

# Design of an Exportable Digital Design Curriculum

by

Christopher M. Cacioppo

B.S.E.E. Electrical Engineering  
University of Connecticut, 1994

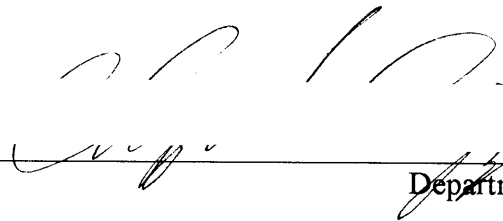
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR  
THE DEGREE OF

MASTERS OF SCIENCE IN ELECTRICAL ENGINEERING  
AT THE  
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE, 1996

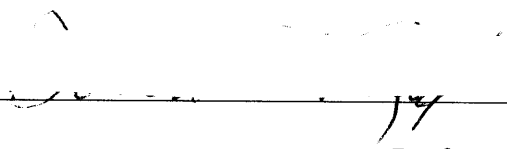
© 1996 Massachusetts Institute of Technology.  
All rights reserved.

Signature of Author: \_\_\_\_\_



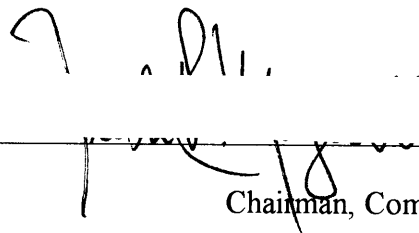
Department of Electrical Engineering  
May 10, 1996

Certified by: \_\_\_\_\_



Donald E. Troxel  
Professor of Electrical Engineering  
Thesis Supervisor

Accepted by: \_\_\_\_\_



Frederic R. Morgenthaler  
Chairman, Committee for Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUL 16 1996

1



LIBRARIES

# Design of an Exportable Digital Design Curriculum

by

Christopher M. Cacioppo

Submitted to the Department of Electrical Engineering on May 10,  
1996 in Partial Fulfillment of the Requirements for the Degree of  
Master of Science in Electrical Engineering.

## ABSTRACT

A digital design curriculum has been developed that is applicable to a wide range of teaching situations. The main focus is on the basic principles of digital design. Economy of effort is achieved by focusing on a single goal that is built up over a number of labs. The chosen goal is a simple game called NetPong, which not only presents information on the basic digital design concepts, but also introduces information on networking and common interfaces.

The materials associated with this curriculum include six lab manuals with associated prelabs, custom software, and sample solutions. This material is not intended to be a complete course, but it should be sufficient for someone knowledgeable about digital design to quickly put together an introductory course. The material is adaptable to a range of audiences and can easily be tailored for different audiences, from advanced high school students to junior level college classes. Furthermore, minimal equipment requirements were chosen to allow programs with more moderate funding to be able to use this material.

Thesis Supervisor: Donald E. Troxel  
Title: Professor of Electrical Engineering

1. Statement of Problem	5
1.1 No lab progression:	5
1.2 Not enough emphasis on design:	5
1.3 No lab manual/text:	6
1.4 Not suitable for exportation:	6
2. Overview	6
2.1 What is provided	6
2.2 Hardware	7
2.3 Software	7
2.4 Lab Manuals	8
3. Hardware Approach and Issues	9
3.1 Requirements	9
3.2 Solution	10
3.2.1 The goal	10
3.2.2 Networking	10
3.3 Components	10
3.4 Documentation	11
4. Software Approach and Issues	11
4.1 Requirements	11
4.2 Solution	11
4.2.1 Design Philosophy	11
4.2.2 C++ Advantages:	12
4.3 Components	12
4.3.1 The Hub	12
4.3.2 The Ports	12
4.3.2.1 Player	12
4.3.2.2 Comm_Port_Master	13
4.3.2.3 Comm_Port_Slave	13
4.3.2.4 Logger	13
4.3.2.5 Video	13
4.3.3 GenComm	13
4.3.4 PITSpeed	14
4.3.5 Message	14
4.4 Documentation	14
4.4.1 Hardware Requirements	14
4.4.2 User Documentation	15
4.4.2.1 Setup Options	15
4.4.2.2 Run-Time Controls	18

5. Lab Manual Design Issues	18
5.1 Requirements	18
5.2 Solutions	19
5.2.1 Prelabs	19
5.2.2 Structure	19
5.2.3 Design information background	19
5.2.4 Clear Presentation of Assignments	19
5.2.5 Design before building	20
5.3 Components	20
5.3.1 Lab Coverage	21
5.4 Documentation	22
6. Lab Manuals and Materials	23
6.1 Overview	25
6.2 Prelab 1 & Lab 1	29
6.3 Prelab 2 & Lab 2	39
6.4 Prelab 3 & Lab 3	47
6.5 Prelab 4 & Lab 4	55
6.6 Prelab 5 & Lab 5	71
6.7 Prelab 6 & Lab 6	81
6.8 Suggested Enhancements	93
7. Conclusion	95
8. Bibliography	96
9. Appendix A: Example Solution	97

# 1. Statement of Problem

The goal of this project was to design a new digital design course which will provide solutions to a number of identified shortcomings of MIT's current course (6.111)<sup>1</sup>. The issues to be addressed are as follows: no lab progression, no lab manual or text and too little emphasis on design issues. In addition, the exportability of the current course is very limited. I will address each of these points individually. The approach taken was not to 'fix' the current course, but to create a new one which teaches much of the same material in a new manner.

## 1.1 No lab progression:

In the current implementation of 6.111, there are three project labs, each requiring students to build a project from scratch. Each of the labs require a significant amount of time on the part of the students, each getting progressively more difficult. Unfortunately, a large portion of this time is spent rebuilding basic building blocks learned in previous labs. In order to address this problem, I designed a course that has a single structured lab goal that the students work on in stages throughout the term. Each lab is built upon the previous one. The amount of components that are built and then torn down is restricted to those that are needed for testing. This will satisfy a number of goals. First, the students will spend less time rebuilding things that they already understand. Second, it will provide the students with a definite goal; they will be able to see why each lab is important and how it is allowing them to progress towards the defined goal. Finally, it will encourage students to stay on track, as each lab will be necessary for subsequent labs.

## 1.2 Not enough emphasis on design:

Currently, a lot of time is spent showing the students the basics of how various components work and how to interface with them. While this is important, little time is spent teaching students how to properly approach engineering decisions and tradeoffs. There are many design decisions that students are required to make during the course of a project; students often make uninformed decisions or may progress without even realizing that they are making decisions. In my new course, time in each lab section will be spent discussing design principles and carefully looking at the design issues associated with integrating the current lab into the overall project. When appropriate, a number of possible solutions to the problem are presented, weighed against each other and finally the best option chosen. It is emphasized that design is an iterative process and discourage the "build and patch later" approach that many students take.

---

<sup>1</sup>The work for this course was funded by an ECSEL block grant from the National Science Foundation (NSF).

### **1.3 No lab manual/text:**

One of the largest criticisms the students have of the current design course is that there is no lab manual or text that is available for their use. The required book for the current course is the TTL Databook, which focuses on specific implementation issues and not on overall design. The students rely heavily on the lecturer and TAs for information on the labs. A large portion of this project was devoted to creating a lab manual presents the material of the course in a coherent and straightforward manner. This will be a great asset to the students and will complement regular lectures well.

### **1.4 Not suitable for exportation:**

Although this is not technically a problem, it would be an asset to a course if it were exportable to other schools. Exportability brings up two major issues: equipment and level of course material. MIT students are fortunate that their university has a large amount of state-of-the-art equipment. This allows the students to design complicated systems with relative ease compared to many schools which do not have these facilities. In order to make this course functional, as well as much more accessible to different schools, I have focused it for schools with mid-range facilities. I am assuming that all of the schools will have access to basic equipment: entry level PCs, low bandwidth oscilloscopes & logic analyzers, and some basic PAL and PROM programmers.

The second issue is the level of the course material. In order for the course to be exportable, it had to be very flexible and be taylorable to audiences with various educational levels.

## **2. Overview**

### **2.1 What is provided**

As previously stated, the purpose of this project was to provide materials that would be suitable for putting together a digital design curriculum. This is not intended to be a turn-key system; instead, it is a collection of materials that will allow someone with a background in digital design to rapidly put together a course suitable to teach the concepts of digital design to students. There is a significant amount of flexibility that the instructor has in molding the course to be most suitable for his students. The flexibility comes from optional assignments, high level design decisions and further suggested enhancements.

The approach that is taken with this material to teaching digital design is to have the students build a significantly complicated digital system, one piece at a time. The overall goal of the project had many requirements from a digital standpoint, but it also had a number of requirements from a more general standpoint. The students were going to be working on the project for a significant amount of time (the exact amount depends upon the instructor) and it

was important to keep the students' interest throughout that time. The net result of the search to fulfill all of the requirements was a game called NetPong.

NetPong is a multiple player version of the classic game of pong. The basic premise is that there is a ball that moves around a 64 X 64 screen. Each player has a paddle that is positioned on one side of the screen. Their job is to keep the ball in play by moving their paddle to block the ball from falling off their side of the screen. If the ball falls off the side of the screen, it is placed back in the center and play starts again.

## **2.2 Hardware**

The main purpose of this lab sequence is to teach the concepts of digital design. In the course of the lab sequence, the students will work with basic units such as UARTs, SRAMs, A/Ds and D/As. Furthermore, they will be controlling their system using Finite State Machines (FSMs) and Micro Control Units (MCUs). In addition to actually building the hardware, they will be responsible for writing the necessary microcode (~500 lines of code), though the professor may want to provide a portion of this code to the students. Although there is a lot of necessary structure in the lab, there are flexibilities that the students have when accomplishing their individual projects.

It is suggested that the students build their projects on bread-boards. In addition, the students will need to have access to basic oscilloscopes, logic analyzers, PROM and PAL programmers. The equipment can be low end or older equipment, as the suggested system clock speed will be less than 200KHz.

One of the most basic elements of NetPong is the network which enables communication between the students. The network is a simple hub-based star network which utilizes UARTs for all input and output. There is only a little extra work that is necessary on the students' part to properly interact with the network. The students will gain some valuable knowledge about networking while building and utilizing the NetPong network.

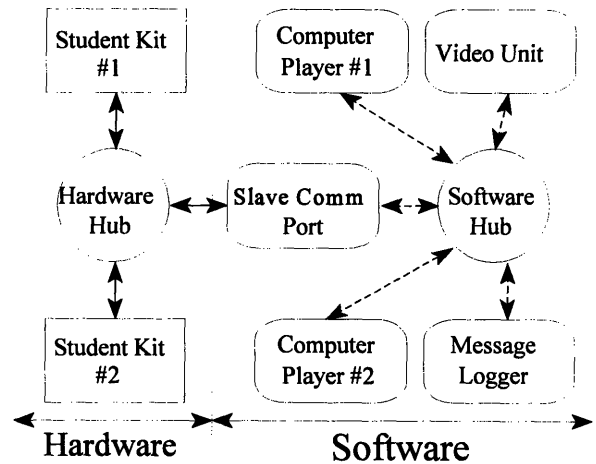
## **2.3 Software**

The software for the course is written in C++ and is intended to run on an IBM compatible PC platform. This was chosen because many universities use this platform, as it is low cost compared to most UNIX workstations. Source code is provided, therefore, if it is desirable, this software can be ported to a UNIX environment. See section **5. Software** for more details. The PC requirements are moderate; for specific details see **5.1 Software Requirements**.

The role of the software is to monitor the network and provide a number of services to the students. The software will provide a video display for the students so they can quickly be provided with visual feedback. This also eliminates the need for students to build their own video which is a reasonably complicated project and involves higher bandwidth scopes and analyzers. In addition to video, the software can add artificial players if less than four human

players are available. Furthermore, the software provides logging of network messages for debugging purposes.

The architecture of the software is similar to the hardware that the students will be designing. At the heart of the software there is a software hub which acts in a similar manner to the hardware hub. All of the software elements, such as the video display, communication port interfaces, message logger and artificial players, are ports that can be 'plugged into' the software hub. All of these ports have identical software interfaces which allow them all to be treated the same. All of the 'plugging in' of modules is controlled by command line switches or commands placed in the environment variable NETPONG.



There are a number of options that the instructor can decide upon that will directly influence how the network will operate. These options include the size (in bytes) and format of the messages and the format of each transmitted byte. In order for the software to accommodate all the possibilities, a flexible configuration is necessary. This results in a significant number of options that can be specified on the command line or in the NETPONG environment variable. Section 5.4 **Software Documentation** has specifics on the various options.

In addition to the software that was specifically made for the course, there are a number of tools that were developed at MIT for FSM and MCU microcoding purposes that will assist the students in their design. More information on these tools are provided in section 5. **Software**

## 2.4 Lab Manuals

Probably the most valuable resource that is provided is the sequence of six lab manuals that guide the students through the design and development of their NetPong kits. The lab manuals are not intended to be the sole source for information for the course. Instead, they are expected to be accompanied by lectures and/or a formal text on digital design. Furthermore, the students should have available a TTL data book and specification sheets on the various parts they will be using.

There are two major ways to learn material: self learning and instructed learning. Self-learning is the process where students figure out concepts and issues for themselves. Instructed learning is when concepts and issues are directly presented to the students. In general, self-learning results in a greater long term comprehension for the students. The advantage of instructed learning is that we can make sure that all of the concepts are adequately covered. This lab sequence tries to strike a compromise through a directed self-learning approach. Each lab is accompanied by an associated prelab. The prelabs present one or more high level digital design



issues that are relevant to the upcoming lab. The students should be required to generate solutions to the issues presented. In the actual lab manual, there is a discussion of the prelab questions and suggestions are made as to the best possible solution(s) for the design of NetPong. Some solutions must be followed or they will significantly change the structure of NetPong. Some suggestions should be decided upon by the instructor as the decisions will effect the entire class. Finally, there are some decisions that only affect the individual student's kit and they are thus free to chose the solution of their choice. These three different levels of freedom in the suggested solutions are clearly marked in the text.

### 3. Hardware Approach and Issues

#### 3.1 Requirements

There were a number of major issues associated with the hardware. The three most important design considerations were:

- 1) Teach major digital design concepts (UART, FSM, MCU, A/D, D/A, SRAM, PROM)
- 2) Be able to break up the project into completable, mostly stand-alone, projects.
- 3) Be interesting to the students

In addition, the following design ideas were desirable, though not necessary:

- 1) Teach how to work with a standard interface: In today's working environment it is very rare that an engineer is designing an entire system by themselves. Instead, most of the time engineers are working in teams and must design systems that interact with each other and also interact with off-the-shelf designs. It is for these reasons that I believe that this skill is very important.
- 2) Promote working together: In addition to just designing system interfaces that match up, it is important that students get some experience working directly with one another on a reasonably difficult task. This is also an invaluable skill.
- 3) Teach basic networking: Networking is a technology that has significantly grown in importance recently. Many aspects of engineering are directly or indirectly involved with networking. Understanding some of the basic issues of networking will also be a valuable asset to the students.
- 4) Spend less time building and tearing down: The current course at MIT that teaches digital design requires a tremendous amount of the student's time. Some of this time is necessary, but a good deal of time is spend building and tearing apart and rebuilding systems. A goal of this project was to maximize the amount of useful time the students spend on the course.

## 3.2 Solution

NetPong was conceived as a solution to all of the above requirements. The NetPong project is broken up into six progressive labs that the students will complete. In addition, NetPong is inherently an interactive project, with the finished goal being a simple multi-player game of pong. The students have to interface to the custom network on both a digital level and at a simple protocol level. In addition, because they are progressive labs, very little hardware has to be built only to be torn down again later. For more information on the hardware in the lab was put together see section 4.3 **Components**.

### 3.2.1 The goal

The goal of this lab sequence is to build hardware that can act as a player in the game of NetPong. The game aspect should keep the students' interest more than many less interactive projects could. The actual game play will be much like the well known game of pong. Each student will have a paddle on one side of the screen. As the ball bounces around the screen it is the player's job to make sure the ball doesn't fall off their side of the screen. They have to use their paddle to deflect their ball into another direction. The actual mechanics of the game are as simple as possible so that the students can concentrate on hardware rather than complicated microcoding.

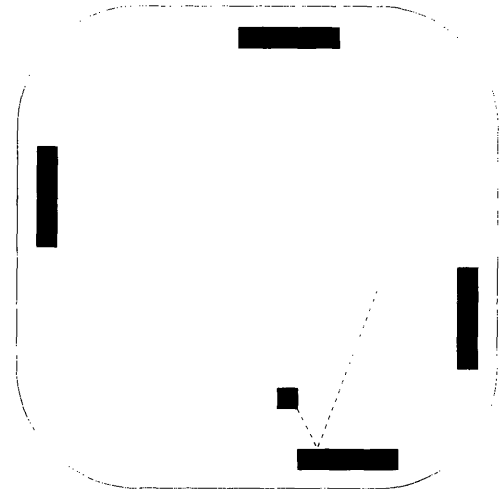


Figure 1: NetPong

### 3.2.2 Networking

The heart of NetPong is a simple network that the students build. To keep the complexity of the network down, it is made up of UARTs and does not require any special purpose analog hardware. Some basic ideas of low level networking are introduced in the earlier labs. In the later labs when they are working with the MCU, higher level networking problems are proposed, presenting such things as synchronization and auto-assignment of network IDs (i.e. player numbers).

## 3.3 Components

The hardware was designed with the intention that we want the students to build it in small pieces. In addition, the sequencing of the labs was such that simpler concepts were presented before more complex ones; this is especially important when one concept is based upon another. For example, FSMs should be presented to the students as simple controllers before MCUs are discussed. For the most part, each of the six labs present one part of NetPong, although a few present more than one.

### **3.4 Documentation**

There are, of course, many possible solutions to the hardware that will result in a fully functional NetPong project. Therefore, it is impossible to come up with a fully comprehensive solution set. Instead, the author designed a working solution that is fully documented in Appendix A. Depending upon the level of the audience, parts of this solution may want to be distributed to the students. In fact, there is a lot that can be learned about digital systems by distributing the provided solution and having the students making the system work.

## **4. Software Approach and Issues**

### **4.1 Requirements**

It was decided that in order to reduce the complexity of the labs, a computer would be used to monitor the network and provide a video display based upon the messages. Furthermore, once a computer is attached to the network, it becomes convenient for it to generate auto-players to fill in for players who are missing. In addition, it was useful for debugging to have some sort of message logging feature.

### **4.2 Solution**

The two major decisions that are made when writing a program are the general approach to the problem and the language that will be used. These two decisions are often related. The first issue is covered in 4.2.1 Design Philosophy and the second issue is covered in 4.2.2 C++ Advantages.

#### **4.2.1 Design Philosophy**

The original software written for this course was done exclusively to test the hardware concepts and the networking protocol for NetPong. As a hardware test was the goal, the software was written in modules that closely resembled the hardware. There was a software hub, which has a similar role as the hardware one the students will build. Furthermore, there were objects called "players" and "video" that plugged into the hub to test functionality, much like student kits plug into the hardware hub.

When it came time to build the software that the students would be using, it was apparent that the test software was a good base to work from. The interface between the player objects and the hub was generalized so that other types of objects could be plugged into the software hub. Communication objects were added, as well as message loggers. Furthermore, a complicated configuration system was added to provide a large amount of flexibility. Finally, hardware timers were implemented so that the software would work the same independently of the speed of the machine it was being used on.

### 4.2.2 C++ Advantages:

The program was written in C++ to take advantage of the reusability of code and the polymorphism properties of the languages. C++ is a good language for modeling systems, due to its class structure, so it was used to build the first simulation of the NetPong design. After this, code reusability was an issue because I could use the code from the original simulation I had already created. In addition, a communication object called GenComm (see 4.3.3 GenComm) was previously written, I was able to reuse the code with some modifications. Polymorphism was useful in creating 'ports' which plugged into the software 'hub' with the same interface. This made the program easier to understand, and also helped in making the configuration options very flexible. The down side to using C++ is that it is not quite as portable as ANSI C.

## 4.3 Components

The NetPong program is made up of a number of important objects. Each object that was built is described in this section.

### 4.3.1 The Hub

The hub is the software equivalent of the hardware network hub. It keeps track of the various ports and distributes messages between them. It also passes tokens to each port to indicate that the port has control of the network and is now able to transfer data. The interface between the hub and each of the ports is identical in all respects except for initialization.

As in the hardware hub, the software hub provides a virtual network token to all of the ports in a round robin fashion. This is done by the hub calling the **token** function for the appropriate port. When the port is finished with the network token, it returns from the token function. Messages are sent by the ports by calling the **send** function of the software hub. Every time a message is received by the software hub it, in turn, sends the message to all of the ports attached to it (with the exception of the sender) by calling the function called **parse\_msg**. The sender is excluded here to prevent the echoing of messages.

### 4.3.2 The Ports

The following objects are all derived from a single virtual object called `np_port`. This base object contains all the virtual functions needed to interact with the software hub.

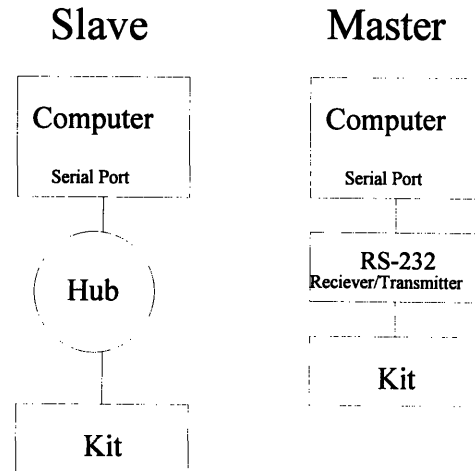
#### 4.3.2.1 Player

The player object is essentially one artificial player. The artificial player's paddle will generally follow the movements of the ball. The artificial player has a difficulty setting which determines how often the paddle will miss the ball. Every time the artificial player receives a 'CLAIM OWNERSHIP' message, it determines if the paddle is going to hit or miss the ball.

Once this is determined, the side of the paddle the ball will miss, or the place on paddle the ball will hit is determined. For debugging purposes, if the miss rate is defined as zero, the paddle will act deterministically and the ball will always hit the paddle in the middle.

#### 4.3.2.2 Comm\_Port\_Master

The `comm_port_master` is used when a kit will be directly attached to the computer without the use of an external hub. This will enable the student to test an individual kit without being concerned about whether or not the hardware hub works correctly. This object will take care of all I/O through the serial port.



#### 4.3.2.3 Comm\_Port\_Slave

The `comm_port_slave` is very similar to the `comm_port_master`. It will be used when the computer will be attached to the network hub, instead of directly to a single kit. This will enable the computer to provide video, message logging, and artificial players to a standard game.

#### 4.3.2.4 Logger

The message logger takes all of the messages that it gets from the software hub and logs them to a user specifiable file. The data can either be saved in a tab separated raw format for analysis with other programs or it can output verbose message information. The main purpose of this object is to provide high level debugging information for the students.

#### 4.3.2.5 Video

The video unit is responsible for receiving messages from the software hub and displaying information on the computer screen. In addition to displaying the location of the paddles and the ball, the video unit tracks and displays network statistics. The last person to claim the ball has his paddle colored red. Furthermore, two statistics, network messages per second and tokens received per second, are tracked and displayed.

#### 4.3.3 GenComm

This is a communications object that is used in both `comm_port_master` and `comm_port_slave` port objects. This object was based upon a communication object previously written for an undergraduate senior design project, but was significantly rewritten to remove

bugs and make a more general interface. This object takes care of programming the UART and servicing all of the serial port interrupts. The incoming and outgoing data is stored in two separate queues that are easily accessible to the programmer through the object's functions.

#### **4.3.4 PITSpeed**

The PITSpeed object was designed as a solution to acquiring a hardware timer on a PC. It was very difficult to obtain a suitable hardware timer on the PC platform. There is only guaranteed to be one hardware PIT (Programmable Interrupt Timer) for each PC. Each PIT has three individual timers. In the PC, timer 0 is used for the system clock and causes interrupt 08h, timer 1 is used for the DRAM refresh, and timer 2 is used for generating tones for the speaker. We can not reprogram the DRAM refresh without possible severe consequences and using timer 2 would aggravate the users by having lots of aggravating tones coming out of their speakers. The solutions would have to be based upon timer 0 (i.e. the system clock). Unfortunately, the system clock is refreshed approximately 18 times a second, which is very slow for the timing requirements we need. The solution was to reprogram the PIT to a faster frequency. Normally, this would cause problems with the system clock and anything else that was using the timer interrupt. In order to correct for this an interrupt handler was placed at the top of the interrupt chain for Interrupt 08h. The interrupt handler only passed on a fraction of the interrupts that it received to correct for the appropriate speed change. For example, if we sped the PIT up eight times, the interrupt handler would only allow one out of eight interrupts to actually pass. This now provided us with an interrupt at appropriate speeds and also preserved the system clock.

#### **4.3.5 Message**

The Message object is a collection of data associated with a message, including sync character, player number, message id, and data fields. This is used more like a data structure than an object, but it was convenient that it would allocate space and initialize itself to default values when created with the 'new' function. All of the formatting modifications for transmission are done at transmission time, and therefore the programmer does not have to worry about them when using this object.

### **4.4 Documentation**

#### **4.4.1 Hardware Requirements**

It was chosen that this software should run on an IBM compatible PC. This choice was made because this platform is inexpensive and is often available in many school labs. The software should work on a 286 or better class of machine. It has, however, only been tested on a 486 class machine. Furthermore, it is necessary that the PC has a VGA display and a serial port so that it can connect to the network.

## 4.4.2 User Documentation

There are two types of commands that the NetPong program will accept: Setup Options and Run-Time Options. Setup Options are options provided to the program as it begins execution and will setup the way the program works. Setup options can either be placed in the environment variable 'NETPONG' or put on the command line when executing NETPONG. Command line arguments will override arguments that are placed in the environment variable. Run-Time options are commands that the user can execute while the program is running.

### 4.4.2.1 Setup Options

All setup commands are to be preceded with a '/' character. If a setup command is invalid, the program will fail and specify which argument is invalid. All character options can be provided in either upper or lower case. Between each command there should be one space, however, command parameters should follow the commands without any spaces.

For example: `netpong /s /p2 /lbci`

or alternately: `set NETPONG /s /p2 /lbci`  
`netpong`

will start netpong in slave mode with two artificial players. In addition, the message logger will be enabled to store all ball moves, claim ownership, and initiate player messages.

There are a number of areas that the setup options effect: communications port, message logging, message format and misc.

#### Computer Players:

'P#' This sets the number of artificial players that the computer should add to the game. The valid numbers are 1 through 4. If you have three students kits attached to the network then you should add one computer player (/P1). If you specify more artificial players than there are empty spaces then the results are undefined. The default players are one.

'D#' This sets the difficulty of the artificial players. The parameter specified is the 1/10's of a percent that the paddle will miss each ball. The valid ranges are 0-100, which means that the maximum error rate can be 10%. It is random where on the paddle the ball will hit. The only exception is a argument of 0, which will result in a deterministic game where the ball always hits the middle of the paddle. This is useful for testing. All of the computer players have the same difficulty. The default is a difficulty of 0 - deterministic play.

Example:

/D15 The computer players will have a 1.5% chance of missing the ball every time the ball hits the paddle's side of the screen.

### Communications Port:

- 'M' This sets the communication port up as a master. This is the mode that you want to use to hook a single kit directly up to the computer. Make sure that you have an RS-232 to TTL voltage converter between the two. The default comm type is slave ('S').
- 'N' This disables the communication port. This is only used for software testing. The default comm type is slave ('S').
- 'S' The sets the communication port up as a slave. This is the mode that is used when you want to attach the computer to the hardware hub. This is the default comm type.
- 'C#' This sets the comm port to use on the PC. The valid numbers are 1 through 4. Comm ports 3 and 4 should work, but were not tested, as no hardware was available to test on. The default comm port is one.
- 'B#' This sets the baud rate for the serial port. The following speeds are supported:  
110, 150, 300, 600, 1200, 2400, 4800, 9600, 19200 and 38400  
The default speed is 9600 baud.
- 'R[1,2,6,7,8,N,E,O]' This sets the various comm port settings, including data bits, stop bits, and parity. The following are the meaning of the options:
  - 1 - 1 stop bit (default)
  - 2 - 2 stop bits
  - 6 - 6 data bits
  - 7 - 7 data bits
  - 8 - 8 data bits (default)
  - N - No parity (default)
  - E - Even parity
  - O - Odd Parity

### Message Logging:

- 'F[name]' This sets the file name for the message logger. If no name is specified then the default file name will be NETPONG.LOG.
- 'L[#,A,B,C,I,P,R,U,V]' This option enables the message logger and sets the options for the message logger. This option can be enabled multiple times to turn on more options, but can not be used to disable older options. Multiple options can be included after the L option and should all be placed together without spaces.
  - # The valid numbers are 0 through 9. This option will log the message with the number specified.



