message1[n+7]};
	end
	/*for (q = 0; q < 224; q=q+8) begin
		message6[q/8] =
{message2[q],message2[q+1],message2[q+2],message2[q+3],message2[q+4],m
essage2[q+5],message2[q+6],message2[q+7]};
	end*/

	for (q = 0; q < 240; q=q+8) begin
		message7[q/8] =
{message3[q],message3[q+1],message3[q+2],message3[q+3],message3[q+4],m
essage3[q+5],message3[q+6],message3[q+7]};
	end
```

```
      for (q = 0; q < 209; q=q+8) begin
            message9[q/8] =
{message8[q],message8[q+1],message8[q+2],message8[q+3],message8[q+4],m
essage8[q+5],message8[q+6],message3[q+7]};
      end

      #1 i_rst = 1;

      #50 i_rst=0;
      #100 i_ready_backend=1;i_ready_backend_down=1;




    end

always #25 i_clk = ~i_clk;

    always_ff@(posedge i_clk) begin
      if(i_rst)begin
            i_clk <= 0;
            i_rst <= 0;
            i_validpkt <= 0;
            i_startofpkt <= 0;
            i_endofpkt <= 0;
            i_pkt<=64'd0;
            i_validpkt_down <= 0;
            i_startofpkt_down <= 0;
            i_endofpkt_down <= 0;
            i_pkt_down<=64'd0;
            i_empty<=3'd0;
            i_empty_down<=3'd0;
            counter4<=0;
            counter5<=0;
            counter6<=0;

      end
      else begin
            if(o_ready_pktz==1) begin

                  if(counter4==0) begin
                        i_startofpkt<=1;
                        i_validpkt<=1;
                        i_pkt<=message5[counter4];
                        counter4<=counter4+1;
                  end
                  else if(counter4<27) begin
                        i_startofpkt<=0;
                        i_validpkt<=1;
                        i_pkt<=message5[counter4];
                        counter4<=counter4+1;
                  end
                  else if(counter4==27) begin
                        i_endofpkt<=1;
                        i_validpkt<=1;
                        i_pkt<=message5[counter4];
                        counter4<=counter4+1;
                        i_empty<=3'd3;
                  end

                  else if(counter5==0) begin
                        i_empty<=3'd0;
                        i_endofpkt<=0;
                        i_startofpkt<=1;
                        i_validpkt<=1;
                        i_pkt<=message5[counter5];
                        counter5<=counter5+1;
                  end
                  else if(counter5<27) begin
                        i_startofpkt<=0;
                        i_validpkt<=1;
                        i_pkt<=message5[counter5];
                        counter5<=counter5+1;
                  end
                  else if(counter5==27) begin
                        i_endofpkt<=1;
                        i_validpkt<=1;
                        i_pkt<=message5[counter5];
                        counter5<=counter5+1;
                        i_empty<=3'd3;
                  end

                  else if(counter6==0) begin
                        i_empty<=3'd0;
                        i_endofpkt<=0;
                        i_startofpkt<=1;
                        i_validpkt<=1;
                        i_pkt<=message5[counter6];
                        counter6<=counter6+1;
                  end
                  else if(counter6<27) begin
                        i_startofpkt<=0;
                        i_validpkt<=1;
                        i_pkt<=message5[counter6];
                        counter6<=counter6+1;
                  end
                  else if(counter6==27) begin
                        i_endofpkt<=1;
                        i_validpkt<=1;
                        i_pkt<=message5[counter6];
```

```
        counter6<=counter6+1;
        i_empty<=3'd3;
end
else if(counter7<100) begin
        i_empty<=3'd0;
        i_endofpkt<=0;
        i_validpkt<=0;
        i_pkt<=64'd0;

        counter7<=counter7+1;

end
/*else if(counter7==100) begin

        i_startofpkt_down<=1;
        i_validpkt_down<=1;
        i_pkt_down<=message9[counter7-100];

        counter7<=counter7+1;
end
else if(counter7<125) begin
        i_startofpkt_down<=0;
        i_validpkt_down<=1;
        i_pkt_down<=message9[counter7-100];

        counter7<=counter7+1;

end
else if(counter7==125) begin
        i_endofpkt_down<=1;
        i_validpkt_down<=1;
        i_empty_down<=3'd0;
        i_pkt_down<=message9[counter7-100];
        i_empty_down<=64'd0;
        counter7<=counter7+1;
end
*/

else if(counter8==0) begin
        i_empty_down<=3'd0;
        i_endofpkt_down<=0;

        i_startofpkt_down<=1;
        i_validpkt_down<=1;
        i_pkt_down<=message7[counter8];
        counter8<=counter8+1;
end

else if(counter8<29) begin
        i_startofpkt_down<=0;
```

```
      i_validpkt_down<=1;
      i_pkt_down<=message7[counter8];

      counter8<=counter8+1;

end
else if(counter8==29) begin
      i_endofpkt_down<=1;
      i_validpkt_down<=1;
      i_empty_down<=3'd5;
      i_pkt_down<=message7[counter8];
      i_empty_down<=64'd0;
      counter8<=counter8+1;
end

else if(counter9<100)begin
      i_empty_down<=3'd0;
      i_endofpkt_down<=0;
      i_validpkt_down<=0;
      counter9<=counter9+1;
end

else if(counter9==100) begin

      i_startofpkt<=1;
      i_validpkt<=1;
      i_pkt<=message5[counter9-100];
      counter9<=counter9+1;
end
else if(counter9<127) begin
      i_startofpkt<=0;
      i_validpkt<=1;
      i_pkt<=message5[counter9-100];
      counter9<=counter9+1;
end
else if(counter9==127) begin
      i_endofpkt<=1;
      i_validpkt<=1;
      i_pkt<=message5[counter9-100];
      counter9<=counter9+1;
      i_empty<=3'd3;
end




else       begin
i_empty<=3'd0;
i_endofpkt<=0;
```

```
                i_validpkt<=0;
                end
            end

        end
end


endmodule
```

Uplink Packetizer:

```
module packetizer( input logic clk, reset,
        input logic readybackend, readypar,
validpkt,startofpkt,endofpkt,validswitch,drop,
        input logic [2:0] empty,
        input logic [63:0] pkt,
        output logic validtopar,validtobackend,
readypktz,startofpacket,endofpacket,sop,eop,
        output logic [7:0] datatopar,
        output logic [2:0] o_empty,
        output logic [63:0] data

