Derek Tzeng

18-341

Project 3: USB

I designed from the bottom up, so I first focused on the bit-level FSMs.

For the encoding half:

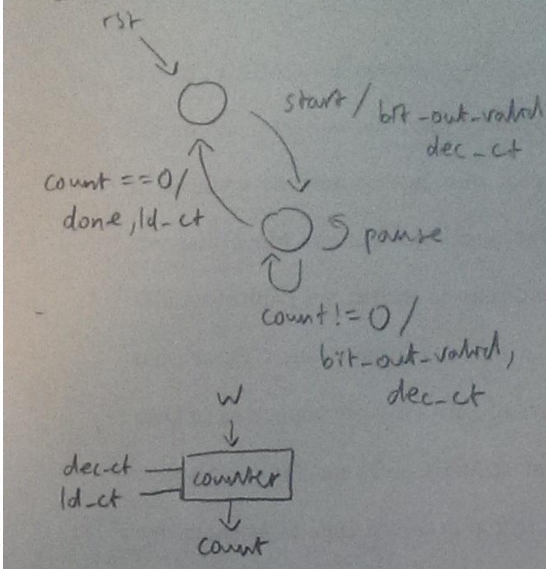
- sendSerialMSBtoLSB and sendSerialLSBtoMSB send data serially with the specified order
- bitStuff receives data serially and stuffs a 0 if it sees six consecutive 1's
- NRZI takes data serially and converts it to a non-Wakerly encoding

For the decoding half:

- unNRZI converts the non-Wakerly encoding back into 0's and 1's
- unStuff receives data serially and ignores the next bit after every six consecutive 1's it encounters
- wait_sync waits for a SYNC, or 0000_0001 to appear on the line. If this does not appear for 255 clock cycles then it times out
- receiveSerial receives data serially and stores it
- wait_idle waits for the lines to go idle for 3 consecutive cycles. I use this to make sure the lines are safe to drive after I receive a packet.
- receive_crc16 receives data serially and puts it through the CRC16 calculation

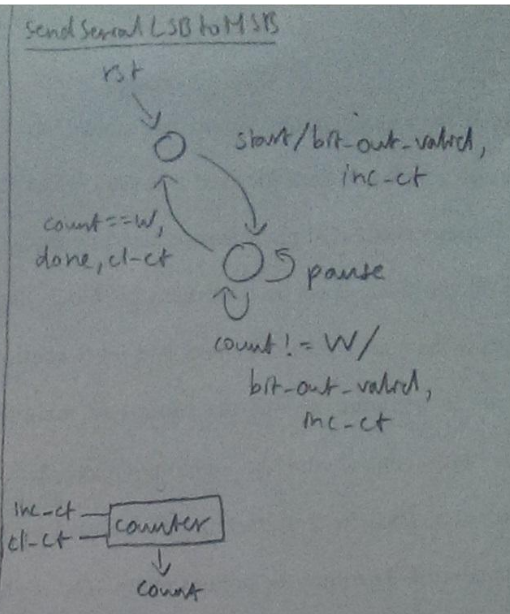
I also have hardware implementations of the crc5_datapath and crc16_datapath, which will be used for CRC calculations. These FSMs can be seen on the 3 next pages.

Send Serial MSB to LSB



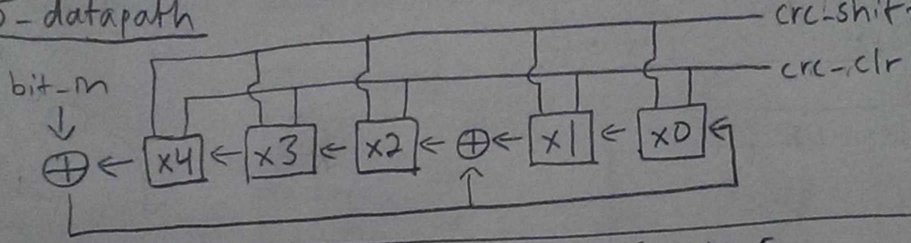
bit-out = in [count-1]

Send Serial LSB to MSB

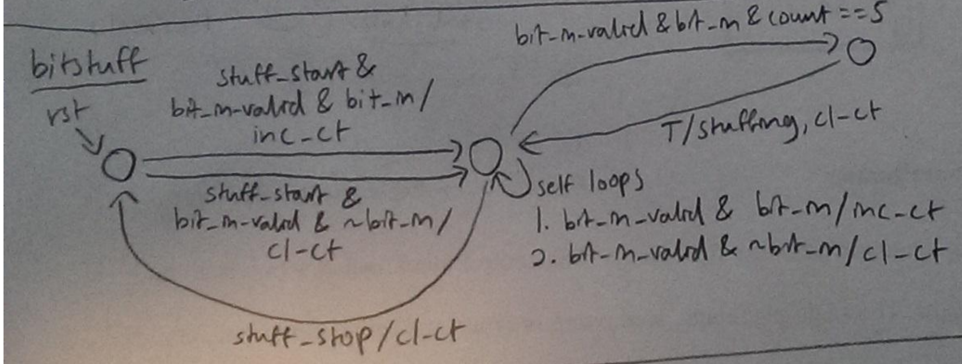


bit-out = in [count]