- A This option logs all messages.
- B This option logs ball move messages - both BALL\_X\_MOVE and BALL\_Y\_MOVE.
- C This option logs the CLAIM\_OWNERSHIP messages.
- I This option logs the INIT\_PLAYER messages
- P This option logs paddle move messages - both PADDLE\_X\_MOVE and PADDLE\_Y\_MOVE.
- R This logs RESET messages
- U This message logs velocity update messages - both UPDATE\_X\_VELOCITY and UPDATE\_Y\_VELOCITY.
- V This enables verbose message handling. By default the message logging is done in a raw, tab separated form. If the 'V' option is specified then the output will be in a more readable form.

Examples:

- /LAV This would log all messages in a verbose format
- /L1234 This would log messages with ID#s 1,2,3 and 4 in a raw format.

Message Format:

'/I[+,-]' This argument sets or clears player number and message ID compression. If compression is set then the second byte of the message will contain both the player number and the message ID compressed (the first byte of the message is the sync character). Compression has the upper most two bits represent player number, and the lower bits being message ID. Keep in mind that actual bit locations depend upon the number of data bits that your UART is sending. If there are eight data bits then the compression will look like PPIIIII, seven data bits PPIIIII, and six data bits PPIIII. /I+ turns on compression and /I- turns it off. Compression is off by default.

'/Y#' This argument will specify the character that should be used for the sync byte. The hex number must immediately follow the 'Y'. (ex, /Y80 would set the sync byte to 80 hex, or 1000 000 binary) The default sync byte is FF hex.

Misc:

'?' This will provide a list of available commands

T# This argument sets the speed of the interrupt counter. The valid numbers that can be chosen are 1 to 9. The number corresponds to the speedup over the standard system clock. This is used to clock the counter that times the ball X and Y movement. The numbers directly translate to the following frequencies:

1	36 Hz	4	291 Hz	7	2330 Hz
2	73 Hz	5	582 Hz	8	4660 Hz
3	146 Hz	6	1165 Hz	9	9318 Hz

/T7 would result in a base frequency of 2330 Hz. The default is 6 ( 1165 Hz ).

V[+/-] This option turns the video unit on and off. The network will somewhat faster if the video unit is disabled, but this is normally not of much use if the game can not be seen. This option is mostly used for debugging purposes. The default is video unit on.

#### 4.4.2.2 Run-Time Controls

The run-time options are very limited; currently there are only three.

'r' - Reset: When the 'r' key is pressed, a RESET message is broadcast to all of the ports on the network (both software and hardware ports). This should result in the game being reset.

'q' - Quit: When the 'q' key is pressed the program will quickly exit, closing all of the appropriate log files.

p' - Pause: This is a toggle that will pause (or un-pause) the game. The computer will hold onto the token so that the students kits will not continue playing.

## 5. Lab Manual Design Issues

### 5.1 Requirements

The important criterion to address in creation of the lab manual is how to properly convey the information to the students in the most efficient and useful manor. There is an important balance between the superior retention of problems that the students figure out themselves and the extra knowledge that is possible when you directly present information. Furthermore, it is important to keep a consistent structure that the students can be familiar with as they progress through the labs.

## 5.2 Solutions

There are a number of methods that are used to properly convey information to the students which attempt to maximize the amount of information presented and retained by the student.

### 5.2.1 Prelabs

Associated with each lab is a prelab, which is intended to be provided to the students before the actual lab. In the prelabs, important design decisions are presented and possible solutions are proposed. The students should be given ample time to analyze the problems and write up their solutions. The actual labs contain solutions to the prelab questions and explain why the choice made is probably the best for our particular problem.

### 5.2.2 Structure

The lab is designed to have a consistent structure in which it is easy to identify the information contained within. At the top of each lab are two sections labeled *Object* and *Topics*. The *Object* contains a list of the piece or pieces of NetPong that will be designed in the particular lab. The *Topic* section explains the digital design concepts that will be covered. This will allow students to easily use the previous labs as reference.

### 5.2.3 Design information background

When the students are in lab, it is often useful to have some easy informational reference to be able to use. Within each lab is a short description of the relevant topics that will be important for completing the lab. This information is abbreviated and is only intended for quick reference or review. There should be a detailed lecture and/or a digital design book which should provide the students with a detailed and formal presentation of the material.

### 5.2.4 Clear Presentation of Assignments

It is important that the assignments the students are expected to complete are marked clearly. This is done in the lab by having a section titled Lab Assignment, which contained a list of the assignments that were expected to be completed. The prelabs would have been broken up too much if assignments were divided into their own sections. Instead, assignments are clearly marked with a special symbol (☞) which the students know to look for. This should prevent possible tension between course personnel and students over the clarity of what is expected.

### 5.2.5 Design before building

The students are encouraged numerous times throughout the lab sequence to design out the projects fully before building them. This is stated explicitly and also implicitly in the carefully posed prelab questions and design discussions.

### 5.3 Components

The sequence of labs contains a consistent format so that the students can easily adjust to one standardized format. Furthermore, the layout enables the labs to be used as reference for the students. The individual sections are as follows:

Prelab: Each prelab is separate from its associated lab which is intended to be handed out before the lab itself. The prelabs present one or more significant design decisions to the students. The students are then required to analyze the situations and then suggest and defend a solution to the problem. The design questions to which the students are responding to are marked with the ⇨ symbol.

Object: This is the first section of every lab. It is simply a list of the relevant piece(s) of NetPong that will be built in the lab.

Topics: This is a list of the digital design concepts which are covered in the lab.

Prelab Discussion: This section discusses the questions proposed in the prelab. If the design decision affects other kits, then a suggestion is provided that is expected to be followed. If, however, the solution only affects the student's kit, they are free to implement any solution they feel is best. Because this section provides answers to the Prelab, it is suggested that the labs be handed out after the prelabs so that the students have time to review the design problems and spend time understanding the issues.

Design Issues: This section provides information that is necessary to complete the lab at hand. Many digital design issues are discussed, but this is not intended to be a comprehensive source on all topics. It is intended to be an additional resource, for quick review or preview, that will compliment a digital design text and/or lecture.

Lab Assignment: The lab assignment section contains the actual lab assignments that the students are expected to complete. There are a few assignments that are for educational purposes only and are not strictly necessary for building NetPong. These assignments are marked accordingly.

### 5.3.1 Lab Coverage

The following is a summary of the topics that are covered in the various labs. These are listed at the top of each lab under the section titled Topics.

#### Lab 1:

Topics: Clocking, Fanout, Counters, and PALs

Object: Gain Familiarity with Equipment

Establish Appropriate Clocks

Designing Network Hub Functional Logic

Description: This lab is intended to get the students familiar with the way labs are set up and how they will progress. The goals here are fairly straight forward and should not take the students a large amount of time. Because of it's importance in making the whole system work, the first thing that the students will work on is the network hub. It is suggested that the students work in groups for this lab, as there will only need to be one network hub for every four student NetPong projects.

#### Lab 2:

Topics: FSMs

Object: Design FSM for the Network Hub

Description: This lab is essentially a continuation of Lab 1. The students will finish construction of the network hub by implementing a FSM that will control the functional hardware they created in the previous lab. Again, as in Lab 1, there only needs to be one network hub per every four kits, and therefore the students may want to be encouraged to work in groups.

#### Lab 3:

Topics: Complicated FSMs, UARTs

Object: Design of Network Unit

Description: This is where the students finally get started working on their own NetPong projects. They will be constructing the network unit which is responsible for taking bytes from the UART and packaging them into messages so that the MCU (discussed in Lab 4) can deal with messages at a higher level.

#### Lab 4:

Topics: MCUs, PROMs, Bus Contention

Object: Design of MCU

Description: This lab has the students designing their own MCU that will be used to control their entire project. As MCUs are fairly complicated and have lots of room for errors, their only goal here is to build and test the MCU.

#### Lab 5:

Topics: A/D, D/A, ALUs, SRAM

Object: Design of Paddle I/O  
Velocity Counters  
Design of Sound Unit (Optional)

Description: This lab has the students building much of the peripheral hardware that will be necessary for NetPong to function properly. Microcode will be written for the MCU to test the various components and make sure that they work properly.

#### Lab 6:

Topics: Message Queue Implementation  
Summary of Messages and Necessary Processing  
Memory Storage Suggestions

Object: Create Stack in SRAM  
Write Game Microcode

Description: This final lab is mostly microcoding of the NetPong game and pulling all of the individual components together. The only hardware modification that the students may be working on is a modification to the SRAM to allow for hardware stack pointers. The students have the option of using a software stack pointer instead.

### **5.4 Documentation**

The actual text of the labs and associated handouts is included in chapter 7. **Lab Manual and Materials.**

## 6. Lab Manuals and Materials

The lab manuals and associated material are included on the following pages. The following is a list of the various materials:

### **6.1 Overview**

### **6.2 Prelab 1 & Lab 1**

### **6.3 Prelab 2 & Lab 2**

### **6.4 Prelab 3 & Lab 3**

### **6.5 Prelab 4 & Lab 4**

### **6.6 Prelab 5 & Lab 5**

### **6.7 Prelab 6 & Lab 6**

### **6.8 Suggested Enhancements**





# NetPong Overview

---

## **Statement of Purpose:**

The purpose of this lab manual is to teach the principles of digital design. In this manual we will, through the course of six labs, develop a complicated digital system. Along the way, we will discuss such issues as digital control, storage and interaction between digital systems. The most important concept that this manual will try to convey is the approach one takes to design.

After a close look at the topic of digital design, it becomes apparent that there is very little previous knowledge necessary before delving into digital design. With this in mind, the lab was designed for audiences ranging from more advanced Juniors and Seniors in high school, to College level students. In order to accomplish this, there are a number of optional extras that can be added to the lab sequence to tailor the difficulty. Furthermore, in a more advanced class, this lab sequence might be followed by student projects, either by groups or individually, which will be created from scratch.

## **The Goal:**

When choosing the final goal of this lab sequence there were many important criteria to consider. First of all it, it had to be complicated enough to maintain the student's interest. Secondly, it had to incorporate all of the main digital systems concepts: control - Finite State Machines(FSMs), and Micro Control Units(MCUs), input/output - Digital to Analog (D/A) and Analog to Digital (A/D) converters, and basic building blocks - Programmable Array Logic (PALs), Static Random Access Memory (SRAM), Universal Asynchronous Receive Transmit (UARTs), and Programmable Read Only Memory (PROMs). Furthermore, the designers wanted to make sure that the project was interesting and that the students' hardware interacted with each other. After much thought about the issues, the designers came up with the concept of NetPong. NetPong will be a four player pong game where each student controls one side of the screen. The students' projects will be connected by a very rudimentary network which they will create.

## **What is Needed:**

In order to complete these labs there is some equipment that is necessary. First and foremost, it is necessary to have an IBM compatible computer with a serial port and a VGA display. The software does not require a fast computer, and a fast 80286 should be sufficient (although this has not been tested).

In addition to the software specifically designed for the course, there are few other general purpose software packages that, while not required, will significantly simplify some of the projects. They include:

**PALASGN:** This software converts more intuitive equation files (\*.eqn) into pal programming format (\*.pal).

**FSMC:** This program stands for Finite State Machine Compiler. This program takes a description of a Finite State Machine (\*.fsm) and produces an equation file compatible with PALASGN (\*.eqn).

**ASSEM:** This program is used to compile microcode. It takes a microcode specification file (\*.sp) and a microcode file (\*.asm) and produces output file suitable for PROM programming (\*.dat).

These three pieces of software are publicly distributed by MIT. The source code can be obtained by anonymous FTP from SUNPAL2.MIT.EDU. These software packages were designed for UNIX, but they were written in C and should be fairly easy to port to any major platform.

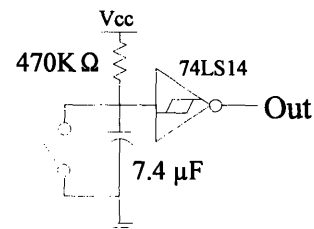
Further tools needed are a basic logic analyzer and an oscilloscope. They do not need to be high speed devices, as the suggested clock speed for the system is merely 153.6 KHz. This clock speed should allow the use of older and/or less expensive equipment. Furthermore, the course requires the use of some sort of PAL and PROM programmer.

It is assumed that the students will be building their projects on proto-boards. There are a few suggestions that will make their work easier. First, before the students start their design, they should be provided with, or told how to, design the following basic elements:

At least four (4) de-bounced switches.

At least four (4) LEDs that can be driven by TTL.

These are suggested minimums that should be followed, more are always desirable. Furthermore, it would be helpful if the students had one or two hex displays with hex decoders, for easy monitoring of data.



**Figure 1:** Suggested Switch Debouncer

## Conventions:

To facilitate ease of use, the lab manual is laid out in a consistent format. The sections and a brief description of what they contain is listed below:

**Prelab:** Each prelab is separate from its associated lab which is intended to be handed out before the lab itself. The prelabs present one or more significant design decisions to the students. The students are then required to analyze the situations and then suggest and defend a solution to the problem. The design questions to which the students are responding to are marked with the  $\Rightarrow$  symbol.

**Object:** This is the first section of every lab. It is simply a list of the relevant piece(s) of NetPong that will be built in the lab.

**Topics:** This is a list of the digital design concepts which are covered in the lab.

**Prelab Discussion:** This section discusses the questions proposed in the prelab. If the design decision affects other kits, then a suggestion is provided that is expected to be followed. If, however, the solution only affects the student's kit, they are free to implement any solution they feel is best. Because this section provides answers to the Prelab, it is suggested that the labs be handed out after the prelabs so that the students have time to review the design problems and spend time understanding the issues.

**Design Issues:** This section provides information that is necessary to complete the lab at hand. Many digital design issues are discussed, but this is not intended to be a comprehensive source on all topics. It is intended to be an additional resource, for quick review or preview, that will compliment a digital design text and/or lecture.

**Lab Assignment:** The lab assignment section contains the actual lab assignments that the students are expected to complete. There are a few assignments that are for educational purposes only and are not strictly necessary for building NetPong. These assignments are marked accordingly.

## **Overall Design**

The main emphasis of this lab manual is to teach design. Design is an iterative process. The process starts with the designer coming up with an exact description of the system to be created. All of the important criteria must be specified so that informed decisions can be made. After all of the criteria are available, the designer must brainstorm some possible overall solutions. Keep in mind that just because one solution was found which satisfies the problem, this doesn't mean that there aren't additional solutions that are possibly simpler and cheaper to implement. These solutions don't have to incorporate every detail, they just have to define an overall approach to the problem. Solutions should include a high degree of modularity. This means designing sub-modules that are clearly defined and appear to be practical. The designer should then make a list of the pros and cons of each design. The list should include complexity, ease of debugging (testability), costs, expendability, and any other criteria that is important to the designer. After the overall approach is decided, the designer should look at each of the sub-modules and again use the same design principles. After investigating the sub-modules, you may feel that your overall approach may not have been the best one. Feel free to revise your previous decisions. You should realize that for every hour you spend designing you will likely save tenfold working on implementing and debugging your design. As the reader progresses through this lab manual, each stage of design will be demonstrated.

## **What you will be building**

In order to design NetPong, we must fully understand what we want it to do. Most people should be familiar with the game of pong - the first video game. The object of the original game is to keep the ball on the playing field by hitting the ball with your paddle. You were only responsible for the bottom edge of the screen. In NetPong we will have a four player version of the game. Each player will control one edge of the screen. Each student or group of students will be responsible for designing their own player. In addition, a computer will be used to generate extra players (if there are less than 4 human players) and to generate the video portion of the game.

For the high level design of NetPong, we are going to use a simple single data bus architecture. This essentially means that there will be a single data path on which information can travel which will connect a number of sub-modules. This architecture is chosen because it is the simplest to understand and implement. In high performance systems, a single bus system can cause problems due to the bottleneck that this design imposes. This shouldn't be a problem with this design because performance will most likely be limited by the speed of the network. As with any other design issue, if later on we notice that performance is an issue, we can revise the architecture.

In addition to an overall architecture, we need to come up with a list of the sub-modules. First and foremost, we need some idea of the functional elements that are required by our project. We will need some way to talk to the other kits to satisfy the "Net" in NetPong. We will also require some sort of storage unit to keep track of all of the game information. A calculation unit and a timing unit will be necessary to keep track of movement and a user I/O unit to provide input. Furthermore, we will need something to control the entire system and a system clock to keep everything synchronized.

Distilling those ideas we come up with a requirement for the following units:

- (1) Network Unit
- (2) Storage Unit
- (3) Arithmetic Logic Unit (ALU)
- (4) Timing Unit
- (5) User I/O
- (6) Control Unit
- (7) System Clock

Each student (or group of students) will be building a game unit. These game units will be called 'kits' or 'players' throughout the lab. Each player will be responsible for receiving messages from the network that will be designed. The players will react to the messages by performing certain actions and/or sending other messages of their own. Messages that are to be sent out are placed in a queue until the kit has permission to use the network. At this point, the queue will be emptied and all of the messages will be sent out. Using these basic elements we will design a multi-player game of pong.

# NetPong Prelab #1

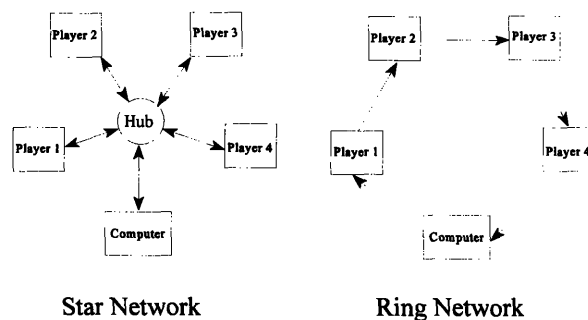
## Network Topology:

Lets take a look at the Network Unit. This unit will be responsible for sending and receiving messages to the network that we are going to build. The first thing that we want to ask ourselves is, "What network?" We, of course, have to design that too. Often, when trying to solve a problem we learn that another problem will have to be solved first.

### Requirements for the network:

- Must allow 4 kits and 1 computer to communicate
- Must be easy to build
- Must be reliable
- Must be resilient to errors

Due to their popularity and ease of use, we are planning to use UARTs to form the network. A UART is a device which takes a byte and transmits it serially over a wire. It also receives serial data and changes it back to a byte. UARTs are used in computer serial ports. This is all that you need to know about UARTs at this time; more information will be necessary when you actually need to use them.



With a little thought, two basic network topologies emerge: a star topology and a ring topology. The star topology has the computer and the four kits plugged into a "hub". The responsibility of the hub is to give each port (the four players and the computer) a turn to talk. Each port then sends its messages to the hub, which redistributes it to all of the other ports. The ring topology, however, has each kit (player) and the computer plugged into both of their neighbors (see diagram). Each kit takes incoming messages, processes them and passes them on. Unlike the star topology, the messages will continually go around the ring until removed. Therefore it is the sending kit's responsibility to removing the messages.

A decision of this nature is crucial to our basic design and will have a very large impact on many decisions in the future. For this reason we should spend some time trying to weigh the advantages and disadvantages of these two designs.

☞ Your task in the prelab is to make a list of advantages and disadvantages of the two designs. Try to keep in mind all of the issues, including speed, ease of implementation (for both kits and computers), reliability, complexity, and any other issues you feel are important. After

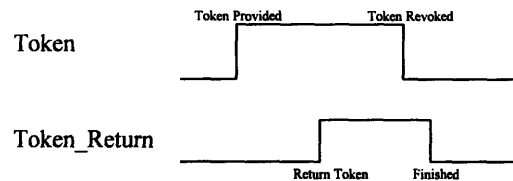
making a list of the advantages and disadvantages, review the important design requirements of the project and make a choice as to which design would be best for our use. Insure that you defend your position.

Token Passing:

Let us form an analogy for our network: Consider all of the kits attached to our network as people in a room, and the hub is a policeman in the same room. The policeman (like our hub) is uninterested in speaking, but is interested in keeping order. Everyone has something that they want to say, but if they all talk at once no one can hear what anyone is saying. What we need to do is to create some type of system which will tell each person (i.e. kit) that it is their turn to talk. Now we are going to introduce the concept of a token, by giving the policeman a microphone. The policeman gives the microphone to one of the speakers, who is the only person in the room who is allowed to speak. When that person is finished, they give the microphone back to the policeman. The policeman then gives the microphone to the next person, etc. Our network is like the room full of people because, if everyone talks at once, no one can hear what anyone is saying. We need to have a 'token' which, like the microphone, can be given to each kit when it is permitted to transmit data. When it is done transmitting, it returns the token to the hub. This analogy is describing a star based network, but all of the concepts are the same with the ring, except the people (and kits) pass the microphone (token) to the next speaker instead of to the policeman.

Another important design decision that has to be made about the network is whether we should be passing the token using a *standard message* or as a *digital signal handshaking*. If we were to pass the token as a standard message, we would do it by creating a message with an ID that said "pass the token". In addition, depending upon the topology chosen, it may or may not be important to say to whom you are passing the token. This, of course, depends upon whether you are sending messages to a single player (as in the ring topology), or broadcasting them to everyone (as in the star topology).

The digital signal handshaking is a method where a digital logic signal is raised when it is time to pass the token to the next player. In order to do this properly, two signal lines would be used to properly handshake between the two digital systems to insure that the signal is passed correctly.



**Figure 1: Digital Signal Handshaking**

⇒ Spend some time thinking about these two token passing methods. There are a number of important issues, such as ease of implementation, speed, and startup issues. If we are going to use a ring topology, which token passing system should we use? How about for the star topology?

# NetPong Lab #1

---

## Object:

Gain Familiarity with equipment  
Establish Appropriate Clocks  
Design Network Hub Functional Logic

## Topics:

Clocking  
Fanout  
Counters  
PALs

## Prelab Discussion:

### Network Topology:

In the prelab we introduced two potential methods for structuring our network. Some of the largest advantages and disadvantages of the two networks are listed below:

#### **Star Network**

##### *Advantages:*

Faster (at most 2 links per message, although it can be brought down to 1 link)  
Easier to hook up computer (only requires one serial port)  
Central Arbitrator eases debugging.

##### *Disadvantages:*

Requires additional hardware (the hub)  
Must change the hub to add more nodes

#### **Ring Network**

##### *Advantages:*

More scalable (any number of nodes can be added without changing hardware)

##### *Disadvantages:*

Slower (requires  $n$  links, where  $n$  is the number of nodes)  
More difficult to hook up computer (uses 2 ports)  
Kit Network Units must be more complicated (know when to take message off of the ring)

Looking at the list of advantages and disadvantages seems to show that for our requirements (listed in the Prelab) the star network is preferable. It will be easier to hook up to a computer and, because of the central arbitrator, it will be easier to debug. Finally it is considerably (five times) faster. It is not obvious that this is important at this stage of designing, but in general faster is always better for networks.

### Token Passing:

Let us take a look at some of the important advantages and disadvantages of the two systems:

## Message Passing

### *Advantages:*

Easier to implement on a PC

### *Disadvantages:*

Slower

Difficult to start token

More decoding logic

## Digital Signal Passing

### *Advantages:*

Faster

Easier to implement

Easier Hub design

### *Disadvantages:*

Harder to design software

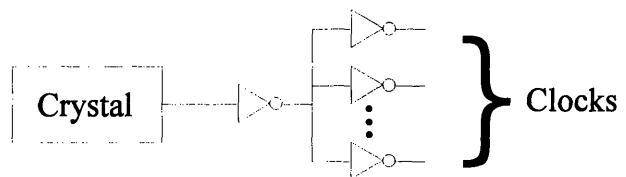
Looking at these two lists it appears that digital signal passing is probably the way to go. This holds true for both of the network topologies (Star and Ring). If we had to pass the token via message passing, it would certainly be easier on a Ring network, because each kit knows how to contact its neighbors, unlike the Star topology. As we have previously chosen to use the Star topology, there is no question that we should use digital signal handshaking to pass the token.

## Design Issues

### Fanout

Although we like to consider the logic gates purely as a digital abstraction, they are real electrical components which have some non-ideal properties, one of which is fanout. Each gate can supply or sink a limited amount of current. In addition, each input requires a little current to make them work properly. What this means in practice is that there is a limit to the number of gates that you can attach to an output. Fanout is defined as the number of gate inputs that can be driven by a single gate output. A typical TTL 74LS gate has a fanout of 20. If you attach too many gates to an output the quality of the signal will degrade and you will often get ringing in the signal, which can cause unpredictable results.

A crystal oscillator has a fanout of 1, so do NOT directly use the output of the crystal oscillator as your system clock. Instead, you should attach the crystal oscillator to an inverter and use the inverter output as the clock. For large circuits, even the output of one inverter may not be enough, so you can take the output of the first inverter and use it to drive a number of other inverters. The clock should be taken from the second row of inverters (see diagram). Do NOT attach the outputs of the second row of inverters together, instead have some devices attached to each inverter.



**Figure 1:** Suggested Clock Circuit for large systems



## Logical '1' and '0':

Often you will find a need to hardwire gate input to logical '1's and '0's. This can be done by connecting the input to ground and Vcc. You should, however, not directly connect the input of the gate to ground or Vcc; instead connect them with a small resistor. The specific value does not matter very much, though a good suggestion is 2.2K $\Omega$ . The resistor is especially necessary when using 7400 or 74S series components. You can directly connect 74LS series components without a resistor.

## Hub Requirements:

Now that we have settled on an architecture, we have to design the hardware to accommodate it. We are going to start with the design of the hub. Let us first make a list of the requirements for the hub:

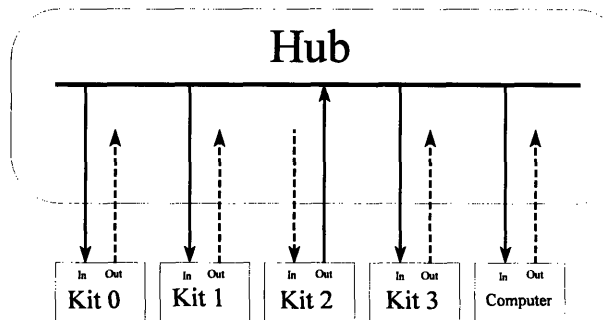
- Must give each port a chance to talk to all of the other players. In other words, when a port is selected, its output should be connected to all of the other port's inputs. In addition, it would make sense that the messages are not echoed back to the port that is transmitting them.

- To ease debugging, the hub should be self contained.

- The hub will have five ports, one of which will be special so that it can accommodate the line voltages that come out of a PC serial port. (-8 for a '1' and +8 for a '0', as apposed to standard TTL levels). These voltage levels provide added reliability when transmitting data over long lines. An RS-232 Line Driver and Receiver will need to be employed to convert back and forth between standard TTL levels and RS-232 transmit levels.

- The hub is a reasonably simple device, but it is certainly more complicated than a simple combinational logic circuit. It has to remember which port is currently active for talking and also which port should talk next. We can therefore divide this project into two parts: the functional hardware and the control hardware. You can think of the functional hardware as a marionette. It has a number of control lines that you can use much like strings of a marionette. The control unit can be thought of as the person controlling the marionette (e.g. the one pulling the strings). For this lab we are just going to build the functional hardware. We will put off building the control unit until Lab 2 where we will discuss controllers.

Let us take a look at how the hub is going to interact with your kit. There will be five lines that will be going between your kit and the hub. They are Serial\_Out, Serial\_In, Token\_Available, Token\_Return and, of course, ground (a ground is very important and something that is often forgotten!). They are further described as follows:

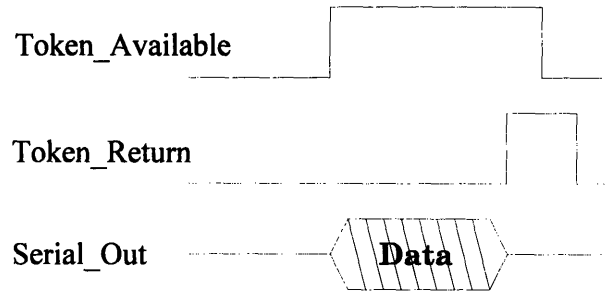


**Figure 2:** Example of Kit 2 being selected

**Serial\_Out (Output from kit):** This is the serial output of the UART from the kits. When this port is selected it will be the digital line that is broadcast to all of the other Serial\_Ins.

**Serial\_In (Input to kit):** This is the serial input to the UART from the other kits. When your port is **not** selected, information from the other ports will come in on this line.

**Token\_Available (Input to kit):** This line will tell your kit that it has the right to send data over its Serial\_Out line. All data sent on Serial\_Out will be echoed to all of the other ports. The Token\_Available signal will stay high until your kit sends a Token\_Return.



**Figure 3: Kit-Hub interaction**

**Token\_Return (Output from kit):** This is a signal from your kit to the hub saying that you have completed transmission. When it is raised, the hub will respond by lowering the Token\_Available line going to your port and raising the Token\_Available line on the next available port. After Token\_Available gets de-asserted, you should de-assert Token\_Return immediately.

