VoIP

Design Document

Sambuddho Chakravarty
Ari Klein
Ashish Sharma
George Sirois

Introduction:

As our project is quite large in scope, we have decided to break it up into two distinct pieces which can be worked on independently. While the general idea behind our project is to create a voice over IP phone system, some of the techniques involved in the compression of the audio signal depart from what is generally regarded as the norm in VoIP communications. Therefore, the first module is a system utilizing the onboard ADC and DAC as well as the FPGA to produce companded signals for transmission. This is a departure from the standard μ-law or similar compression used on VoIP transmissions. The second module is more a more straightforward implementation of all the necessary networking subsystems for VoIP (SIP, RTP, UDP, TCP, and IP). The summaries of the two distinct modules are given below.

Summary of Companding Module:

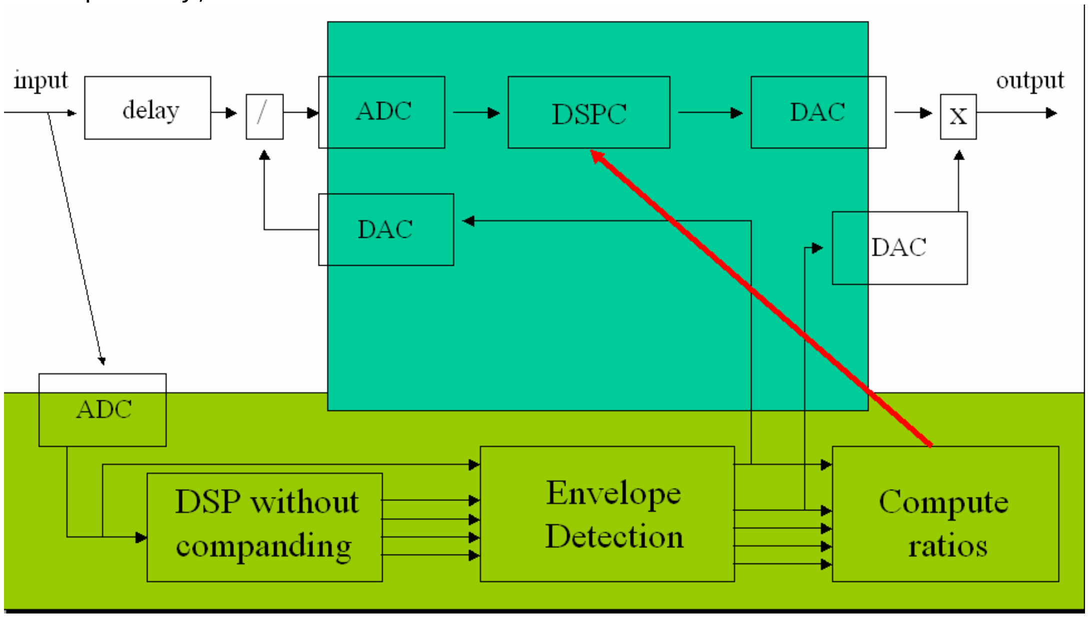
In his research with Professor Yannis Tsividis, Ari Klein extended a technique called companding to digital signal processors. The idea was to reduce the dynamic range of the signals at the input to the ADC, output of the DAC, and internal to the DSP, so that these signals are always large, and thus take full advantage of the available bits in these devices. This should increase the signal to quantization noise ratio at low signal levels. All this should be done without otherwise disturbing the output (in other words, in the absence of quantization, the outputs of the companded and non-companded systems should be identical). For this part of the project, we will be implementing this digital companding technique in hardware on the FPGAs for the case study of a simple digital reverberator.

The formulas for accomplishing this have already been derived by Ari Klein and Yannis Tsividis. For the purposes of this project, the important result from those formulas is that for the companding DSP to compute its next state (at time-step n+1) and its companded (always large) output, it requires:

- ratios of envelopes of NEXT states (time-step n+1) of the prototype DSP
- current state of companded DSP
- companded (always large) input

Block Diagrams:

Conceptually, we want:



The input is passed to the prototype (non-companded DSP), digital envelope detection is done on its states, ratios of envelopes are computed. The input envelope is then used to modify the input (the input is divided by its envelope), giving the companded (always large) input, which is passed to the upper ADC. The companded input and the ratios of envelopes are used by the companding DSP (DSPC), along with its current state, to give the companded output. This companded output is then converted back to analog, and given back its envelope, to become the output of the system.

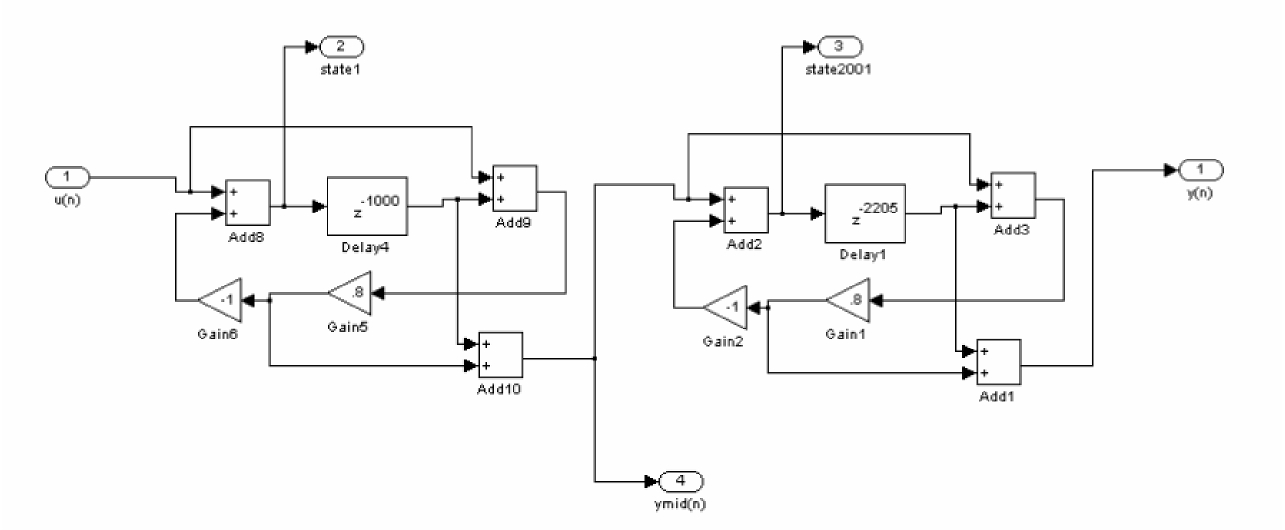
Even though the conceptual system on the previous page would “theoretically” work, it is impractical for implementation in this project for several reasons. For example, we only have one ADC, and one DAC, and we also don’t ever want to do division on an FPGA, so computing ratios directly is a bad idea.

