NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

NEURAL NETWORK DESIGN ON THE SRC-6 RECONFIGURABLE COMPUTER

by

Scott P. Bailey

December 2006

Thesis Advisor:	Douglas J. Fouts
Second Reader:	Jon T. Butler

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 2006	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE Neural Network Design on the SRC-6 Reconfigurable Computer	**5. FUNDING NUMBERS**
6. AUTHOR Scott P. Bailey	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Security Agency Fort Meade, MD	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

This thesis presents an approach to image classification via a Multi-Layer Perceptron (MLP) Artificial Neural Network (ANN) on the SRC-6 reconfigurable computer for use in classifying Low Probability of Intercept (LPI) radar emitters. The rationale behind the previously unexplored use of new reconfigurable computers combined with neural networks for this application is the potential for near real-time classification. Current potential near-peer competitors have access to LPI technology, so development of quick classification methods is crucial for ships to determine intent and to enable the possibility for self-defense against these types of emitters. The neural network, based on work conducted by Professor Phillip E. Pace of the Naval Postgraduate School (NPS), generates integer-cast weights by first using a sequential processor to conduct floating-point backpropagation to train the network on potential time-frequency images that allows generation of weights with lower overall Root Mean Squared (RMS) errors. The weights are then used in a parallel-processing reconfigurable computer for close to real-time classification. A second method of direct pixel comparison using Exclusive-Or (XOR) logic is presented as an alternative image classification method. Comparisons to similar representations in C++ are provided, for use in judging comparative error levels and timing between parallel and sequential processing methods.

14. SUBJECT TERMS Image Classification, Neural Network, SRC-6, Reconfigurable Computer, Backpropagation, LPI Emitter	**15. NUMBER OF PAGES** 130
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

NEURAL NETWORK DESIGN ON THE SRC-6 RECONFIGURABLE COMPUTER

Scott P. Bailey
Lieutenant, United States Navy
B.S., Illinois Institute of Technology, 1997

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 2006

Author: Scott P. Bailey

Approved by: Douglas J. Fouts
Thesis Advisor

Jon T. Butler
Second Reader

Jeffrey B. Knorr
Chairman, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis presents an approach to image classification via a Multi-Layer Perceptron (MLP) Artificial Neural Network (ANN) on the SRC-6 reconfigurable computer for use in classifying Low Probability of Intercept (LPI) radar emitters. The rationale behind the previously unexplored use of new reconfigurable computers combined with neural networks for this application is the potential for near real-time classification. Current potential near-peer competitors have access to LPI technology, so development of quick classification methods is crucial for ships to determine intent and to enable the possibility for self-defense against these types of emitters. The neural network, based on work conducted by Professor Phillip E. Pace of the Naval Postgraduate School (NPS), generates integer-cast weights by first using a sequential processor to conduct floating-point backpropagation to train the network on potential time-frequency images that allows generation of weights with lower overall Root Mean Squared (RMS) errors. The weights are then used in a parallel-processing reconfigurable computer for close to real-time classification. A second method of direct pixel comparison using Exclusive-Or (XOR) logic is presented as an alternative image classification method. Comparisons to similar representations in C++ are provided, for use in judging comparative error levels and timing between parallel and sequential processing methods.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1
- **A. THE CENTRAL PROBLEM AND PURPOSE 1**
  - **1. Low Probability of Intercept (LPI) Emitters 1**
    - ***a. First-generation Systems and Information Communication 1***
    - ***b. Low Probability of Intercept (LPI) Radars 1***
    - ***c. Potential Detection Methodology 2***
  - **2. Purpose of this Thesis 2**
- **B. DESIGN OVERVIEW 3**
  - **1. Overview of the SRC-6 Reconfigurable Computer 3**
    - ***a. IMPLICIT + EXPLICIT™ Architecture 3***
    - ***b. Hardware Environment 5***
  - **2. Data Input and Preprocessing 6**
  - **3. ANN Image Classifier 6**
  - **4. Alternative Image Classification Method 7**
- **C. THESIS ORGANIZATION 8**

II. BACKGROUND 9
- **A. NEURAL NETWORKS 9**
  - **1. History of Development 9**
  - **2. Multi-Layer Perceptron Networks 10**
    - ***a. Basics of Multi-Layer Perceptron Design 10***
    - ***b. Backpropagation in Detail 12***
- **B. IMAGE CLASSIFICATION 14**
  - **1. Current Research 14**
  - **2. An Application for Detection of LPI Emitters 15**