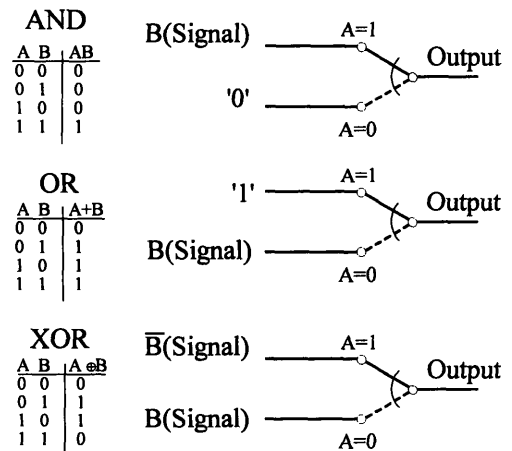
The important two lines that we want to look at now are the Serial\_Out, and Serial\_In lines. The functional hardware that you will be creating in this lab will correctly connect the selected port's Serial\_Out with all of the other port's Serial\_In. The important question to ask now is, "how do I know which port is selected?" Obviously, you need some more information.

You can assume that there will be five Token\_Available lines (one for each port), only one of which will be raised high at a time. When a port's line is raised high, it will be in control, meaning that the Serial\_Out of the port will be connected to all the other Serial\_Ins, except its own. This control is none other than the Token\_Available signal that is presented to the port. These lines can be thought of as the marionette's "strings" from the earlier analogy.

In order to design large digital systems, you will have to get a feel for what a logic gate does, above and beyond simply memorizing the truth tables. The three most important examples are as follows:

**AND gate:** Consider this logic gate as a device that can switch a stream of data on and off, where the output of the gate has a default of logic '0'. For instance, consider the B input the signal, and the A the control. If the control is a '1' then the signal passes through, unhindered. However, if the control is a '0' then the output of the gate is forced to a '0'

**OR gate:** This gate can also be considered a switch. The difference is that the on and off for the



**Figure 4: Logic Gate Representations**

control is reversed and the default output is a '1'. Again, we will consider B the input signal, and A the control. If the control is a '0', then the signal passes through unhindered. If the control is a '1' then the output is forced to a '1'.

**XOR gate:** This gate can be considered an optional inverter. Once again, we will consider the B input the signal, and the A input the control. If the control is '0' then the signal B is passed through unhindered. If the control is '1' then the input signal B is inverted at the output.

Although you probably do not always want to think of these gates in this manner, it is often helpful when trying to design or understand complicated digital circuits.

Let's look at the problem at hand using the insight about digital gates we have just discussed. For the functional hardware, the three signals that we would like to think about are Serial\_Out, Serial\_In, and Token\_Available. First of all, let us think about the Serial\_Out. This line has the data which is being transmitted from the kit's UART. In normal operation this port should not be sending data if Token\_Available for this port is not asserted, and therefore it should default to a '0', the waiting state. We would like our system to be reasonably fault tolerant so, just in case one of the kits is not working properly and sends data out of turn, we do not bring down the whole network. What this means is that we want to physically cut off the path to the other units. We will do this using an AND gate as described above. The Serial\_Out goes into one part of the AND gate, and the Token\_Available goes into the other. That way, if Token\_Available is asserted, then the Serial\_Out flows through the gate, otherwise the output is a '0'. The output of this AND gate from all of the ports should then be ORed together so that the resulting output will contain what is coming from the selected (via the correct Token\_Available signal) Serial\_Out.

If this resultant signal was directly connected to all of the Serial\_In's, then the basic layout would be almost complete. The only problem would be that we would be feeding the data back to the port which originally sent it. Since we would like to prevent this from happening, let us consider adding one more gate to each port to prevent this feedback of data. Right before the Serial\_In of each port leaves the hub, we will insert an additional AND gate. This gate will prevent data from going through if the Token\_Available signal is selected for that port. We put the output of the central OR gate into one of the inputs of the AND gate. We will then invert the Token\_Available signal and put it into the other input of the AND gate (i.e. we want the data to **not** pass when the port has it's Token\_Available signal selected). We now have the entire functional unit of the hub designed. We will need a controller to make it work, but this will be a project for the next lab. As part of the exercises, you will have to build this hardware and demonstrate that it works.

### RS-232 Line Driver/Receiver

As previously mentioned, the output voltages that come out of a standard RS-232 serial port are **not** TTL compatible. Standard RS-232 ports utilize line drivers, which increases the voltage difference between a logical '1' and '0', so that they can get lower error rates when transmitting over long cables. A logic '1' is mapped to -8 volts and a logic '0' is mapped to +8

volts, which greatly increase the noise margin over standard TTL. The line receiver takes these voltages and returns them to standard TTL levels. **It is important that you do not attach the output of a standard RS-232 output port to a TTL input. The high voltages will very likely destroy the TTL chip!**

The line drivers and receivers are usually contained in a single chip. Due to the higher line voltages, the chip will often require access to positive and negative 10 or 12 volts. Some have incorporated charge pumps into the chip, so you only need to supply the standard +5 and ground. These chips will need external capacitors for use with the charge pumps.

The following is a table of the mapping of NetPong signals to standard RS-232 signals. A 9-pin or 25-pin cable will have to be made that corresponds to these specifications:

<b>NetPong Signal</b>	<b>RS-232 Signal</b>	<b>pin #(9-pin)</b>	<b>pin #(25-pin)</b>
Sout	TD	3	2
Sin	RD	2	3
Token	CTS	8	5
Token_Return	RTS	7	4

## Lab Assignment:

### 1. Ring Oscillator:

Connect five inverting buffers together so that the output of one goes into the input of the others. Take your oscilloscope and examine one of the outputs of the inverters. What is the frequency of the oscillation? Using this frequency, calculate the propagation delay of one of the inverters. Measure the rise and fall time of the square wave.

Next, insert a 0.1  $\mu\text{f}$  capacitor at one of the junctions of the inverters (simulating a fanout of about 20). The capacitor represents the connection of a number of gates to the output and shows the effects of fanout. Notice how the square wave degrades at this location. Again, measure the rise and fall time at the junction with the capacitor. What happened to the oscillation frequency? Why?

*Note: This assignment is for training and is not used for NetPong. It can be disassembled after Lab 1 is completed.*

### 2. Clocks:

In this part of the lab you will be designing the clocks that will be used in other parts of the overall project. Using a 10 MHz crystal oscillator, divide it by 64 using ripple counters to generate a 156.3KHz clock. Built two of these clocks, each on separate boards. You will use one of the clocks for the hub. The other will be the main system clock for your kit. Make sure that these clocks are designed to be glitch free, as this will cause problems for your system.

### 3. Hub Functionality:

We have discussed how the hub's functional logic should work earlier in the chapter. As the final part of your lab, build the required hardware for a five port hub out of logic gates and/or PALs. Be sure that you test each port to insure that it correctly sends and receives data. This should be built on a separate board from the one that is going to be used to build your NetPong kit. Also, make sure that one port is attached to a RS-232 line driver/receiver, so that it can be attached to a computer serial port. You should not only attach serial in and serial out, but also the Token\_Available (to CTS) and Return\_Token (from RTS). Demonstrate this functionality to your Teaching Assistant(TA) or professor.



# NetPong Prelab #2

---

## Ball Ownership Issues

Now that we have defined a network topology and token passing method, we have to start looking more carefully at how the game itself should be conducted. The basic idea is that a ball will bounce around the screen. If the ball tries to fall off your side of the screen, then you will try to stop it by blocking it with your paddle. If you manage to block it with your paddle, its velocity changes and it continues in another direction. However, if you miss, the ball is put back into the center of the screen and play continues.

There are a number of ways that we could design this digital system. One method, probably the most obvious one, is to have all of the kits tracking the ball. In this situation the only information that would have to be passed from kit to kit would be the location of all of the paddles. This seems straight forward, as each kit would move the ball at their own rate and check to see if it collided with its paddle, or any of the other kits paddles, and then take appropriate action. We will call this method group ownership.

Another option is to have one kit designated as the ball controller. This kit moves the ball and tells all of the other kits the current location of the ball. When any kit notices that the ball has reached its edge of the screen, it will report to the ball controller whether the kit's paddle was hit or missed. The ball controller will take this information and update the ball's velocity and or position. We will call this sole ownership.

One step further is to have a distributed control of the ball. In this scenario, a player would wait for the ball to reach its side of the playing field. When the ball reaches its side, the kit will first claim ownership of the ball and will then be responsible for updating the velocity and position of the ball until another kit claims ownership. We will call this distributed ownership.

⇒ There are some important issues to think about with regard to these systems. First, what happens if one or more kits get out of sync (i.e. disagree on ball or paddle location)? When designing digital systems that communicate with each other, it is often a very important that they be able to recover from communication errors and synchronization problems. How well do the three systems described recover from these type of errors? Are there any other important differences which encourage us to pick one of these systems over the other?

## Messages

At this point we should introduce the concept of messages to our network. The network that we have designed with UARTs has set up a good framework which allows us to send bytes from one kit to the others. We now need a higher level of communication so that the kits can

talk to each other intelligently. Messages, in the most basic sense, are a collection of bytes. They have prearranged fields which contain specific information. Each message must contain enough information to convey its meaning, but also be as small as possible to cut down on transmission time. What data fields are necessary in our message? Obviously we need a message ID field. We also should have a player field, which will tell who sent the message. Furthermore, we need at least one data field. Finally we should have a sync byte to recognize the beginning of the message. If we did not have a sync field and we lost one or more bytes, there would be no easy way to tell that our messages were incorrect and no way to recover from this problem.

⇒ What are the requirements of the sync byte? Are 00h or FFh a good choice? Why or why not?



# NetPong Lab #2

---

## **Object:**

Design FSM for hub

## **Topics:**

FSMs

## **Prelab Discussion:**

### Ball Ownership Issues:

Let us look at the recoverability of the systems, one at a time.

#### *Group Ownership:*

This system has some serious flaws with respect to recovery from lost messages and synchronization errors. The major reason for this is that the data that is going between the systems is so limited. Once a system goes out of sync (i.e. disagree where the ball is) there is no way to recover short of resetting the whole system. For this reason we will not choose this system.

#### *Sole ownership:*

This system has an advantage in that one kit, which we will call the owner, has sole responsibility for the ball and all of the other kits are just slaves. In this case there is some important information that is being shared - the location of the ball. If one of the kits loses a message about a ball move it will most likely not be that important and will be recovered once the next ball move has been broadcast by the owner.

#### *Distributed Ownership:*

This system has much the same advantages of the Sole Ownership system in respect to recovery. In addition it has two very nice features for us. The first is that the work is distributed equally among all of the kits. This is nice because all of the kits are interchangeable, unlike the sole ownership where one kit is different. In addition this system allows us to do something very interesting - all of the kits do not have to be running exactly the same rules and can still work together. Some kits could implement features such as 'ball hold' where the ball is held to the paddle as long as a button is pressed. As long as all of the kits play by the basic rules agreed upon (i.e. broadcast paddle locations and, when you own it, the ball location), what they do with the ball while they have ownership doesn't really matter. For these reasons, this is the system that we will use for our implementation of NetPong.

### Messages:

The sync byte, as was stated, is the flag that we use to recognize the beginning of a message. As such, it is important that the sync byte be a unique byte that is not used anywhere

else in any message. 00h and FFh are probably not the best choices for sync bytes. These two byte are too easy to get in hardware or software if you make a mistake. Quite often the default state of a malfunctioning device is a 00h or FFh. Therefore, in order to make debugging easier we should choose something else. We will end up using 80h (1000 0000b) because this byte will not be used in any of our data. We will verify this later when we fully define the data fields. 80h is especially nice when using data with sign/magnitude data (first bit sign, seven bits data). This is because there is very rarely a use for negative zero.

## Design Issues:

Now that we have investigated the functional logic for the hub, we need to have some sort of controller to "pull the strings". The simplest form of controller is the finite state machine (FSM). This device, as the name says, has a finite number of states and some rules as to how to traverse from one to another. In each state there is an output associated with all of the digital lines. The important new concept here is keeping track of which state you are in; as a result of this requirement, we need some sort of memory. The memory that we will use are called flip-flops.

Flip-Flops are simply a one-bit storage device. Flip-Flops are often made from combinational logic, but they rely upon the time delays inherent in gates to store their memory. We will not go into exactly how a flip-flop internally works; instead we will concentrate upon functionality.

There are three common types of flip-flops used today: D, T, and J-K. We will start by investigating the D, as it is the simplest conceptually. In its simplest form, the D flip-flop has two inputs (D and CLK) and one output(Q). Inside the D-flip flop is a current state, which is reflected at the output, Q. This state, and hence Q, will remain unchanged until the CLK input rises from a zero to a one. (This is assuming a rising edge trigger. Some devices are falling edge triggered; see the specification sheet for more information your the particular device.) At the point that the CLK rises, the value at D is the new state, and Q changes to reflect that. The reader should remember that there are also falling edge triggers, where the new value is latched in when the clock goes from a one to a zero. There are some common variations to this theme. Two other common inputs are Clear and Set. Both of these come in two varieties: synchronous and asynchronous. Lets start by looking at the Clear input. If the Clear is **synchronous**, then, regardless of the D input, it will set the state of the flip flop to a '0' when the clock rises (assuming rising edge). An **asynchronous** clear will set the state of the flip flop to a '0' immediately, regardless of the CLK input. The Set input is identical to the Clear, except it sets the input to a '1' instead of '0'. Different flip flops may or may not have Set and Clear, and they may be

D	CLK	Q <sub>old</sub>	Q <sub>new</sub>
X	0	Q <sub>old</sub>	Q <sub>old</sub>
X	1	Q <sub>old</sub>	Q <sub>old</sub>
0	Rise	X	0
1	Rise	X	1

D flip-flop truth table

synchronous or asynchronous. Make sure you read the specification sheet to determine this information. Furthermore, this flip flop usually has an additional output  $/Q$ , simply the inverse of the internal state.

The second common type of flip flop is the Toggle or T-flip flop, which is similar to the D-flip flop. The functional difference is that instead of setting the internal state to the T input when the CLK rises, it inverts the internal state if the T is '1' and leaves it the same if the input is a '0'. T flip-flops can also contain Clear and Set inputs.

T	CLK	$Q_{old}$	$Q_{new}$
$\bar{X}$	0	$Q_{old}$	$Q_{old}$
X	1	$Q_{old}$	$Q_{old}$
0	Rise	X	$Q_{old}$
1	Rise	X	$/Q_{old}$

T flip-flop truth table

The third type of common flip-flop is the J-K flip-flop. This flip flop is slightly more complicated than the previous two. It has three inputs J, K, and CLK, and one output Q. The main advantage of using J-K flip-flops is that they support multiple don't care states which may simplify the combinational logic needed.

In general, for FSM designs, we end up using D flip-flops because they are the most common type (possibly the only type) that come packaged in PALs. PALs are ideal for creating FSMs because there are a number of tools which allow us to take an abstract description of an FSM and compile it directly into a PAL descriptor file.

There are two types of FSM: Moore and Mealey. The difference between the two is that with Moore we are restricted to having the outputs only as functions of state. Mealey is less restrictive and can have the outputs be a function of both inputs and state. In general we will be using Mealey, due to its less restrictive qualities. When designing an FSM we have to remember that we are dealing with a finite predefined number of states. Each state has a defined output (possibly a function of the inputs in the case of Mealey). Any time that you want to change this mapping, you have to create a new state. You may also want to make a new state to keep track of changes in the system or to keep track of time. Usually the states should be reasonably easy to identify with a little practice.

J	K	CLK	$Q_{old}$	$Q_{new}$
X	X	0	$Q_{old}$	$Q_{old}$
X	X	1	$Q_{old}$	$Q_{old}$
0	0	Rise	X	$Q_{old}$
0	1	Rise	X	0
1	0	Rise	X	1
1	1	Rise	X	$/Q_{old}$
X	X	Fall	$Q_{old}$	$Q_{old}$

J-K flip-flop truth table

Each state should have a defined output (which may be a function of the input). In addition, each state must fully describe when and how they will transition to other states. Keep in mind that FSMs are clocked devices (as they use flip-flops) and therefore transitions may occur every clock cycle. Furthermore, more than one input signal may change each clock cycle. When you are defining transition conditions, remember that you must look at all combinations of the signals that you choose to use. If you do not specify one or more of the signal combinations, the FSM may end up going into an incorrect state.

There are a number of other problems that can arise when using FSMs. One of these is the presence of "dead-end" states. These states are ones which you did not expect to be there. When your FSM gets first powered on it will end up in a random state. This state may or may not be one that you defined. If you have  $n$  flip-flops in your FSM, then you can represent  $2^n$  states. Usually, the number of states that you have in your FSM does not turn out to be exactly a power of two. What happens to these other states? They get assigned some random output conditions and random transition equations, because in your logic equations they were "don't care" states. It is possible that these random transitions loop back on themselves and you get stuck in these "dead-end" states. There are two solutions to this problem: The first is to define all of the remaining states, up to  $2^n$  to go directly to a known state (preferably a starting state). The other option is to have some way of resetting the FSM to an initial state, say through the use of a reset signal that clears all of the flip-flops.

Another common problem that occurs when using FSMs is related to the non-ideal properties of the flip-flops. The flip-flops need the input signals (D, T, J, K) to be stable for a certain amount of time before the clock rises (set-up time) and a certain amount of time after the clock rises (hold time). If these requirements are not obeyed, it is possible that the flip-flop will not go to the right state. The set-up and hold time is reasonably small, and therefore, for most cases, it is unlikely that a signal will transition during these times. Also, for many applications, it doesn't matter what state the flip-flop ends up when a transition occurs very close to the clock edge. However, it is important that setup and hold times are not violated in your FSM controller flip-flops. This violation can cause the system to take a state transition that you did not intend, or even one that is not possible given your FSM description. If you see a state transition that does not seem possible given your design, then this is a likely place to look. The solution to this problem is reasonably straight forward. All asynchronous signals (ones not dependent upon the same clock as your FSM) that go into the FSM must be latched through flip-flops, usually 'D' types, that are using the same clock as the FSM. This will move the setup and hold time problems away from the flip-flops that hold state information for your FSM to another flip-flop where setup and hold time problems will not be important. Keep this in mind, as many students ignore the warning and get stuck trying to fix a difficult illogical bug that only happens sporadically (the worst type!).

## **Lab Assignment:**

### 1. Resource Sharing:

In order to properly share a resource between two or more independent controllers, there has to be some sort of method for controlling who has access to the resource. For this exercise, we will assume that there are two controllers which both have access to a resource. We want to have a system where the controllers request permission to use the resource from the resource manager. They will do this by asserting their Request line. The controller will then look for the granted line to be raised (keeping the Request line asserted). At this point the controller is free to use the resource (again, keeping Request asserted). When the controller is finished with the resource, it finishes by de-asserting the Request line. For the first part of this lab you are to

design and build the FSM for the resource manager to mediate a resource between two other controllers. Make sure that you latch the inputs to the resource manager. (This is not necessary if the two other controllers are using the same clock as the resource manager).

Hint: This FSM can be created using only two states.

*Note: This assignment is for training and is not used for NetPong. It can be disassembled after Lab 2 is completed.*

## 2. Hub Controller:

At this point let us look at the controller necessary for the hub. We want the hub FSM to control switching between the five different ports. Each time the port which has the token raises it's Token\_Return signal we want to switch to the next available port. The easiest way to approach this FSM is to make one state for each port (5). In each state the output will be the Token\_Available signal for the associated port. Furthermore, when the associated Token\_Return signal is raised, the state will change to the next port (See state diagram). Implement and test this FSM. It is always a good idea to figure out a reasonable clock speed for the FSM that you are using. In this case, we are switching serial communication links, which are clocked at 9600 Hz. A good number might be to run the hub at 16 times this speed, which should make the switching time reasonably transparent.

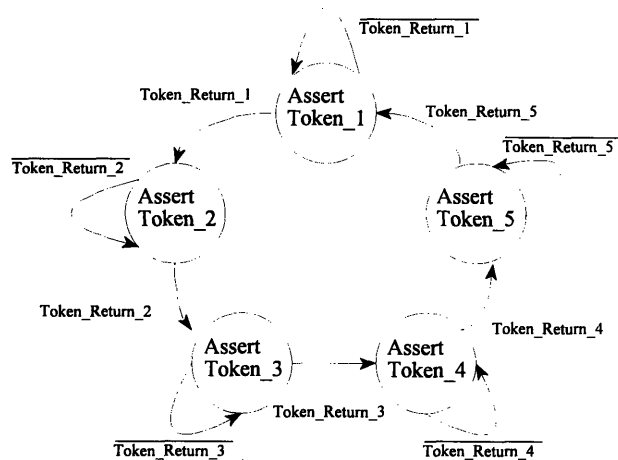


Figure 1: Hub FSM State Diagram

After the hub controller is tested you will want to hook it up to the hub's functional logic that you built in Lab 1. Make sure that you latch all of the inputs to the system (all of the Token\_Return signals). After this is completed you should have a fully functional Hub. You can test some of the functionality of the hub by "switching off" all of the ports except the computer port and running the computer software with the external hub option on. Keep in mind, you must always "switch off" ports that are not being used. This is easily done by attaching the appropriate Token\_Return signal to Vcc (logical '1'). The computer should have four auto-players going. Furthermore, if you attach another serial port to the hub you should be able to see the messages that the computer players are generating to talk to each other.



# NetPong Prelab #3

---

## Message Types:

In the previous two labs, we have established a basic format for our messages. It is time that we start dealing with the specifics of what the message types will be. We have established that we are going to be using a system with distributed control of the ball. This fact will have a significant influence upon the messages that are going to be passed. The first question that we should ask is what type of messages are we going to need. Certainly we need to specify that the ball has moved and also to tell when a paddle has moved. In addition we need to specify when a ball velocity has changed. We also need a message to claim ownership and a message to reset the game.

We have decided that each message will have four parts: sync byte, player number, message id, and data field(s). For our implementation we are going to use fixed length messages. This will make decoding the messages a little bit easier in hardware. This means that we must decide how many data fields there will be. Let us look at the messages necessary:

**Paddle Move:** The paddle has an X and a Y coordinate. However, under normal operation (after startup) the top and bottom players will only modify the X coordinate and the left and right players will only modify the Y coordinate. The X and Y coordinates will range from 0 to 64, which is the size of our playing field.

**Ball Move:** The ball will need both an X and a Y coordinate. Both will need to be updated regularly.

**Ball Velocity:** The ball velocity will also have an X and a Y attribute that will need to be updated. Velocity will normally only be updated when a ball reaches the edge of the screen and therefore will be much less common than the previous two types of messages. The velocity attribute will be a count of a number of clock cycles (the frequency of which will be defined later). Keep in mind, this means a small number here is faster than a large number. Furthermore, the most significant bit is reserved for direction (0=increasing, 1=decreasing).

**Claim Ownership:** This message has no necessary data. Again, this message will only be broadcast rarely compared to Paddle Move and Ball Move.

**Reset:** This message has no necessary data. This message is only for resetting the game and will therefore will be the least frequent message.

☞ From this quick analysis it seems that our messages should have two data fields to satisfy both X and Y attributes. This is a straight forward approach and will satisfy our requirements. Of course there are always other options. Take, for example, the option of having one data field. In this case we would need a separate message ID for X and Y attributes when we wanted to do a Paddle Move, Ball Move, or Velocity change. What are the advantages and

disadvantages of this proposal? In general would this result in slower or faster operation of the system?

### How Many Messages?

An important question to ask is how fast our network is going to transmit data. As we will see, the network will be the bottleneck in data transfer, and we want to make sure that it will provide enough bandwidth to satisfy our game. In order to do some rough estimations we have to know a little more about the output of the UART. The speed of the UART is measured in bits per second (baud). This is only half the story, however. For each byte of data that you send out a certain number of serial bits are sent from one UART to the other(s). The format of the UART is as follows: one start bit, logic '0'. Five to eight data bits, from lowest to highest. If odd or even parity is selected then one bit of parity follows. Finally, the transition ends with one or two stop bits, logic '1'. Our transmission needs 8 data bits, and in order to keep transmission speeds high we will work with no parity and one stop bit.

⇒ Use the information provided to calculate the maximum number of messages per second we can transmit, ignoring any processing time for the kits. Do this calculation for the single data field (total of 4 bytes per message) and the double data byte (total of 5 bytes per message). Let's assume very roughly that each time the ball moves there will be approximately five messages (one for a ball move and four paddles). Does this seem like a reasonably speed? What can we do to increase this speed? Provide some speed estimates for these improvements.



# NetPong Lab #3

---

## Object:

Design of the Network Unit

## Topics:

Complicated FSMs

UARTs

## Prelab Discussion:

### Message Types

In trying to decide whether our messages should have one or two data fields, there are two important things to consider: ease of implementation and speed. Single data field messages are 20% smaller but two such messages will be needed if we want to update both X and Y components. If, in truth, we are normally only updating one or the other, this type of message will save us a little time. When a kit moves a paddle (after setup) it will only be moving it in one direction (Y or X depending upon what side of the screen they are on). Furthermore, the ball usually has different X and Y velocities and therefore it usually moves in either the X or the Y direction and not both at the same time. When updating the velocity, both often want to be updated at the same time, but velocity updates occur rarely when compared to ball and paddle moves. Using these assumptions it seems that the one data field messages will be somewhat faster than the double data field messages.

In addition to the speed issue, there is another advantage to the single data field messages: they provide us with information about which attribute has changed (X or Y). If it was necessary, we could obtain this information from the two data field message, but it would require a number of compares. In the one field data messages we get this information free. This is lost in the double data field message. This information will be advantageous for us later when we want to do things like synchronizing the kits counters after a ball move in one direction (done by loading a counter).

### How Many Messages?

The first thing that we must do to calculate the throughput is to determine the number of bits per message. For our setup, this was given as one start bit, eight data bits, and one stop bit, for a total of ten bits. Therefore, the calculation for the single data field message becomes:

$$4 \text{ bytes/msg} * 10 \text{ bits/msg} / 9600 \text{ bits/sec} = 4.167 \text{ ms/msg}, \text{ and therefore } 240 \text{ msg/sec.}$$

Using similar calculations, the double data field example comes to 5.208 ms/msg or 192 msg/sec.

As far as improvements go, there are two areas that should be reasonably straight forward to improve if it becomes necessary:

**UART Baud Rate:** It has been suggested that we use 9,600 baud for the transmission speed of our UARTs. If necessary, however, we can increase the speed. The software that comes with this course will also support faster speeds: 19,200 and 38,400 baud. In order to increase our speed, we have to check to make sure that the UART that we are using can support the faster speed and we have to provide the correct clock to the UART. For more details on this see the Design Issue section below.

**Decrease Byte Size:** Another possibility that can increase the messages rate is to shorten the length of the bytes. We already removed the parity bit and are using only one stop bit, so the only way left to us to shorten the bytes is to transmit less data bits per byte. In order to see if this is possible, let us look at the data that we are going to transmit:

**Sync Byte:** This byte is just for message identification, and therefore we can redefine it to be any appropriate (unique) number.

**Player ID:** There will only be four players, so we will only need two bits.

**Message ID:** If we count two message IDs each for paddle move, ball move, and velocity update, and one message ID for Reset, Claim Ownership, we have 8 messages. Leaving some room for expansion we still only use up four data bits.

**Data Field:** For the X and Y coordinates we need to range from 0 to 63 (the size of the playing field) and therefore we take up 6 data bits. The only thing that causes us a problem is the velocity, which has been defined as 1 direction bit followed by seven cycle bits (where the higher the number of cycles the slower the ball). We could of course redefine velocity to be one direction bit followed by six cycle bits (seven total data bits), or even 1 direction bit followed by five cycle bits (six total data bits). These two schemes would allow us to use seven and six data bits respectively and therefore lower our transmission time. They are, however, trading off accuracy of velocity for network speed, but this could be worthwhile if necessary.

The following table provides calculations for the messages per second with respect to data bytes per message and baud rate. This table assumes one data field (4 bytes/msg) and ignores any overhead the kits may require:

baud\data bits	8	7	6
9,600	240 msg/sec	267 msg/sec	300 msg/sec
19,200	480 msg/sec	533 msg/sec	600 msg/sec
38,200	960 msg/sec	1067 msg/sec	1200 msg/sec

**Join Player and Message ID:** After looking at the data bit requirements for our various messages, it becomes evident that another optimization can be made. The Message ID is going to take up four data bits, and the player ID is going to take up two data bits. We can optimize these by putting them together in the same byte when we transmit. This will cut down one character per message that we have to send - a 25% improvement!