We will effectively SIMULATE the analog components and the ADCs and DACs on the FPGA. In other words, the actual FPGA’s ADC and DAC will give us inputs and outputs that have more bits than we will use in the “digital processor.” The “analog” operations will be done digitally, but with these extra bits of precision. The ADC’s in the diagram will be simulated by storing the “analog” signals in smaller registers. We plan to use 8 bits for the “digital” part, and either 12 or 16 bits for the “analog” part.

To avoid division, we will further restrict the envelopes to be integer powers of 2. This means using a lookup table or some combinational logic to “round” the envelopes (always up) to an integer power of 2. To compute ratios, we need only subtract the powers. The benefits of using integers are that multiplication by these ratios (as is done in the companding DSP) may be accomplished by simple shifts, and that very few bits (at most 3 bits) are required to store the envelope information. This leads to another simplification. The envelope of the input will be computed in the analog domain. We will only make ADCs for the companded input and the power of 2 corresponding to the input envelope, so we only need an 8-bit ADC and a 3-bit ADC. To get the original input back (for processing in the non-companded system), we simply shift the companded input back.

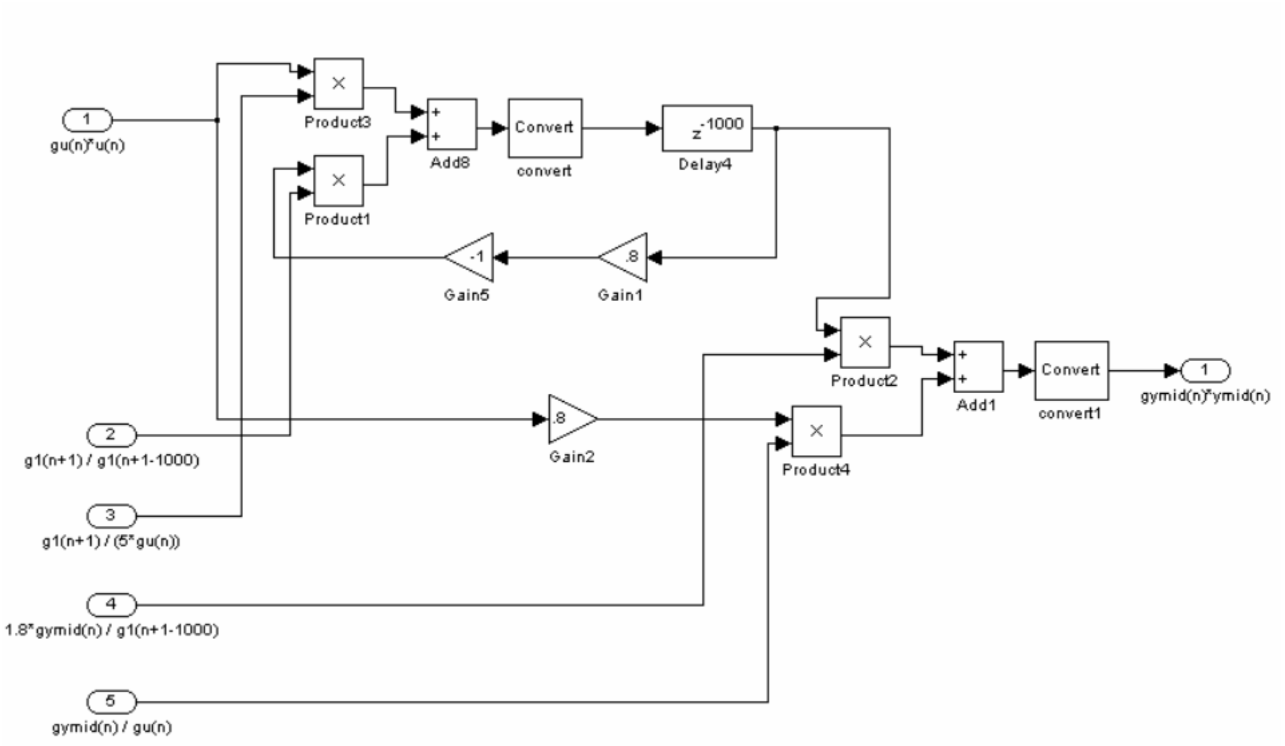
Here is some more detail for some of the blocks:

The original prototype (non-companding) reverberator:



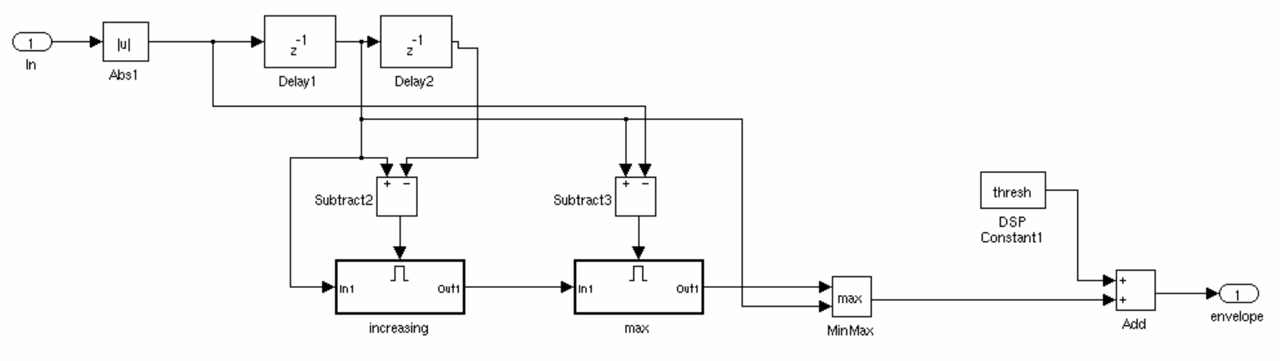
Envelope detection will be done on the input and on the 4 other signals shown (state1, state2001, ymid, and y). We might end up changing the amount of the delays. Also, if we find we have been too ambitious, and are very pressed for time, we might drop the second stage (everything to the right of ymid). Also, we might change the .8 gain to .75 to make it easier to implement.