III. IMAGE AND NETWORK WEIGHT GENERATION 17
- **A. IMAGE GENERATION 17**
  - **1. Image Source 17**
  - **2. Selection of Training Images 17**
- **B. WEIGHT TRAINING SEQUENTIAL-PROCESSOR NETWORK 18**
  - **1. Background 18**
  - **2. Sequential Weight-Generation Program Design 19**
  - **3. Program Operation 20**

IV. RECONFIGURABLE-ENVIRONMENT ARTIFICIAL NEURAL NETWORK (RANN) DESIGN AND OPERATION 23
- **A. DESIGN OVERVIEW 23**
  - **1. Network Input 23**
    - ***a. Connection Weights File 23***
    - ***b. Preprocessed Image Input 24***
  - **2. Input-to-Hidden Layer Processing 25**
    - ***a. Hidden-Layer Connection Weighting and Summation 25***

b. Sigmoid Transfer Function Processing *26*
3. Hidden-to-Output Layer Processing 27

V. NETWORK PERFORMANCE COMPARISON 29
A. PERFORMANCE COMPARISON METHODOLOGY 29
B. SRC-SPECIFIC PERFORMANCE 31
C. SUMMARY 32

VI. EXCLUSIVE-OR (XOR) IMAGE COMPARITOR 33
A. RECONFIGURABLE PROGRAM DESIGN 33
B. RECONFIGURABLE PROGRAM EXECUTION 34
1. Hardware Demands 34
2. Performance Gains: 35
C. SEQUENTIAL COMPARISON PROGRAM 35

VII. CONCLUSION 37
A. SUMMARY OF WORK 37
B. SUGGESTED FUTURE WORK 38
1. Comprehensive Analysis of the SRC LPI Detection System 38
2. Program Optimization 39

APPENDIX A. IMAGES CREATED FOR NETWORK TESTING 41

APPENDIX B. PUBLIC DOMAIN NEURAL NETWORK CODE 43

APPENDIX C. WEIGHT GENERATION NEURAL NETWORK CODE 49

APPENDIX D. OUTPUT OF WEIGHT GENERATION NEURAL NETWORK CODE 65

APPENDIX E. MLP NEURAL NETWORK CODE FOR THE SRC 69
MAIN.C CODE: 69
EX07.MC CODE: 72

APPENDIX F. IMAGE CONVERSION PROGRAM 77

APPENDIX G. SIGMOID FUNCTION VHDL FILES 79
SIGFOUR.VHD: 79
BLK.V: 80
INFO FILE: 81

APPENDIX H. NETWORK OUTPUT GRAPHS 83

APPENDIX I. RECONFIGURABLE-ENVIRONMENT ARTIFICIAL NEURAL NETWORK (RANN) INSTRUCTION GUIDE 87

APPENDIX J. SRC-6 EXCLUSIVE-OR COMPARITOR CODE 89
MAIN.C CODE: 89
EX07.MC CODE: 90

APPENDIX K. OUTPUT OF RECONFIGURABLE XOR COMPARITOR 99

APPENDIX L. SEQUENTIAL-PROCESSOR EXCLUSIVE-OR (XOR) COMPARITOR CODE 101

LIST OF REFERENCES **105**
INITIAL DISTRIBUTION LIST **107**

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1. IMPLICIT + EXPLICIT™ Architecture (From [4]) 4
Figure 2. MAP® Direct Execution Logic (DEL) Processor (From [4]) 5
Figure 3. 1024-5-5 ANN Design 7
Figure 4. Biological vs. Artificial Neurons 10
Figure 5. Common Transfer Functions 12
Figure 6. Learning Rate Effects with (a) smaller and (b) larger than desired rate 13
Figure 7. QMFB Contour Frequency-Time Image (From [1]) 15
Figure 8. File Design Architecture for 'weightout' Program 23
Figure 9. Hidden-Layer Weight Accumulator 26
Figure 10. Hidden-to-Output Layer Processing 27
Figure 11. P4 Image Network Output Comparison 30
Figure 12. 'XOR-Mask' Comparator 33
Figure 13. P4 Image Network Output Comparison 83
Figure 14. T4 Image Network Output Comparison 83
Figure 15. T3 Image Network Output Comparison 84
Figure 16. T2 Image Network Output Comparison 84
Figure 17. No Image Network Output Comparison 85