);
    logic flag,flag2,flag3,flag4;
    logic [7:0] FIFO [0:255];
    logic [2:0] Empty;
    integer counter0=0, counter1=66, counter2=0,i;

    initial begin
     for(i=0;i<256;i++)begin
           FIFO[i]=8'd0;
     end
     Empty=3'd0;
     o_empty=3'd0;
     readypktz=1;
     validtopar=0;
     validtobackend=0;
     startofpacket=0;
     endofpacket=0;
     sop=0;
     eop=0;
     datatopar=8'd0;
     data=64'd0;
     flag=0;
     flag2=0;
     flag3=0;
     flag4=0;
    end
    always_ff@(posedge clk) begin
        if(reset)begin

      for(i=0;i<256;i++)begin
           FIFO[i]=8'd0;
      end
      Empty<=3'd0;
      o_empty<=3'd0;
      flag<=0;
      flag2<=0;
      flag3<=0;
      flag4<=0;
      readypktz<=1;
      validtopar<=0;
      validtobackend<=0;
      startofpacket<=0;
      endofpacket<=0;
      sop<=0;
      eop<=0;
      datatopar<=8'd0;
      data<=64'd0;
      counter0<=0;
      counter1<=66;
      counter2<=0;

         end

         else begin
      if(validpkt) begin
        if(startofpkt && counter0==0)begin

           FIFO[counter0]<=pkt[63:56];
           FIFO[counter0+1]<=pkt[55:48];
           FIFO[counter0+2]<=pkt[47:40];
           FIFO[counter0+3]<=pkt[39:32];
                      FIFO[counter0+4]<=pkt[31:24];
           FIFO[counter0+5]<=pkt[23:16];
           FIFO[counter0+6]<=pkt[15:8];
           FIFO[counter0+7]<=pkt[7:0];
           counter0<=counter0+8;
           flag<=1;
        end
        if(flag) begin

           /*{FIFO[counter0-56],FIFO[counter0-55],FIFO[counter0-
54],FIFO[counter0-53],FIFO[counter0-52],FIFO[counter0-
51],FIFO[counter0-50],FIFO[counter0-49]}<=pkt;*/
           if(empty==3'd0)begin
                FIFO[counter0]<=pkt[63:56];
```

```
          FIFO[counter0+1]<=pkt[55:48];
          FIFO[counter0+2]<=pkt[47:40];
          FIFO[counter0+3]<=pkt[39:32];
               FIFO[counter0+4]<=pkt[31:24];
          FIFO[counter0+5]<=pkt[23:16];
          FIFO[counter0+6]<=pkt[15:8];
          FIFO[counter0+7]<=pkt[7:0];
     end
     else if(empty==3'd1) begin
          FIFO[counter0]<=pkt[63:56];
          FIFO[counter0+1]<=pkt[55:48];
          FIFO[counter0+2]<=pkt[47:40];
          FIFO[counter0+3]<=pkt[39:32];
               FIFO[counter0+4]<=pkt[31:24];
          FIFO[counter0+5]<=pkt[23:16];
          FIFO[counter0+6]<=pkt[15:8];
     end
     else if(empty==3'd2) begin
          FIFO[counter0]<=pkt[63:56];
          FIFO[counter0+1]<=pkt[55:48];
          FIFO[counter0+2]<=pkt[47:40];
          FIFO[counter0+3]<=pkt[39:32];
               FIFO[counter0+4]<=pkt[31:24];
          FIFO[counter0+5]<=pkt[23:16];
     end
     else if(empty==3'd3) begin
            FIFO[counter0]<=pkt[63:56];
          FIFO[counter0+1]<=pkt[55:48];
          FIFO[counter0+2]<=pkt[47:40];
          FIFO[counter0+3]<=pkt[39:32];
               FIFO[counter0+4]<=pkt[31:24];
     end
     else if(empty==3'd4) begin
          FIFO[counter0]<=pkt[63:56];
          FIFO[counter0+1]<=pkt[55:48];
          FIFO[counter0+2]<=pkt[47:40];
          FIFO[counter0+3]<=pkt[39:32];
     end
     else if(empty==3'd5) begin
          FIFO[counter0]<=pkt[63:56];
          FIFO[counter0+1]<=pkt[55:48];
          FIFO[counter0+2]<=pkt[47:40];
     end
     else if(empty==3'd6) begin
          FIFO[counter0]<=pkt[63:56];
          FIFO[counter0+1]<=pkt[55:48];
     end
     else    FIFO[counter0]<=pkt[63:56];
     counter0<=counter0+8;
```

```
       end
       if(endofpkt) begin
          Empty<=empty;
          if(counter0<66) flag4<=1;
          else if(FIFO[66]!=8'h38 || FIFO[67]!=8'h3d ||
FIFO[68]!=8'h46 || FIFO[69]!=8'h49 || FIFO[70]!=8'h58)
               flag4<=1;

          else begin

                         flag2<=1;
          end
          flag<=0;
          readypktz<=0;
       end
     end

     if(flag2 && readypar) begin /*original:counter0>64*/
          if(counter1>65 && counter1<counter0) begin

               if(counter1==66) begin //start
                    startofpacket<=1;
               end
               else if(counter1==counter0-1) begin //end
                    endofpacket<=1;
               end
               else begin
                    startofpacket<=0;
                    endofpacket<=0;
               end
               validtopar<=1;
               datatopar<=FIFO[counter1];
               counter1<=counter1+1;
          end

          else begin
               endofpacket<=0;
               startofpacket<=0;
               validtopar<=0;
               datatopar<=8'd0;
          end
     end
     if(validswitch && !drop) begin
          flag3<=1;
     end
     if((flag2 && readybackend && flag3) || flag4) begin
/*original:counter0>64*/
          if(counter2<counter0) begin

               if(counter2==0) begin //start
```

```
                        sop<=1;
                   end
                   else if(counter2==counter0-8) begin //end
                        eop<=1;
                        o_empty<=Empty;
                   end
                   else begin
                        sop<=0;
                        eop<=0;
                   end
                   validtobackend<=1;