The corresponding companding reverberator (DSPC):



The above is only the left half. The right half is topologically the same; the number 1000 should be replaced by 2205, and the inputs and outputs are slightly different. The “product” blocks will simply be implemented as shifters, since the ratios are all integer powers of 2.

The envelope detection is accomplished by taking the absolute value and connecting the local maxima. Suppose we want the envelope of x(n). We would connect the points where |x(n)|>|x(n-1)| and |x(n)|>|x(n+1)|. If I am not at such a point, I simply hold my envelope detector output constant (using FFs and MUXs). The block diagram is:



For the enabled subsystems (increasing and max), if the enable input is positive, the input is passed to the output. Otherwise, the output is held at its previous value. The “MinMax” block ensures that the envelope never goes below the input, and a small positive number thresh is added to ensure that the envelope is never zero.

Required Components:

At this point it looks like we will need:

- ADC, DAC, audio codec
- adders (and sutractors)
- lots of MUXs to route signals
- lots of FFs for holding state, handling timing, and doing envelope detection
- lookup table (or combinational logic) for getting the closest power of 2 larger than a signal (this will decide the shift amount for a particular signal, to ensure that the signal always takes full advantage of the available bits)
- comparators for envelope detection and lookup table
- memory to implement the delays. If we can get away with it, we would like to use only the BRAM, but if we need more than 8KB of memory, we will use the SRAM. Fortunately, we have already used the SRAM in lab6.

We will configure the ADC and DAC as OPB peripherals, just as we did with the SRAM in lab6. Once George and Sambuddho have the ethernet controller implementing VoIP, we will replace either the ADC or the DAC with the ethernet controller (bits will be sent or received from the ethernet controller instead of DAC or ADC.)

Timing issues:

At this point, we don't know enough about how the ADC and DAC timing works to give very detailed timing analysis for the whole system.

It looks like the critical path will be:
get input from ADC ---> shift input ---> shift it back ---> get x(n+1) and output with prototype (no companding) ---> do envelope detection on states, input, output from prototype ----> find closest power of 2 (get shift amount for every signal) ---> use shift amount, and shifted signals, and ratios to get the next state and output from the companding DSP ---> shift the output back to full dynamic range ----> send output to DAC

It is a pretty long path. If the sample rate is r (say, r=44.1kHz, for example), it would need to complete in time T=1/r. This might prove unrealistic. There are fortunately more options:
We could do the processing of the current sample in the prototype in PARALLEL with processing the PREVIOUS sample in the companding processor. This would add a latency of 1 sample, which shouldn't be a problem. It will also mean duplicating hardware (the companding and prototype systems will no longer be able to share the same adders and shifters)
Finally, if it is still not fast enough, then we can lower the sampling rate (22.05k, 16k, etc.)

To test that our system works, we plan to add something to the state machine to let us skip everything but the prototype. Then we can listen to the output of the prototype alone, and compare to the output of the full companding system.

Summary of VoIP Module:

We are implementing a SIP/RTP based VoIP soft phone: This is the main program used to initiate connection to a peer FPGA based SIP/RTP based VoIP soft phone. It encapsulates the SIP connection packets in TCP packets (using the TCP Runtime Library) and connects to the peer FPGA soft phone. The other half of the module is used to carry out communication by transferring and receiving RTP based voice packets encapsulated in UDP packets (using the UDP Runtime Library). This function forks into three threads: the sending thread, the receiving thread and the connection management (SIP) thread. These threads are scheduled in a pre-emptive round robin fashion using the timer interrupt which passes control to the task scheduler to context switch between these tasks.

TCP Runtime Library:

The TCP Runtime Library is the core TCP subsystem. The main functionalities of this library are :

1. Connection Request / TCP connect() operation for the SIP based connection initiation and setup.
2. Maintain per connection connected socket descriptor block whose index into the connected socket descriptor index gives the source port of the connection as well as the socket descriptor number (similar to what is implemented in UNIX systems)
3. Maintaining per connection TCP send / receive buffers and TCP timers for connection retransmission timeouts and TCP states like the the TIME_WAIT state.
   This is also used to maintain and timeout the SIP session on behalf of the user program – i.e. the VoIP soft phone.
4. Implement the basic TCP functions like the TCP connect() , accept() , read() , write
   and close().
5. Implement a very simple TCP state machine.

UDP Runtime Library:

The UDP runtime library is used to communicate to the connected sockets' descriptor array and the update the connection details such as populate the UDP send buffers and remove the packets from the UDP frame buffer. Both the UDP and the TCP subsystems talk to wrapper functions to encapsulate the packet into UDP/IP or TCP/IP encapsulation routines to encapsulate the contents into datagrams which are transferred to the IP subsystem to be transferred out through the Ethernet subsystem. The following are the functions of the UDP Runtime Library:

1. Encapsulate a RTP voice packet to send and receive to the peer entity / peer FPGA based soft phone.
2. Remove the RTP payload from the received RTP datagram and send it to the user application which is the local FPGA based soft phone.
3. Implements simple UDP functions like the sendto() and recievefrom() to send and receive the UDP packets.

IP Layer (The Network Layer) Runtime Library:

The IP layer library is performs the following functions:

1. Receive UDP / TCP segments and encapsulate them into IP datagrams
2. Lookup the routing table to determine the appropriate network interface to be used to send the data out.
3. Associate appropriate source address to the packet which is being sent out.
4. If there is no entry for the IP to MAC mapping communicate with the ARP module which takes inputs only from the IP layers and returns appropriate MAC address for the frame to be associated with for the appropriate destination IP address .
5. Encapsulate the packet with a MAC layer header and send it to the Ethernet Packet Creation / Reception subsystem which sends it out of the MAC interface.
6. The important functionality of the IP layer is to implement the IP send and recv functions to send the packet to the MAC subsystem which copies it to the On-chip SRAM of the Ethernet controller which is read up by the OPB_Ethernet / On board Ethernet Processor Chip. The same functionality is also implemented for the receiver function which is used to read out the packets from the OPB_Ethernet. Whenever there is an Etherent DMA completion interrupt, the program (Ethernet packet creation/reception subsystem) takes out the frame from the On-chip SRAM buffer and passes it to the higher layers thereby renewing the operations of the Ethernet controller chip and the local DMA operations associated with it.