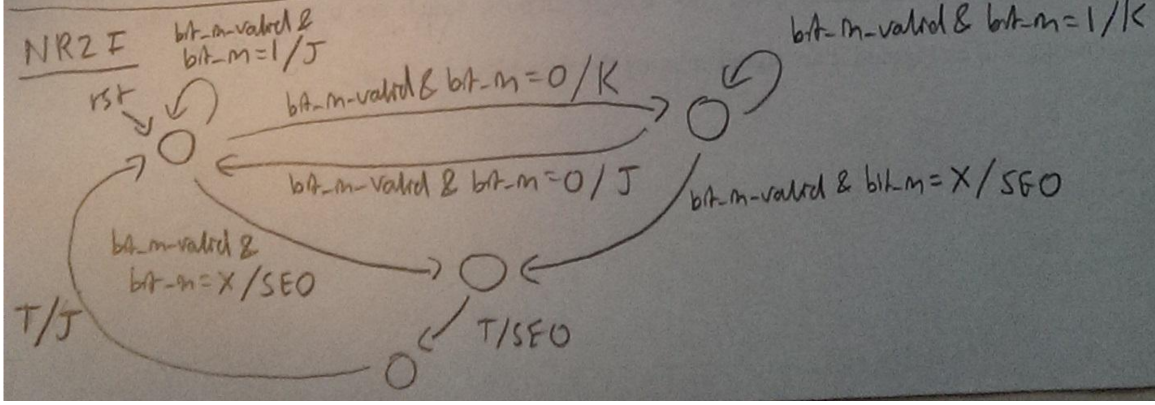
crc5-datapath



bitstuff



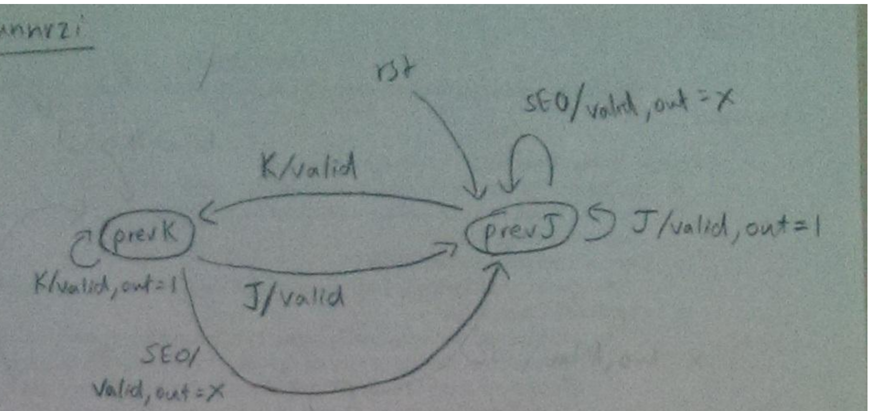
NRZI


bit-in-valid &
bit-in=1/J
bit-in-valid & bit-in=1/K
rst
bit-in-valid & bit-in=0/K
bit-in-valid & bit-in=0/J
bit-in-valid & bit-in=X/SE0
bit-in-valid &
bit-in=X/SE0
T/J
T/SE0

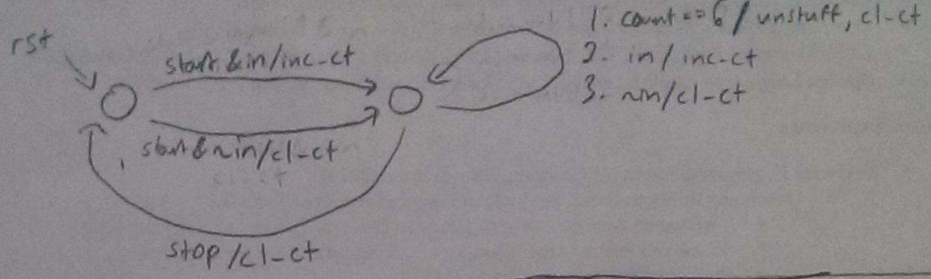
undpdm

DP & ~DM : J=1
~DP & DM : K=1
~DP & ~DM : SEO=1

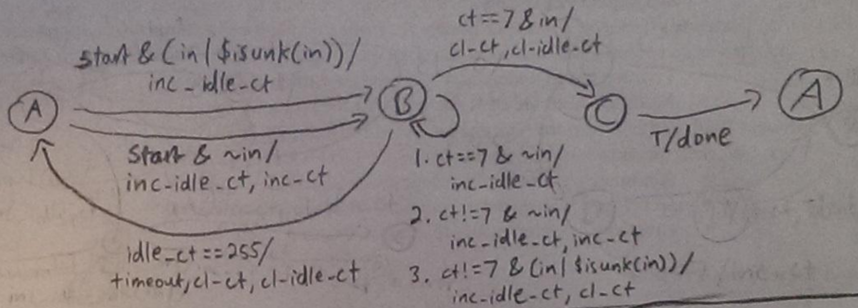
unnrzi



unstuff

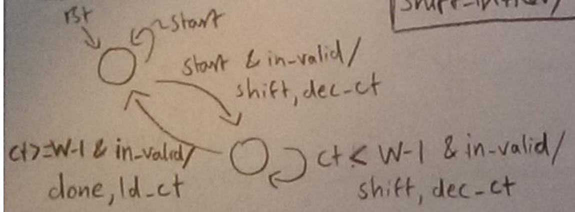


wait_sync'