The new performance table will look as follows (3 bytes per message):

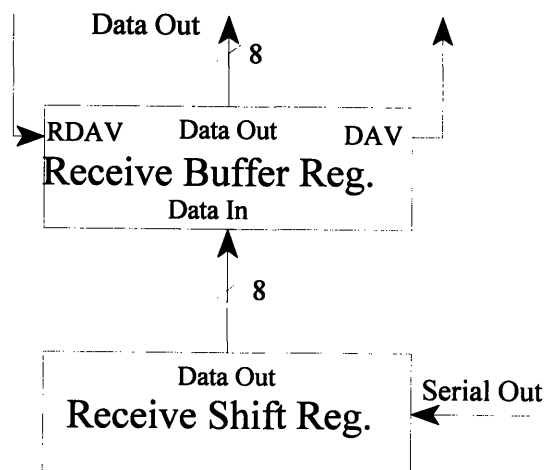
baud\data bits	8	7	6
9,600	320 msg/sec	356 msg/sec	400 msg/sec
19,200	640 msg/sec	711 msg/sec	800 msg/sec
38,200	1280 msg/sec	1422 msg/sec	1600 msg/sec

After looking at the performance figures, by just using data optimization (keeping the same baud rate) we managed to increase our throughput by 67%. Keep in mind, however, that everyone on the same network **must use the same optimizations** or the communication will not work. Also make sure that you set the correct parameters on the computer program provided. Furthermore, remember that these are performance estimates, they do not include delays for token passing and other processing times; therefore your true throughput may differ significantly.

## Design Issues:

### Interfacing with the UART:

In this lab we will be designing the network unit that will be responsible for receiving the network messages. This responsibility entails looking for the sync byte which indicates the start of the message and then copying the next three bytes into a register file. When this is completed it raises a signal to say that it has a message ready. It is at this point that the external controls have access to the register file. When the external source is finished it raises an acknowledgement, at which point the network unit starts the process over again. Note: if the external device does not return control to the network unit before two bytes come into the UART, then a byte will be overwritten and therefore a message will be lost. This is, of course, a reasonable amount of time, but one should be aware of the constraint.



**Figure 1: UART Receive Diagram**

The only necessary component that we haven't worked with yet is the UART. As you know by now, the UART is a device that takes parallel data (a byte) and sends it out serially (one bit at a time). Then a receiving UART takes the serial stream and reconstructs the original data byte. The UART is especially nice because it is an easy to use device which arbitrates data transfer between two or more devices. The functional interface for receiving information from the UART is very simple. When a byte is available the UART asserts its Data Available (DAV) signal. The output of the UART is tri-stated, so in order to actually read the byte you have to assert Received Data Enable (RDE), and the byte will be put on the output bus. When you are finished reading the byte, you must assert the Reset Data Available (RDAV) signal, which tells the UART you are finished. The UART then lowers DAV until the next byte is available. The only important stipulation is that if a new byte requires servicing before you have finished processing the previous byte (i.e. by asserting RDAV) the new byte will be lost.

Sending a data byte is just as simple. The first thing that we do is wait for Transmit Buffer Empty (TBMT) to be asserted. As the name suggests, this signifies that the transmit buffer is empty and that we are free to send the next byte. We do this by placing the desired byte on the input bus and asserting Data Strobe (DS). This clocks in the data and starts transmitting. We then wait for TBMT to be asserted again and repeat the cycle. Normally, this is all that we are interested in. There is one additional signal, End Of Character (EOC), that we are going to find useful due to the nature of our network. First, we have to look at how the transmit part of the UART works internally.

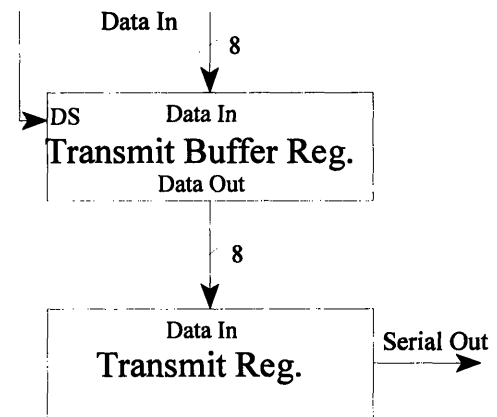


Figure 2: UART Transmit Diagram

The UART is double buffered, meaning that there are two registers used for transmitting. The Transmit Register is the one that actually get shifted out serially. This register is not loaded directly by the user; instead it is loaded from the Transmit Buffer Register. This transfer occurs automatically as soon as the last byte is shifted out, assuming that there is another character waiting in the Transmit Buffer Register. When you actually Data Strobe the last byte that you want to send into the UART, there is at least one and possibly two characters that need to be sent out. Keep this in mind, for if you tell our network hub that you are finished transmitting as soon as you stuff your last byte into your UART, then the end of your last message is going to get cut off. In order to fix this problem there is an End of Character (EOC) signal that the UART asserts when it has finished transmitting a character out of its Transmit Register. Normally you can ignore this signal, for the UART will automatically load the next character from the Transmit Buffer if available. If, however, you want to make sure that the UART has finished transmitting, you can look for both EOC and TBMT both to be asserted. This assures you that the UART is finished.

### Setting up the UART:

Before we can make the UART transfer bytes, we must set it up properly. There are a number of parameters which must be set up to make a UART function properly. Probably the most important thing is to set the proper baud rate (bits per second). Although we can choose any baud rate we want, both the sender and the receivers must all agree or the data will not be transferred properly. For the purpose of consistency you should set your UART up for 9600 baud. This is done by providing a clock to the transmit clock (TCP) and the receive clock (RCP) which has a frequency 16 times faster than the baud rate. For the case of 9600, we want to supply a clock rate of 153.6KHz. The provided frequency can vary a little from this frequency. For example, one easy way to obtain the desired frequency is by dividing a 10MHz clock by 64, thus providing 156.3Khz. Conveniently, this is the system clock frequency. This frequency will be close enough for our purposes.

There are a number of other parameters which can be set on the UART. These include parity, data bits and stop bits. Parity is a check to test if the data is received correctly. We will not be using this option, and therefore it should be turned off. Data bits are the number of data bits that will be transmitted. We will need to have 8 data bits for our project. Finally, stop bits should be set at one. Keep in mind, all of the sending and receiving kits must be set up for the same parameters or the communication will not work properly; this includes the computer. If you change the baud rate, parity, or stop bits, make sure that you configure the included software to reflect these changes. The default for the software is 9600 baud, no parity, 8 data bits and one stop bit.

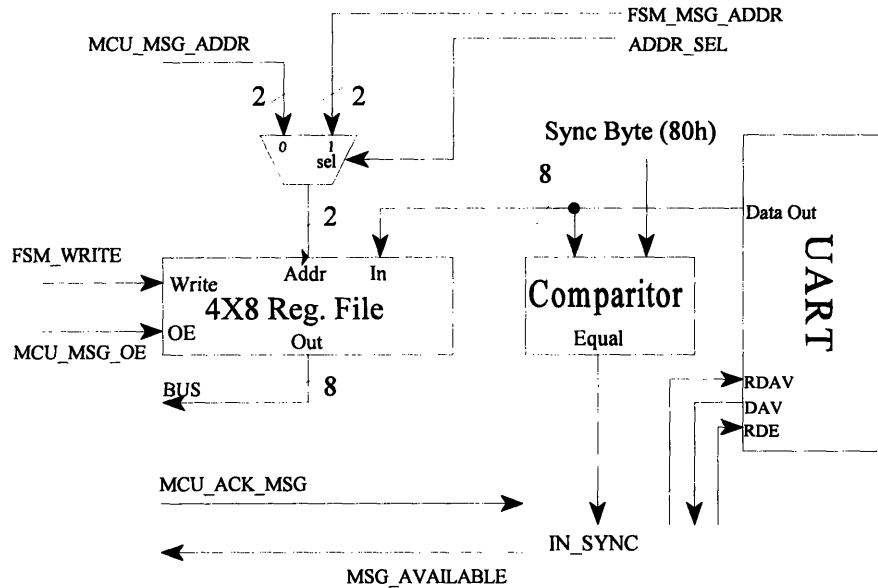
### FSMs revisited:

When designing more complicated controllers, it is important to have a well-defined method of approaching the problem. The first suggestion is to make a list of the inputs to the controller and the outputs that the controller must provide. Furthermore, it is often beneficial if the inputs and outputs that are related are grouped together. The next step is to start laying out the states. After you have laid out the states, make sure that you go through a few mental simulations to insure that the FSM is doing exactly what you expect it to.

### **Lab Assignment:**

You are to build the network receive unit. As previously explained, this unit starts by looking for a sync byte of a message from the UART. When it finds one it then receives the remaining three bytes of the message and places them in a small register file. When the entire message is read in, it asserts the signal, MSG\_AVAILABLE, to this effect. At this point it allows outside access to the addresses of the register file. It stays in this state until MCU\_ACK\_MSG is received, signifying that the other controller has finished with the message. This controller then de-asserts MSG\_AVAILABLE and starts waiting for another message to begin.

The basic components for this system are the UART, a 4X8 register file, a 2X2 MUX, and a comparator. The basic block diagram is as follows:



Note: if you use a register file like the 74LS670, with separate read and write addresses, you can simplify this design by removing the address MUX, which would be unnecessary.

Keep in mind that the message format will be:

Offset	Name	Mem Location
0	sync byte	Discard
1	player #	0
2	msg id	1
3	data field	2

At this point you have all of the information that you need to construct the network unit. You can test this unit by attaching the computer (as a slave) to the hub and configure it to run four computer players. This way your unit can be decoding the serial messages that come out of the computer. You only want to listen and you do not want to send anything, therefore make sure that you keep the Token Return attached to `Vcc`.

# NetPong Prelab #4

---

## How fast do we need to process messages?

The system that we are building is a real time system, meaning that you will have to process messages in real time. If the speed at which you can process messages can not keep up with the speed that messages are coming into your system, then you will lose messages and the system will not work properly. When dealing with real time systems, it is important to consider how much time you will have to process events, in our case incoming messages. There is, of course, some control you have over how fast you can process messages. Messages come in at a constant rate, therefore if you increase the system clock speed you will be able to execute more instructions in the intervals between message.

⇒ Assuming that the system clock rate is 156.3KHz (as suggested in Lab 1) and your baud rate is one sixteenth of this (~9600), calculate the number of instructions that you will have available to process each message. Do this for messages with Player#/ID compression and those without, assuming 8 data bits. Furthermore, calculate the number of instructions that you will have if you lower the number of data bits to 6 or 7. Does this seem like enough instructions to process our messages?

⇒ What happens if we take an especially long time processing a message and the Network Unit asserts the MESSAGE\_AVAILABLE line, signifying that the next message is available? When will we really lose a message and what message will be lost?





# NetPong Lab #4

---

**Object:**

Design of the MCU

**Topics:**

MCU, PROMs  
Bus Contention

**Prelab Discussion:**How fast do we need to process messages?

Assuming no Player#/ID compression, we have 4 bytes per message. Furthermore, if we have 8 data bits, one stop bit and one start bit we have a total of 10 bits per byte. This leads us to the following calculation:

$$156,300 \text{ cycles/sec} / 9,769 \text{ bits/sec} * 10 \text{ bits/byte} * 4 \text{ bytes/msg} = 640 \text{ cycles/msg}$$

We can do quite a lot of processing in 640 cycles, and it should be far more than we need for our purposes.

The calculations for the other transmit configurations are as follows:

Compress\Data bits	8	7	6
Player#/ID Compress	640	576	512
No Player#/ID Compression	480	432	384

Looking at these numbers, with Player#/ID compression and six data bits we have only 384 cycles per message, half of what we have with no compression and eight data bits. Still, 384 instructions is a lot and you should not run into a problem as long as you are somewhat careful.

When the Network Unit raises the MESSAGE\_AVAILABLE line, it is no longer servicing the UART. This means that when the next byte comes in the UART will assert DAV and it will be ignored by the Network Unit. The Network Unit can still recover until the second byte comes in and overruns the first one. The message that will be lost will be the message after the one that the Network Unit is currently holding.

## **Design Issues:**

### Programmable Read Only Memory (PROM)

PROMs are memory devices which store information to be read out. In standard operation, the PROMs can only be read. In order to read a value from the PROM, one places the address that you want on the input address lines. After a short delay, the output is available on the data pins. Many varieties of PROMs are available. Some of the most popular that are used for development are Electrically Erasable PROMs (EEPROMs). These PROMs can be written to, although the procedure is usually much slower and more complicated than using SRAM, and often requires higher voltages (+12V). PROMs also come in a wide variety of sizes, from 1KB or smaller to 1MB and larger. PROMs are usually programmed in a machine in much the same way as PALs. You usually provide the PROM programmer with the data corresponding to each address. The exact format necessary depends upon the software you are using.

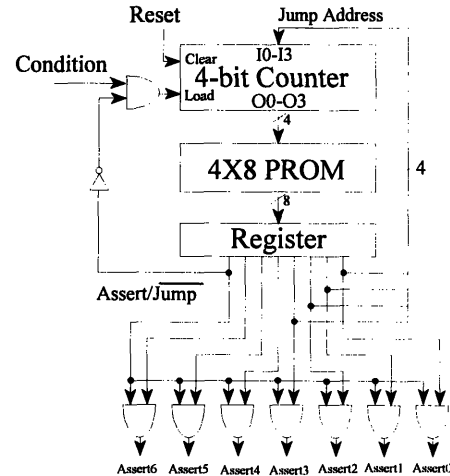
PROMs have many useful applications. The most basic application is as a lookup table. For example, let's assume that you want to compute the cosine of a 12-bit digital number. The actual calculations for performing a cosine operation are quite complicated and your system may not have the capability or the time to do this calculation. What you can do, instead, is to use the PROM as a cosine lookup. The 12-bit number is placed at the address lines of the PROM and the data read out of the PROM will contain the cosine of the number. Of course you will have to precompute (most likely using a computer program) all of the cosines possible and store them in the PROM at the appropriate locations.

### Micro Control Unit - a more advanced controller

We have already looked at FSMs and how to use them as controllers. The problem with FSMs, as you have probably already noticed, is, as they get larger, they get complicated very quickly. The thought of a fifty or a hundred state FSM should not be a pleasant one. It is often necessary to have advanced controllers that do a large number of tasks. The Micro Control Unit (MCU) is a more advanced controller that will allow us to do more complex tasks. The MCU executes instructions that we specify (called microcode) that tells the MCU and the system what to do.

The simplest of MCUs can be built using a counter, a PROM and a register. The counter is attached to the address inputs of the PROM. The PROM contains the microcode. As the counter counts up from location zero, the outputs of the PROM is the microcode. The outputs of the PROM should be latched in a register and the outputs of the register should be attached to the digital hardware that we want to control. In this way we can produce a reliable, consistent waveform that can control the system. The reason for the register is that the output of the PROM is often glitchy and we do not want the glitches to be transferred in our control signals. In this most basic form, this system has very limited applications.

The system just described only asserts signals. In order to make this basis of an MCU more useful, we have to incorporate some way to react to information. We are going to do this by incorporating a conditional jump instruction to our counter. We will use the first bit of our microcode to determine whether this is a conditional jump or an assert instruction. This first bit will be called the microinstruction (either assert or conditional jump). A conditional jump instruction will need to include the location of the microcode to which we want to jump. This address in the microcode will be fed back to the input of the counters. If the instruction code is a jump (i.e. the first bit is a zero) and the condition is true, the new microcode address is loaded into the counter. If the instruction is an assert, or a conditional jump where the condition is zero, the counter will not load and instead will increment to the next piece of microcode. The final change that must be made is to have some assertion logic that only asserts the signals from the PROM if the instruction code is an assert (i.e. first bit is a one). Otherwise when we have a conditional jump instruction, the jump address would be interpreted as assertions.



*Note: Just as in the FSMs, if the condition bit is asynchronous to the MCU, we need to put it through a latch with the same clock as the FSM to prevent violating the setup and hold time of the counter.*

Because microcode is at such a low level, it will be different for every implementation. If we are to describe the microcode we have just designed, it would be done as follows:

Conditional Jump	0	X	X	X	A	A	A	A
Assert	1	S0	S1	S2	S3	S4	S5	S6

- where X - don't care
- A - address
- S# - assert signal

The MCU that was just described is limited in a number of ways. First, with only 4 address bits, there can only be 16 microinstructions. This can of course be remedied by increasing the size of the counter. In order to keep our example simple, this expansion is not included in the example below. The second problem is that since there is only one condition bit we can only make decisions on the value of one signal. This can be expanded using a MUX. If we use the three don't care bits on the Conditional Jump instruction as the select to our MUX and then attach the output of the MUX to our condition bit, we can selectively test each of the eight signals. We usually want to fix one of the MCU inputs to logical one to allow for unconditional jumps. The final problem we face is that we only have seven assert signals. Often complicated systems will have many more signals than this. There are a number of remedies. The first is expanding the size of the PROM. Usually PROMs are eight bits wide, if we put them in parallel we can get eight, sixteen, twenty four or more bits wide MCU. Another option is to have different banks of assertions. A suggested microcode format would be:

Conditional Jump	0	C0	C1	C2	A	A	A	A
Assert1	1	0	S0	S1	S2	S3	S4	S5
Assert2	1	1	S6	S7	S8	S9	S10	S11

where A - address  
C# - select bits to condition MUX  
S# - assert signal

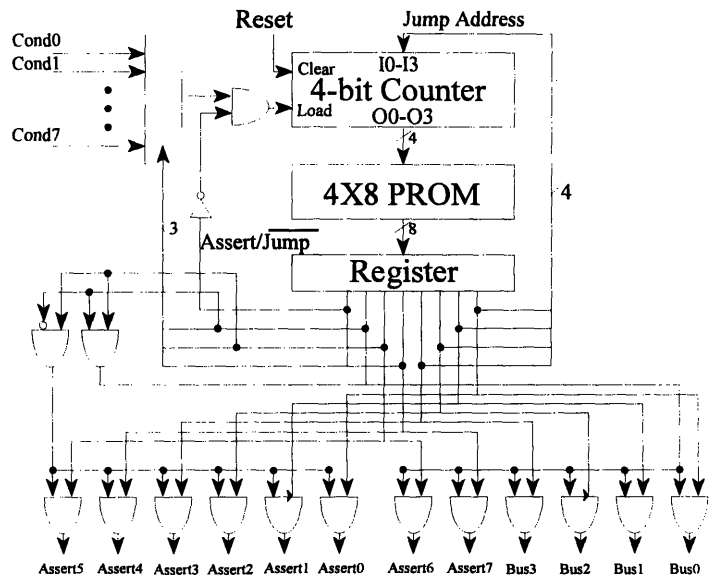
This new microcode format allows us to output 12 different assert signals (as opposed to the original 7) without increasing the size of our microcontroller. The drawback is that signals from Assert bank 1 (S0-5) and Assert bank 2 (S6-11) can not be asserted on the same clock cycle. Sometimes this is acceptable and sometimes it is not; thus, when you use this method, you have to arrange your assertion signals so that the ones you need to assert together are in the same bank. If necessary, some signals can be in both banks (i.e. have both S0 and S6 attached to the same register load). The hardware that accomplishes this is straight forward and is easily created out of logic gates or in a PAL.

Still another useful method for obtaining more signals is encoding them. For example we can encode four signals (S0..S3) into two bits (Z0, Z1). If Z0=0 and Z1=0 then assert S0 (and don't assert S1, S2 or S3). If Z0=1 and Z1=0 assert only S1, etc. This provides very high density packing of signals that do not need to be asserted together. This is especially useful for tightly packing signals that are output enables to the system bus, as you will only want one device driving the bus at a time.

One final note on squeezing the most signals out of your microcode: it is often possible to use the same signals for different purposes. For example, if you have two devices that both use a one bit address, then as long as you don't ever need to use both devices at the same time you can likely use the same signal for both address lines.

### Immediate Values

One feature that is nice to have, although not always necessary, is the ability to embed data in the microcode (i.e. Load 06h into register A). We will have to create a special instruction to allow the immediate value. In essence, an immediate value is nothing more than a number of asserts that attach (and therefore drive) the bus. In addition to driving data, you will need one or more regular asserts that allow you to store the immediate value in a register or main memory. Let us further enhance the microcode that we have suggested by adding an instruction for an immediate value:



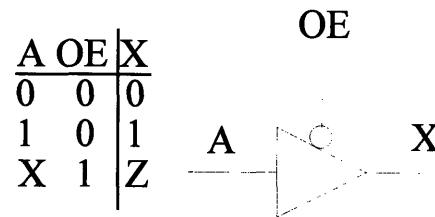
Conditional Jump	0	C0	C1	C2	A	A	A	A
Assert	1	0	S0	S1	S2	S3	S4	S5
Immediate Value	1	1	S6	S7	I3	I2	I1	I0

where A - address  
 C# - select bits to condition MUX  
 S# - assert signal  
 I - immediate

Keep in mind, if we only allow four bits of immediate data, a range from 00h to 0Fh, we must drive the upper four bits of the bus to logic zero on an Immediate Value instruction.

### Tri-States

Tri-states are a type of logic gate that has three output states: '1', '0', or High Impedance (essentially unconnected). Tri-states are important to understand when talking about busses. A bus is a set of digital lines with a number of devices connected. Normally, if we connect more than one output device to the same set of digital lines, they will conflict, resulting in an indeterminable output and possibly even damage to one or both components. Tri-states allow you to avoid this issue by electronically attaching and detaching various components to the bus. Because they are so useful, many components will have tri-states built into their outputs.

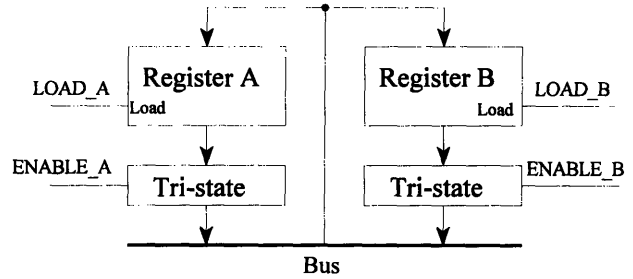


Tri-states also introduce another problem - floating inputs. Because it is possible, and often desirable, to have no devices driving the bus, we have the issue of floating logic lines. Floating logic lines are digital lines that do not have any specific value and can be a logical '0', logical '1' or in the gray area in between. Floating lines can make debugging difficult if you are accidentally loading a register from the floating bus. Floating lines are especially bad if you are using CMOS, which draws large amounts of power when not at a solid logic '1' or logic '0'. This can be fixed through the use of pull-up or pull-down resistors. These resistors are usually 2.2K resistors that are attached from each line of the bus to power or ground. This results in pulling the bus up to logic '1' or down to logic '0' respectively.

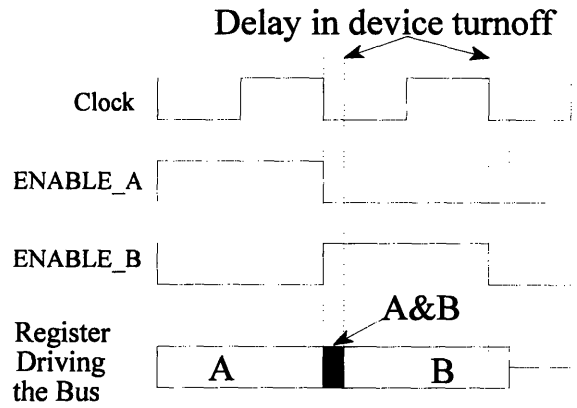
### Bus Contention:

Bus contention is a problem that plagues many designs. Before we discuss how to avoid bus contention, let us describe what it is. Bus contention occurs any time two or more devices try to drive the same logic line. This is especially significant when they try to drive the same line to a different value. There are two types of bus contention: transient and steady state, each of which will be discussed separately.

On a data bus, many devices are attached through the use of tri-states. Tri-states are used to connect and disconnect the device from the bus. Steady state bus contention occurs when two or more devices are told to output to the bus at the same time. As an example, let us take a simple bus with two eight bit registers attached through tri-states (see diagram). Let us assume that the four signals (LOAD\_A, ENABLE\_A, LOAD\_B, and ENABLE\_B) are controlled by an MCU. Let us further assume that Register A and B contain 0Fh and A7h respectively. If the MCU executes an instruction that asserts both ENABLE\_A and ENABLE\_B then both devices are trying to drive the bus to their levels. Bit zero, for example, wants to be pulled to a one by Register A and to a zero by Register B. The net result is a very low resistance path from power to ground (through the two chips) which uses a lot of power. Furthermore, you can burn out one or both of the chips that you are using due to the large amount of current flowing. If, in addition, we attempt to load one of the registers from the bus, the value that we read in from the contested bits will be indeterminable.



Transient bus contention is a less severe condition, though it is harder to recognize. Transient bus contention occurs when you turn a device on and another off on the same instruction. There is often a brief overlap where both are trying to drive the bus at the same time. This results in much the same situation as the steady state bus contention, though for a much shorter period of time. This still draws large amounts of power and can potentially damage chips. The most obvious solution is to place a dummy instruction between any two instructions that want to use the bus. Unfortunately, this will slow our MCU or FSM to half speed. A better solution is to only use the bus for half of the clock cycle; in other words, AND all of your tri-state enable bits with the clock. This will prevent bus contention altogether and let your controller perform at full speed.

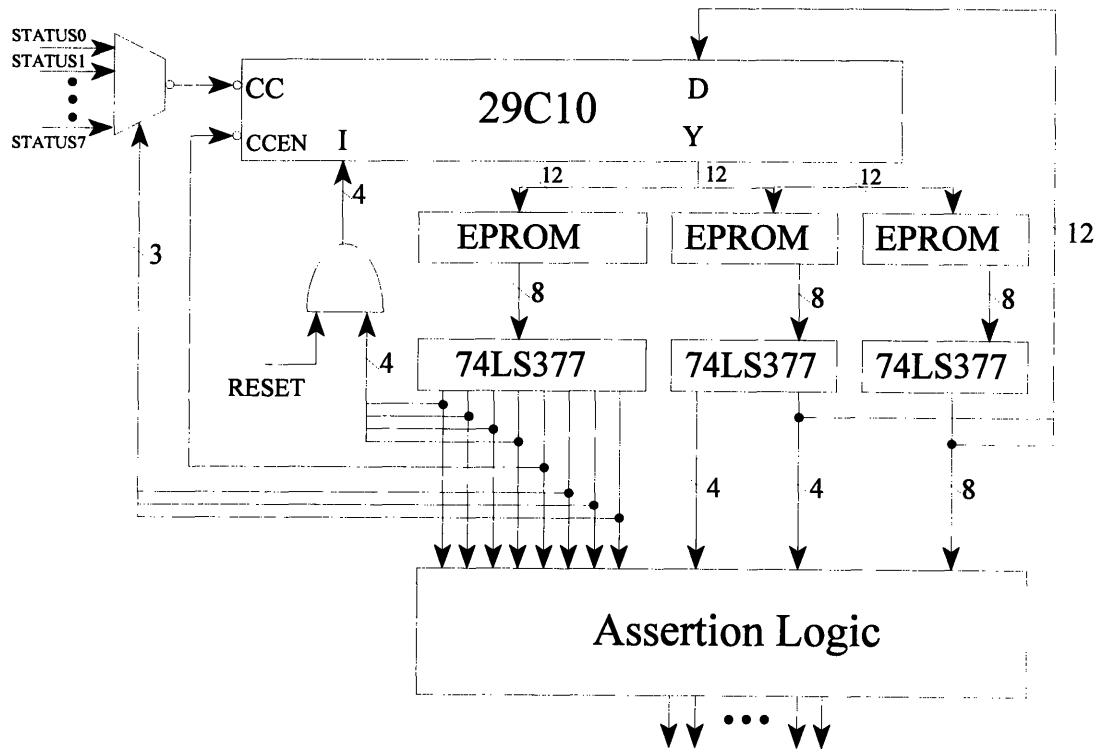


### Lab Assignments

It is suggested that for this project you replace the simple counter with a more advanced microprogram controller, such as the Advanced Micro Devices(AMD) Am29C10A. This chip has 16 built-in instruction formats that allow looping, conditional check, subroutines, and other advanced features that make for shorter microprograms and easier programming. These features are explained in detail in the specification sheet.

You are to build and test the MCU that you will use to control your kit. You are to use the test software that is provided to test your MCU and make sure that it works properly. Details on the operation of the test code are provided as comments in the code.

A suggested diagram for creating your MCU is provided below.