    /*datatoswitch<={FIFO[counter2],FIFO[counter2+1],FIFO[counter2+1],
FIFO[counter2+3],FIFO[counter2+4],FIFO[counter2+5],FIFO[counter2+6],FI
FO[counter2+7]}*/
                   data[63:56]<=FIFO[counter2];
                   data[55:48]<=FIFO[counter2+1];
                   data[47:40]<=FIFO[counter2+2];
                   data[39:32]<=FIFO[counter2+3];
                   data[31:24]<=FIFO[counter2+4];
                   data[23:16]<=FIFO[counter2+5];
                   data[15:8]<=FIFO[counter2+6];
                   data[7:0]<=FIFO[counter2+7];

                   counter2<=counter2+8;
              end

              else begin
                   Empty<=3'd0;
                   o_empty<=3'd0;
                   flag4<=0;
                   flag3<=0;
                               flag2<=0;
                   readypktz<=1;
                   sop<=0;
                   eop<=0;
                   validtobackend<=0;
                   data<=64'd0;
                   for(i=0;i<256;i++)begin
                   FIFO[i]=8'd0;
                   end
                   counter0<=0;
                   counter1<=66;
                   counter2<=0;

              end
         end

         if(validswitch && drop && !flag4) begin
              for(i=0;i<256;i++)begin
```

```
            FIFO[i]=8'd0;
            end
            Empty<=3'd0;
            o_empty<=3'd0;
            flag<=0;
                       flag2<=0;
            flag3<=0;
            readypktz<=1;
            validtopar<=0;
            validtobackend<=0;
            startofpacket<=0;
            endofpacket<=0;
            sop<=0;
            eop<=0;
            datatopar<=8'd0;
            data<=64'd0;
            counter0<=0;
            counter1<=66;
            counter2<=0;
      end

         end
    end


endmodule
```

Downlink Packetizer

```
module packetizer_down( input logic clk, reset,
      input logic readybackend, readypar,
validpkt,startofpkt,endofpkt,validswitch,
      input logic [2:0] empty,
      input logic [63:0] pkt,
      output logic validtopar,validtobackend,
readypktz,startofpacket,endofpacket,sop,eop,
      output logic [7:0] datatopar,
      output logic [2:0] o_empty,
      output logic [63:0] data

);
    logic flag,flag2,flag3,flag4;
    logic [7:0] FIFO [0:255];
    logic [2:0] Empty;
    integer counter0=0, counter1=66, counter2=0,i;

    initial begin
      for(i=0;i<256;i++)begin
            FIFO[i]=8'd0;
      end
```

```
  Empty=3'd0;
  o_empty=3'd0;
  readypktz=1;
  validtopar=0;
  validtobackend=0;
  startofpacket=0;
  endofpacket=0;
  sop=0;
  eop=0;
  datatopar=8'd0;
  data=64'd0;
  flag=0;
  flag2=0;
  flag3=0;
  flag4=0;

end
always_ff@(posedge clk) begin
    if(reset)begin

  for(i=0;i<256;i++)begin
       FIFO[i]=8'd0;
  end
  Empty<=3'd0;
  o_empty<=3'd0;
  flag<=0;
  flag2<=0;
  flag3<=0;
  flag4<=0;
  readypktz<=1;
  validtopar<=0;
  validtobackend<=0;
  startofpacket<=0;
  endofpacket<=0;
  sop<=0;
  eop<=0;
  datatopar<=8'd0;
  data<=64'd0;
  counter0<=0;
  counter1<=66;
  counter2<=0;

    end

    else begin
  if(validpkt) begin
    if(startofpkt && counter0==0)begin

       FIFO[counter0]<=pkt[63:56];
       FIFO[counter0+1]<=pkt[55:48];
```

```
            FIFO[counter0+2]<=pkt[47:40];
            FIFO[counter0+3]<=pkt[39:32];
                        FIFO[counter0+4]<=pkt[31:24];
            FIFO[counter0+5]<=pkt[23:16];
            FIFO[counter0+6]<=pkt[15:8];
            FIFO[counter0+7]<=pkt[7:0];
            counter0<=counter0+8;
            flag<=1;
        end
        if(flag) begin

            if(empty==3'd0)begin
                  FIFO[counter0]<=pkt[63:56];
                  FIFO[counter0+1]<=pkt[55:48];
                  FIFO[counter0+2]<=pkt[47:40];
                  FIFO[counter0+3]<=pkt[39:32];
                        FIFO[counter0+4]<=pkt[31:24];
                  FIFO[counter0+5]<=pkt[23:16];
                  FIFO[counter0+6]<=pkt[15:8];
                  FIFO[counter0+7]<=pkt[7:0];
            end
            else if(empty==3'd1) begin
                  FIFO[counter0]<=pkt[63:56];
                  FIFO[counter0+1]<=pkt[55:48];
                  FIFO[counter0+2]<=pkt[47:40];
                  FIFO[counter0+3]<=pkt[39:32];
                        FIFO[counter0+4]<=pkt[31:24];
                  FIFO[counter0+5]<=pkt[23:16];
                  FIFO[counter0+6]<=pkt[15:8];
            end
            else if(empty==3'd2) begin
                  FIFO[counter0]<=pkt[63:56];
                  FIFO[counter0+1]<=pkt[55:48];
                  FIFO[counter0+2]<=pkt[47:40];
                  FIFO[counter0+3]<=pkt[39:32];
                        FIFO[counter0+4]<=pkt[31:24];
                  FIFO[counter0+5]<=pkt[23:16];
            end
            else if(empty==3'd3) begin
                     FIFO[counter0]<=pkt[63:56];
                  FIFO[counter0+1]<=pkt[55:48];
                  FIFO[counter0+2]<=pkt[47:40];
                  FIFO[counter0+3]<=pkt[39:32];
                        FIFO[counter0+4]<=pkt[31:24];
            end
            else if(empty==3'd4) begin
                  FIFO[counter0]<=pkt[63:56];
                  FIFO[counter0+1]<=pkt[55:48];
                  FIFO[counter0+2]<=pkt[47:40];
                  FIFO[counter0+3]<=pkt[39:32];
            end
            else if(empty==3'd5) begin
                 FIFO[counter0]<=pkt[63:56];
                 FIFO[counter0+1]<=pkt[55:48];
                 FIFO[counter0+2]<=pkt[47:40];
            end
            else if(empty==3'd6) begin
                 FIFO[counter0]<=pkt[63:56];
                 FIFO[counter0+1]<=pkt[55:48];
            end
            else    FIFO[counter0]<=pkt[63:56];
            counter0<=counter0+8;

        end
        if(endofpkt) begin
            if(counter0<66) flag4<=1;
            else if(FIFO[66]!=8'h38 || FIFO[67]!=8'h3d ||
FIFO[68]!=8'h46 || FIFO[69]!=8'h49 || FIFO[70]!=8'h58)
                 flag4<=1;

            else begin

                           flag2<=1;
            end
            flag<=0;
            readypktz<=0;
        end
      end

      if(flag2 && readypar) begin /*original:counter0>64*/
            if(counter1>65 && counter1<counter0) begin

                 if(counter1==66) begin //start
                      startofpacket<=1;
                 end
                 else if(counter1==counter0-1) begin //end
                      endofpacket<=1;
                 end
                 else begin
                      startofpacket<=0;
                      endofpacket<=0;
                 end
                 validtopar<=1;
                 datatopar<=FIFO[counter1];
                 counter1<=counter1+1;
            end

            else begin
                 endofpacket<=0;
                 startofpacket<=0;
```