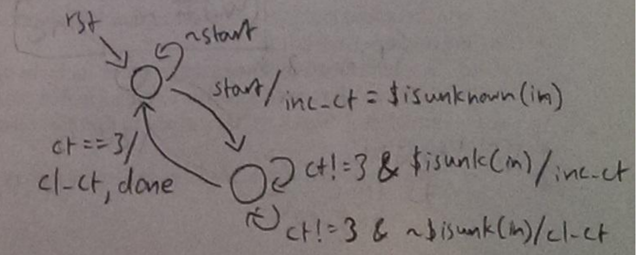
receveSerral

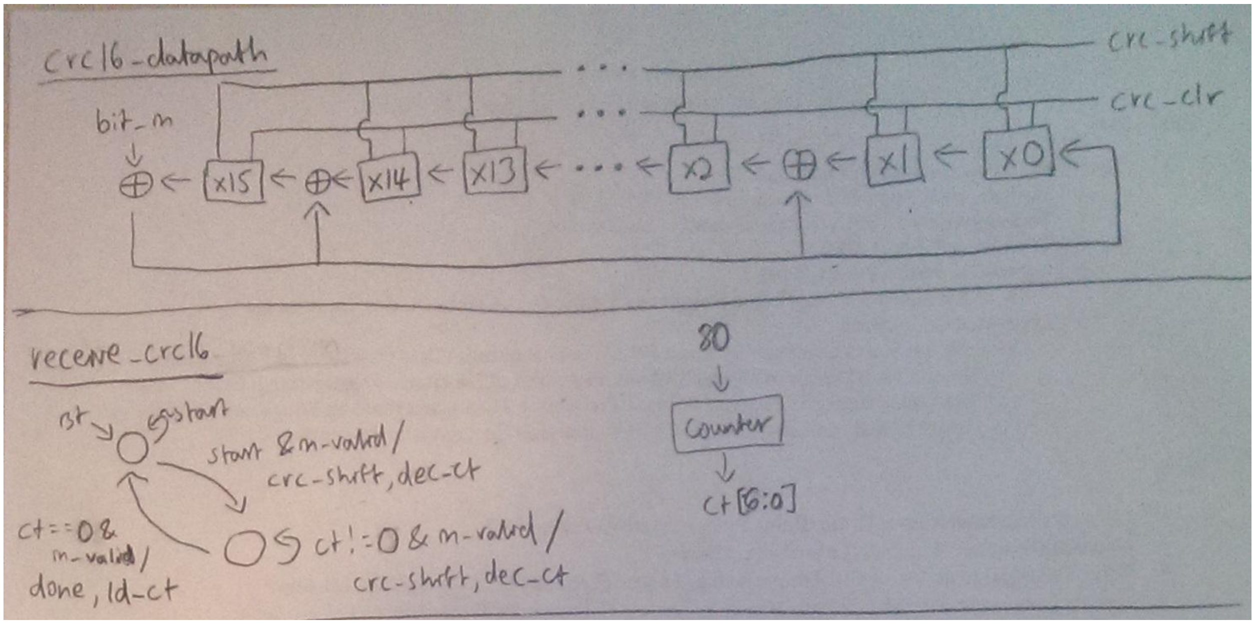
shift_in#(W)



ct[$clog2(W)-1:0]

wait_idle


rst
~start
start/ inc-ct = $isunknown(in)
ct==3/
cl-ct, done
ct!=3 & $isunk(in)/inc-ct
ct!=3 & ~$isunk(in)/cl-ct