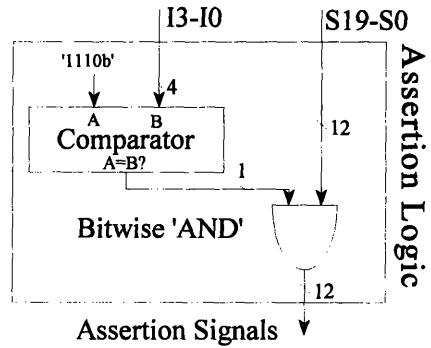
The following is the microcode format that our MCU will use:

Conditional Statements	IIIIUCCC	XXXXAAAA	AAAAAAAA
Immediate Value*	IIIISSSS	XXXXXXXX	VVVVVVVV
Assert	11100SSS	SSSSSSSS	SSSSSSSS
Bus Immediate Value*	11101SSS	SSSSSSSS	VVVVVVVV

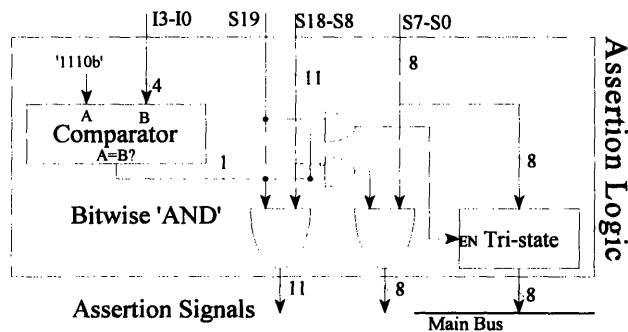
- where
- I - microinstruction
  - A - address
  - U - unconditional bit
  - C - select bits to condition MUX
  - S - assert signal
  - X - don't care
  - V - immediate

\* The Immediate Value instruction format is used by the 29C10 when the desired instruction loads the internal counter. When you want to load values onto the system bus, you should use the Bus Immediate Value format.

The job of the assertion logic is to appropriately interface the signals that need to be controlled to the raw MCU output. At the most basic level, before we concern ourselves with compression or immediate values to the bus, the assertion logic wants to allow the twenty signal outputs (S19-S0) of the MCU (twenty four minus 4 instruction bits) to be reflected in the control signals only when an assert instruction is executed (I=1110b).



The assertion logic gets a little more complicated when we want to add the possibility for an immediate value to the bus. We then use signal number eleven (S19) to control whether to assert an immediate value onto the bus or not. In either case, signals S18-S8 are always asserted. Signals S7-S0 will be asserted if S19 is a '0'. If, however, S19 is a '1', then the lower eight bits from the MCU will be placed directly on the bus. Keep in mind that this means we can not use signals S7-S0 when using an immediate value. Make sure that any signal that is needed to latch or write the immediate value is not placed within these bits.



In addition, when you need to compress signals in your microcode, the assertion logic is very likely the correct place to uncompress the signals (see description of various compression techniques in the Discussion section).

One final note on the assertion logic. There are numerous devices that use negatively asserted logic in some or all of their controls. Most people are much more comfortable dealing with positive asserted logic and therefore make more mistakes when building a controller that deals with negatively asserted signals. A suggestion that might make coding easier is for the microcode to consider all signals to be positively asserted and, in the assertion logic, invert the ones that need to be negatively asserted.



```

/* mcu.sp is the specification file for the MCU debugging code */
/* Last revised 3-9-92 */

/*****
/* Instruction Word Organization:
/* conditional statements   iiiiuccc xxxxaaaa aaaaaaaa
/* immediate data statement iiiissss ssssssss vvvvvvvv
/* assertion statements    11100sss ssssssss ssssssss
/* immediate assert to bus 11101sss ssssssss vvvvvvvv
/* where i = opcode selection
/* u = unconditional bit
/* c = status selection
/* a = alternative address, i.e. jump address
/* v = immediate values, i.e. binary data
/* s = assertion signals
*****/

op <23:0>;          /* Indicates the available bits          */
address op <11:0>;  /* Indicates bit locations for addresses          */
value op <7:0>;     /* Indicates bit locations for immediate val     */

/*
 * There is nothing magic about upper case.
 * You may change things to lower case as you wish.
 * Remember, the assembler maps all characters to lower case anyway!
 */

/*
 * Instruction Set of AM29C10A
 * All of the instructions available are defined.
 * In all likelihood, you will only use a subset.
 */

RESET op<23:20>=%b0000; /* Jump Zero (RESET)          */
CCALL op<23:20>=%b0001; /* Cond CALL subroutine       */
CJMP  op<23:20>=%b0011; /* Cond JuMP                  */
PUSH  op<23:20>=%b0100; /* PUSH stack down / cond load counter */
CJSRP op<23:20>=%b0101; /* Cond Jump to Subroutine via Register Pipe */
CJRP  op<23:20>=%b0111; /* Cond Jump via Register Pipeline */
RFCT  op<23:20>=%b1000; /* Repeat loop with File when CounTer != 0 */
RPCT  op<23:20>=%b1001; /* Repeat loop with Pipeline when CounTer!=0 */
CRTN  op<23:20>=%b1010; /* Cond ReTURn                */
CJPP  op<23:20>=%b1011; /* Cond Jump Pipeline and Pop  */
LDCT  op<23:20>=%b1100; /* Load CounTer              */
LOOP  op<23:20>=%b1101; /* test end-of-LOOP          */
ASSERT op<23:20>=%b1110; /* continue, perform the ASSERTions */
TWB   op<23:20>=%b1111; /* Three-Way-Branch          */

/* Unconditional branch statements */

CALL  op<23:19>=%b00011; /* CALL subroutine          */
JMP   op<23:19>=%b00111; /* JuMP                    */
JSRP  op<23:19>=%b01011; /* Jump to Subroutine via Register Pipeline */
JRP   op<23:19>=%b01111; /* Jump via Register Pipeline */
RETURN op<23:19>=%b10101; /* RETURN                  */
JPP   op<23:19>=%b10111; /* Jump Pipeline and Pop   */

/* These are defined so that you may use them to make your code more

```

```

* readable. Their use is not required. */

IF    nop;
THEN  nop;
TRUE  op<19>=1;          /* This makes sure /CCEN is deasserted */

/* Assertions */

L0      op<0>=1;
L1      op<1>=1;
L2      op<2>=1;
L3      op<3>=1;
L4      op<4>=1;
L5      op<5>=1;
L6      op<6>=1;
L7      op<7>=1;

L8      op<8>=1;
L9      op<9>=1;
L10     op<10>=1;
L11     op<11>=1;
L12     op<12>=1;
L13     op<13>=1;
L14     op<14>=1;
L15     op<15>=1;

L16     op<16>=1;
L17     op<17>=1;
L18     op<18>=1;

IMMEDIATE op<19>=1;
BUS       op<19>=1;

/*
* Status signals: Switches and frequency divider output OSC
* Make sure that all status signals that change during mcu operation
* are synchronized to the system /CLK
*/

S0      op<18:16>=0;
S1      op<18:16>=1;
S2      op<18:16>=2;
S3      op<18:16>=3;
S4      op<18:16>=4;
S5      op<18:16>=5;
OSC     op<18:16>=6;
S7      op<18:16>=7;

/* End of debugging software specification file */

```

```

/* Debugging software assembler code      Last revised 4-10-96      */
# SPEC_FILE = test.sp; /* This statement is required at the
                        beginning of the ASSEM_FILE. It tells
                        where the SPEC_FILE can be found. */

```

```

# LIST_FILE = test.lst; /* This statement specifies the name for
                        the assembler listing file. If not
                        included, no listing will be created */

```

```

# MASK_COUNT = 8; /* This statement is required to mask out 8
                  bits of the 16 bit op-code to produce 2 PROM
                  files. Use with the 'assem16to8' command. */

```

```

# SET_ADDRESS = 0; /* This statement tells the program at what
                  address to start assembling. The address
                  given is a hexadecimal number. */

```

```

# LOAD_ADDRESS = 0; /* This statement, if used AFTER the
                    SET_ADDRESS statement, determines the
                    beginning PROM address for the program
                    image. The address is in HEX. */

```

/\* To execute this code, you should use the PAL file mcu.pal. The code has been placed in memory starting at HEX 100 so you can use the same PROM for the debugging code as well as your regular code. Wire address bit A8 of your PROM to +5 to execute this code. Check the pal file, mcu.pal, and the specification file, mcu.sp, to determine wiring from the pipeline register to the PAL.

The OSC input to the multiplexor is the output of a frequency divider counter that divides the clock frequency of the MCU by 256. If the LEDs flash too slowly, either change the LDCT instruction in this code (there is only one) to load a smaller number into the sequencer register or choose a smaller output from your frequency divider as the OSC input. If the LEDs flash too quickly, change the LDCT instruction to load a larger number into the register.

If nothing works, hook up your logic analyzer and follow the addresses being executed. Use the mcu.lst file to follow the flow of instructions and to see exactly what should be on the prom outputs and the sequencer inputs. Check to see that the operand bits from the AND gate are correct. Given that operand, does the sequencer select the correct next address from the D inputs, its stack, or the program counter? Remember that the address inputs to the PROMs will be one clock cycle ahead of the current instruction. \*/

```

/* Begin debugging code */

```

```

START:    ASSERT;          /* Test to see if program counter increments */
          ASSERT;          /* If PC does not increment, test to see */
          JMP TESTC;       /* that Ci is connected to +5 */
FAIL1:   RESET;           /* If code reaches here, then test your */

```

```

TESTC: CALL TESTE; /* TRUE condition bit(s) */
        JMP BEGIN;
TESTE: RETURN;
FAIL2:  RESET; /* If code reaches here, see that the proper */
        /* instruction is being seen by the sequencer */

BEGIN: CJMP S5 PROC2; /* If Switch 5 = 0 then execute PROC 1 */
        CALL PROC1; /* otherwise execute PROC 2 */
        JMP BEGIN; /* PROC 1 lights individual LEDs */
PROC2:  CALL BF; /* PROC 2 lights different patterns */
        JMP BEGIN;

PROC1:  CJMP S0 LIGHT0; /* This checks LEDs 0 -> 4 . Each LED can */
        JMP CHECKS1; /* be turned on or off independently */
LIGHT0: ASSERT L0;

CHECKS1: CJMP S1 LIGHT1;
        JMP CHECKS2;
LIGHT1:  ASSERT L1;

CHECKS2: CJMP S2 LIGHT2;
        JMP CHECKS3;
LIGHT2:  ASSERT L2;

CHECKS3: CJMP S3 LIGHT3;
        JMP CHECKS4;
LIGHT3:  ASSERT L3;

CHECKS4: CJMP S4 LIGHT4;
        RETURN;
LIGHT4:  ASSERT L4;
        RETURN;

BF:     CJMP S0 ODD; /* If Switch S0 then flash odd LEDs */
        CJMP S1 EVEN; /* If Switch S1 then flash even LEDs */
        CJMP S2 TEST_BUS; /* If Switch S2 then alternate odd & even */
        CJMP S3 ROTATE; /* If Switch S3 then rotate LEDs */
        RETURN; /* If proper function is not executed, test */
        /* to see if condition bit select and */
        /* inputs are wired correctly. If random */
        /* behavior, check instruction code bits */
        /* and AND gate inputs/outputs */

ODD:   ASSERT L1 L3 L5 L7 L9 L11 L13 L15 L17;
        /* Flash odd LEDs. Switch S4=1 doubles */
        CALL DELAY; /* the rate of flashing */
        ASSERT ;
        CALL DELAY;
        CJMP S0 ODD;
        ASSERT; /* Turn LEDs off */
        RETURN;

```

```

EVEN:ASSERT L0 L2 L4 L6 L8 L10 L12 L14 L16 L18;
        /* Flash even LEDs. */
        CALL DELAY; /* Switch S4 changes the flash rate */
        ASSERT;
        CALL DELAY;
        CJMP S1 EVEN;
        ASSERT; /* Turn LEDs off */
        RETURN;

```

```

TEST_BUS:
    ASSERT BUS %h00; /* test the bus immediate values */
    ASSERT BUS %h01;
    ASSERT BUS %h02;
    ASSERT BUS %h04;
    ASSERT BUS %h08;
    ASSERT BUS %h10;
    ASSERT BUS %h20;
    ASSERT BUS %h40;
    ASSERT BUS %h80;
    ASSERT BUS %hFF;
    CALL DELAY;
    CJMP S2 TEST_BUS;
    RETURN;

```

```

ROTATE:    ASSERT L0 L18; /* Flash LEDs from outer to inner */
            CALL DELAY; /* and back */
            ASSERT L1 L17;
            CALL DELAY;
            ASSERT L2 L16;
            CALL DELAY;
            ASSERT L3 L15;
            CALL DELAY;
            ASSERT L4 L14;
            CALL DELAY;
            ASSERT L5 L13;
            CALL DELAY;
            ASSERT L6 L12;
            CALL DELAY;
            ASSERT L7 L11;
            CALL DELAY;
            ASSERT L8 L10;
            CALL DELAY;
            ASSERT L9;
            CALL DELAY;
            CJMP S3 ROTATE;
            ASSERT; /* Turn LEDs off */
            RETURN;

```

```

DELAY:
    CJMP OSC DELAY;
LOW:    CJMP OSC NEXT;

```

```

        JMP LOW;
NEXT:
        CJMP S4 TWICE;
        CALL REG;      /* Call REG to wait for given number    */
        CALL REG;      /* of OSC cycles.                                           */
        CALL REG;
        CALL REG;
        CALL REG;
        CALL REG;
        CALL REG;
        RETURN;

TWICE:
        CALL REG;      /* Call REG only half as many times to                    */
        CALL REG;      /* double the frequency                                     */
        CALL REG;
        RETURN;

REG:
        LDCT %H40;      /* Wait for this many cycles of OSC. */
HERE:
        CJMP OSC HERE; /* Change this value to change */
HERE2:
        CJMP OSC THERE; /* the LED flash rate. */
        JMP HERE2;
THERE:
        RPCT HERE;
        RETURN;

/* End of Debugging Code */

```

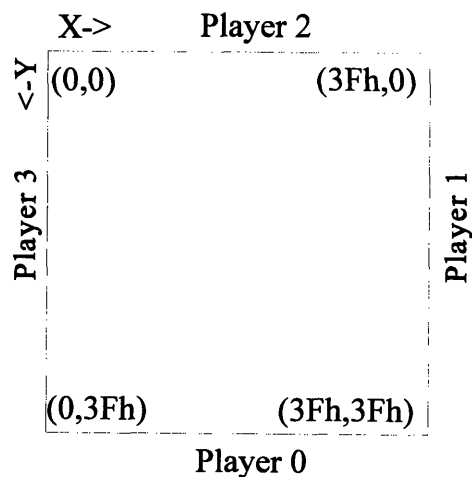
# NetPong Prelab #5

## Where am I?

There is an interesting problem that we have not yet touched upon. Each kit will have a player number from zero to three. Player zero will be on the bottom, player one on the right, player two on the top, and player three on the left. Your kit will be assigned a number from which you will have to determine on which side of the screen your paddle will be located. This has a couple of implications. First, given the ball's X and Y coordinates, you will have to know whether it is at your edge of the screen. Also, you will have to know whether your paddle should be going up and down or left and right. There are two general approaches that can be used.

The first approach is to make extensive use of 'if' statements in your microcode. As in any type of code, large numbers of 'if' statements makes for larger and more bug prone code.

There is an alternative, which is to make a hardware translator. This translator would take the player number and then modify the true addresses for the ball in such a way that it would always look like your player was at the bottom of the screen. For our purposes, assume that the top left hand corner of the screen has (X, Y) address (0,0) and the bottom right corner of the screen has address (3Fh, 3Fh).



☞ Figure out a way to translate the address in hardware so that no matter which player you are, the translated addresses look the same as those of player zero (bottom of screen). For example, if you are player one (right), then the true address for the top right part of the screen (3Fh, 0h) would translate to a virtual address at the bottom right hand side of the screen (3Fh, 3Fh). The net effect of the hardware translator is as if you tilted the screen so that the player you specified was at the bottom. Hint: Assume that the X and Y coordinates for the ball are stored in addressable registers whose addresses differ by the least significant bit. In order to switch X and Y values you can use an XOR gate to optionally invert the least significant bit of the address to the register storing the values. This way, when the controller tries to read the X value out of the register, it will get the 'virtual' X coordinate (maybe X or Y depending upon the player number). Provide some advantages and disadvantages of these two systems. Which one is more suited to our task? Are there any other complications that the hardware translator causes?

## Passing Ownership

As described in Lab 2, we are using a distributed ball ownership approach to designing NetPong. A player will claim the ball when the ball reaches their side of the playing field. The ball is physically claimed by sending a CLAIM\_OWNERSHIP message, with the data field being irrelevant (should be zero). This player is then responsible for updating the ball position and velocity until another player sends a CLAIM\_OWNERSHIP message, at which point the first kit stops updating the ball information. For this system to work properly, there must be one and always one ball controller.

In its simplest implementation we will have a variable to represent whether we, the player, own the ball or not. We will use the following rules for passing ownership:

- (1) When the ball hits our side of the screen, we set ball\_ownership to 01h, meaning we own the ball. We also put a CLAIM\_OWNERSHIP message in our output queue.
- (2) When we get the network token, we send our message queue to the UART, including our CLAIM\_OWNERSHIP message (as usual).
- (3) When the other players receive the CLAIM\_OWNERSHIP message they set their ball\_ownership to 00h meaning they no longer own the ball.

⇒ For most situations these rules work very nicely, but what happens when we put the protocol under stress? For example what happens when the ball goes into the corner, say for example the ball moves from (01h, 3Eh) to (00h, 3Fh). Why does the protocol fail? How can we fix it?



# NetPong Lab #5

---

## Object:

Design of Paddle I/O  
Velocity Counters  
Design of Sound Unit (Optional)

## Topics:

A/D, D/A, ALU, SRAM

## Prelab Discussion:

### Where am I?

In order to perform the correct conversion based upon the two player bits, we should first make a list of the required conversions for each player.

Player 0(00b)       $x = x$   
                          $y = y$

Player 1(01b):     $x = /y$  (Bitwise inversion of y)  
                          $y = x$

Player 2(10b):     $x = /x$  (Bitwise inversion of x)  
                          $y = /y$  (Bitwise inversion of y)

Player 3(11b):     $x = y$   
                          $y = /x$  (Bitwise inversion of x)

In order to make investigation easier, let us put this in a form similar to a Karnoff map:

$p1 / p0$	0	1
0	$x=x, y=y$	$x = /x, y = /y$
1	$x = /y, y=x$	$x=y, y=/x$

The first thing that should be evident is that if  $p1$  (player bit 1) is 1 then we should be switching  $x$  and  $y$ , otherwise we should not switch them. Furthermore, the  $y$  value (after the switch) should be inverted if  $p0=0$  and  $x$  should be inverted if  $p0 \oplus p1=1$ . Knowing these values, we could build a converter to do this for us. Before we do, let us look at the advantages and disadvantages of doing this:

### Numerous Ifs

#### **Advantages:**

Simpler  
Easier to conceptualize

### Hardware Correction

#### **Advantages:**

Slightly simpler code(?)  
Slightly faster

Looking at these two short lists, we could implement either system. In order to keep complexity down, however, it might be preferable to stick with the 'if' statement approach.

### Token Passing

In order for the ball to move into corner (00h, 3Fh), it must have last been in contact with (and hence owned by) either player one or player two. Lets assume that the ball is owned by player two. The following is a short simulation. At the beginning, assume that the ball is at location (01h, 3Eh):

*Player 2 has Network Token :*

Player two notices that both the X and Y velocity counters have reached zero and therefore places both a BALL\_X\_MOVE(00h) and a BALL\_Y\_MOVE(3Fh) in his output queue.

*Player 3 has Network Token:*

Don't Care.

*Player 0 has Network Token:*

Don't Care

*Player 1 has Network Token:*

Don't Care

*Player 2 has Network Token:*

Player 2 sends the data from its output queue, notifying all the kits that the ball has moved to (0h, 3Fh). Player three notices that the ball has reached his side, therefore he sets his ball\_ownership to 01h and places a CLAIM\_OWNERSHIP message in his output queue. Player zero also realizes that the ball has reached his side and also sets his ball\_ownership to 01h and places a CLAIM\_OWNERSHIP message in his output queue.

*Player 3 has Network Token:*

Player three broadcasts the CLAIM\_OWNERSHIP message that was in his output queue. Player two receives the message and clears his ball\_ownership to 00h. Player zero also receives the message and clears his ball\_ownership.

*Player 0 has Network Token:*

Player zero broadcasts the CLAIM\_OWNERSHIP message in his output queue. Player three receives the message and clears his ball\_ownership bit. **Nobody owns the ball anymore!**

In this presented situation, both player three and player zero want to claim the ball. It doesn't really matter which one ends up with ownership, but one, and only one, should have it at the end of the exchange. The simple rules that we came up with worked well up until the last step. The ownership exchange was working well, player three had claimed the ball and players two and zero both relinquished control. What failed was that there was a pending message in Player zero's output queue that caused player three to also clear his ownership.

The first shot at a solution is to add an additional rule:

- (4) When you receive a CLAIM\_OWNERSHIP message, go through your output queue and remove any CLAIM\_OWNERSHIP messages of your own.

This rule would fix our problem, but it is rather awkward to do. Searching through our network queue will not be very easy and actually removing a message will be difficult. There is another simpler solution!

In addition to having a ball\_ownership bit, each kit will have a pending\_ball\_ownership variable. The rules for token passing will be rewritten as follows:

- (1) When the ball hits our side of the screen, we set the pending\_ball\_ownership to 01h, meaning we are going to own the ball. We also put a CLAIM\_OWNERSHIP message in our output queue.
- (2) When we get the network token, we empty our message queue, including our CLAIM\_OWNERSHIP message. If pending\_ball\_ownership is 01h, set ball\_ownership to 01h. Set pending\_ball\_ownership to 00h.
- (3) When the previous ball owner (and all other players) receives our CLAIM\_OWNERSHIP message they set their ball\_ownership to 00h meaning they no longer own the ball. They do **not** clear pending\_ball\_ownership to 00h!

If you go through the scenario presented above with the new rules, you will see that one player and only one player (player zero) ends up with the ball ownership.

## **Design Issues:**

### Analog to Digital Converters (A/D):

Analog to digital converters are very useful devices which allow us, as the name implies, to convert between the analog and digital domains. These devices come in many varieties, often differentiated by speed of conversions (usually samples per second), bits of accuracy, and method of input (voltage -vs- current sensing). The basic operation starts by telling the A/D to

begin a new conversion. The A/D starts to measure the quantity, eventually coming up with a digital number. When the number is available, the A/D raises a signal to this effect. Often the output of the A/D are tri-stated and you must enable the output to the digital lines.

### Digital to Analog Converter (D/A):

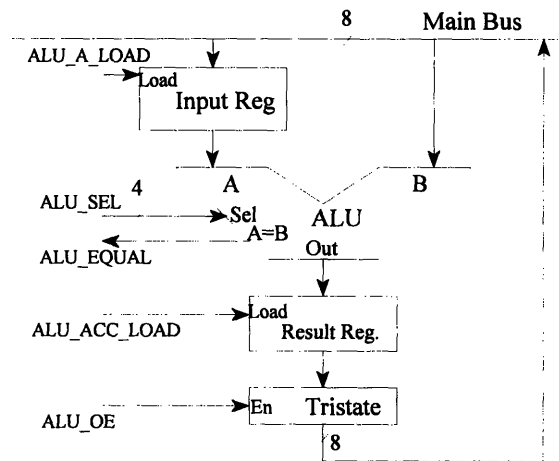
This counterpart to the A/D is considerably simpler. All one has to do is place a digital number at the inputs and latch them into the D/A. The D/A then updates the output to reflect the new value. As in the A/D converters, the D/A come in different speeds of conversions, bits of accuracy, and methods of output.

### Arithmetic Logic Unit (ALU):

The ALU is the basic element that is used for doing calculations. Various ALUs have different functions, but they often include addition, subtraction, shifting and comparisons. ALU's, like the 74LS181 (which you should use), are often 4-bit devices that are set up to be chained together to make ALUs of 8 bits or more. The basic operation of the ALU is started by placing the two operands in the A and B inputs, respectively, and selecting the appropriate function using the function input. As the circuit is combinational, the result will be available after a short propagation delay. The 74LS181 can also act as a comparator. In order to use the comparator function, you must place the two operands in A and B as usual and select the subtract function.. The output pin A=B will be asserted if they are equal.

In order to use the ALU most efficiently in our system, we should attach it to our main bus. To do this we will have to make use of some registers, for our bus can only hold one 8-bit number at a time and we have two operands and a result. We are going to store one of the operands and a result in a register. Furthermore, the result, like everything attached to the bus, will have to be tri-stated to prevent bus contention. A suggestion of a block diagram is included in Figure 1.

The operation will be done by loading operand A into the Input Register. Operand B is placed on the bus and then the Result Register is loaded. Finally the result is placed on the bus by enabling the tri-state.



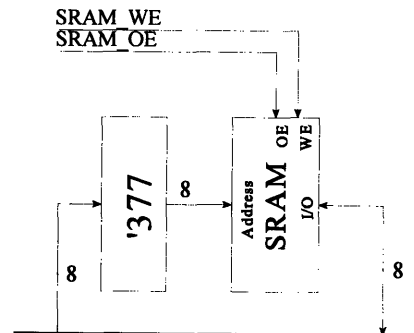
**Figure 3:** ALU Suggested Setup

### Static Random Access Memory (SRAM):

SRAM is a device used to store large amounts of information. It is a little more difficult to use than a register file, but its storage potential is much greater. SRAM can be found in ranges from hundreds of bytes to megabytes, and in speeds from 200ns to 12ns. The various memory locations are addressable via an address bus. In addition, input and output both use the same data bus.

There are three important signals that must be used to access the read and write functions of the SRAM: Chip Select (CS), Output Enable (OE), and Write Enable (WE). CS must be asserted on the SRAM for any operation to be successful. For our purposes, CS can always be asserted. If we want to read from the SRAM, we place the address that we want to read on the address bus and assert OE; WE is de-asserted the entire operation. The value stored at the specified location will be placed on the data bus. For a write operation, the address to be written to is placed on the address bus, and the data to be written is placed on the data bus; then WE is asserted. When WE is de-asserted, the data is stored in the SRAM. It is necessary that OE is de-asserted during a write operation. For more detailed timing information, see the specification sheet on the SRAM you are using.

In order to make the most efficient use of the SRAM we are going to use a register, which is attached to the bus, to hold the address. This way, by using an immediate value from the MCU, we can load the address that we want to access in one cycle and then read or write the actual location on the next cycle. Keep in mind that with this configuration we are **always** loading the address register. Thus, an SRAM read or write must be done in the instruction immediately following the one which placed the address on the bus.

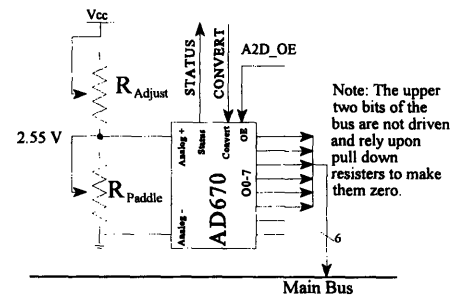


## Lab Assignments

1. **ALU construction:** Build the ALU, as described earlier. Attach the MCU to the ALU controls and do some tests to make sure that the ALU works properly. There are many inputs to control on the '181 ALU, including four selection bits(S0-S3), one carry in(Cn), and a Logic/Arithmetic Selector(M). This is going to take up a large number of assert signals in our MCU, especially considering that there are only a few functions we need the ALU to perform. The necessary functions are incrementing a number(A +1), adding two numbers(A+B) and comparing two numbers, for which we need to use the subtract function(A-B). What you should do when you are implementing the ALU is to have the MCU only provide two bits, which you decode to the proper function that you want to execute (eg. 00 = A+1, 01 = A+B, 10 = A-B (Compare), 11 = any other function you might need).

2. **User Input:** For this part of the lab assignment you are to build the paddle controls for NetPong. You will also need to write new microcode for the MCU that you built in Lab 4 to control this unit. Furthermore, because the MCU will be controlling many things, it would be advisable if you wrote the paddle controls as a function. This way, after you debug this system, you can just copy the function into your larger program later on.

When this function is called, it should have the A/D make a current reading. Then you should use the ALU to compare this value to the value of the last reading from the A/D (which will be stored in the SRAM). If the two values differ, then you should compose a message saying that your paddle has moved and place it in the output message queue. Furthermore, you should update the old A/D value in the SRAM.



A schematic for the AD670 A/D converter is provided. The AD670 should be used in the 0-2.55V range. The  $R_{adjust}$  should be changed so that the voltage across the paddle is 2.55V when the paddle is at the maximum resistance. Furthermore, we take the top six bits of the AD670 and pad the top with two 0's. This provides us with numbers from 0 to 63, which is the size of our playing field and consequently the range we need.

3. **Velocity Counters:** Two simple but important units that you will need to construct are the velocity counters. These devices keep track of the time that has elapsed since the ball has moved in the X and the Y positions. These devices will need to be seven bit counters. You will load these counters from the bus, and when the counter reaches zero, the Ripple Carry Out(RCO) should go high and stay high until the next value is loaded. Furthermore, the RCO of the X and Y counters should be status bits for the MCU, so that you can tell when it is time for the ball to be moved.