```
                validtopar<=0;
                datatopar<=8'd0;
            end
        end
        if(validswitch) begin
            flag3<=1;
        end
        if((flag2 && readybackend && flag3) || flag4) begin
/*original:counter0>64*/
            if(counter2<counter0) begin

                if(counter2==0) begin //start
                    sop<=1;
                end
                else if(counter2==counter0-8) begin //end
                    o_empty<=Empty;
                    eop<=1;
                end
                else begin
                    sop<=0;
                    eop<=0;
                end
                validtobackend<=1;

    /*datatoswitch<={FIFO[counter2],FIFO[counter2+1],FIFO[counter2+1],
FIFO[counter2+3],FIFO[counter2+4],FIFO[counter2+5],FIFO[counter2+6],FI
FO[counter2+7]}*/
                data[63:56]<=FIFO[counter2];
                data[55:48]<=FIFO[counter2+1];
                data[47:40]<=FIFO[counter2+2];
                data[39:32]<=FIFO[counter2+3];
                data[31:24]<=FIFO[counter2+4];
                data[23:16]<=FIFO[counter2+5];
                data[15:8]<=FIFO[counter2+6];
                data[7:0]<=FIFO[counter2+7];

                counter2<=counter2+8;
            end

            else begin
                Empty<=3'd0;
                o_empty<=3'd0;
                flag4<=0;
                flag3<=0;
                            flag2<=0;
                readypktz<=1;
                sop<=0;
                eop<=0;
                validtobackend<=0;
                data<=64'd0;
                for(i=0;i<256;i++)begin
                FIFO[i]=8'd0;
                end
                counter0<=0;
                counter1<=66;
                counter2<=0;

           end
      end

         end
     end


endmodule
```

Parser:

```
module parser(input logic clk,reset,
      input logic
validreg,readypreproc,startofpacket,endofpacket,validcontroller,
      input logic[7:0] mesg,
      output logic readypar,validpar,
      output logic [31:0] tag,
      output logic [167:0] value);

    logic [31:0] tag0;
    logic [167:0] value0;
    logic flag;

    int i,j;


    typedef enum logic[3:0]{IDLE, READTAG, READVALUE, DONE} state_t;
    state_t state;

    initial begin
      state = IDLE;
      readypar = 1;
      validpar = 0;
      tag[31:0] = 32'd0;
      value[167:0] = 168'd0;
      tag0[31:0] = 32'd0;
      value0[167:0] = 168'd0;
      i=31;
      j=167;
    end

    always_ff@(posedge clk) begin
      if(reset) begin
           state <= IDLE;
```

```
        readypar = 1;
        validpar = 0;
        tag0[31:0] = 32'd0;
        value0[167:0] = 168'd0;
        tag[31:0] = 32'd0;
        value[167:0] = 168'd0;
        i=31;
        j=167;
end

else if(validcontroller) begin
        state <= IDLE;
        readypar = 1;
        validpar = 0;
        tag0[31:0] = 32'd0;
        value0[167:0] = 168'd0;
        tag[31:0] = 32'd0;
        value[167:0] = 168'd0;
        i=31;
        j=167;
end

else case(state)
        IDLE: begin
                if(validreg == 0) ;
                else if(startofpacket==1)begin
                        if(i == 31)begin
                                tag0[31:24] <= mesg[7:0];
                                i <= i-8;
                                state <= READTAG;
                        end
                end
                end

        READTAG:begin
                if(validreg == 0) ;
                else if(mesg!=8'h3D) begin
                        if(i == 31) tag0[31:24] <= mesg[7:0];
                        else if(i == 23) tag0[23:16] <= mesg[7:0];
                        else if(i == 15) tag0[15:8] <= mesg[7:0];
                        else if(i == 7) tag0[7:0] <= mesg[7:0];
                        //state <= READTAG;
                        i <= i-8;
                end
                else begin
                        if(tag0 == 32'h31300000) flag <= 1;
                        state <= READVALUE;
                end
                end
```

```
            READVALUE:begin
                      if(validreg == 0) ;
                      else if(mesg!=8'h7C) begin
                           if(j==7) value0[7:0] <= mesg[7:0];
                           else if(j==15) value0[15:8] <= mesg[7:0];
                           else if(j==23) value0[23:16] <= mesg[7:0];
                           else if(j==31) value0[31:24] <= mesg[7:0];
                           else if(j==39) value0[39:32] <= mesg[7:0];
                           else if(j==47) value0[47:40] <= mesg[7:0];
                           else if(j==55) value0[55:48] <= mesg[7:0];
                           else if(j==63) value0[63:56] <= mesg[7:0];
                           else if(j==71) value0[71:64] <= mesg[7:0];
                           else if(j==79) value0[79:72] <= mesg[7:0];
                           else if(j==87) value0[87:80] <= mesg[7:0];
                           else if(j==95) value0[95:88] <= mesg[7:0];
                           else if(j==103) value0[103:96] <=
mesg[7:0];
                           else if(j==111) value0[111:104] <=
mesg[7:0];
                           else if(j==119) value0[119:112] <=
mesg[7:0];
                           else if(j==127) value0[127:120] <=
mesg[7:0];
                           else if(j==135) value0[135:128] <=
mesg[7:0];
                           else if(j==143) value0[143:136] <=
mesg[7:0];
                           else if(j==151) value0[151:144] <=
mesg[7:0];
                           else if(j==159) value0[159:152] <=
mesg[7:0];
                           else if(j==167) value0[167:160] <=
mesg[7:0];
                           //state <= READVALUE;
                           j <= j-8;
                           end
                        else begin
                           tag[31:0] <= tag0[31:0];
                           value[167:0] <= value0[167:0];

                           state <= DONE;
                           validpar <= 1;
                           readypar <= 0;
                           if(endofpacket == 1) flag <= 1;
                           else flag <= 0;