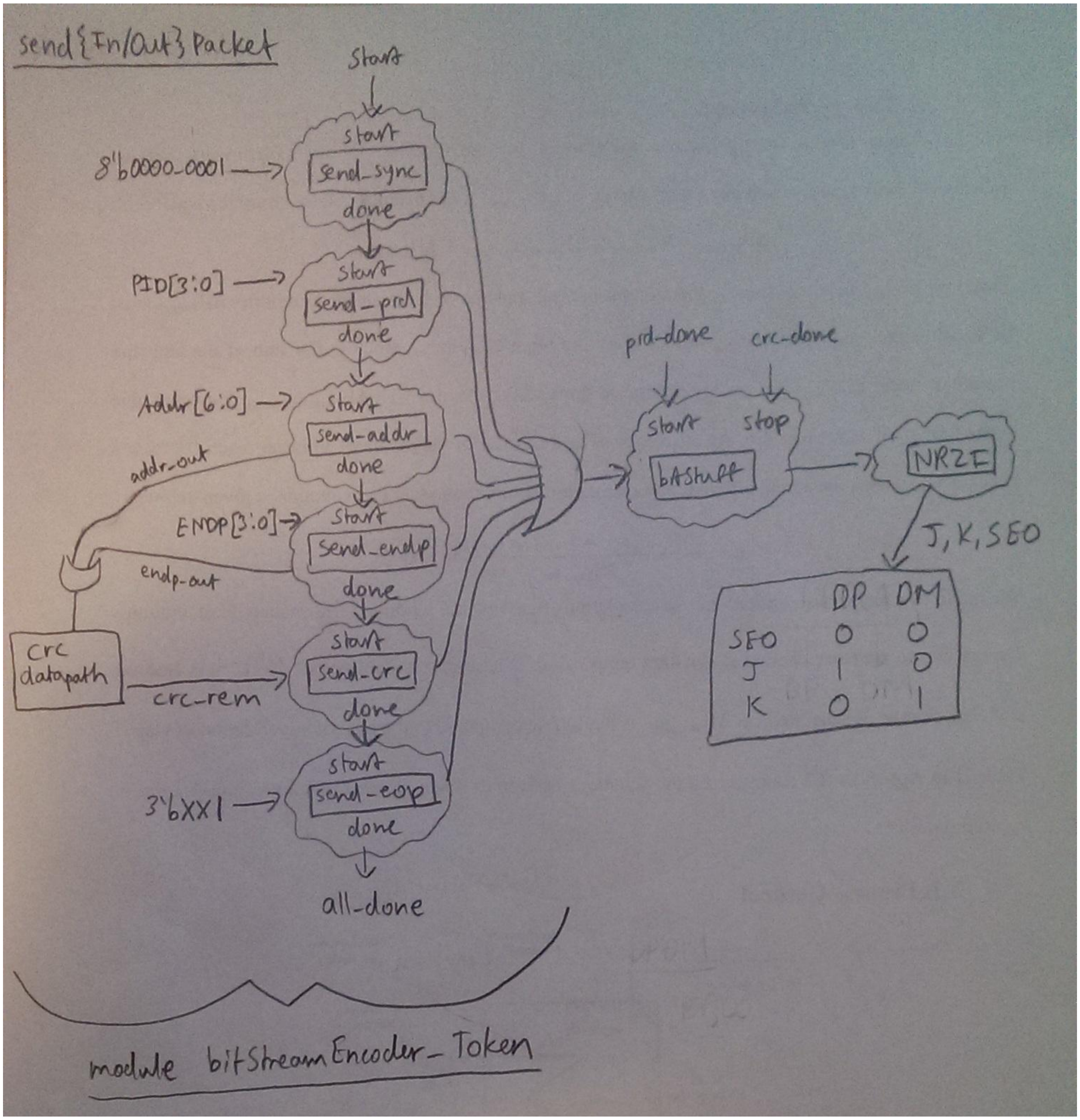


Now that I have the bit-level FSMs, I can implement the packet level FSMs.