The only thing left to specify is the speed at which the counters will count. In order to keep the game interesting, the slowest that we could conceive of moving the ball across the screen would be eight seconds. Therefore, if we assign the largest velocity number (7Fh) to mean eight seconds, then we can calculate the speed at which our counters should count:

$$8 \text{ sec/board} / 64 \text{ spaces/board} / 128 \text{ counts/space} = 0.000976 \text{ sec or } 1024 \text{ Hz.}$$

Due to some constraints in the PC computer, the standard rate that it will use for its hardware counter is 1165Hz. The computer's base frequency can be changed to a limited number of values. (See the software documentation for more information.) Assuming that we will go with the computer's default frequency, we will need to create a signal that will oscillate at this rate. Assuming that our system clock is 156.3 KHz, we should divide it by 134 to get a frequency of 1166 Hz. What we need to create now is a circuit that provides us with one pulse every 134 (86h) system clock pulses. Furthermore, we want the pulse to be the same width as a system clock pulse. A simple divider circuit can be made from an 8-bit loadable counter. We place the two's complement of the number we are dividing by, in our case 7Ah, in the counter load input. We take the RCO and attach it to the counter load signal. We then attach the system clock to the counter clock. We now have a counter that counts from 7Ah to FFh and then resets back to 7Ah again, raising RCO for one cycle. Make sure that you hook up ENP and ENT correctly to make the RCO pulse last for one system clock cycle. Note that this pulse is glitchy and should not be used as a clock to a counter!

We now have a 1166Hz pulse, which is glitchy, so we do NOT want to attach it to the clock input of our Velocity Counters. Instead, we are going to put the system clock into the clock input of the Velocity Counters and use the 1166 Hz pulse as the enable for these counters. (This is why the pulse wants to be one clock cycle wide). At this point we have a system that constantly counts down at the proper speed. You have to create some logic to tell the counter to stop counting when it reaches 00h, and to keep the Velocity Counter's RCO asserted, for it will be used as a Status bit for the MCU and needs to stay asserted until a new value is loaded

4. **Sound Unit (Optional):** This part of the lab assignment has a suggestion for building a sound unit which will add sound effects to your project. The sound unit will be responsible for the 'ping' and 'pong' noises that were common in the original game of Pong. It is suggested that these sounds be played whenever a player receives or sends a CLAIM\_OWNERSHIP message. The Sound unit will be a simple device to interface to: we will provide a sound number, via the bus, and tell it to play. The sound unit will then play the sound without any more intervention by the MCU. The sound unit will then be quiet until the MCU tells it to start again. It also makes sense that if the MCU tells the sound unit to play a new sound in the middle of playing the previous sound that it stops playing the old one and starts the new one.

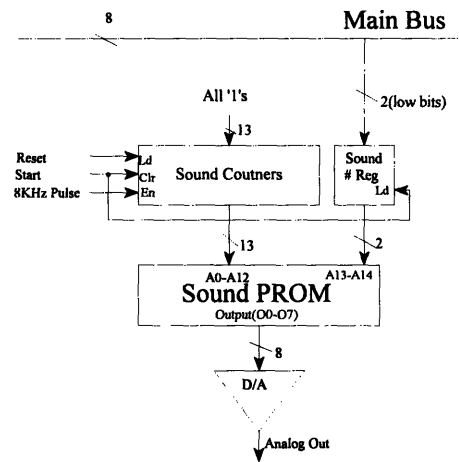
The sounds will be stored in a 32K PROM. Commonly, 8KHz is used for low quality sounds (similar in quality to AM radio). We will therefore use this sampling frequency. The selection for sounds (there will be four sounds, one for each player) will be the two most significant bits of the address to the 32K PROM. This leaves 13 address lines, or 8192 addresses

per sound effect. This will provide us with approximately 1 second of recording for each sound effect, which should be enough for our purposes.

The next important thing that we have to consider is that we must have a counter which will go through all 8192 addresses at a rate of 8KHz. Our system clock has a rate of 156.3KHz, therefore if we divide this by 19 we get a rate of 8.224KHz. This will be close enough for this application. The design of this counter should be done in a similar way to that done for the Velocity Registers.

After the counters are properly built, we will have a system that constantly outputs whatever sound is specified by the sound # register. At this point you have to create some logic to tell the counter to stop counting when it reaches all '1's. Furthermore, you should set up the counters so that a global reset signal initializes the counter to all '1's, thereby preventing it from playing a sound at startup.

The values in the PROM that actually make the sound can come from a number of places. For simple sound effects, you can write a simple program which will generate certain tones (i.e. sine waves). For more advanced sound effects, you can use a sound card to record sound effects and then use a utility to turn the data into raw magnitude number form. The specifics on how to do this are not appropriate for this lab manual.



**Figure 4: Sound Unit**



# NetPong Prelab #6

---

## Assignment of Player Numbers

An interesting thing to consider is how we are going to assign player numbers. The player numbers, as we have already seen, should range from zero to three, and represent which part of the screen the players paddle will defend. There are many ways that player numbers can be assigned. We could just hardwire each kit with a player number. However, we have decided that each kit should be identical and should work together when we attach them to the hub. This would imply that hardwiring player numbers would be a mistake.

Another option would be to assign two switches that the MCU could read to determine the player numbers. This would have the advantage that all of the kits would be identical and could be assigned any valid number. The only thing that would be necessary is to make sure that all of the player numbers are unique in any particular game.

⇒ The switch based method would work well but, in today's day and age, manually configured systems are being replaced by ones that automatically configure. Propose a system where player numbers were automatically and uniquely assigned to the kits on startup. You can assume that a RESET message will be sent by a kit or the computer on startup. Would the system you proposed be easy to implement, or should we stick with the switched based system?

## Message Queue Implementation:

The output message queue is where the kit will store the outgoing messages while waiting to get the Network Token. When the kit receives the Network Token, the messages stored in the output message queue are then sent out to the network, one byte at a time, until the queue is empty. As we come to the end of the lab sequence, it is important that we look into the proper design of a message queue.

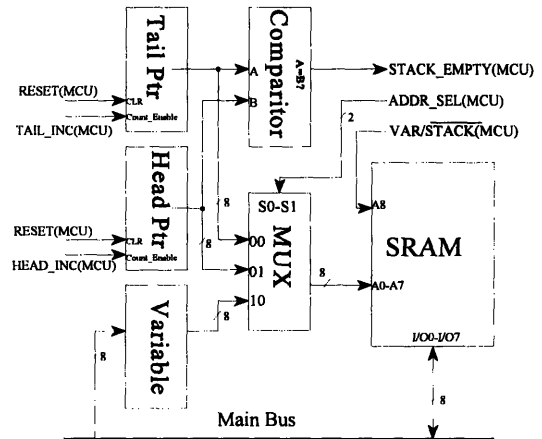
As with all queues, the network queue will have a head and a tail. As a new message is added to the queue, the bytes are added to the head of the queue. When the messages are sent out, they are taken from the tail of the queue. Keep in mind that this means that you want to place the bytes of the message to be sent out in the order that you want them sent out. You should place the Sync Byte first, then the player number and id, and finally the data field. If you don't keep this format, the other kits will not be able to correctly receive the messages.

Up until now, the SRAM you are using should be set up to use the lower 256 bytes. All of the address lines above 0-7 should be tied to ground. For the message queue, we are going to use another block of 256. This will be done by tying address line 8 to an output of the MCU. This way the MCU can switch between the message queue address space and the variable storage space. A stack of 256 bytes will allow you to store 85 messages if you are compressing

id and player number fields and 64 if you are not. This will be more than enough to store messages until you get the Network Token, even in the worst possible case.

As with most designs, there are options that we must consider. The simplest implementation would be to store the head and tail pointers in the SRAM variable and load them into the SRAM address register when needed. Furthermore, we can use the ALU to increment the head and tail pointers when necessary and compare them to see if the stack is empty.

An alternate implementation strategy would be to create separate hardware registers (counters) for the tail and head pointers. The MCU would have controls to increment the tail and head counters separately. There would also have to be a MUX at the input of the SRAM to choose between the head, tail, and variable address registers. Furthermore, we would have a comparator to test if the head and tail pointers are equal (i.e. stack empty).



☞ The issue that is presented here is a common one. The software approach of storing the head and tail pointers in the SRAM as variables is nice since you do not have to add any additional hardware; however it is slower. Often, the implementation of special purpose hardware can significantly increase the speed of a task, but at the expense of added complexity. For the two schemes, calculate the number of instructions that it will take to output the byte referenced by the tail pointer to the UART, increment the tail pointer and test to see if the stack is then empty. You can assume that the UART has asserted TBMT, signifying that the transmit buffer is empty. Don't forget that this operation will be performed for every byte sent out and is therefore going to have a significant impact on the network speed. (Actually, with intelligent microcoding, we need only test to see if the stack is empty once a message.) Which system should we implement?

# NetPong Lab #6

---

## Object:

Create Stack in SRAM  
Write Game Microcode

## Topics:

Message Queue Implementation  
Summary of Messages and Necessary Processing  
Memory Storage Suggestions

## Prelab Discussion:

### Assignment of Player Numbers

In order to implement a self-configuring system we will need to implement a new message type which we shall call INIT\_PLAYER. The four rules that we have to follow for this implementation are:

- (1) When the player gets reset, in addition to the standard setup that they have to do, they place a RESET message followed by an INIT\_PLAYER message in their output queue. In addition, they set their Player\_Number to 00h, set Ball\_Ownership to 01h, set Pending\_Ball\_Ownership to 00h, and a variable, called Player\_Setup, is set to 00h (meaning that the player number is already setup). After this is completed, the rest of the processing for the reset is done. After the RESET and INIT\_PLAYER messages are sent out, this kit will be the owner of the ball. Therefore, BALL\_X/Y\_MOVE and UPDATE\_X/Y\_VELOCITY messages should be placed in the output message queue to start the ball.
- (2) When a player receives a RESET message, in addition to the standard setup that they have to do, they place an INIT\_PLAYER message in their output queue. In addition, they set Player\_Number to 00h, set Ball\_Ownership to 01h, set Pending\_Ball\_Ownership to 00h, and Player\_Setup to 01h (meaning that the player is **not** setup).
- (3) Whenever the kit receives an INIT\_PLAYER message, it checks the Player\_Setup variable to see if it is still 01h. If it is, the kit increments its Player\_Number and sets Ball\_Ownership to zero.
- (4) Finally, when the player gets the Network Token, it sets Player\_Setup to 00h, thereby locking the Player\_Number.

An example of the player setup scheme follows. Initially, let us arbitrarily label the kits A through D just to keep straight which is which.

*Kit B has Network Token: (A=?, B=?, C=?, D=?)*

The reset button on Kit C is pressed. Kit C performs an internal setup, including setting `Player_Setup` to 00h, `Player_Number` to 00h, `Ball_Ownership` to 01h, and `Pending_Ball_Ownership` to 00h.

*Kit C has Network Token: (A=?, B=?, C=0, D=?)*

Kit C empties its message queue, containing the `RESET`, `INIT_PLAYER`, `BALL_X_MOVE`, `BALL_Y_MOVE`, `VELOCITY_X_UPDATE`, and `VELOCITY_Y_UPDATE` messages. Kits A, B, and D, upon receiving the `RESET`, should perform an internal setup, place `INIT_PLAYER` messages in the output queue, set `Player_Number` to 00h, and set `Player_Setup` to 01h, `Ball_Ownership` to 01h, and `Pending_Ball_Ownership` to 00h. Upon receipt of the `INIT_PLAYER` message, kits A, B, and D will check their `Player_Setup` variable, notice that it is still 01h, and therefore increment their `Player_Number` and set `Ball_Ownership` to 00h. The next four messages will be received by kits A, B, and D and should setup the initial ball position and velocity.

*Kit D has Network Token (A=1, B=1, C=0, D=1):*

Kit D sets `Player_Setup` to 00h, locking its `Player_Number` at 01h. Kit D empties its output message queue, sending the `INIT_PLAYER` message stored there. Kits A and B upon receipt of `INIT_PLAYER` increment their `Player_Numbers`, and set `Ball_Ownership` to 00h (it should already be 00h). Kit C has already set `Player_Setup` to 00h, and therefore ignores this message.

*Kit A had Network Token (A=2, B=2, C=0, D=1):*

Kit A sets `Player_Setup` to 00h, locking its `Player_Number` at 02h. Kit A empties its output message queue, sending the `INIT_PLAYER` message stored there. Kit B, upon receipt of the `INIT_PLAYER` message, increments its `Player_Number` and sets `Ball_Ownership` to zero (it should already be zero).

*Kit B had Network Token (A=2, B=3, C=0, D=1):*

Kit B sets `Player_Setup` to 00h, locking its `Player_Number` at 03h. Kit B empties its output message queue, sending the `INIT_PLAYER` message stored there. All kits have their `Player_Setup` set to 00h, so this message gets ignored.

At this point, all of the kits have their player numbers uniquely assigned and player zero owns the ball. The effect of this number assigning scheme is that the person who sends the `RESET` message will be assigned player number zero. There can be many variations of this setup scheme, but it is important that the class agrees upon one, otherwise they will not work together properly. This particular scheme allows anyone on the network to issue a `RESET` and have it work properly, not just a kit. This might be desirable if we want the attached computer to be able to reset the game.

### Message Queue Implementation:

With well optimized code, we should be able to cut down the number of instructions for the load/inc/compare to two for the hardware implementation, and six for the software implementation. A comparison between the two systems would look like:

#### **Hardware Pointers**

##### *Advantages:*

Faster  
Simpler Microcode

#### **Software Pointers**

##### *Advantages:*

No Hardware changes

The choice that you make for this sub-system does not effect the way that your kit outputs packets, and therefore you can safely choose either one of these and still maintain compatibility. Although the hardware solution accomplished the same task in one third the time of the software solution, the difference in speed in this situation is probably not very important. You should implement the system that you believe is the better choice.

### **Design Issues:**

#### Token Processing Responsibilities:

When you receive the token, you have a number of things that you must accomplish in short order. First, you want to check the Pending\_Ball\_Ownership variable. If it is 01h, then set Ball\_Ownership to 01h and set Pending\_Ball\_Ownership to 00h and load both X and Y counters with the current velocities. Also, if Player\_Setup is 01h, then set it to 00h, locking your Player\_Number. This can be done more efficiently by always setting Player\_Setup to 00h, thus eliminating the check.

The next important thing that needs to be done is to send the messages in your output message queue. The Network Unit is very convenient since it takes care of receiving and packaging all of the incoming messages from the UART. However, the MCU must take responsibility for outputting messages to the UART. The information on interfacing to the UART for sending bytes is described in Lab 3. You should keep sending out bytes from the message queue until it is empty.

After the message queue is empty, you should check the paddle to make sure that its position hasn't moved. If it has, then update the position and place the appropriate PADDLE\_X/Y\_MOVE message in the output message queue. If you own the ball, then check both the X and Y counters' RCO to see if they have expired (reached 00h). If either has, send out the appropriate BALL\_?\_MOVE message(s) if necessary and reload the counter(s).

You should now be finished processing but, before you return the Network Token, make sure that the UART is finished sending out the last byte by checking TBMT and EOC. When these signals are asserted, return the Network Token by asserting TOKEN\_RETURN.

### Summary of Messages and Necessary Processing:

The main part of the MCU's responsibilities is to respond to various messages that come in from the Network Unit. When the Network Unit receives a complete message, it raises MSG\_AVAILABLE signifying that a new message is available. The MCU has to process the message before the next byte is overrun in the UART. Provided that your clock is 156.3 KHz, this gives you approximately 256 instructions to work with. If you can not process the message in this amount of time, then you should copy the message out of the Network Unit's register file and into the SRAM and assert MCU\_ACK\_MSG. This will provide an additional 256 instructions (128 if you are using 3-byte messages) before the next message arrives, which you can use for processing. You must first determine the message type (through a series of comparisons) and then you can process it properly. When processing is done, the MCU should assert MCU\_ACK\_MSG (assuming that you haven't already) to tell the Network Unit that you are finished with the current message. It is important to allow the Network Unit to get back to its responsibilities as fast as possible, otherwise messages can be lost.

Following is a list of all of the standard message types, including their ID number and the data they contain. For each message, there is a summary of what the message means and information on how to process it.

#### **PADDLE\_X\_MOVE (ID#1, Data=X position)**

**Summary:** This message is used by the video display (in the computer) to track the location of the paddles. Any time a kit notices that a paddle has moved in the X direction, it should send this message. The data field contains the absolute X position of the lower end (closer to zero) of the paddle; for example, if the data field is 10h, then the paddle stretches from 10h to 15h. Note that in normal operation, players 1 and 3 will only issue this message once (to place their paddles at 00h and 3Fh respectively) because their paddles should move only in the Y direction.

**Processing:** Unless you are generating your own video, you can safely ignore these messages.

#### **PADDLE\_Y\_MOVE (ID#2, Data=Y position)**

**Summary:** This message is the counterpart to PADDLE\_X\_MOVE. Any time a kit notices that a paddle has moved in the Y direction, it should send this message. The data field contains the absolute Y position of the lower end (closer to zero) of the paddle; for example, if the data field is 21h, then the paddle stretches from 21h to 26h. Note that in normal operation, players 0 and 2 will only issue this message once (to place their paddles at 3Fh and 00h respectively), because their paddles should move only in the X direction.

**Processing:** Unless you are generating your own video, you can safely ignore these messages.

### **BALL\_X\_MOVE (ID#3, Data=X position)**

**Summary:** This message is issued by the owner of the ball to indicate that the ball has moved in the X direction. The absolute X coordinate of the ball is given in the data field.

**Processing:** First, the kit should check to see if the ball has reached its side of the playing field (this is only a concern to players 1 and 3, as the ball has only moved in the X position). If the ball has reached your side of the screen, you should issue a CLAIM\_OWNERSHIP message. You should determine if the ball has hit your paddle or missed it. If the paddle was missed, place the ball in the center of the screen and change the velocity. If the ball hit your paddle, you should change the velocity to make the ball 'bounce' back into play.

To keep the game interesting, when the ball hits your paddle, you should do more than just invert the velocity of the ball, as that will result in a very boring game. One suggestion, is to divide the paddle into sections and have each section act differently. The paddle is six units long, so it easily divided into three sections. If the ball hit the middle section, the X velocity is inverted (just invert the most significant bit). If it hits the top section, the X velocity is inverted, and divided by two. The Y velocity is multiplied by two. If it hits the bottom section, the X velocity is inverted and multiplied by two, and the Y velocity was divided by two. This method keeps the ball motion interesting and somewhat unpredictable. Feel free to use whatever system you like.

### **BALL\_Y\_MOVE (ID#4, Data=Y position)**

**Summary:** This message is issued by the owner of the ball to indicate that the ball has moved in the Y direction. The message is the counterpart of the BALL\_X\_MOVE. The absolute Y coordinate of the ball is given in the data field.

**Processing:** This message should be processed similarly to the BALL\_X\_MOVE, except all of the information should be changed with respect to the Y coordinate.

### **VELOCITY\_X\_UPDATE (ID#5, Data=X Velocity)**

**Summary:** This message is issued by the owner of the ball to indicate that the ball has changed velocity. This message is sent whenever the ball reaches the edge of the screen and has to 'bounce' off a paddle. The data field contains the new X velocity value and direction. The most significant bit is the sign (direction) bit and the rest is magnitude. For eight data-bits, the format would be SMMMMMMM, for seven it would be XSMMMMMM and for six it will be XXSMMMMM, where 'X' is invalid, 'S' is sign, and 'M' is magnitude.

**Processing:** The only action that has to be taken when a VELOCITY\_X\_UPDATE occurs is to replace the old X velocity value in the SRAM with the new value from the data field.

### **VELOCITY\_Y\_UPDATE (ID#6, Data=Y Velocity)**

**Summary:** This message is issued by the owner of the ball to indicate that the ball has changed velocity. This message is the counterpart to the VELOCITY\_X\_UPDATE. This message is sent whenever the ball reaches the edge of the screen and has to 'bounce' off a paddle. The data field contains the new Y velocity value.

**Processing:** The only action that has to be taken when a VELOCITY\_X\_UPDATE occurs is to replace the old Y velocity value in the SRAM with the new value from the data field.

### **CLAIM\_OWNERSHIP (ID#7, No Data)**

**Summary:** This message is issued by a kit when the ball reaches its side of the screen. It signifies that the kit wants to take over control of the ball.

**Processing:** In order to properly maintain ownership we follow the following rules:

- (1) When the ball hits our side of the screen, we set the Pending\_Ball\_Ownership bit to '1', meaning we are going to own the ball. We also put a CLAIM\_OWNERSHIP message in our output queue.
- (2) When we get the Network Token, we empty our message queue, including our CLAIM\_OWNERSHIP message. If Pending\_Ball\_Ownership is '1' set Ball\_Ownership to '1'. Set Pending\_Ball\_Ownership to '0'.
- (3) When the previous ball owner (and the other kits) receives our Claim Ownership message, they set their ownership bit to '0' meaning they no longer own the ball. They do **not** clear the Pending\_Ball\_Ownership bit!

See Lab 5 for more details on processing this message.

### **INIT\_PLAYER (ID#8, No Data)**

**Summary:** This message is only used at the start of the game to assign player numbers. The start of the game is considered to be any time the Reset button is pressed or a RESET message is received.

**Processing:** Whenever the kit receives an INIT\_PLAYER message, it checks the PLAYER\_SETUP variable to see if it is still 01h. If it is, then the kit increments its Player\_Number, and sets Ball\_Ownership to zero. If PLAYER\_SETUP is 00h, then ignore this message. See the Prelab Discussion of this lab for more information on this message.

### **RESET ID#9 (No Data)**

**Summary:** This message is sent out by a kit to signify that a Reset has taken place. Each kit should have a Reset button that, as the name suggests, resets all of the electronics to the



initial start state. In addition, when the Reset button is pressed, the kit sends a RESET message to the other kits so that the whole game will start from the beginning.

**Processing:** Upon receipt of a RESET message, you will have to perform a number of tasks to reset your kit. First, you want to empty your stack (without sending them to the UART). This can be done by setting both the head and tail pointers to 00h (or any other number, as long as they are the same). Furthermore, you should set the Ball\_Ownership to 01h (signifying that you own the ball), Pending\_Ball\_Ownership to 00h, and PLAYER\_SETUP to 01h (signifying that Player\_Number has not yet been set). Player\_Number should also be set to zero. The ball X and Y position should be set to the center of the screen (1Fh, 1Fh), and the velocity should be set to some initial value (preferably slow).

Furthermore, when you next get a token, you lock your Player\_Number (as discussed in the INIT\_PLAYER information in the Prelab Discussion). You should also set your paddle to the middle of your side of the screen (as determined by your new player number) and send both a PADDLE\_X\_MOVE and PADDLE\_Y\_MOVE message.

Memory Storage Suggestions:

Your SRAM is divided into two logical 256 byte sections: the variable storage section and the stack section. The table provided makes suggestions for the placing of variables within the SRAM variable storage space. It is not necessary to follow this suggestion; it is solely provided to try facilitate debugging between kits (for Professors and TAs especially) and so that you do not forget any important variables. With that said, here is the suggested layout:\

00h	Temp Storage
01h	Temp Storage
02h	Ball X Position
03h	Ball Y Position
04h	Ball X Velocity
05h	Ball Y Velocity
06h	Paddle X Position
07h	Paddle Y Position
08h	Player Number
09h	Player_Setup
0Ah	Ball_Ownership

0Bh	Pending_Ball_Ownership
0Ch	Unused
0Dh	Stack Head (If software stack is used)
0Eh	Stack Tail (If software stack is used)
0Fh	Unused

Network Message Format:

The network messages have the following format. When you send and receive messages, the lower byte should be transmitted first. Make sure that you all agree to use compression or not, as the two types are not compatible.

<u>Compression</u>		<u>Without Compression</u>	
1	Sync Byte	1	Sync Byte
2	Player/ID byte	2	Player Byte
3	Data Byte	3	ID Byte
		4	Data Byte

**Lab Assignment:**

**1. Implementation of Message Queue:** Implement and test the message queue that you have decided to use (as discussed in the Prelab Discussion). Write a test program for your MCU to insure that your stack works correctly. You may want to pay special attention to when the stack rolls around from FFh back to 00h to insure that data is not lost.

**2. Write NetPong Game Microcode:** It has come time to wrap the lab up and write the microcode to tie together all of the systems that you have designed so far. The summary of how to process all of the messages was provided in the Discussion section of this lab and should be used to help your microcoding.

This is a big project and you should approach it in pieces. Instead of writing all of the microcode and debugging it at once, write pieces and debug each part one at a time. For example, a good starting point might be to echo the messages that you receive to test receive and send function. When you receive messages from the Network Unit, place them into the output message queue; then when you receive the Network Token, send them back out. You can have the NetPong software running a game with one computer player to generate messages and use the Message Logger feature to make sure that you are echoing the messages properly.

Continue to build upon the system in pieces, fully debugging each piece before adding another one. This will make your time writing code easier and debugging a whole lot faster.



# NetPong Suggested Enhancements

---

After completing the NetPong lab sequence, you should have a very good understanding of how the system works. If you are interested in further work on the NetPong system here are a few suggestions that you could implement.

1. Ball Hold Feature: This feature implements the ability to 'catch' the ball and hold onto it while you move the paddle. This project will require modification to the paddle input unit, namely a Ball Hold button will have to be installed. Furthermore, the microcode will have to be modified to support this feature.
2. Video Unit: A more ambitious project is to implement a custom video unit. This unit would be responsible for displaying the paddles and ball, like the computer does now. Furthermore, it would be a convenient feature to twist the board so that the player that the video unit is associated with is always at the bottom of the screen. This is a fairly intensive modification and should be carefully thought out before modification begins.
3. Alternate Input Sources: This suggestion is based upon a project that a couple of MIT students built in MIT's 6.111 Digital Design Course. These students used two flashlights in front of a CCD camera as paddle controls to a Pong game. The basic idea is to look for the brightest spot in the incoming video and interpret it as the paddle position. You may also want to experiment with other sources of input.

Remember to carefully design the modifications to your circuits before actually implementing them. This will save you a significant amount of time.



## 7. Conclusion

In conclusion, the course materials that were created satisfy all of the proposed specifications. The lab sequence and the supporting software and documentation should satisfy the needs of professors that want to create a digital design class. After the students go through the lab sequence they will have a good background of digital design and integration. Furthermore, because many of the design issues are related to the simple network that they will be building, they will end up with an understanding of some of the basic issues in networking. Regrettably, due to time and budget constraints, this lab sequence has not been fully tested on students. In order to compensate for this, much effort was spent having people with the same background as a student proofread and comment on the labs. As a result, the labs should be easily comprehensible by the students.

## 8. Bibliography

Dettmann and Johnson. DOS Programmer's Reference, 3rd Edition. (Que Corporation, 1992).

Hans-Peter Messmer. The Indispensable PC Hardware Book. (Addison-Wesley, Massachusetts, 1995).

LaMothe, Ratcliff, Seminatore & Tyler, Tricks of the Game Programming Gurus (SAMS Publishing , 1994) p483-487.

Texas Instruments. The TTL DataBook. (Texas Instruments Press, 1994).

Troxel and Kirtley. "6.111 Course Notes"

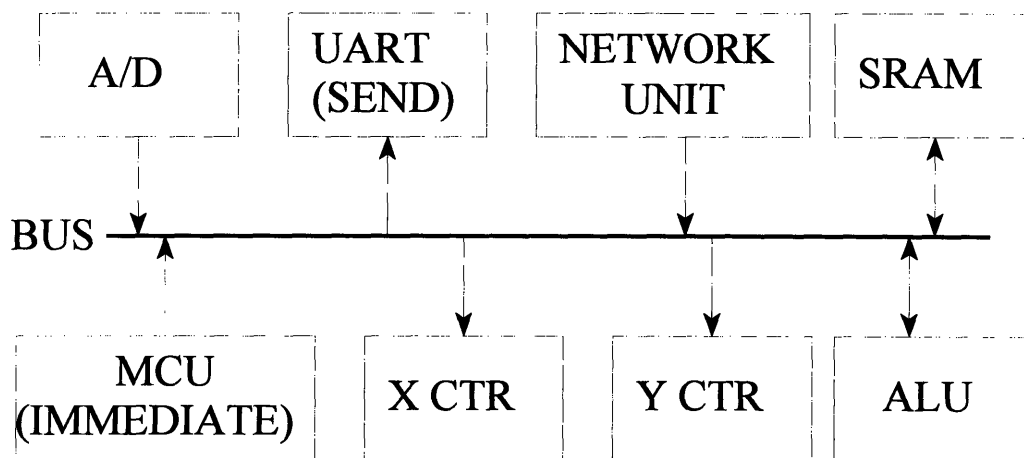


## 9. Appendix A: Example Solution

In a design course such as this, there are an almost infinite number of variations of this project that will still work. I felt that it is still important to provide one possible solution so that an instructor can have something to look at and evaluate.

The solutions themselves are provided in four forms: detailed block diagrams, PAL equation files, FSM descriptor files, and Microcode (with descriptor file). All four of these are necessary to have a complete working netpong.