                           end
                 end

            DONE: begin
```

```
                    if(readypreproc == 0) ;
                    else begin
                         i <= 31;
                         j <= 167;
                         tag0[31:0] <= 32'd0;
                         value0[167:0] <= 168'd0;
                         tag[31:0] <= 32'd0;
                         value[167:0] <= 168'd0;
                         readypar <= 1;
                         validpar <= 0;
                         if(flag == 1)begin
                              state <= IDLE;
                              flag <= 0;
                         end
                         else state <= READTAG;
                    end
                    end
          endcase
         end

endmodule
```

Uplink Preprocessor:

```
module preprocessor( input logic clk,reset,
           input logic validpar,
           input logic [31:0] tag,
           input logic [167:0] value,
           output logic readypreproc, validpreproc,
           output logic [15:0] vmsgtype,
           output logic[31:0] vseqnum, vordqty,
           output logic[31:0] vprice,
           output logic[63:0] vsendid, vtargid, vsymbol,
           output logic[95:0] vsendtime1,
           output logic[31:0] vsendtime2,
           output logic[95:0] vtranstime1,
           output logic[31:0] vtranstime2,
           output logic[7:0] vside, vordtype);

    integer i;
    integer j;

    logic[9:0][7:0] value0;
    logic found_seq;
    logic found_qty;
    logic found_pric;

    assign value0[9][7:0] = value[167:160];
    assign value0[8][7:0] = value[159:152];
    assign value0[7][7:0] = value[151:144];
    assign value0[6][7:0] = value[143:136];
    assign value0[5][7:0] = value[135:128];
    assign value0[4][7:0] = value[127:120];
    assign value0[3][7:0] = value[119:112];
    assign value0[2][7:0] = value[111:104];
    assign value0[1][7:0] = value[103:96];
    assign value0[0][7:0] = value[95:88];

    typedef enum logic [1:0]{READ,OUTPUT} state_t;
    state_t state;

    initial begin
    readypreproc = 1;
    validpreproc = 0;
    state = READ;
    found_seq = 0;
    found_qty = 0;
    found_pric = 0;
    vmsgtype[15:0] = 16'd0;
    vseqnum = 32'd0;
        vordqty = 32'd0;
    vprice = 32'd0;
    vsendid[63:0] = 64'd0;
    vtargid[63:0] = 64'd0;
    vsymbol[63:0] = 64'd0;
    vsendtime1 = 96'd0;
    vsendtime2 = 32'd0;
    vtranstime1 = 96'd0;
    vtranstime2 = 32'd0;
    vside[7:0] = 8'd0;
    vordtype[7:0] = 8'd0;
    end

    always_ff@(posedge clk) begin
      if(reset) begin
           readypreproc = 1;
           validpreproc = 0;
           state <= READ;
           found_seq = 0;
           found_qty = 0;
           found_pric = 0;
           vmsgtype[15:0] = 16'd0;
           vseqnum = 32'd0;
                vordqty = 32'd0;
           vprice = 32'd0;
           vsendid[63:0] = 64'd0;
           vtargid[63:0] = 64'd0;
           vsymbol[63:0] = 64'd0;
           vsendtime1 = 96'd0;
           vsendtime2 = 32'd0;
           vtranstime1 = 96'd0;
```

```
          vtranstime2 = 32'd0;
          vside[7:0] = 8'd0;
          vordtype[7:0] = 8'd0;
          end

     else case(state)
          READ: begin
               if(validpar == 0) ;
               else if(tag == 32'h38000000) begin
                              vmsgtype[15:0] = 16'd0;
                              vseqnum = 32'd0;
                                   vordqty = 32'd0;
                              found_seq = 0;
                              found_qty = 0;
                              found_pric = 0;
                              vsendid[63:0] = 64'd0;
                              vtargid[63:0] = 64'd0;
                              vsymbol[63:0] = 64'd0;
                              vsendtime1 = 96'd0;
                              vsendtime2 = 32'd0;
                              vtranstime1 = 96'd0;
                              vtranstime2 = 32'd0;
                              vside[7:0] = 8'd0;
                              vordtype[7:0] = 8'd0;
                              end

               else if(tag == 32'h33350000)    vmsgtype[15:0] <=
value[167:152];

               else if(tag == 32'h33340000)    begin
                                   for(i=0; i<10; i=i+1)begin
                                        if(value0[i][7:0] &&
!found_seq)begin
                                             found_seq = 1;
                                             for(j=0; j<10-i;
j=j+1)begin
                                                  vseqnum =
vseqnum + (value0[j+i][7:0]-8'h30)*((32'd10)**j);
                                             end
                                        end
                                   end
                                   end

                    else if(tag == 32'h33380000)    begin
                                   for(i=0; i<10; i=i+1)begin
                                        if(value0[i][7:0] &&
!found_qty)begin
                                             found_qty = 1;
                                             for(j=0; j<10-i;
j=j+1)begin
```

```
                                                     vordqty =
vordqty + (value0[j+i][7:0]-8'h30)*((8'd10)**j);
                                                end
                                           end
                                      end
                                      end
                      else if(tag == 32'h34340000)    begin
                                      for(i=0; i<10; i=i+1)begin
                                           if(value0[i][7:0] &&
!found_pric)begin
                                                found_pric = 1;
                                                for(j=0; j<10-i;
j=j+1)begin
                                                     vprice = vprice
+ (value0[j+i][7:0]-8'h30)*((8'd10)**j);
                                                end
                                           end
                                      end
                                      end
                 else if(tag == 32'h34390000) vsendid[63:0] <=
value[167:104];
                 else if(tag == 32'h35360000) vtargid[63:0] <=
value[167:104];
                 else if(tag == 32'h35350000) vsymbol[63:0] <=
value[167:104];
                 else if(tag == 32'h35320000)    begin
                                      vsendtime1[95:0] <=
value[167:72];
                                      vsendtime2[31:0] <=
((value[71:64]-8'h30)*32'd600000) + ((value[63:56]-8'h30)*32'd60000) +
((value[47:40]-8'h30)*32'd10000) + ((value[39:32]-8'h30)*32'd1000) +
((value[23:16]-8'h30)*32'd100) + ((value[15:8]-8'h30)*32'd10) +
((value[7:0]-8'h30)*32'd1);
                                      end
                 else if(tag == 32'h36300000)    begin
                                      vtranstime1[95:0] <=
value[167:72];
                                      vtranstime2[31:0] <=
((value[71:64]-8'h30)*32'd600000) + ((value[63:56]-8'h30)*32'd60000) +
((value[47:40]-8'h30)*32'd10000) + ((value[39:32]-8'h30)*32'd1000) +
((value[23:16]-8'h30)*32'd100) + ((value[15:8]-8'h30)*32'd10) +
((value[7:0]-8'h30)*32'd1);
                                      end
                 else if(tag == 32'h35340000) vside[7:0] <=
value[167:160];
                 else if(tag == 32'h34300000) vordtype[7:0] <=
value[167:160];
                 else if(tag == 32'h31300000)    begin
                                      state <= OUTPUT;
                                      readypreproc <= 0;
```