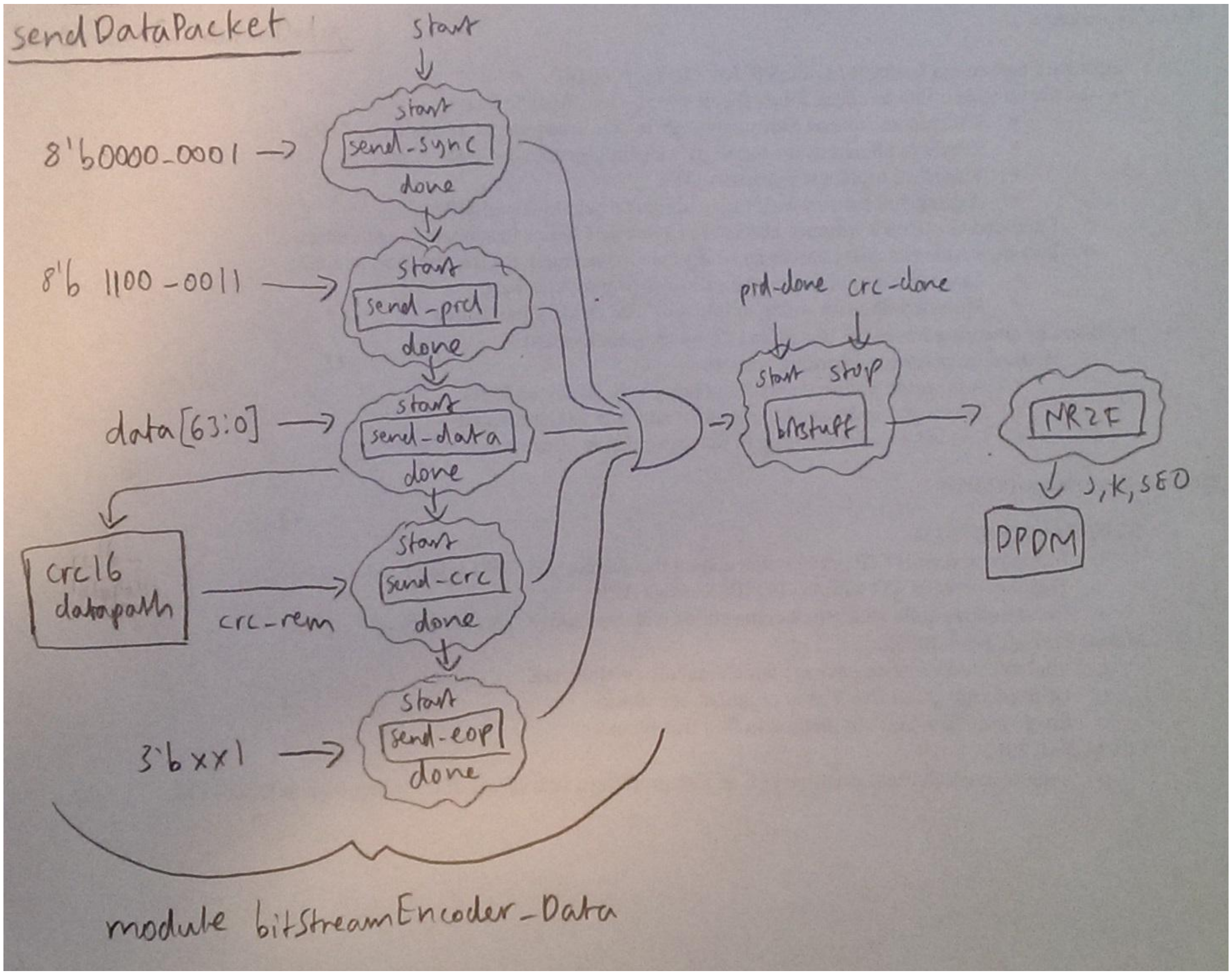
To send an IN or OUT packet, I create a chain of bit-level FSMs which send the SYNC, PID, ADDR, ENDP, CRC5, EOP. These FSMs are tied together such that the 'done' signal of one is connected the 'start' of the next. All these FSMs know what they will output except for the CRC5 FSM, which must calculate the complemented remainder at runtime. So, I instantiate a crc5_datapath on the side, which takes in the outputs of ADDR and ENDP when they are sent. Now, the outputs of all the FSMs are basically fed into an OR gate which feeds into the bitStuffer, then NRZI, then DPDM converter. The bitStuffer starts when PID is done and stops when CRC5 is done. All these components form the basis of sending an IN or OUT packet. The same idea is applied for sending a DATA packet, except the ADDR and ENDP fields are replaced by DATA, and a CRC16 is used instead.

On the other hand, to receive a DATA packet, the opposite needs to be done. The DPDM converter feeds into the unNRZI, then the unStuffer. The output of the unStuffer is connected to my chain of bit-level FSMs which wait for the SYNC, PID, DATA, CRC16, EOP. If a timeout is encountered, then the chain exits immediately and signals an error. As decoding is symmetric, the unStuffer starts when PID is done and stops when CRC16 is done. Errors are evaluated combinationally, so the caller can check for any errors when the whole operation is done. As a precaution, I also wait for 3 consecutive idle's at the very end to make sure that I can recover after an abnormally long packet. All these components form the basis of receiving a DATA packet. The same idea is applied for receiving a handshake packet, except it is simpler because unstuffing and CRC verification are not necessary.

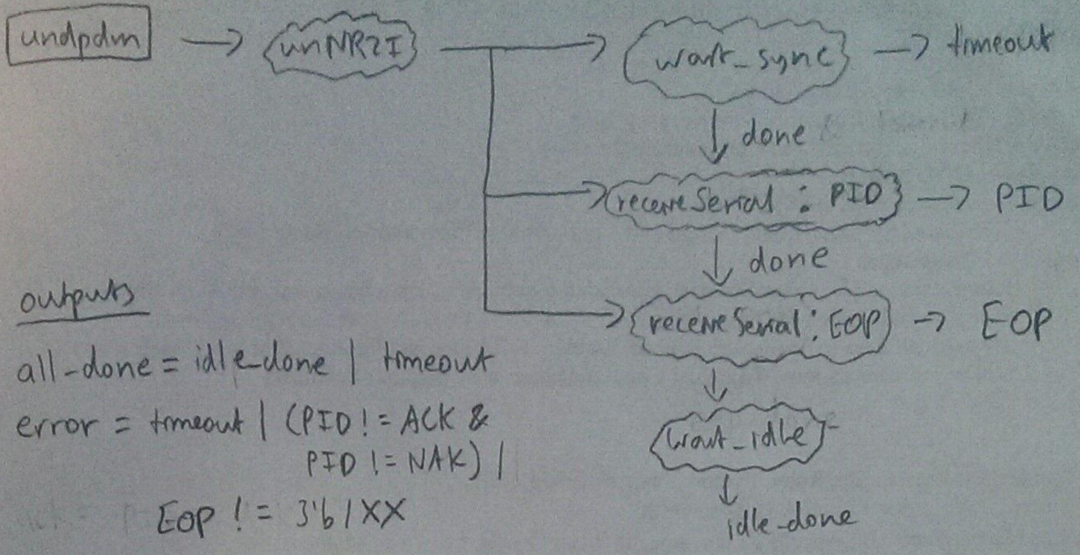
The diagrams for these packet level FSMs are on the next 3 pages.