ARP Module:

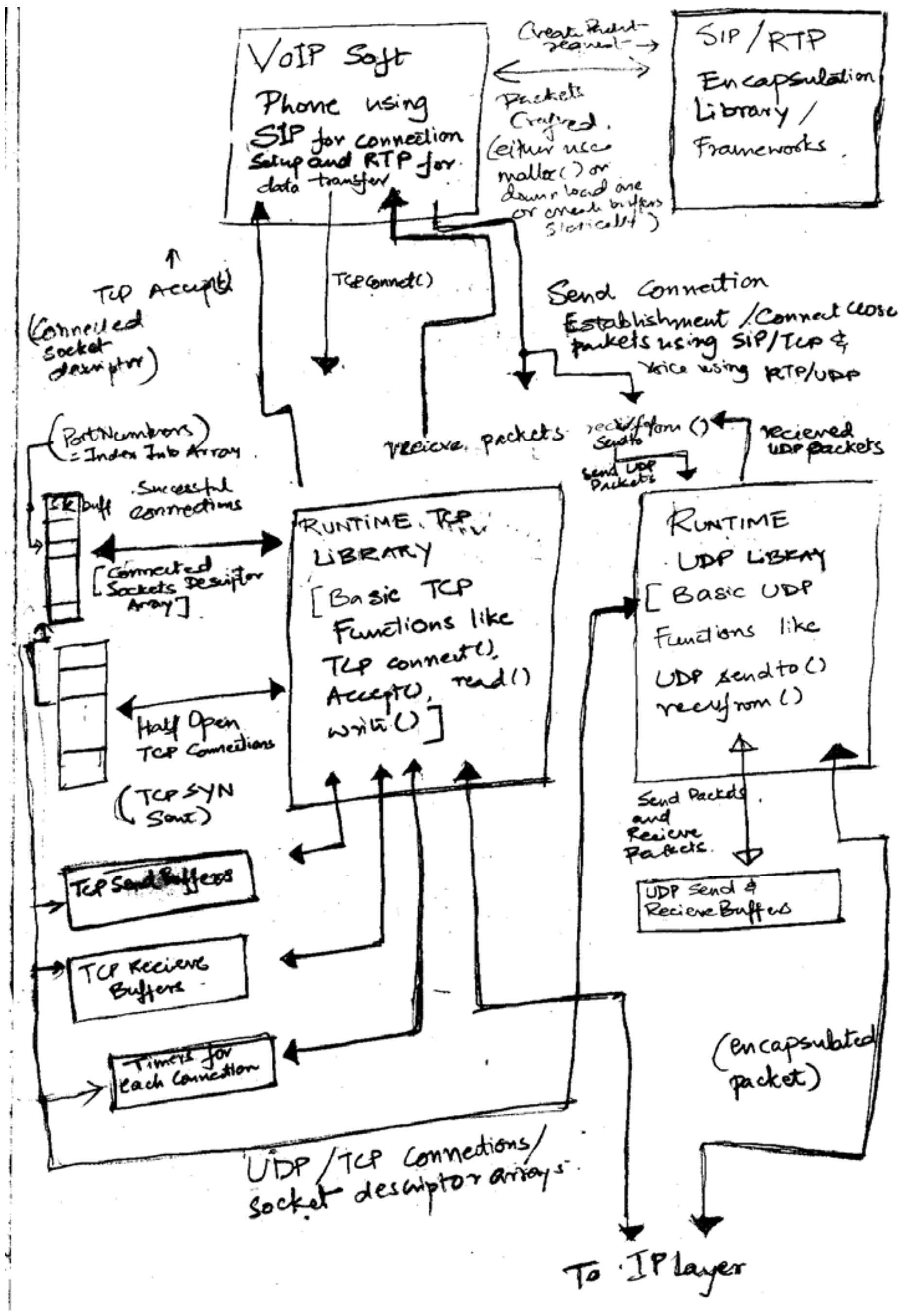
The APR (Address Resolution Module) is the one that actually maps IP address to MAC layer addresses and appends the MAC layer header to the outgoing datagram with the MAC layer header with appropriate MAC layer destination and source address which is what is understandable to the Ethernet Controller (which independently handles the physical layer signaling , channel coding , carrier sensing , collision detections and binary exponential backoff in case of collision detection (the standard MAC layer mechanism used for collision avoidance in case of shared medium like the Ethernet). It further talks to the Ethernet Packet creation , transmission and reception subroutines to send and receive the ARP requests and replies and to the the IP layer routing table / route – ARP cache to refresh entries for the destination MAC layer addresses for the IP addresses selected.

Send / Receive Packets Module:

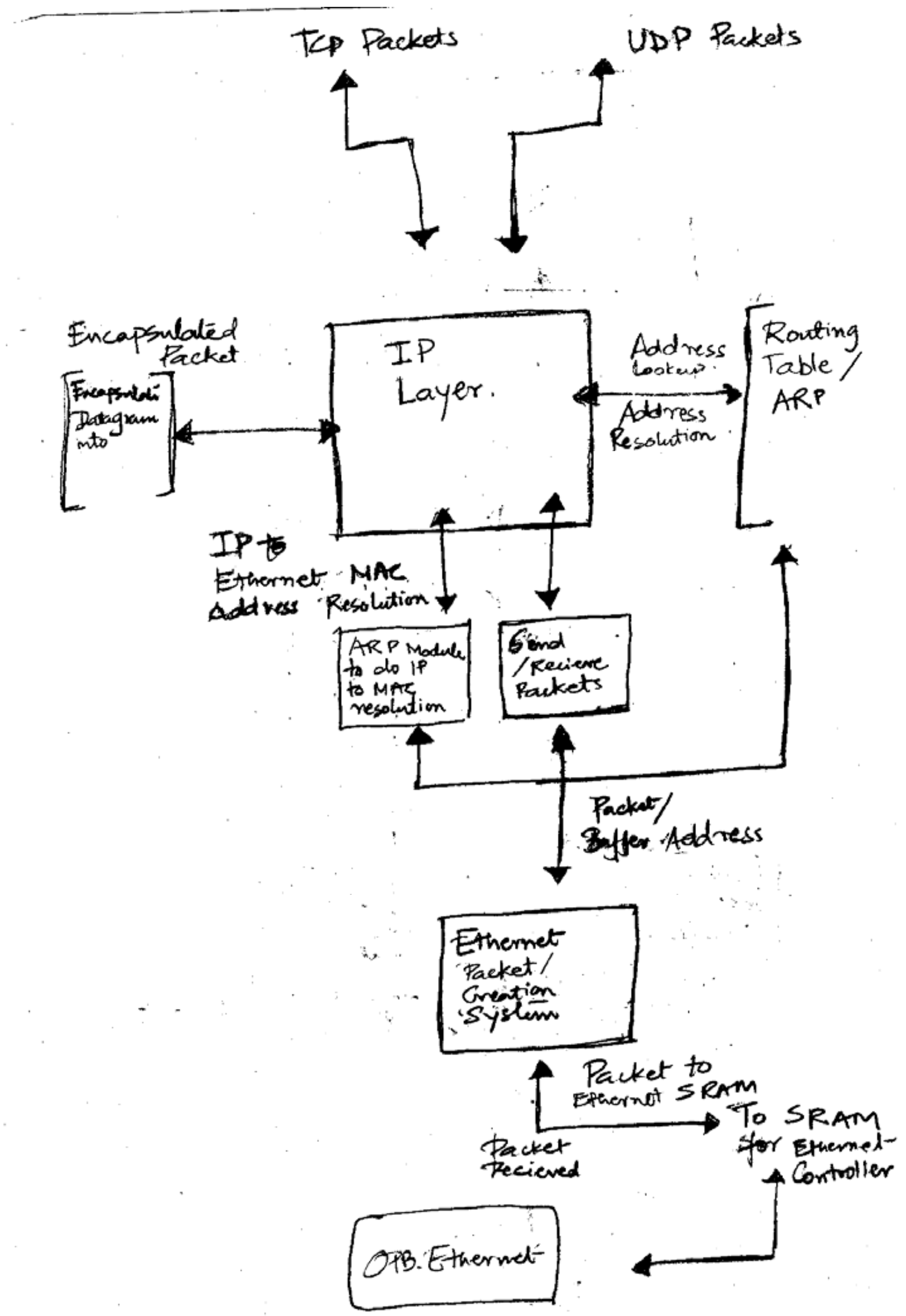
This module simply acts as a bridge to send packets from the IP layer / ARP packets and send it to the Ethernet Packet creation module to transfer it to the Ethernet controller via the OPB_Etherent SRAM contoller.

Block Diagrams:

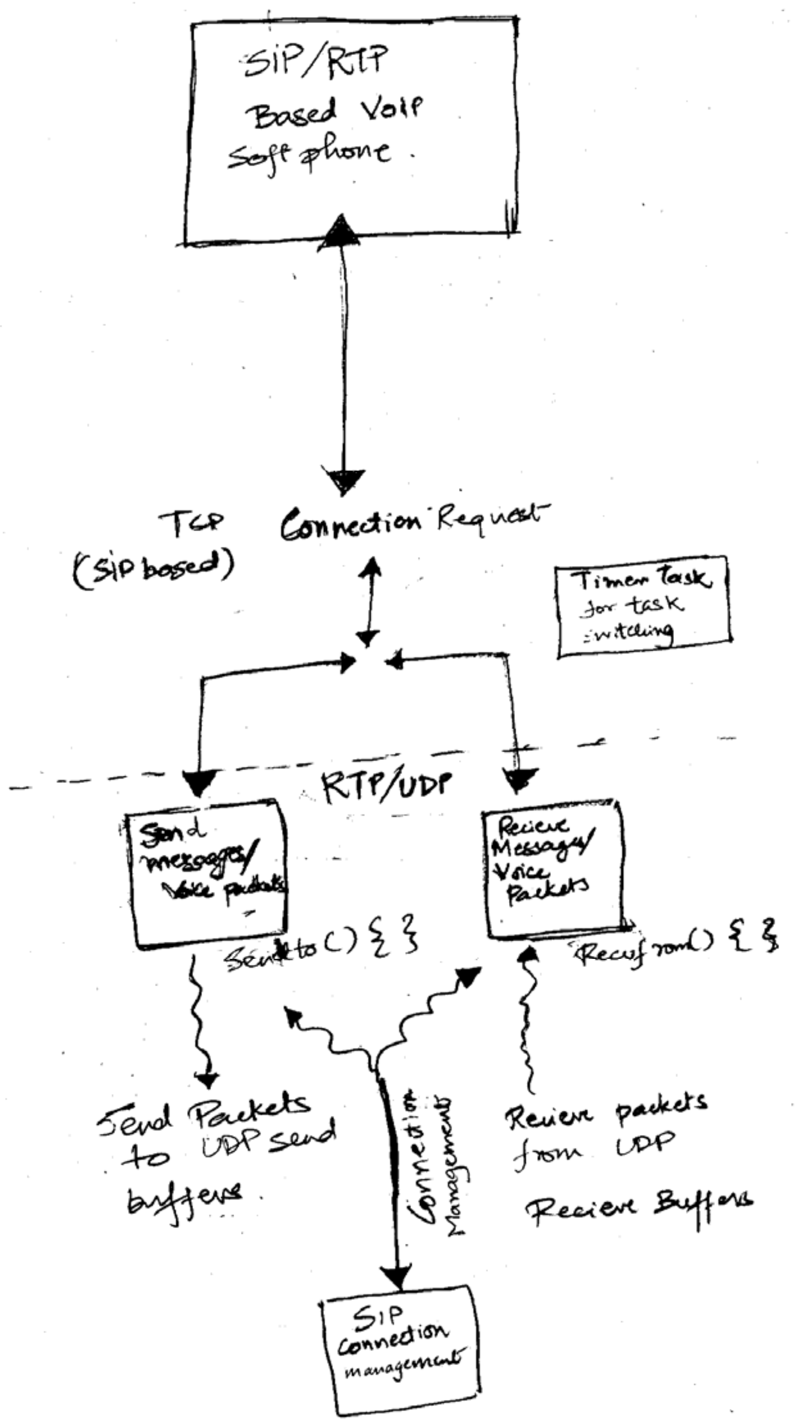
VoIP SoftPhone Module:


VoIP Soft Phone using SIP for connection Setup and RTP for data transfer
Create Packet request
Packets Crafted. (either use malloc() or download one or create buffers statically)
SIP / RTP Encapsulation Library / Frameworks.
TCP Accept()
(Connected socket descriptor)
TCPConnect()
Send Connection Establishment / Connect close packets using SIP/TCP & Voice using RTP/UDP
(PortNumbers) = Index Into Array
recieve packets
recvfrom()
Sendto
Send UDP Packets
recieved UDP packets
1k buff
Successful connections
[Connected Sockets Desciptor Array]
RUNTIME TCP LIBRARY [Basic TCP Functions like TCP connect(), Accept(), read() write()]
RUNTIME UDP LIBRARY [Basic UDP Functions like UDP sendto() recvfrom()]
Half Open TCP Connections
(TCP SYN Sent)
TCP Send Buffers
TCP Recieve Buffers
Timers for each Connection
Send Packets and Recieve Packets.
UDP Send & Recieve Buffers
(encapsulated packet)
UDP / TCP Connections / Socket descriptor arrays
To IP layer

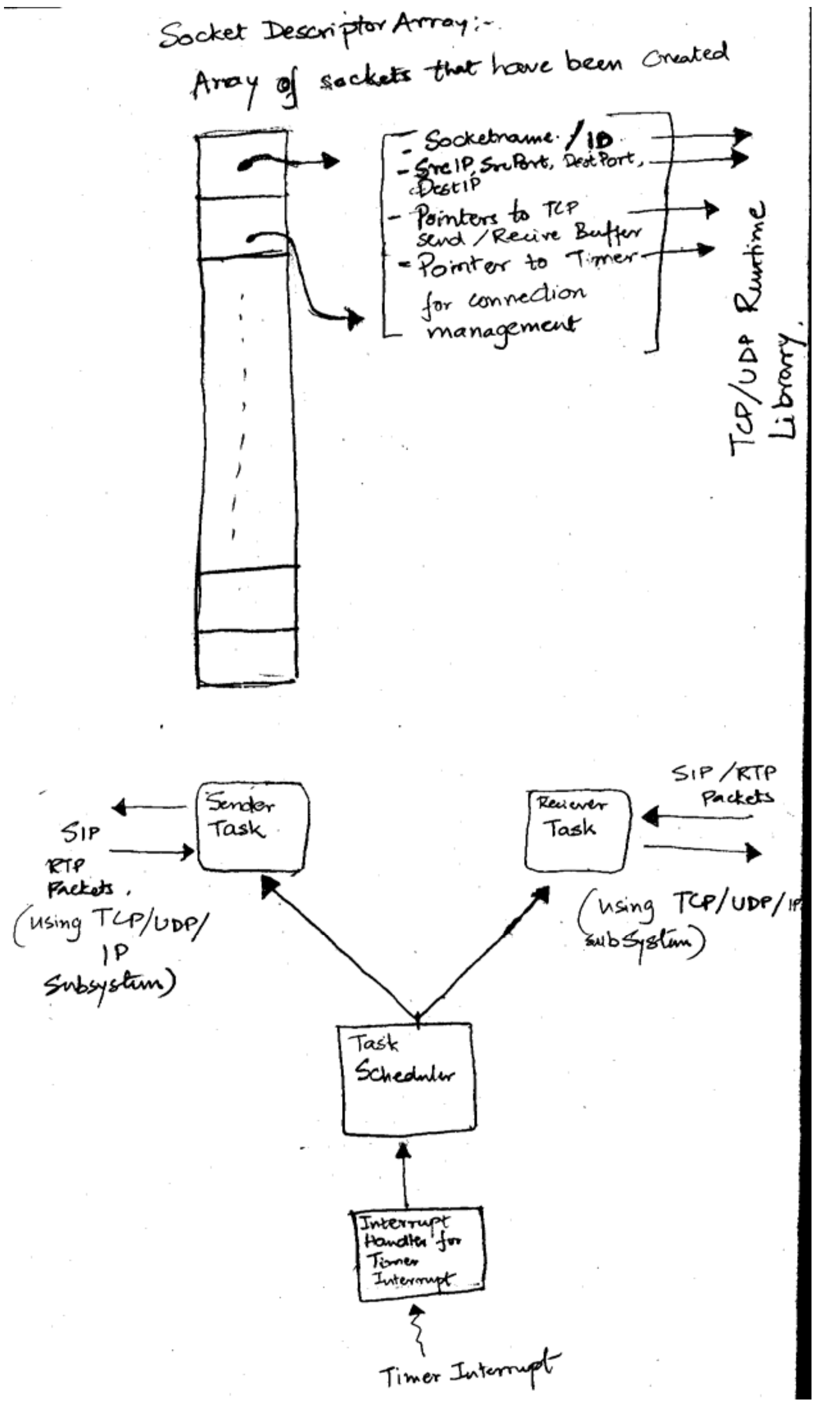
IP Layer Module:



SIP/RTP Module:


SIP/RTP Based VoIP Soft phone.
TCP (SIP based)
Connection Request
Timer Task for task switching
RTP/UDP
Send messages/ Voice packets
Recieve Messages/ Voice Packets
Sendto() { }
Recvfrom() { }
Send Packets to UDP send buffers.
Connection Management
Recieve packets from UDP
Recieve Buffers
SIP Connection management

Task Scheduling Module:


Socket Descriptor Array:-
Array of sockets that have been Created
- Socketname / ID
- SrcIP, SrcPort, DestPort, DestIP
- Pointers to TCP send / Recive Buffer
- Pointer to Timer for connection management
TCP/UDP Runtime Library.
Sender Task
SIP RTP Packets.
(using TCP/UDP/IP Subsystem)
Reciever Task
SIP/RTP Packets
(using TCP/UDP/IP subSystem)
Task Scheduler
Interrupt Handler for Timer Interrupt
Timer Interrupt

Other Figures and Timing Diagrams:

Figure (1) Ethernet Controller Ring Buffer

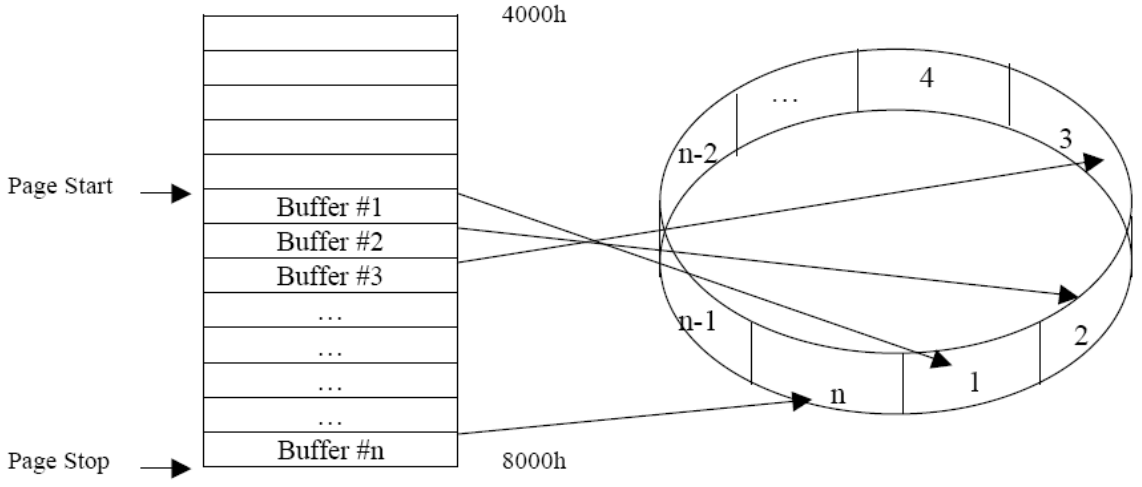


Figure (2) FPGA To SRAM Block Diagram

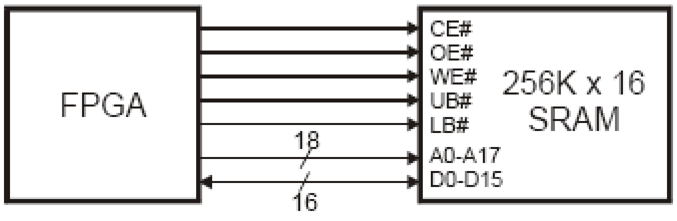


Figure (3) FPGA To Ethernet Controller Block Diagram

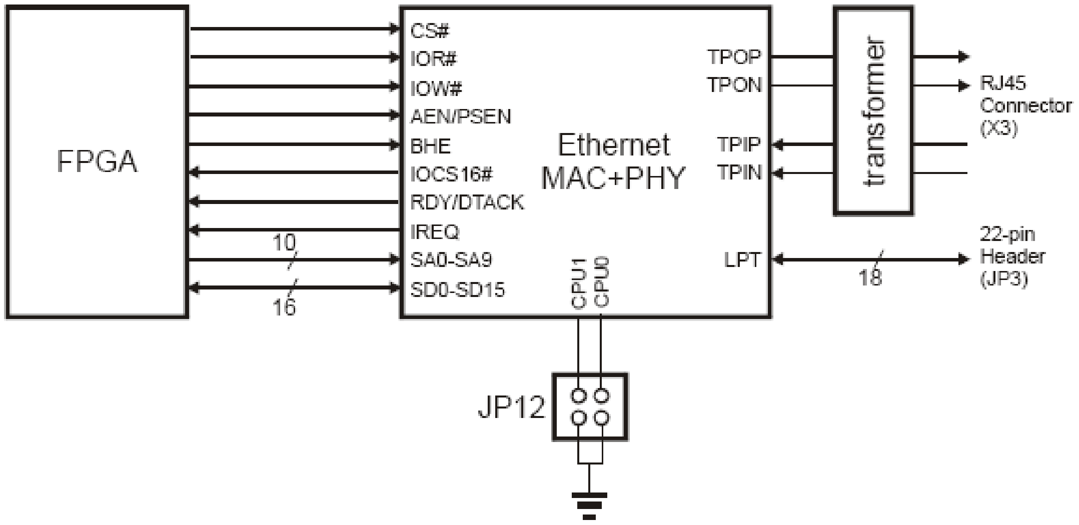


Figure (4) Ethernet Controller Timing Diagram

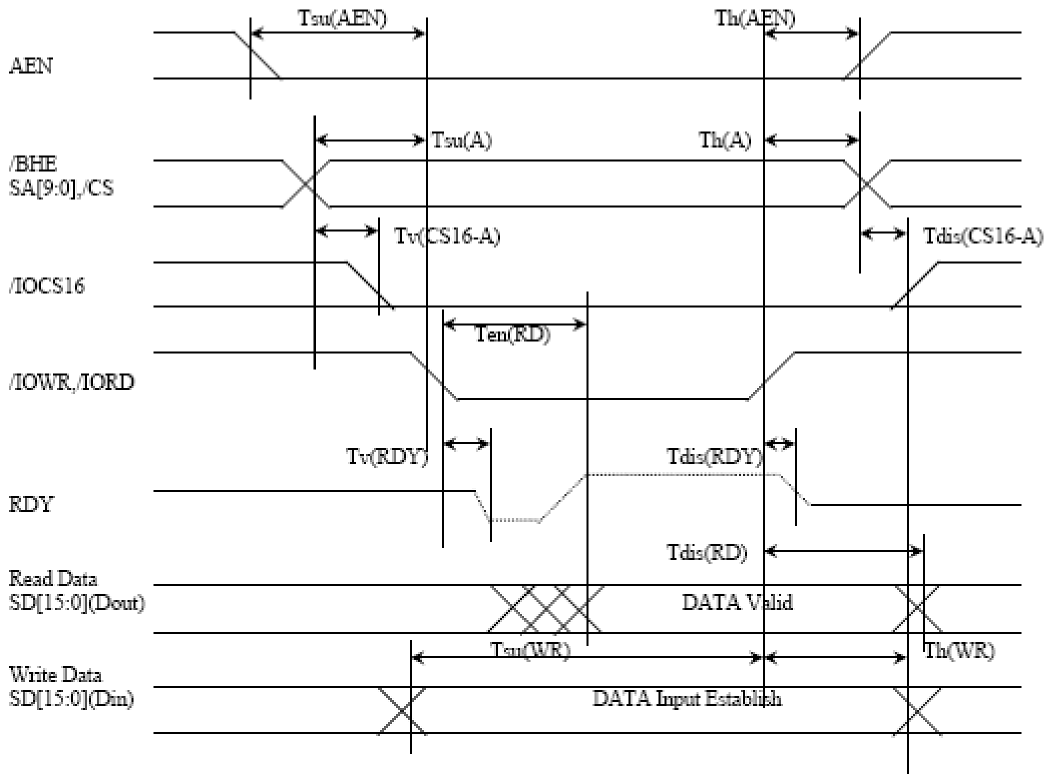


Figure (5) SRAM Read Timing Diagram

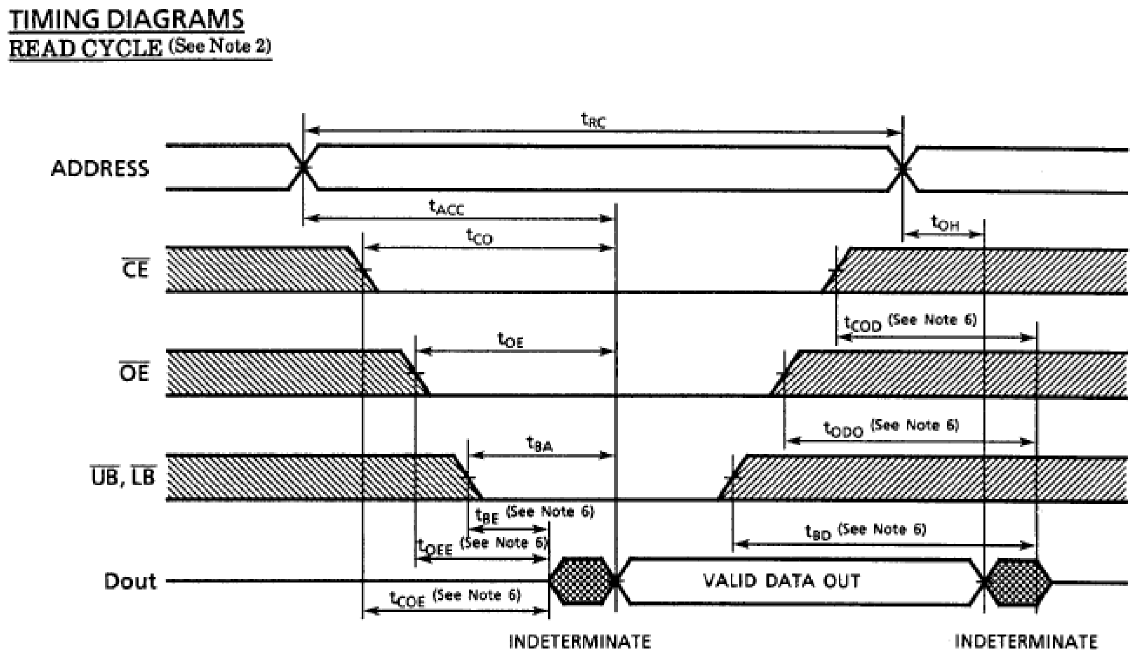
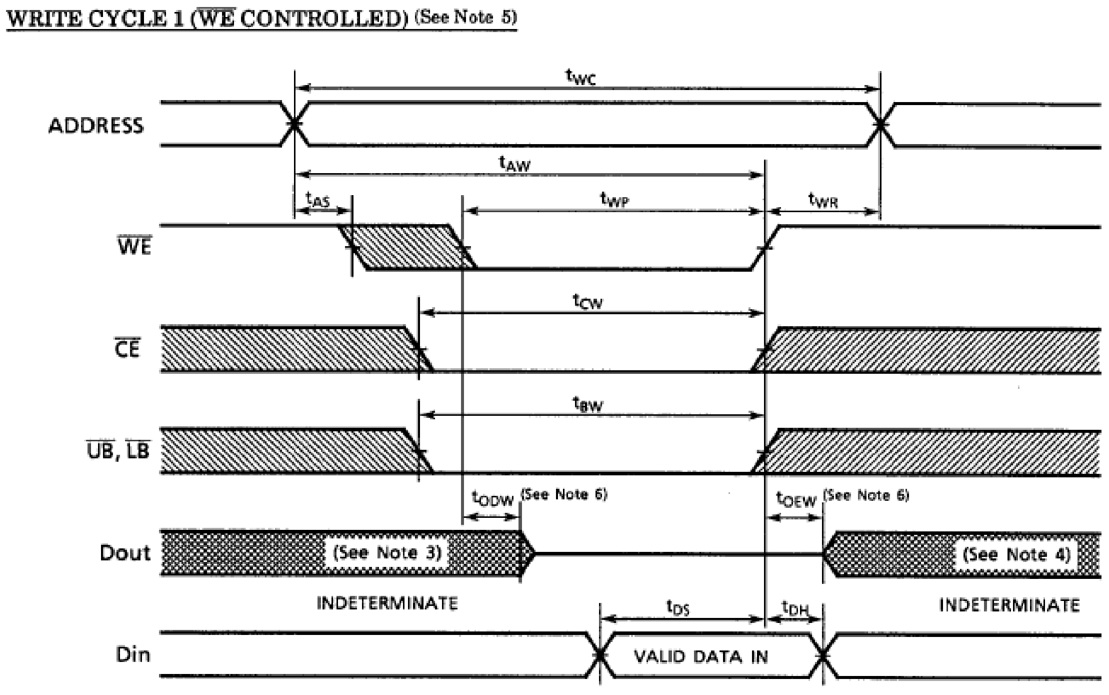


Figure (6) SRAM Write Timing Diagram



Component Integration:

Our plan is to integrate the two modules of the system using shared memory. Using a mutex or some similar device for sharing resource access, each of the modules will be able to read and write from a particular block of memory on the SRAM component. The companding component will therefore be able to write out data to the buffer on the fly in discretely sized chunks. The VoIP module will sample the buffer at a fixed rate and then packetize the data for network transmission. At the receiving end, the VoIP module will extract the data from the received packets and place it in a buffer in the SRAM from which the companding module can read. The compressed signal will then be decompressed and routed out through the DAC.