```
                                   validpreproc <= 1;
                                   end
                end

            OUTPUT: begin

                        readypreproc <= 1;
                        validpreproc <= 0;
                        state <= READ;


                end
        endcase
       end
endmodule
```

Downlink Preprocessor:

```
module preprocessor_down( input logic clk,reset,
           input logic validpar,
           input logic [31:0] tag,
           input logic [167:0] value,
           output logic readypreproc, validpreproc,
           output logic [15:0] vmsgtype,
           output logic[31:0] vlastqty,
           output logic[63:0] vsendid, vtargid, vsymbol,
           output logic[7:0] vside);

    integer i;
    integer j;

    logic[9:0][7:0] value0;
    logic found_qty;

    assign value0[9][7:0] = value[167:160];
    assign value0[8][7:0] = value[159:152];
    assign value0[7][7:0] = value[151:144];
    assign value0[6][7:0] = value[143:136];
    assign value0[5][7:0] = value[135:128];
    assign value0[4][7:0] = value[127:120];
    assign value0[3][7:0] = value[119:112];
    assign value0[2][7:0] = value[111:104];
    assign value0[1][7:0] = value[103:96];
    assign value0[0][7:0] = value[95:88];

    typedef enum logic [1:0]{READ,OUTPUT} state_t;
    state_t state;

    initial begin
    readypreproc = 1;
    validpreproc = 0;
    state = READ;
    found_qty = 0;
    vmsgtype[15:0] = 16'd0;
    vlastqty[31:0] = 32'd0;
    vsendid[63:0] = 64'd0;
    vtargid[63:0] = 64'd0;
    vsymbol[63:0] = 64'd0;
    vside[7:0] = 8'd0;
    end

    always_ff@(posedge clk) begin
      if(reset) begin
           readypreproc = 1;
           validpreproc = 0;
           state <= READ;
           found_qty = 0;
           vmsgtype[15:0] = 16'd0;
           vlastqty[31:0] = 32'd0;
           vsendid[63:0] = 64'd0;
           vtargid[63:0] = 64'd0;
           vsymbol[63:0] = 64'd0;
           vside[7:0] = 8'd0;
           end

      else case(state)
           READ: begin
                if(validpar == 0) ;
                else if(tag == 32'h38000000) begin
                                vmsgtype[15:0] = 16'd0;
                                     vlastqty = 32'd0;
                                found_qty = 0;
                                vsendid[63:0] = 64'd0;
                                vtargid[63:0] = 64'd0;
                                vsymbol[63:0] = 64'd0;
                                vside[7:0] = 8'd0;
                                end

                else if(tag == 32'h33350000)    vmsgtype[15:0] <=
value[167:152];

                     else if(tag == 32'h33320000)    begin
                                     for(i=0; i<10; i=i+1)begin
                                          if(value0[i][7:0] &&
!found_qty)begin
                                               found_qty = 1;
                                               for(j=0; j<10-i;
j=j+1)begin
                                                    vlastqty =
vlastqty + (value0[j+i][7:0]-8'h30)*((8'd10)**j);
                                               end
```

```
                                                      end
                                                 end
                                                 end

                       else if(tag == 32'h35360000) vsendid[63:0] <=
value[167:104];
                       else if(tag == 32'h34390000) vtargid[63:0] <=
value[167:104];
                       else if(tag == 32'h35350000) vsymbol[63:0] <=
value[167:104];
                       else if(tag == 32'h35340000) vside[7:0] <=
value[167:160];
                       else if(tag == 32'h31300000)     begin
                                                 state <= OUTPUT;
                                                 readypreproc <= 0;
                                                 validpreproc <= 1;
                                                 end
                       end

                 OUTPUT: begin
                            readypreproc <= 1;
                            validpreproc <= 0;
                            state <= READ;

                       end
      endcase
     end
endmodule
```

ID cam:

```
module IDcam(    input logic clk,reset,
      input logic [63:0] senderid,targid,
      output logic validcam,error,
      output logic [7:0] addr);

      parameter namewidth = 2;
      parameter depth =1 << namewidth;  //4 combinations of ID

      logic [127:0] lutable [0:depth-1];
     logic flag;
      integer i;

     assign lutable[0] = {32'h44415644,32'd0,32'h434D4500,32'd0};
     //DAVD,CME
     assign lutable[1] = {48'h514955534849,16'd0,32'h434D4500,32'd0};
     //QIUSHI,CME
     assign lutable[2] = {32'h4D4F4449,32'd0,32'h434D4500,32'd0};
     //MODI,CME
     assign lutable[3] = {40'h4B61746879,24'd0,32'h434D4500,32'd0};
     //Kathy,CME
```

```
    initial begin
        error=1;
        validcam=0;
        addr=8'd0;
    flag=0;
     end


    always_ff@(posedge clk) begin
     if(reset) begin
                addr<=8'd0;
                error<=1;
                validcam<=0;
     flag<=0;
                end

           else begin
           if(senderid==64'd0 && targid==64'd0)begin
                validcam<=0;
                error<=1;
                addr<=8'd0;
                flag<=0;
           end

           else if (senderid!=64'd0 && targid!=64'd0) begin
                validcam <= 1;
                for(i=0; i<depth; i++) begin
                     if({senderid,targid}==lutable[i][127:0] &&
flag==0)begin
                          flag <= 1;
                          error <= 0;
                          addr <= (8'd1)*i;
                     end

                end
           end
     end
    end

endmodule
```

Symbol cam:

```
module symbolcam(     input logic clk,reset,
           input logic [63:0] symbol,
           output logic validcam,error,
           output logic [7:0] addr);

     parameter symbolwidth = 4;
     parameter depth = symbolwidth;
```

```
      logic [63:0] lutable [0:depth-1];
     logic flag;
      integer i;

     assign lutable[0] = {24'h414D43,40'd0};      //AMC
     assign lutable[1] = {24'h4E5954,40'd0};      //NYT: New York Times
Company
     assign lutable[2] = {24'h51544D,40'd0};      //QTM: Quantum Corp
     assign lutable[3] = {32'h594F4B55,32'd0};    //YOKU: Youku.Com Inc

     initial begin
         error=1;
         validcam=0;
         addr=8'd0;
     flag=0;
      end


     always_ff@(posedge clk) begin
      if(reset) begin
                 addr<=8'd0;
                 error<=1;
                 validcam<=0;
      flag<=0;
            end

      else begin
            if(symbol==64'd0)begin
                 validcam<=0;
                 error<=1;
                 addr<=8'd0;
                 flag<=0;
            end

            else if(symbol!=64'd0)begin
                 validcam <= 1;
                 for(i=0; i<depth; i++) begin
                      if(symbol==lutable[i][63:0] && flag==0)begin
                           flag <= 1;
                           error <= 0;
                           addr <= (8'd1)*i;
                      end

                 end
            end
      end
     end

endmodule
```

Whole cam:

```
module cam_2(   input logic clk,reset,
     input logic valid,
     input logic [63:0] senderid,targid,
     output logic readycam,hitcam,validcam,
     output logic [7:0] addr);

     parameter namewidth = 8;
     parameter depth =1 << namewidth;  //256 combinations of ID


     logic [0:depth-1][127:0] lutable;
    integer i;
    integer j;
    integer lu;
    logic [127:0] name;

    typedef enum logic[1:0]{LOOKUP, DONE} state_t;
    state_t state;

    initial begin
        readycam=1;
    hitcam=0;
    validcam=0;
        addr=8'd0;
    i=0;
    j=0;
        name=128'd0;
        state=LOOKUP;
    for(lu=0;lu<depth;lu++) lutable[lu][127:0]=128'd0;
     end


    always_ff@(posedge clk) begin
     if(reset) begin
              addr<=8'd0;
              readycam<=1;
              hitcam<=0;
              validcam<=0;
     i=0;
     j=0;
         name<=128'd0;
     state<=LOOKUP;
         end

     else case(state)
     LOOKUP:begin
         if(valid==1 && senderid!=64'd0 && targid!=64'd0)begin
              name<={senderid,targid};
```

```
                            for(i=0;i<depth;i++)begin

         if({senderid,targid}==lutable[i][127:0]) begin
                                                hitcam<=1;
                                            addr<=(8'd1)*i;
                            break;
                                            end


                     end
                            state<=DONE;
                     validcam<=1;
                     readycam<=0;
                 end
                        end

                     DONE:begin
                 if(i==depth) begin
                     for(j=0;j<depth;j++)begin
                         if(lutable[j][127:0]==128'd0)begin
                             lutable[j][127:0]<=name;
                             addr<=(8'd1)*j;
                             break;
                         end
                     end

                 end
                         readycam<=1;
                         validcam<=0;
                 hitcam<=0;
                     state<=LOOKUP;
                         end
                 endcase
         end

endmodule
```

Maximum number of NewOrderSingle messages per second:

```
module pktnum(   input logic clk,
     input logic reset,
     input logic[15:0] msgtype,
     output logic flag,
     output logic valid);   //flag == 1, through; flag == 0, drop

   parameter numlimit = 1000000;

   logic[31:0] numcounter;
   logic[31:0] scounter;

   initial begin
```

```
    flag = 1;
    valid = 0;
    numcounter = 32'd0;
    scounter = 32'd0;
    end

    always_ff@(posedge clk) begin
      if(reset)begin
           flag <= 1;
           valid <= 0;
           numcounter <= 32'd0;
           scounter <= 32'd0;
      end
      else if(scounter == 32'd200000000) begin
           numcounter <= 32'd0;
           scounter <= 32'd0;
      end
      else if(msgtype==16'h4400 && valid == 0) begin
           if(numcounter == numlimit) begin
                scounter <= scounter + 1;
                flag <= 0;
                valid <= 1;
           end
           else begin
                scounter <= scounter + 1;
                numcounter <= numcounter +1;
                flag <= 1;
                valid <= 1;
           end
      end
      else if(msgtype==16'h0000) begin
           flag <= 1;
           valid <= 0;
           scounter <= scounter + 1;
      end
      else scounter <= scounter + 1;
    end



endmodule
```

Maximum number of contracts a single FIX message can contain:

```
module qtysingle(     input logic clk,
      input logic reset,
      input logic[15:0] msgtype,
      input logic[7:0] idindex,
      input logic[7:0] symbolindex,
      input logic iderror,
      input logic symbolerror,
      input logic idvalid,
      input logic symbolvalid,
      input logic[7:0] side,
      input logic[31:0] qty,
      output logic flag,
      output logic valid);       //flag == 1, through; flag == 0, drop

    parameter idnum = 4;
    parameter symbolnum = 4;
    parameter depth = idnum*symbolnum;

    logic[0:depth-1][31:0] buylimit;
    logic[0:depth-1][31:0] selllimit;

    assign buylimit[0] = 32'd0;
    assign buylimit[1] = 32'd10;
    assign buylimit[2] = 32'd10;
    assign buylimit[3] = 32'd100;
    assign buylimit[4] = 32'd100;
    assign buylimit[5] = 32'd1000;
    assign buylimit[6] = 32'd1000;
    assign buylimit[7] = 32'd10000;
    assign buylimit[8] = 32'd10000;
    assign buylimit[9] = 32'd100000;
    assign buylimit[10] = 32'd100000;
    assign buylimit[11] = 32'd1000000;
    assign buylimit[12] = 32'd1000000;
    assign buylimit[13] = 32'd100000;
    assign buylimit[14] = 32'd10000;
    assign buylimit[15] = 32'd1000;

    assign selllimit[0] = 32'd0;
    assign selllimit[1] = 32'd10;
    assign selllimit[2] = 32'd10;
    assign selllimit[3] = 32'd100;
    assign selllimit[4] = 32'd100;
    assign selllimit[5] = 32'd1000;
    assign selllimit[6] = 32'd1000;
    assign selllimit[7] = 32'd10000;
    assign selllimit[8] = 32'd10000;
    assign selllimit[9] = 32'd100000;
    assign selllimit[10] = 32'd100000;
    assign selllimit[11] = 32'd1000000;
    assign selllimit[12] = 32'd1000000;
    assign selllimit[13] = 32'd100000;
    assign selllimit[14] = 32'd10000;
    assign selllimit[15] = 32'd1000;