	DP	DM
SE0	0	0
J	1	0
K	0	1


sendDataPacket
start
start
send-sync
done
8'b0000-0001
8'b 1100-0011
start
send_pid
done
data[63:0]
start
send-data
done
crc16 datapath
crc-rem
start
send-crc
done
start
send-eop
done
3'b xx1
pid-done
crc-done
start
stop
bitstuff
NRZI
J,K,SE0
DPDM
module bitStreamEncoder_Data

receive HandshakePacket



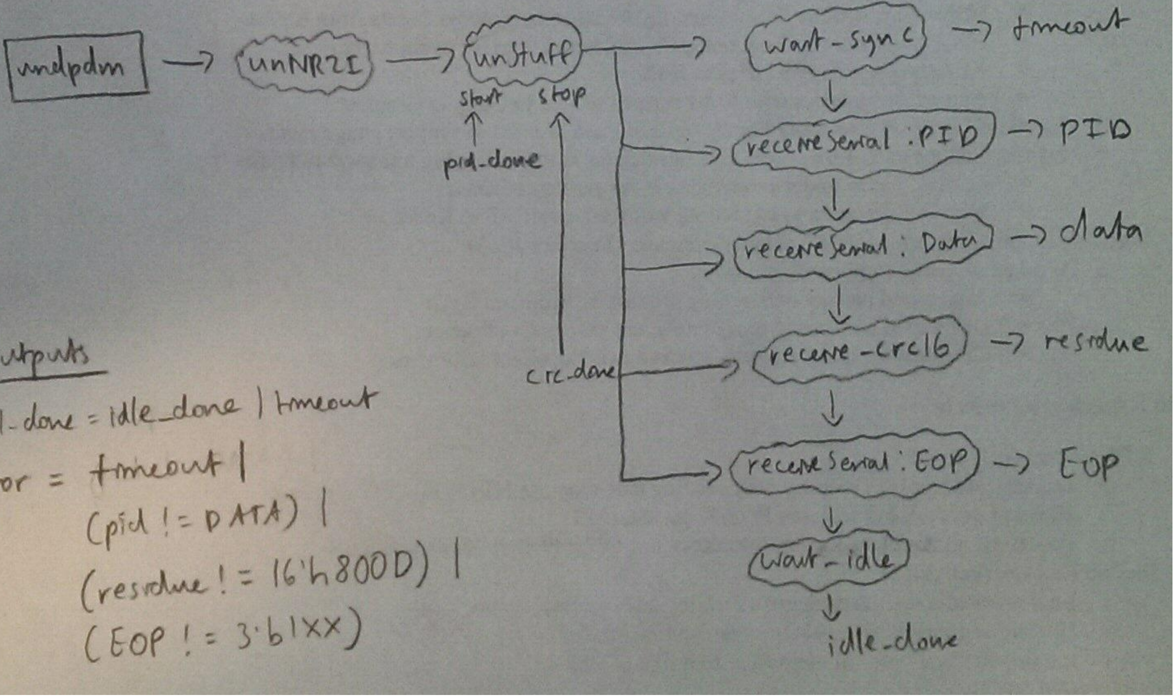
outputs

all-done = idle-done | timeout

error = timeout | (PID != ACK &
PID != NAK) |
EOP != 3'b1XX

ack = PID == ACK

receive DataPacket



outputs

all-done = idle_done | timeout

error = timeout |
(pid != DATA) |
(residue != 16'h800D) |
(EOP != 3'b1XX)

With my packet level FSMs, I can now implement my transaction FSMs.