### NetPong System Diagram



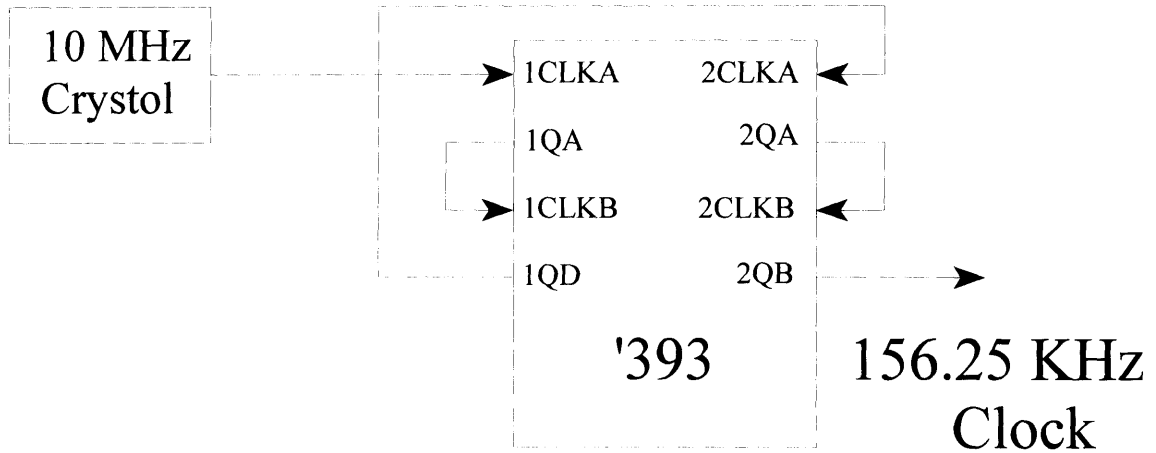
The NetPong system diagram is an overview of the entire system. All of the block are further broken down into individual block diagrams

Note: The designed system does not use player number/ID compression. In addition it uses 8 data bits and the Sync character used was 3Fh. This means that for only one kit the correct parameters to use are:

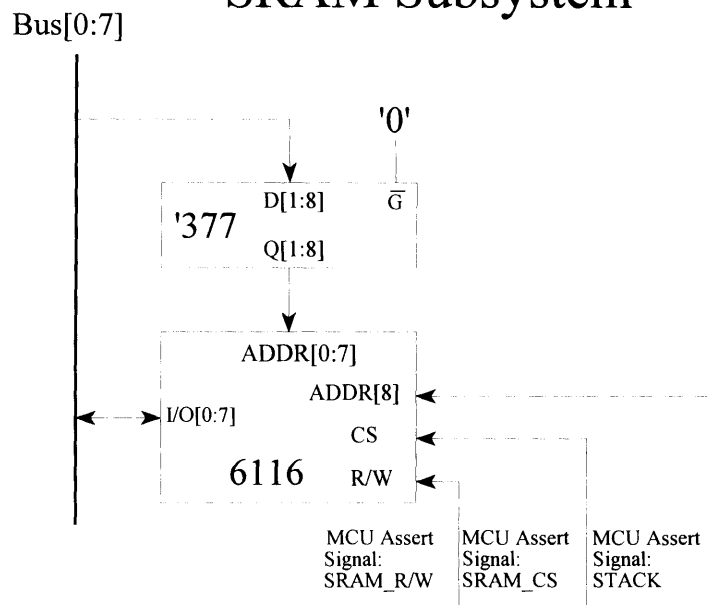
NETPONG /P3 /S /Y3F

The relies upon the defaults of 8 data bits and not player/ID compression.

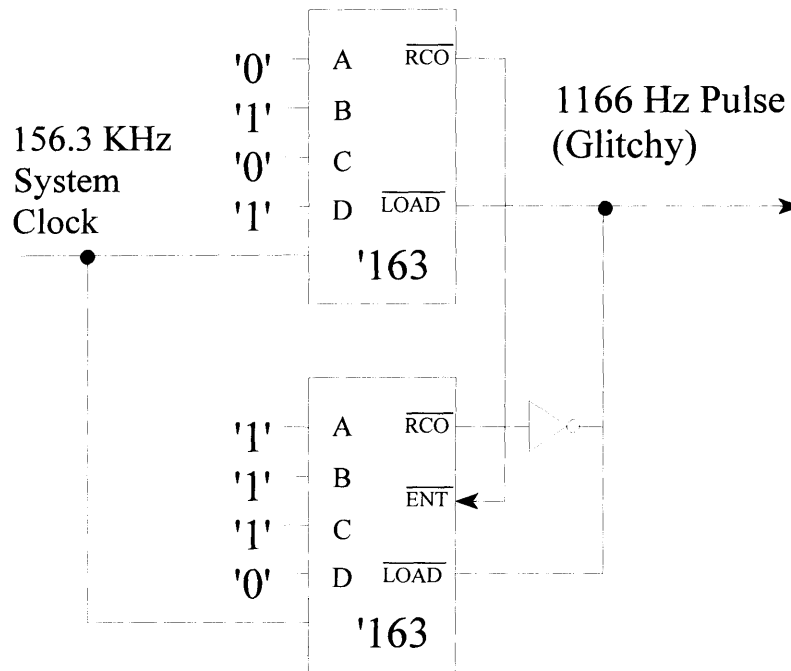
# System Clock Subsystem



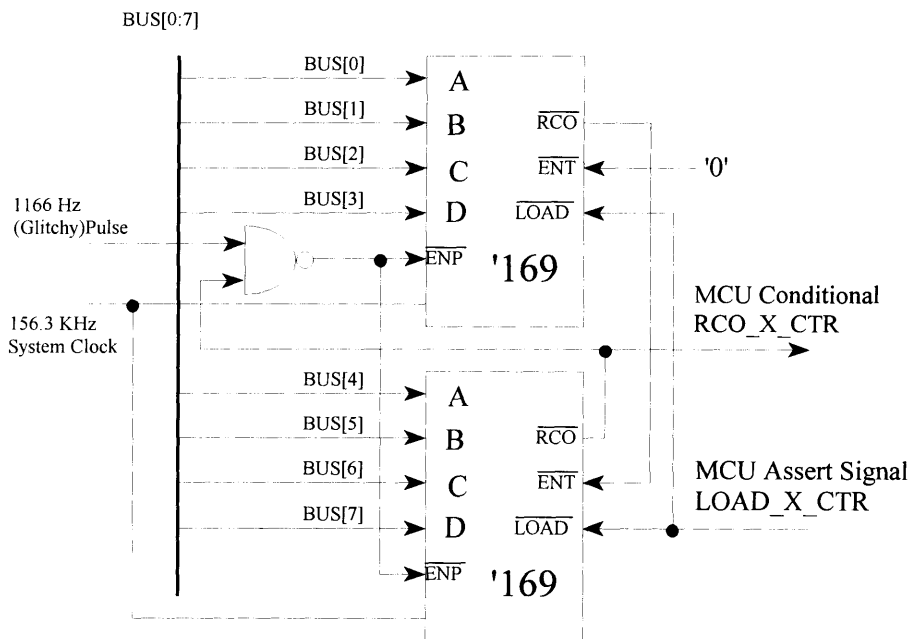
# SRAM Subsystem



# 1166 Hz Timing Pulse Generator

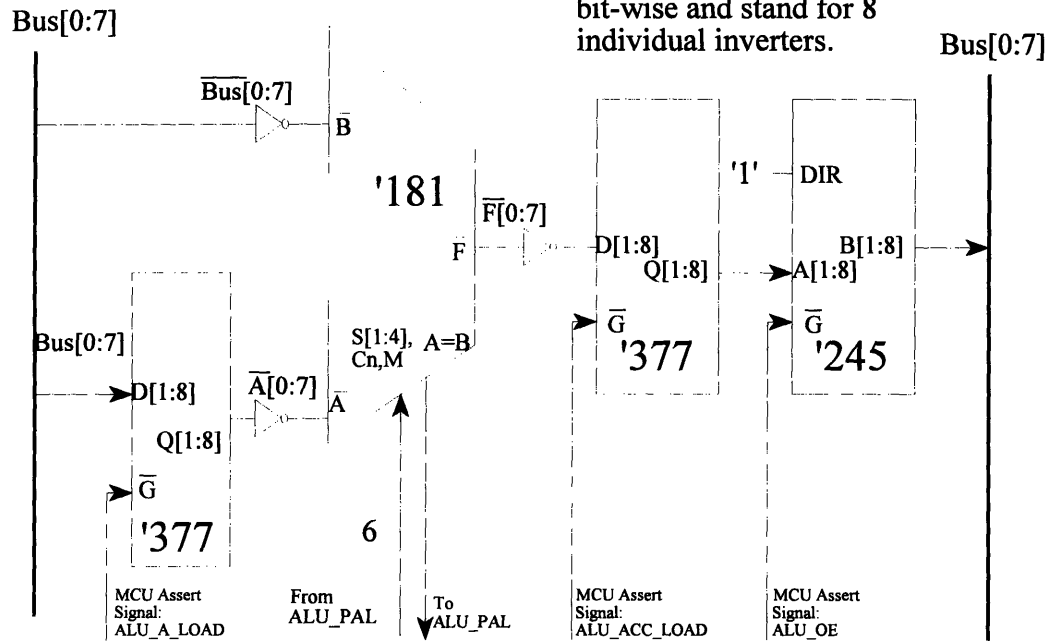


# X Counter (Y Counter)



# ALU Subsystem

Note: All inveter symbols are bit-wise and stand for 8 individual inverters.



## ALU\_PAL Equation File:

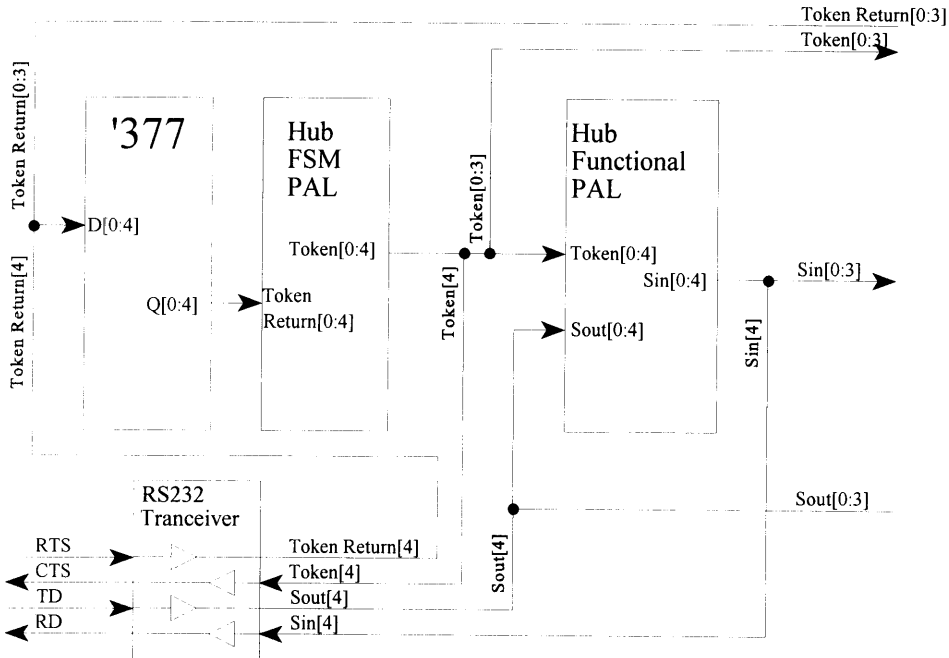
This PAL has two functions. First it takes three bits of ALU function from the MCU and converts them to the correct six signals (4 Select, 1 Function/Logic, and 1 Carry in). Furthermore, it stores the results of the last compare - resulting in a "sticky" alu\_equal signal that stays until the next compare instruction is executed.

```

device 20v8
positive S3@18,S2@17,S1@16,S0@15,M@19,C0@20;
positive A@2, B@3, C@4;
negative alu_acc_load@5;
positive alu_equal@6;
register mcu_alu_equal@21;

S3 = /B+C+A;
S2 = /B*C + /A*B;
S1 = A*/B + /A*C + B*/C;
S0 = /A*/B + B*C + A*B;
M = A;
C0 = /B*C + B*/C;
mcu_alu_equal = /alu_acc_load*/A*B*/C*alu_equal +
(alu_acc_load+A+/B+C)*mcu_alu_equal;
    
```

# Network Hub



## EQN file for Hub Functional PAL:

```
device 20v8
positive Token_0@2, Token_1@4, Token_2@6, Token_3@8, Token_4@10;
positive Sout0@3, Sout1@5, Sout2@7, Sout3@9, Sout4@11;
positive Broadcast@21;
positive Sin0@20, Sin1@19, Sin2@18, Sin3@17, Sin4@16;
```

```
Broadcast =      Token_0*Sout0 +
                Token_1*Sout1 +
                Token_2*Sout2 +
                Token_3*Sout3 +
                Token_4*Sout4;
```

```
Sin0 = Broadcast + Token_0;
```

```
Sin1 = Broadcast + Token_1;
```

```
Sin2 = Broadcast + Token_2;
```

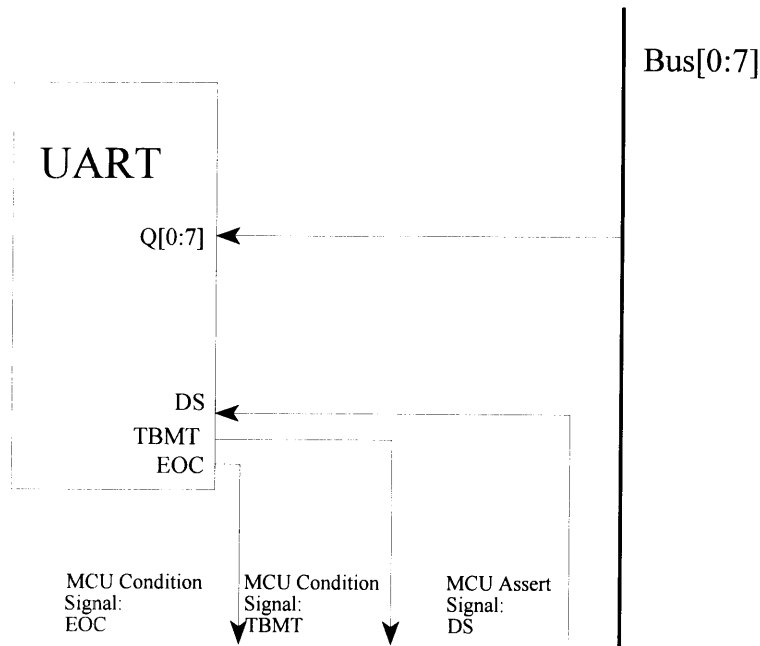
```
Sin3 = Broadcast + Token_3;
```

```
Sin4 = Broadcast + Token_4;
```

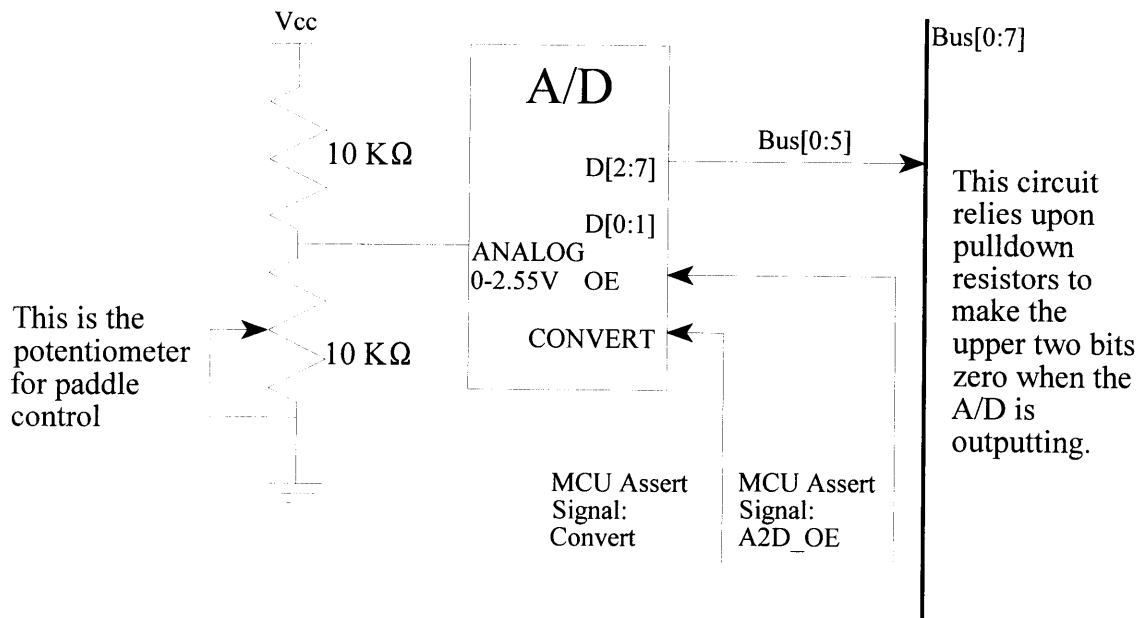
## FSM descriptor file for Hub FSM:

```
#0:
{
  Token_0 = 1;
  Token_1 = 0;
  Token_2 = 0;
  Token_3 = 0;
  Token_4 = 0;
  if Token_0_Return goto #1;
  if /Token_0_Return stay;
}
#1:
{
  Token_0 = 0;
  Token_1 = 1;
  Token_2 = 0;
  Token_3 = 0;
  Token_4 = 0;
  if Token_1_Return goto #2;
  if /Token_1_Return stay;
}
#2:
{
  Token_0 = 0;
  Token_1 = 0;
  Token_2 = 1;
  Token_3 = 0;
  Token_4 = 0;
  if Token_2_Return goto #3;
  if /Token_2_Return stay;
}
#3:
{
  Token_0 = 0;
  Token_1 = 0;
  Token_2 = 0;
  Token_3 = 1;
  Token_4 = 0;
  if Token_3_Return goto #4;
  if /Token_3_Return stay;
}
#4:
{
  Token_0 = 0;
  Token_1 = 0;
  Token_2 = 0;
  Token_3 = 0;
  Token_4 = 1;
  if Token_4_Return goto #0;
  if /Token_4_Return stay;
}
#5: goto #0;
#6: goto #0;
#7: goto #0;
```

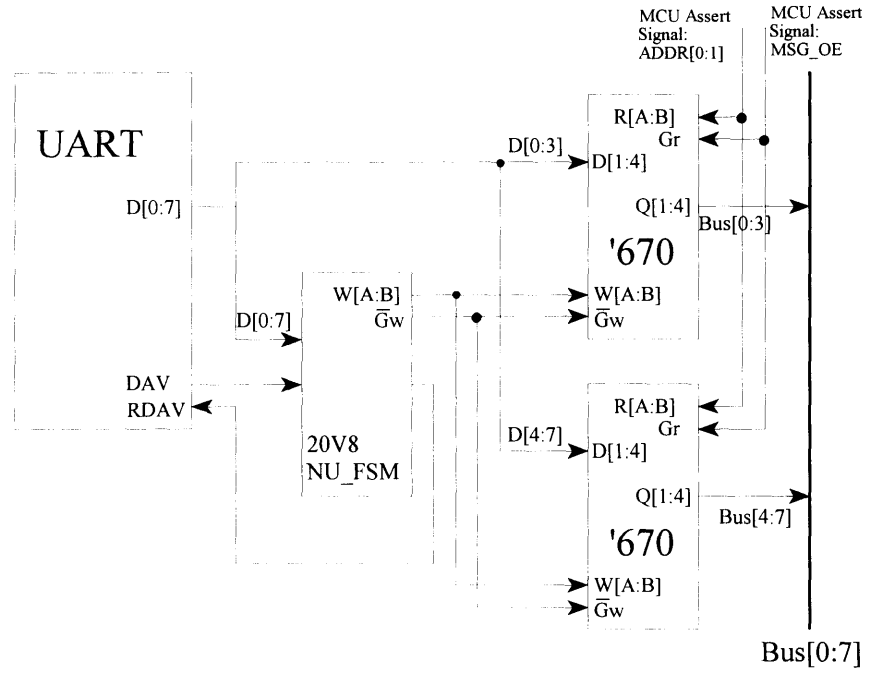
# UART (SEND) Subsystem



# A/D Subsystem



# Network Unit Subsystem





## Network Unit FSM File:

```
begin_p
device 22v10
negative WE@23;
negative RDAV@22;
positive ADDR0@21, ADDR1@20,
MSG_AVAIL@18;
register Q0@14;
register Q1@15;
register Q2@16;
register Q3@17;
positive DAV@2, MCU_ACK_MSG@3,
IN_SYNC@4;
end_p

FIND_SYNC:
{
  if DAV* IN_SYNC goto CLEAR_SYNC;
  if DAV*/IN_SYNC goto
CLEAR_WRONG_SYNC;
  stay;
}

CLEAR_WRONG_SYNC:
{
  RDAV = 1;
  goto FIND_SYNC;
}

CLEAR_SYNC:
{
  RDAV = 1;
  goto WAIT_1;
}

WAIT_1:
{
  ADDR0 = 0;
  ADDR1 = 0;
  WE = DAV;
  if DAV goto CLEAR_1;
  stay;
}

CLEAR_1:
{
  RDAV = 1;
  goto WAIT_2;
}

WAIT_2:
{
  ADDR0 = 1;
  ADDR1 = 0;
  WE = DAV;
  if DAV goto CLEAR_2;
  stay;
}

}

CLEAR_2:
{
  RDAV = 1;
  goto WAIT_3;
}

WAIT_3:
{
  ADDR0 = 0;
  ADDR1 = 1;
  WE = DAV;
  if DAV goto CLEAR_3;
  stay;
}

CLEAR_3:
{
  RDAV = 1;
  goto WAIT_FOR_MCU;
}

WAIT_FOR_MCU:
{
  MSG_AVAIL = 1;
  if MCU_ACK_MSG goto FIND_SYNC;
  stay;
}
```

## Network Unit EQN File: (Modified to include IN\_SYNC)

```
device 22v10
negative WE@23;
negative RDAV@22;
positive ADDR0@21, ADDR1@20, MSG_AVAIL@18;
register Q0@14;
register Q1@15;
register Q2@16;
register Q3@17;
positive DAV@2, MCU_ACK_MSG@3;
register IN_SYNC@19;
positive D0@11, D1@10, D2@9, D3@8, D4@7, D5@6, D6@5, D7@4;
```

```
register Q0;
register Q1;
register Q2;
register Q3;
```

```
Q0 =          /DAV * Q1 +
              /Q0 * Q1 +
              /DAV * Q2 +
              /Q0 * Q2 +
              /MCU_ACK_MSG * Q3 +
              /Q0 * Q3 +
              DAV * /IN_SYNC * /Q0;
```

```
Q1 =          /Q0 * Q1 +
              DAV * Q0 * /Q1 * Q2 +
              DAV * IN_SYNC * /Q0 * /Q2 * /Q3 +
              /DAV * Q1;
```

```
Q2 =          /Q0 * Q2 +
              /Q1 * Q2 +
              DAV * Q0 * Q1 * /Q2 +
              /DAV * Q2;
```

```
Q3 =          /MCU_ACK_MSG * Q3 +
              /Q0 * Q3 +
              DAV * Q0 * Q1 * Q2;
```

```
RDAV =        /Q0 * Q2 +
              /Q0 * Q3 +
              Q0 * /Q1 * /Q2 * /Q3 +
              /Q0 * Q1;
```

```
ADDR0 =       Q0 * /Q1 * Q2;
```

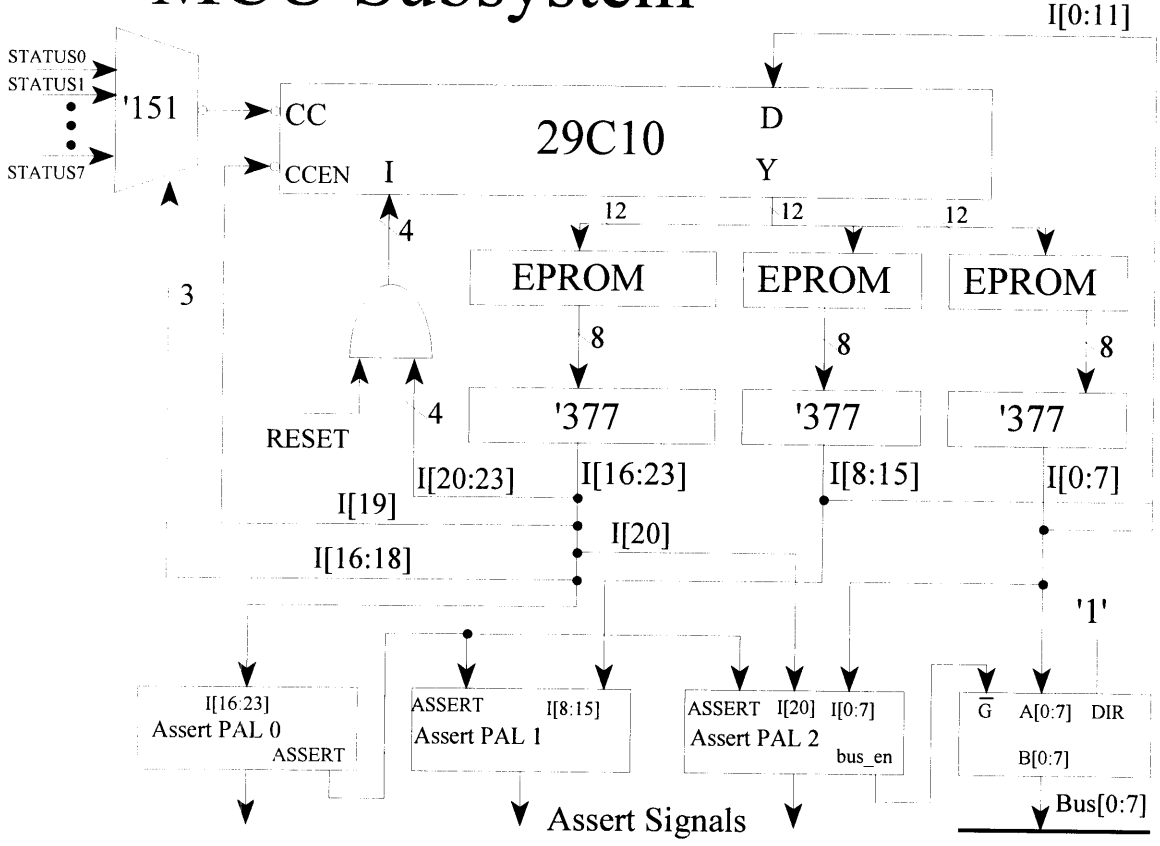
```
ADDR1 =       Q0 * Q1 * Q2;
```

```
WE =          DAV * Q0 * Q2 +
              DAV * Q0 * Q1;
```

```
MSG_AVAIL =   Q0 * Q3;
```

```
IN_SYNC =     /D7*/D6*D5*D4*D3*D2*D1*D0;
```

# MCU Subsystem



# MCU Assert PALS

## Assert 0 PAL File:

```
device 22v10
positive i0@5, i1@4, i2@3, i3@2;
positive assert@22, si0@6, si1@7, si2@8;
positive start_sound@19, stack@18;
negative sram_we@17;

assert          = /i0*i1*i2*i3;
start_sound    = assert*si0;
stack          = assert*si1;
sram_we        = assert*si2;
```

## Assert 1 PAL File:

```
device 22v10
positive assert@2;
positive si3@3, si4@4, si5@5, si6@6;
positive si7@7, si8@8, si9@9, si10@10;
positive sel0@22, sel1@21, sel2@20;
negative load_x_ctr@18, load_y_ctr@17;
negative alu_a_load@16, ds@15, alu_acc_load@19;

sel0           = si3 * assert;
sel1           = si4 * assert;
sel2           = si5 * assert;
alu_acc_load   = si6 * assert;
load_x_ctr     = si7 * assert;
load_y_ctr     = si8 * assert;
alu_a_load     = si9 * assert;
ds             = si10 * assert;
```