    initial begin
```

```
    flag = 1;
    valid = 0;
    end

    always_ff@(posedge clk) begin
      if(reset) begin
            flag <= 1;
            valid <= 0;
      end
      else if(msgtype==16'd0 && side==8'h00 && qty==32'd0) begin
            flag <= 1;
            valid <= 0;
      end
      else if(idvalid && symbolvalid && iderror==0 && symbolerror==0 &&
msgtype==16'h4400 && side!=8'd0 && qty!=32'd0 && valid == 0) begin
            if(side==8'h31) begin
                  if(qty <= buylimit[idindex*symbolnum+symbolindex])
begin
                        flag <= 1;
                        valid <= 1;
                  end
                  else begin
                        flag <= 0;
                        valid <= 1;
                  end
            end

            if(side==8'h32) begin
                  if(qty <= selllimit[idindex*symbolnum+symbolindex])
begin
                        flag <= 1;
                        valid <= 1;
                  end
                  else begin
                        flag <= 0;
                        valid <= 1;
                  end
            end

      end
    end
endmodule
```

Maximum number of floating contracts:

```
module floatram( input logic clk,
      input logic reset,
      input logic[7:0] idindex,
      input logic[7:0] symbolindex,
      input logic iderror,
      input logic symbolerror,
  input logic idvalid,
  input logic symbolvalid,
  input logic[15:0] msgtype,
  input logic[7:0] side,
  input logic[31:0] qty,
  input logic[15:0] msgtypeD,
  input logic[7:0] idindexD,
  input logic[7:0] symbolindexD,
  input logic idvalidD,
  input logic symbolvalidD,
  input logic iderrorD,
  input logic symbolerrorD,
  input logic[7:0] sideD,
  input logic[31:0] qtyD,
  output logic flag,
  output logic valid,
  output logic validD);      //flag == 1, through; flag == 0, drop

parameter idnum = 4;
parameter symbolnum = 4;
parameter ramdepth = idnum*symbolnum;

logic[0:ramdepth-1][31:0] buylimit;
logic[0:ramdepth-1][31:0] selllimit;
logic[0:ramdepth-1][31:0] buyfloat;
logic[0:ramdepth-1][31:0] sellfloat;
integer i;
logic flag_up;
logic flag_down;

assign buylimit[0] = 32'd0;
assign buylimit[1] = 32'd0;
assign buylimit[2] = 32'd10;
assign buylimit[3] = 32'd100;
assign buylimit[4] = 32'd1000;
assign buylimit[5] = 32'd10000;
assign buylimit[6] = 32'd100000;
assign buylimit[7] = 32'd1000000;
assign buylimit[8] = 32'd10000000;
assign buylimit[9] = 32'd100000000;
assign buylimit[10] = 32'd10000000;
assign buylimit[11] = 32'd1000000;
assign buylimit[12] = 32'd100000;
assign buylimit[13] = 32'd10000;
assign buylimit[14] = 32'd1000;
assign buylimit[15] = 32'd100;

assign selllimit[0] = 32'd0;
assign selllimit[1] = 32'd0;
assign selllimit[2] = 32'd10;
```

```
    assign selllimit[3] = 32'd100;
    assign selllimit[4] = 32'd1000;
    assign selllimit[5] = 32'd10000;
    assign selllimit[6] = 32'd100000;
    assign selllimit[7] = 32'd1000000;
    assign selllimit[8] = 32'd10000000;
    assign selllimit[9] = 32'd100000000;
    assign selllimit[10] = 32'd10000000;
    assign selllimit[11] = 32'd1000000;
    assign selllimit[12] = 32'd10000;
    assign selllimit[13] = 32'd10000;
    assign selllimit[14] = 32'd1000;
    assign selllimit[15] = 32'd100;

    initial begin
    for(i=0; i<ramdepth; i=i+1) buyfloat[i] = 32'd0;
    for(i=0; i<ramdepth; i=i+1) sellfloat[i] = 32'd0;
    flag = 1;
    valid = 0;
    validD = 0;
    flag_up = 0;
    flag_down = 0;
    end

    always_ff@(posedge clk) begin
      if(reset) begin
           for(i=0; i<ramdepth; i=i+1) buyfloat[i] = 32'd0;
           for(i=0; i<ramdepth; i=i+1) sellfloat[i] = 32'd0;
           flag <= 1;
           valid <= 0;
           validD <= 0;
           flag_up <= 0;
           flag_down <= 0;
      end
      else begin
           if(msgtype==16'd0 && side==8'h00 && qty==32'd0) begin
                flag <= 1;
                valid <= 0;
                flag_up <= 0;
           end
           if(msgtypeD==16'd0) begin
                validD <= 0;
                flag_down <= 0;
           end

           if(msgtypeD==16'h3800 && idvalidD && symbolvalidD &&
sideD!=8'd0 && qtyD!=32'd0 && flag_down==0)begin
                flag_down <= 1;
                validD <= 1;
                if(symbolerrorD==0 && iderrorD==0)begin
```

```
                      if(sideD==8'h31)begin
                           buyfloat[idindexD*symbolnum+symbolindexD]
<= buyfloat[idindexD*symbolnum+symbolindexD] - qtyD;
                      end
                      if(sideD==8'd32)begin
                           sellfloat[idindexD*symbolnum+symbolindexD]
<= sellfloat[idindexD*symbolnum+symbolindexD] - qtyD;
                      end
                 end
           end
           else if(msgtype==16'h4400 && idvalid && symbolvalid &&
iderror==0 && symbolerror==0 && side!=8'd0 && qty!=32'd0 && flag_up ==
0) begin
                 flag_up <= 1;
                 if(side==8'h31)begin
                      if(qty+buyfloat[idindex*symbolnum+symbolindex] >
buylimit[idindex*symbolnum+symbolindex]) begin
                           flag <= 0;
                           valid <= 1;
                      end
                      else begin
                           buyfloat[idindex*symbolnum+symbolindex] <=
buyfloat[idindex*symbolnum+symbolindex] + qty;
                           flag <= 1;
                           valid <= 1;
                      end
                 end
                 if(side==8'h32)begin
                      if(qty+sellfloat[idindex*symbolnum+symbolindex]
> selllimit[idindex*symbolnum+symbolindex]) begin
                           flag <= 0;
                           valid <= 1;
                      end
                      else begin
                           sellfloat[idindex*symbolnum+symbolindex] <=
sellfloat[idindex*symbolnum+symbolindex] + qty;
                           flag <= 1;
                           valid <= 1;
                      end
                 end

           end
      end


    end
endmodule
```