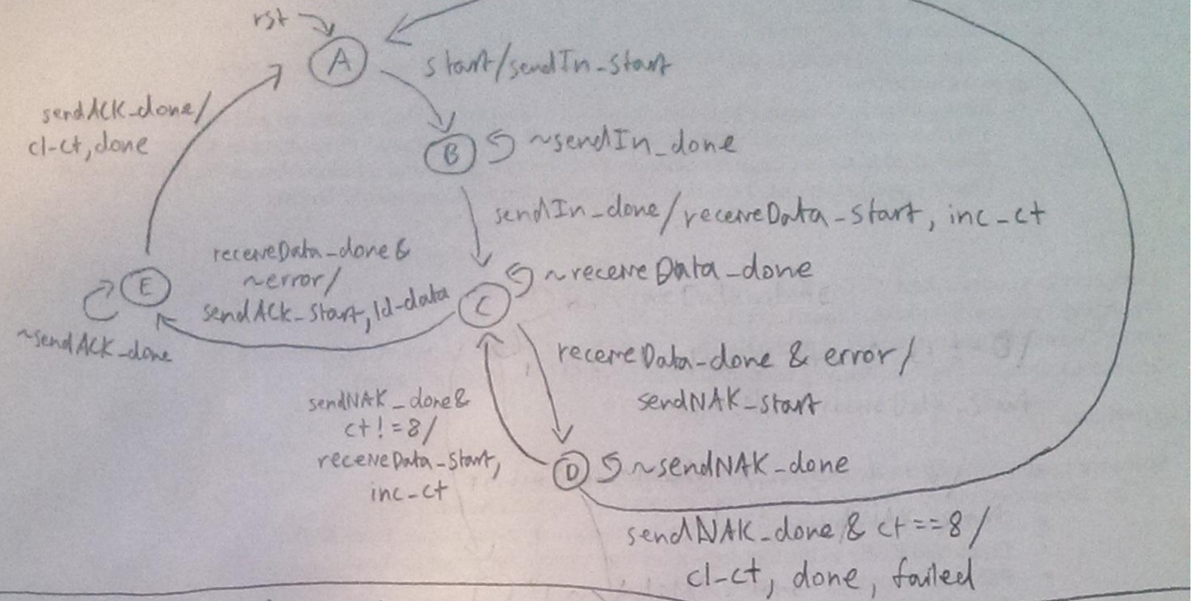
For an IN transaction, I will first send an IN packet. Then, I wait for the DATA. If it is corrupted, or it times out, then I send a NAK and retry up to 8 times, after which I will finish and signal an error. Otherwise, if I get a good DATA, then I send an ACK and finish with no errors.

For an OUT transaction, I will first send an OUT packet. Then, I send a DATA. If I get back a NAK, or it times out, I will retry sending the DATA up to 8 times, after which I will finish and signal and error. Otherwise if I get an ACK, I will finish with no errors.

The FSM diagrams for the transactions are on the next page.

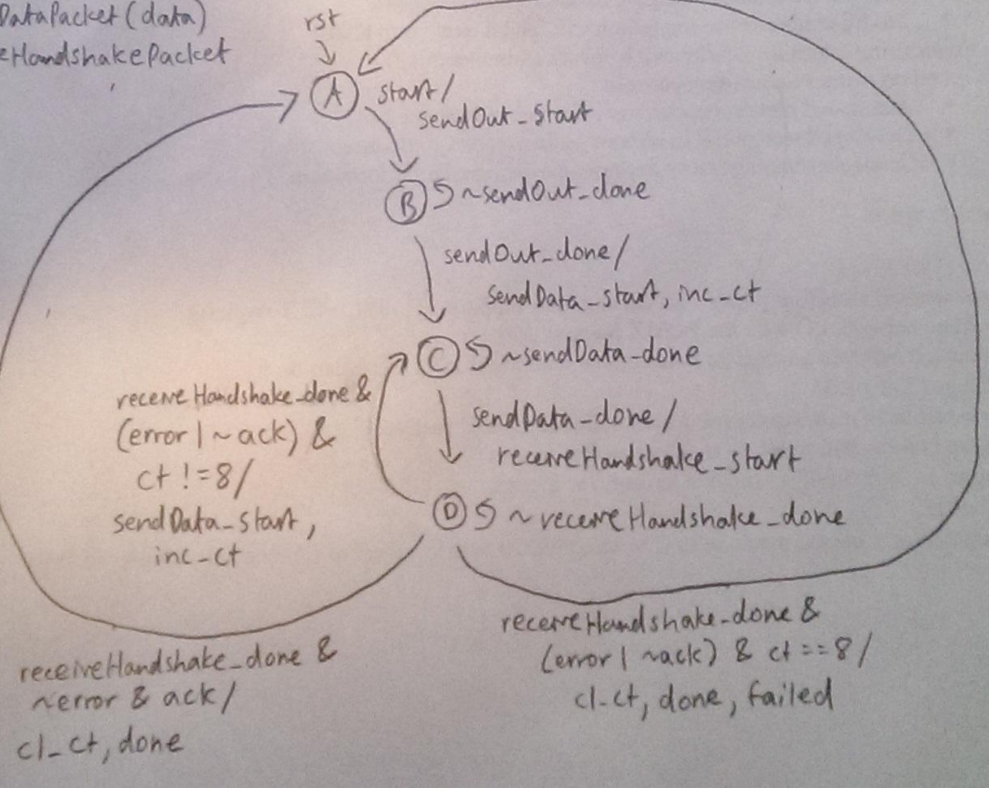
doInTransaction (addr, endp, data)