## Assert 2 PAL File:

```
device 22v10
positive assert@2, i5@11;
positive si11@3, si12@4, si13@5, si14@6;
positive si15@7, si16@8, si17@9, si18@10;
negative a2d_rw@22, a2d_cs@21;
positive mcu_ack_msg@20, token_return@19;
negative alu_oe@18, mcu_msg_oe@17, sram_oe@16;
positive unused@15;
negative bus_en@14;

a2d_rw         = si11 * assert * /i5;
a2d_cs         = si12 * assert * /i5;
mcu_ack_msg    = si13 * assert * /i5;
token_return   = si14 * assert * /i5;
alu_oe         = si15 * assert * /i5;
mcu_msg_oe     = si16 * assert * /i5;
sram_oe        = si17 * assert * /i5;
unused         = si18 * assert * /i5;
bus_en         = assert * i5;
```

```

/* mcu.sp for the MCU of NetPong Game      */
/* Last revised 4-20-96                    */

/*****
/* Instruction Word Organization:          */
/* conditional statements   iiiiuccc xxxxa aaa          */
/* immediate data statement  iiiissss ssssssss vvvvvvvv */
/* assertion statements     11100sss ssssssss ssssssss */
/* immediate assert to bus   11101sss ssssssss vvvvvvvv */
/*   where i = opcode selection          */
/*       u = unconditional bit           */
/*       c = status selection            */
/*       a = alternative address, i.e. jump address      */
/*       v = immediate values, i.e. binary data         */
/*       s = assertion signals           */
*****/

op <23:0>;          /* Indicates the available bits          */
address op <11:0>;  /* Indicates bit locations for addresses */
value op <7:0>;     /* Indicates bit locations for immediate val */

/*
 * Instruction Set of AM29C10A
 * All of the instructions available are defined.
 * In all likelihood, you will only use a subset.
 */

RESET      op<23:20>=%b0000; /* Jump Zero (RESET)          */
CCALL      op<23:20>=%b0001; /* Cond CALL subroutine       */
CJMP       op<23:20>=%b0011; /* Cond JuMP                  */
PUSH       op<23:20>=%b0100; /* PUSH stack down / cond load counter */
CJSRP      op<23:20>=%b0101; /* Cond Jump to Subroutine via Register Pipe */
CJRP       op<23:20>=%b0111; /* Cond Jump via Register Pipeline */
RFCT       op<23:20>=%b1000; /* Repeat loop with File when Counter != 0 */
RPCT       op<23:20>=%b1001; /* Repeat loop with Pipeline when Counter!=0 */
CRTN       op<23:20>=%b1010; /* Cond ReTURn                */
CJPP       op<23:20>=%b1011; /* Cond Jump Pipeline and Pop  */
LDCT       op<23:20>=%b1100; /* Load Counter               */
LOOP       op<23:20>=%b1101; /* test end-of-LOOP           */
ASSERT     op<23:20>=%b1110; /* continue, perform the ASSERTions */
TWB        op<23:20>=%b1111; /* Three-Way-Branch           */

/* Unconditional branch statements */

CALL       op<23:19>=%b00011; /* CALL subroutine          */
JMP        op<23:19>=%b00111; /* JuMP                    */
JSRP       op<23:19>=%b01011; /* Jump to Subroutine via Register Pipeline */
JRP        op<23:19>=%b01111; /* Jump via Register Pipeline */
RETURN     op<23:19>=%b10101; /* RETURN                  */
JPP        op<23:19>=%b10111; /* Jump Pipeline and Pop   */
*/

/* These are defined so that you may use them to make your code more
 * readable. Their use is not required. */

IF      nop;
THEN    nop;

```

```

TRUE  op<19>=1;          /* This makes sure /CCEN is deasserted */

/* Assertions */

convert      op<1:0>=%b11;    /* bit zero is CS & CE (inverted) */
a2d_oe      op<1:0>=%b01;    /* but one is R/W (inverted) */
mcu_ack_msg  op<2>=1;
token_return op<3>=1;
alu_oe      op<4>=1;
mcu_msg_oe  op<5>=1;
sram_oe     op<6>=1;
L7          op<7>=1;

/* These are for the Network Unit Register File interface (shared with ALU)*/
byte0       op<9:8> = 0;
byte1       op<9:8> = 1;
byte2       op<9:8> = 2;
byte3       op<9:8> = 3;

/* These control the ALU function (shared with Network Unit Interface) */
/* all except compare load the accumulator */
add         op<11:8> = %b1000;
inc         op<11:8> = %b1001;
sub         op<11:8> = %b1010;
a           op<11:8> = %b1011;
b           op<11:8> = %b1100;
and         op<11:8> = %b1101;
or          op<11:8> = %b1110;
xor         op<11:8> = %b1111;
compare     op<11:8> = %b0010; /* subtract w/o acc load */

alu_acc_load op<11>=1; /* should never need to be used directly */
load_x_ctr   op<12>=1;
load_y_ctr   op<13>=1;
alu_a_load   op<14>=1;
ds           op<15>=1;

/* will be used for ALU operations and addresses of NetworkUnit */
start_sound op<16>=1;
stack       op<17>=1;
sram_we     op<18>=1;

IMMEDIATE   op<19>=1;
bus         op<19>=1;

msg_available op<18:16>=0;
token_available op<18:16>=1;
rco_x_ctr    op<18:16>=2;
rco_y_ctr    op<18:16>=3;
alu_equal    op<18:16>=4;
tbmt         op<18:16>=5;
eoc          op<18:16>=6;
lsb         op<18:16>=7;

```

```

/* NetPong assembler code          */
/* Last revised 5-10-96            */
# SPEC_FILE = netpong.sp; /* This statement is required at the
                           beginning of the ASSEM_FILE. It tells
                           where the SPEC_FILE can be found. */

# LIST_FILE = netpong.lst; /* This statement specifies the name for
                           the assembler listing file. If not
                           included, no listing will be created */

# MASK_COUNT = 8; /* This statement is required to mask out 8
                  bits of the 16 bit op-code to produce 2 PROM
                  files. Use with the 'assembl6to8' command. */

# SET_ADDRESS = 0; /* This statement tells the program at what
                  address to start assembling. The address
                  given is a hexadecimal number. */

# LOAD_ADDRESS = 0; /* This statement, if used AFTER the
                   SET_ADDRESS statement, determines the
                   beginning PROM address for the program
                   image. The address is in HEX. */

/* Variable Storage Locations in SRAM */
/* 2 - Ball X Position */
/* 3 - Ball Y Position */
/* 4 - Ball X Velocity */
/* 5 - Ball Y Velocity */
/* 6 - Paddle X Position */
/* 7 - Paddle Y Position */
/* 8 - Player Number */
/* 9 - Player Setup (1=not setup) */
/* A - Ball Ownership */
/* B - Ball Ownership Pending */
/* D - Head of Stack */
/* E - Tail of Stack */

/* clear stack */
assert bus 13;
assert bus 0 sram_we;
assert bus 14;
assert bus 0 sram_we;

/*
; add RESET message to if hardware reset
*/

/* First Add Message Header */
call MSG_HDR;

/* write ID #9 (RESET) to stack */
/* and put updated pointer in alu A reg */
assert alu_oe alu_a_load;
assert sram_we bus 9 stack inc;

/* zero out data field */
assert alu_oe alu_a_load;
assert sram_we bus 0 stack inc;

```

```

/* put updated head pointer back in SRAM */
assert bus 13;
assert sram_we alu_oe;

/* start A/D conversion */
assert convert;

/* skip reinitilization of stack */
jmp CONTINUE_RESET;

BALL_INIT:

/* This sets the ball starting position */
/* and velocity - on startup and on a */
/* ball missing a paddle */

/* x-velocity = -0x20 */
assert bus 4;
assert bus %hA0 sram_we;

/* y-velocity = 0x40 */
assert bus 5;
assert bus %h40 sram_we;

/* x-position = middle of screen */
assert bus 2;
assert bus %h1F sram_we;

/* y-position = middle of screen */
assert bus 3;
assert bus %h1F sram_we;
return;

SW_RESET:

/* clear last message */
assert mcu_ack_msg;
assert mcu_ack_msg;
assert mcu_ack_msg;
cjmp SW_RESET if msg_available;

/* clear stack */
assert bus 13;
assert bus 0 sram_we;
assert bus 14;
assert bus 0 sram_we;

CONTINUE_RESET:

/* init ball*/
call BALL_INIT;

/* player-number = 0 */
assert bus 8;
assert bus 0 sram_we;

/* player-setup = 1 */
assert bus 9;
assert bus 1 sram_we;

/* ball-ownership = 1 */

```



```

assert bus 10;
assert bus 1 sram_we;

/* pending-ball-ownership = 0 */
assert bus 11;
assert bus 0 sram_we;

/* add INIT_PLAYER message to queue */

/* first add message header */
call MSG_HDR;

/* write ID #8 (INIT_PLAYER) to stack */
/* and put updated pointer in alu A reg */
assert alu_oe alu_a_load;
assert sram_we bus 8 stack inc;

/* zero out data field */
assert alu_oe alu_a_load;
assert sram_we bus 0 stack inc;

/* put updated head pointer back in SRAM */
assert bus 13;
assert sram_we alu_oe;

MAIN_LOOP:
    cjmp IDENTIFY if MSG_AVAILABLE;
    cjmp TOKEN if token_available;
    jmp MAIN_LOOP;

MSG_ACK:
    /* Make sure HUB receives our ack */
    assert mcu_ack_msg;
    assert mcu_ack_msg;
    assert mcu_ack_msg;
    cjmp MSG_ACK if msg_available;
    jmp MAIN_LOOP;

IDENTIFY:
    /* identify message */
    assert bytel mcu_msg_oe alu_a_load;
    assert compare bus 3;
    cjmp BALL_X_MOVE if alu_equal;
    assert compare bus 4;
    cjmp BALL_Y_MOVE if alu_equal;
    assert compare bus 5;
    cjmp VELOCITY_X_UPDATE if alu_equal;
    assert compare bus 6;
    cjmp VELOCITY_Y_UPDATE if alu_equal;
    assert compare bus 7;
    cjmp CLAIM_OWNERSHIP if alu_equal;
    assert compare bus 8;
    cjmp INIT_PLAYER if alu_equal;
    assert compare bus 9;
    cjmp SW_RESET if alu_equal;
    jmp MSG_ACK;

TOKEN:
    /* check pending_ball_ownership */

```

```

assert bus 11;
assert sram_oe alu_a_load;
assert compare bus 0;
cjmp CHECK_SETUP if alu_equal;

/* set ownership to one */
assert bus 10;
assert bus 1 sram_we;

/* clear pending_ball_ownership */
assert bus 11;
assert bus 0 sram_we;

/* load X and Y velocity counters */
assert bus 4;
assert sram_oe load_x_ctr;
assert bus 5;
assert sram_oe load_y_ctr;

CHECK_SETUP:

/* check player_setup */
assert bus 9;
assert sram_oe alu_a_load;
assert compare bus 0;
cjmp SKIP_SETUP if alu_equal;

/* Identify which player we are */
assert bus 8;
assert sram_oe alu_a_load;
assert bus 1 compare;
cjmp INIT_PLAYER_1 if alu_equal;
assert bus 2 compare;
cjmp INIT_PLAYER_2 if alu_equal;
assert bus 3 compare;
cjmp INIT_PLAYER_3 if alu_equal;
/* default of player zero */

INIT_PLAYER_0:

/* put paddle on the bottom of the screen */
assert bus 7;
assert sram_we bus %h3F;

/* set initial ball position - middle of screen */
assert bus 2;
assert bus %h1F sram_we;
assert bus 3;
assert bus %h1F sram_we;

/* send y paddle message */
call SEND_Y_PADDLE;

/* claim the ball */
call SEND_CLAIM;
call SEND_X_BALL;
call SEND_Y_BALL;
call SEND_X_VELOCITY;
call SEND_Y_VELOCITY;
jmp SKIP_SETUP;

```

```

INIT_PLAYER_1:
    /* put the paddle on left side of the screen */
    assert bus 6;
    assert sram_we bus %h3F;

    /* send x paddle message */
    call SEND_X_PADDLE;
    jmp SKIP_SETUP;

INIT_PLAYER_2:
    /* put the paddles on top of screen */
    assert bus 7;
    assert sram_we bus 0;

    /* send y paddle message */
    call SEND_Y_PADDLE;
    jmp SKIP_SETUP;

INIT_PLAYER_3:
    /* put the paddle on right side of the screen */
    assert bus 6;
    assert sram_we bus 0;

    /* send x paddle message */
    call SEND_X_PADDLE;

SKIP_SETUP:
    /* set player_setup to zero */
    assert bus 9;
    assert sram_we bus 0;

OUTPUT_QUEUE:
    /* load tail pointer into A register */
    assert bus 14;
    assert sram_oe alu_a_load;

XMIT_LOOP:
    /* load head pointer for comparison */
    assert bus 13;
    assert sram_oe compare;
    cjmp DONE_XMIT if alu_equal;

    /* put tail ptr into alu result reg */
    assert add bus 0;

BUFFER_FULL:
    /* wait for transmit buffer to be empty */
    cjmp BUFFER_EMPTY if tbmt;
    jmp BUFFER_FULL;

BUFFER_EMPTY:
    /* time to output a character */
    /* tail ptr is in alu result reg */
    /* After use inc tail pointer(still also in A reg)*/
    assert alu_oe;
    assert sram_oe stack b;
    assert alu_oe;
    assert alu_oe ds inc;

```

```

/* place incremented tail pointer back in A reg */
assert alu_oe alu_a_load;
jmp XMIT_LOOP;

DONE_XMIT:
/* store tail pointer; currently in both alu */
/* result reg and alu A reg */
assert bus 14;
assert alu_oe sram_we;

A2D_WAIT:
/* Now we are going to check the paddle */

/* Make sure the A/D has finished converting */
/* A/D converts in 2 clock cycles so there is */
/* no concern about it not being done */

/* Determine which paddle we are checking */
assert bus 8;
assert sram_oe alu_a_load;
assert bus 1 compare;
cjmp Y_PADDLE if alu_equal;
assert bus 3 compare;
cjmp Y_PADDLE if alu_equal;

/* our paddle moves on the X-axis */
/* compare and store new value in SRAM */
assert bus 6;
assert sram_oe alu_a_load;
assert bus 6;
assert a2d_oe sram_we;
assert bus 6;
assert sram_oe compare;
cjmp NO_CHANGE if alu_equal;

/*
; make a message to notify change of paddle position
*/
call SEND_X_PADDLE;
jmp NO_CHANGE;

Y_PADDLE:
/* our paddle moves on the y-axis */
/* compare and store new value in SRAM */
assert bus 7;
assert sram_oe alu_a_load;
assert bus 7;
assert a2d_oe sram_we;
assert bus 7;
assert sram_oe compare;
cjmp NO_CHANGE if alu_equal;

/*
; make a message to notify change of paddle position
*/
call SEND_Y_PADDLE;

```

```

NO_CHANGE:
    /* start new A/D conversion */
    assert convert;

    /* if we own the ball we have to check the coutners */
    assert bus 10;
    assert sram_oe alu_a_load;
    assert bus 0 compare;
    cjmp RETURN_TOKEN if alu_equal;

    /* check X counter for RCO */
    cjmp X_MOVE if rco_x_ctr;
    jmp CHECK_Y_MOVE;

X_MOVE:
    /* check direction of x velocity */
    assert bus 4;
    assert sram_oe alu_a_load;
    assert bus %h80 and;
    assert alu_oe alu_a_load;
    assert bus %h80 compare;
    cjmp X_SUBTRACT_ONE if alu_equal;

    /* load x position into ALU A reg */
    assert bus 2;
    assert sram_oe alu_a_load;

    /* we should add one to the ball position */
    assert inc;
    jmp SEND_X_MSG;

X_SUBTRACT_ONE:
    /* load x position into ALU A reg */
    assert bus 2;
    assert sram_oe alu_a_load;

    /* subtract one from the X position */
    assert sub bus 1;

SEND_X_MSG:
    /* make sure ball is within the boundry */
    assert alu_oe alu_a_load;
    assert bus %h3F and;

    /* first we need to update the X position in SRAM */
    assert bus 2;
    assert alu_oe sram_we;

    /* now send message and reload counter */
    call SEND_X_BALL;

CHECK_Y_MOVE:
    /* check y coutner for RCO */
    cjmp Y_MOVE if rco_y_ctr;
    jmp RETURN_TOKEN;

Y_MOVE:
    /* check direction of y velocity */
    assert bus 5;

```

```

assert sram_oe alu_a_load;
assert bus %h80 and;
assert alu_oe alu_a_load;
assert bus %h80 compare;
cjmp Y_SUBTRACT_ONE if alu_equal;

/* load y position into ALU A reg */
assert bus 3;
assert sram_oe alu_a_load;

/* we should add one to the ball position */
assert inc;
jmp SEND_Y_MSG;

Y_SUBTRACT_ONE:
/* load y position into ALU A reg */
assert bus 3;
assert sram_oe alu_a_load;

/* subtract one from the Y position */
assert sub bus 1;

SEND_Y_MSG:
/* make sure ball is within the boundry */
assert alu_oe alu_a_load;
assert bus %h3F and;

/* first we need to update the Y position in SRAM */
assert bus 3;
assert alu_oe sram_we;

/* now send message and reload counter */
call SEND_Y_BALL;

RETURN_TOKEN:
/* Make sure we the transmit buffer is empty */
cjmp RETURN_TOKEN2 if tbmt;
jmp RETURN_TOKEN;

RETURN_TOKEN2:
/* And that the transmit shift buffer is empty */
cjmp RETURN_TOKEN3 if eoc;
jmp RETURN_TOKEN2;

RETURN_TOKEN3:
assert token_return;
assert token_return;
assert token_return;
cjmp RETURN_TOKEN3 if token_available;
jmp MAIN_LOOP;

BALL_X_MOVE:
/* first update ball position in SRAM */
assert bus 2;
assert byte2 mcu_msg_oe sram_we;

/* check player number - return if not 1 or 3 */

```

```

assert bus 8;
assert sram_oe alu_a_load;
assert compare bus 1;
cjmp X_CHECK if alu_equal;
assert compare bus 3;
cjmp X_CHECK if alu_equal;
jmp MSG_ACK;

```

X\_CHECK:

```

/* check to see if ball has reached our edge */
assert byte2 mcu_msg_oe alu_a_load;
assert bus 6;
assert sram_oe compare;
cjmp X_SIDE_HIT if alu_equal;
jmp MSG_ACK;

```

X\_SIDE\_HIT:

```

/* it hit our side so claim the ball */
/* and see if ball hit our paddle */
call SEND_CLAIM if alu_equal;

/* see if ball hit our paddle */
assert bus 7;
assert sram_oe alu_a_load;
assert bus 3;
assert sram_oe compare;
cjmp X_PADDLE_HIT if alu_equal;
/* paddle + 1 */
assert inc;
assert alu_oe alu_a_load;
assert bus 3;
assert sram_oe compare;
cjmp X_PADDLE_HIT if alu_equal;
/* paddle + 2 */
assert inc;
assert alu_oe alu_a_load;
assert bus 3;
assert sram_oe compare;
cjmp X_PADDLE_HIT if alu_equal;
/* paddle + 3 */
assert inc;
assert alu_oe alu_a_load;
assert bus 3;
assert sram_oe compare;
cjmp X_PADDLE_HIT if alu_equal;
/* paddle + 4 */
assert inc;
assert alu_oe alu_a_load;
assert bus 3;
assert sram_oe compare;
cjmp X_PADDLE_HIT if alu_equal;
/* paddle + 5 */
assert inc;
assert alu_oe alu_a_load;
assert bus 3;
assert sram_oe compare;
cjmp X_PADDLE_HIT if alu_equal;

```

```

/* The ball missed our paddle - reset ball */
call BALL_INIT;
call SEND_X_BALL;
call SEND_Y_BALL;
call SEND_X_VELOCITY;
call SEND_Y_VELOCITY;
jmp MSG_ACK;

X_PADDLE_HIT:
/* Invert the X velocity */
assert bus 4;
assert sram_oe alu_a_load;
assert bus %h80 xor;
assert bus 4;
assert alu_oe sram_we;

/* send UPDATE_X_VELOCITY Message */
call SEND_X_VELOCITY;

jmp MSG_ACK;

BALL_Y_MOVE:
/* first update ball position in SRAM */
assert bus 3;
assert byte2 mcu_msg_oe sram_we;

/* check player number - return if not 0 or 2 */
assert bus 8;
assert sram_oe alu_a_load;
assert compare bus 0;
cjmp Y_CHECK if alu_equal;
assert compare bus 2;
cjmp Y_CHECK if alu_equal;
jmp MSG_ACK;

Y_CHECK:
/* check to see if ball has reached our edge */
assert byte2 mcu_msg_oe alu_a_load;
assert bus 7;
assert sram_oe compare;
cjmp Y_SIDE_HIT if alu_equal;
jmp MSG_ACK;

Y_SIDE_HIT:
/* it hit our side so claim the ball */
/* and see if ball hit our paddle */
call SEND_CLAIM if alu_equal;

/* see if ball hit our paddle */
assert bus 6;
assert sram_oe alu_a_load;
assert bus 2;
assert sram_oe compare;
cjmp Y_PADDLE_HIT if alu_equal;
/* paddle + 1 */
assert inc;
assert alu_oe alu_a_load;
assert bus 2;

```



```

assert sram_oe compare;
cjmp Y_PADDLE_HIT if alu_equal;
/* paddle + 2 */
assert inc;
assert alu_oe alu_a_load;
assert bus 2;
assert sram_oe compare;
cjmp Y_PADDLE_HIT if alu_equal;
/* paddle + 3 */
assert inc;
assert alu_oe alu_a_load;
assert bus 2;
assert sram_oe compare;
cjmp Y_PADDLE_HIT if alu_equal;
/* paddle + 4 */
assert inc;
assert alu_oe alu_a_load;
assert bus 2;
assert sram_oe compare;
cjmp Y_PADDLE_HIT if alu_equal;
/* paddle + 5 */
assert inc;
assert alu_oe alu_a_load;
assert bus 2;
assert sram_oe compare;
cjmp Y_PADDLE_HIT if alu_equal;

/* The ball missed our paddle - reset ball */
call BALL_INIT;
call SEND_X_BALL;
call SEND_Y_BALL;
call SEND_X_VELOCITY;
call SEND_Y_VELOCITY;
jmp MSG_ACK;

```

Y\_PADDLE\_HIT:

```

/* Invert the Y velocity */
assert bus 5;
assert sram_oe alu_a_load;
assert bus %h80 xor;
assert bus 5;
assert alu_oe sram_we;

/* send UPDATE_Y_VELOCITY Message */
call SEND_Y_VELOCITY;

jmp MSG_ACK;

```

SEND\_CLAIM:

```

/*
; set the ball_ownership_pending bit and
; add CLAIM_OWNERSHIP message to stack
*/

/* set ball_ownership_pending bit */
assert bus 11;
assert bus 1 sram_we;

```

```

/* first add message header */
call MSG_HDR;

/* write ID #7 (CLAIM_OWNERSHIP) to stack */
/* and put updated pointer in alu A reg */
assert alu_oe alu_a_load;
assert sram_we bus 7 stack inc;

/* zero out data field */
assert alu_oe alu_a_load;
assert sram_we bus 0 stack inc;

/* put updated head pointer back in SRAM */
assert bus 13;
assert sram_we alu_oe;
return;

VELOCITY_X_UPDATE:
/* update X velocity in SRAM */
assert bus 4;
assert byte2 mcu_msg_oe sram_we;
jmp MSG_ACK;

VELOCITY_Y_UPDATE:
/* update Y velcoty in SRAM */
assert bus 5;
assert byte2 mcu_msg_oe sram_we;
jmp MSG_ACK;

CLAIM_OWNERSHIP:
/* clear ownership bit */
assert bus 10;
assert bus 0 sram_we;
jmp MSG_ACK;

INIT_PLAYER:
/* if not setup incriment player number */
assert bus 9;
assert sram_oe alu_a_load;
assert compare bus 1;
cjmp INC_PLAYER if alu_equal;
jmp MSG_ACK;

INC_PLAYER:
/* incriment player number */
assert bus 8;
assert sram_oe alu_a_load;
assert inc bus 8;
assert alu_oe sram_we;

/* and clear ball ownership */
assert bus 10;
assert bus 0 sram_we;

jmp MSG_ACK;

MSG_HDR:
/* Add sync byte & player number to message queue */

```

```

/*      leaves head pointer in ALU result reg */
/*      Note: head pointer is NOT updated in SRAM! */

/*      load stack head into alu and sram mem reg */
assert bus 13;
assert sram_oe alu_a_load;

/**/

/*      write sync byte and inc head pointer */
assert bus %h3F sram_we stack inc;

/*      update head pointer in SRAM */
assert bus 13;
assert sram_we alu_oe;

/*      place player number in alu result reg */
assert bus 8;
assert sram_oe b;

/*      write player number in stack at head ptr */
assert bus 13;
assert sram_oe alu_a_load;
assert alu_oe sram_we stack inc;
return;

SEND_X_PADDLE:

/*
; make a message to notify change of paddle position
*/

/*      first add message header */
call MSG_HDR;

/*      write ID #1 (PADDLE_X_MOVE) to stack */
/*      and put updated pointer in alu A reg */
assert alu_oe alu_a_load;
assert sram_we bus 1 stack inc;

/*      put new A/D value as data field */
assert bus 13;
assert alu_oe sram_we;
assert bus 6;
assert sram_oe b; /* place A2D val in acc */
assert bus 13;
assert sram_oe alu_a_load;
assert alu_oe sram_we stack inc;

/*      put updated head pointer back in SRAM */
assert bus 13;
assert sram_we alu_oe;

return;

SEND_Y_PADDLE:

/*
; make a message to notify change of paddle position
*/

/*      first add message header */

```

```

call MSG_HDR;

/* write ID #2 (PADDLE_Y_MOVE) to stack */
/* and put updated pointer in alu A reg */
assert alu_oe alu_a_load;
assert sram_we bus 2 stack inc;

/* put new A/D value as data field */
assert bus 13;
assert alu_oe sram_we;
assert bus 7;
assert sram_oe b; /* place A2D val in acc */
assert bus 13;
assert sram_oe alu_a_load;
assert alu_oe sram_we stack inc;

/* put updated head pointer back in SRAM */
assert bus 13;
assert sram_we alu_oe;

return;

SEND_X_BALL:

/* We need to send a BALL_X_MOVE message */
/* first add message header */
call MSG_HDR;

/* write ID #3 (BALL_X_MOVE) to stack */
/* and put updated pointer in alu A reg */
assert alu_oe alu_a_load;
assert sram_we bus 3 stack inc;

/* store head pointer in SRAM */
assert bus 13;
assert sram_we alu_oe;

/* get new ball x position out of SRAM and */
/* place into the ALU result reg */
assert bus 2;
assert sram_oe b;

/* write the value as the data field */
assert bus 13;
assert sram_oe alu_a_load;
assert sram_we stack alu_oe inc;

/* put updated head pointer back in SRAM */
assert bus 13;
assert sram_we alu_oe;

/* re-load x counter with correct velocity */
assert bus 4;
assert sram_oe alu_a_load;
assert bus and %h7F;
assert alu_oe load_x_ctr;

return;

```

```

SEND_Y_BALL:
    /* We need to send a BALL_Y_MOVE message */
    /* first add message header */
    call MSG_HDR;

    /* write ID #4 (BALL_Y_MOVE) to stack */
    /* and put updated pointer in alu A reg */
    assert alu_oe alu_a_load;
    assert sram_we bus 4 stack inc;

    /* store head pointer in SRAM */
    assert bus 13;
    assert sram_we alu_oe;

    /* get new ball y position out of SRAM and */
    /* place into the ALU result reg */
    assert bus 3;
    assert sram_oe b;

    /* write the value as the data field */
    assert bus 13;
    assert sram_oe alu_a_load;
    assert sram_we stack alu_oe inc;

    /* put updated head pointer back in SRAM */
    assert bus 13;
    assert sram_we alu_oe;

    /* re-load y counter with correct velocity */
    assert bus 5;
    assert sram_oe alu_a_load;
    assert bus and %h7F;
    assert alu_oe load_y_ctr;

    return;

SEND_X_VELOCITY:
    /* We need to send a UPDATE_X_VELOCITY message */
    /* first add message header */
    call MSG_HDR;

    /* write ID #5 (UPDATE_X_VELOCITY) to stack */
    /* and put updated pointer in alu A reg */
    assert alu_oe alu_a_load;
    assert sram_we bus 5 stack inc;

    /* store head pointer in SRAM */
    assert bus 13;
    assert sram_we alu_oe;

    /* get new ball x velocity out of SRAM and */
    /* place into the ALU result reg */
    assert bus 4;
    assert sram_oe b;

    /* write the value as the data field */
    assert bus 13;
    assert sram_oe alu_a_load;

```

```

assert sram_we stack alu_oe inc;

/* put updated head pointer back in SRAM */
assert bus 13;
assert sram_we alu_oe;

return;

SEND_Y_VELOCITY:
/* We need to send a UPDATE_Y_VELOCITY message */
/* first add message header */
call MSG_HDR;

/* write ID #6 (UPDATE_6_VELOCITY) to stack */
/* and put updated pointer in alu A reg */
assert alu_oe alu_a_load;
assert sram_we bus 6 stack inc;

/* store head pointer in SRAM */
assert bus 13;
assert sram_we alu_oe;

/* get new ball x velocity out of SRAM and */
/* place into the ALU result reg */
assert bus 5;
assert sram_oe b;

/* write the value as the data field */
assert bus 13;
assert sram_oe alu_a_load;
assert sram_we stack alu_oe inc;

/* put updated head pointer back in SRAM */
assert bus 13;
assert sram_we alu_oe;

return;

```