SendInPacket (addr, endp)
receiveDataPacket (data)
sendACK, sendNAK



doOutTransaction (addr, endp, data)

sendOutPacket (addr, endp)
sendDataPacket (data)
receiveHandshakePacket

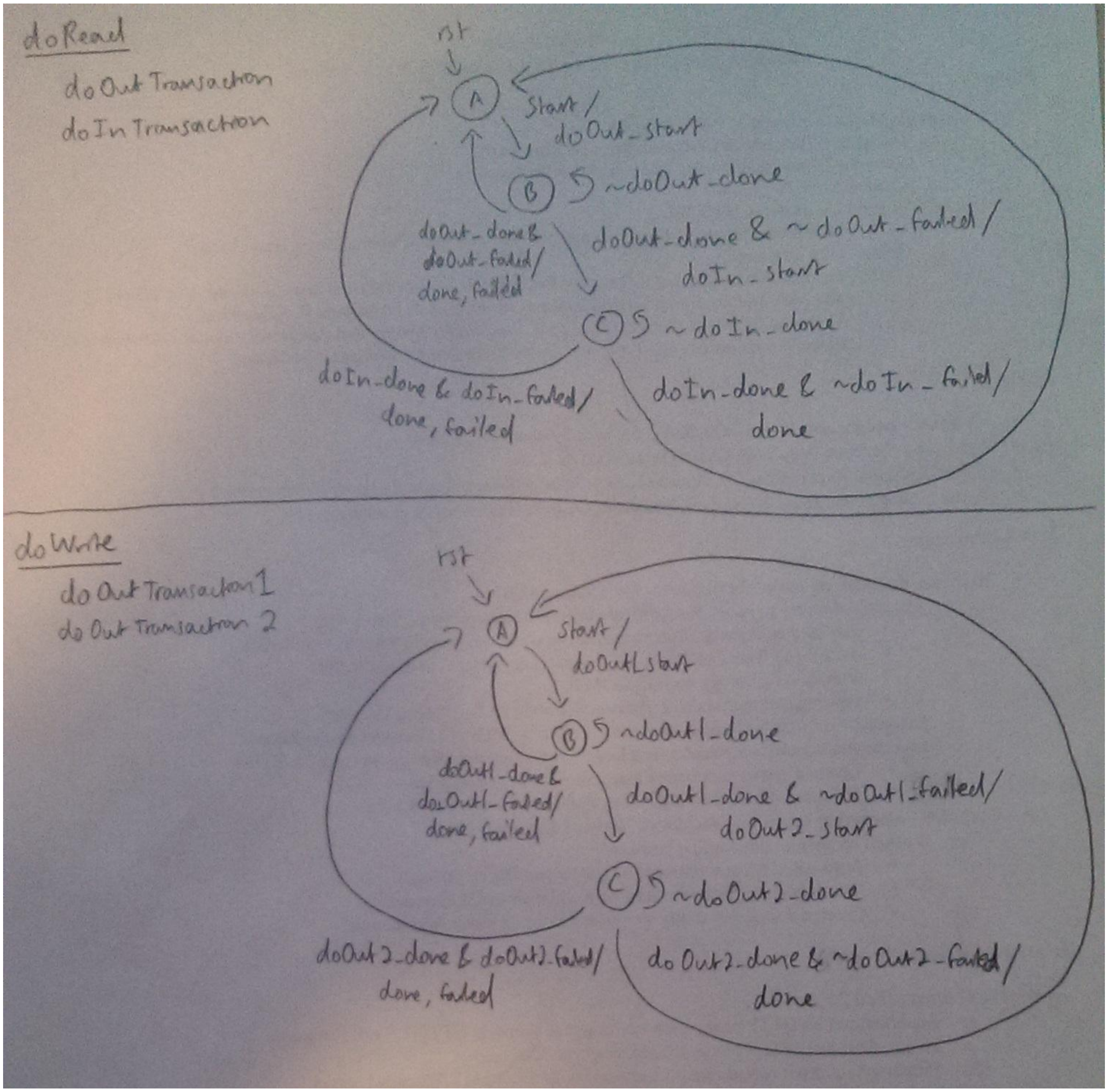


With the transaction FSMs, all that is left is a read and a write FSM.

A read FSM simply performs an OUT and IN transaction. If either of them fail, then it finishes with an error.

Similarly, a write FSM performs two OUT transactions. If either of them fail, then it finishes with an error.

The diagrams for these FSMs are shown below.


doRead
doOutTransaction
doInTransaction
rst
A
Start /
doOut_start
B
~doOut_done
doOut_done &
doOut_failed/
done, failed
doOut_done & ~doOut_failed /
doIn_start
C
~doIn_done
doIn_done & doIn_failed/
done, failed
doIn_done & ~doIn_failed/
done
doWrite
doOutTransaction1
doOutTransaction 2
rst
A
start /
doOut1start
B
~doOut1_done
doOut1_done &
doOut1_failed/
done, failed
doOut1_done & ~doOut1_failed/
doOut2_start
C
~doOut2_done
doOut2_done & doOut2_failed/
done, failed
doOut2_done & ~doOut2_failed /
done