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1. Network Execution Times 30
Table 2. RMS Error Value Comparison.....31

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF SYMBOLS, ACRONYMS, AND/OR ABBREVIATIONS

ANN	Artificial Neural Network
ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuit
BRAM	Block Random Access Memory
DARPA	Defense Advanced Research Projects Agency
DEL	Direct Execution Logic
DLD	Dense Logic Device
DSP	Digital Signal Processing
EA	Electronic Attack
EM	Electromagnetic
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input/Output
LPI	Low Probability of Intercept
LUT	Look-Up Table
MLP	Multi-Layer Perceptron
MRI	Magnetic Resonance Imaging
NPS	Naval Postgraduate School
OBM	On-Board Memory
QMFB	Quadrature Mirror Filter Bank
RAM	Random Access Memory
RANN	Reconfigurable-Environment Artificial Neural Network
RMS	Root Mean Squared
USMC	United States Marine Corps
USNR	United States Naval Reserve
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XOR	Exclusive-Or

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank Jon Huppenthal and David Caliga of SRC Computers, Inc. Their assistance in understanding the intricacies of programming on the SRC-6 was instrumental in developing the methodology of implementation.

Professor Phil Pace and Professor Monique Fargues proved invaluable in aiding my understanding of the development and use of Neural Networks. In particular, Professor Fargues' EC 4460 Neural Networks class was extremely valuable in providing a solid background in Artificial Neural Networks.

I would also like to thank Professor Douglas Fouts and Professor Jon Butler, whose instruction in Computer Architecture inspired me towards the choice of this particular thesis.

The financial support of the National Security Agency was an essential component towards completion of this thesis.

Finally, I would like to thank my wife, Ruri Kaneko Bailey, and my children, Lucy Mitsuko and Meg Masako for their unwavering love and support.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The purpose of this thesis is to design and test an artificial neural network (ANN) architecture for the SRC-6 reconfigurable computer. An ANN is a model that attempts to emulate the complex processing capabilities of the brain, in order to achieve better results than standard programming models. This ANN is used as an image classifier as a part of a project to design a complete Low Probability of Intercept (LPI) detection system in a reconfigurable computing environment. LPI emitters have been developed in an effort to render current passive detection systems useless. The potential threat of use of LPI emitters by hostile entities against current military units is the reason behind the design of this complete LPI detection system. The potential threat of anti-ship cruise missiles with LPI seeker heads is significant enough to warrant a careful study. The LPI detection system consists of three parts, a data input and Quadrature Mirror Filter Bank (QMFB) that conducts Digital Signal Processing (DSP), a preprocessing step that converts the data into a useable form, and an ANN classification system to interpret the data. The design goals of the overall project were to realize real-time classification of LPI signals through the use of a reconfigurable computer.

Our ANN design is based on a feedforward Multi-Layer Perceptron (MLP) architecture. Significant changes to a typical MLP were required in order to take full advantage of the abilities inherent in the SRC-6 reconfigurable computer. These design decisions were the separation of the network weight training program from the network execution program, execution of the network using fixed-point integer math, and realization of the nonlinear transfer function via a Look-Up Table (LUT). Design decisions were also made according to the goals specific to this particular work, which were minimization of SRC-6 hardware requirements and reusability of code. The result of these decisions is a network that executes at approximately ten times the speed of a sequential-processor network. The output of this network is compared to sequential-processor output and is found acceptable for the purpose of classification. This project is fully capable of future integration into the complete LPI detection system.

An alternative methodology of image comparison is shown that provides potentially quicker image comparison. The alternative method uses direct pixel-to-pixel comparison between input images and stored comparison images and selects the ‘least different’ result. While simplistic in nature, this method takes advantage of the ability of a reconfigurable computer to conduct simultaneous parallel processes. This method has a larger demand on hardware resources in its current configuration and thus may not be desirable for use in the complete LPI detection system.

I. INTRODUCTION

A. THE CENTRAL PROBLEM AND PURPOSE

1. Low Probability of Intercept (LPI) Emitters

a. First-generation Systems and Information Communication

A typical radar system encounters an information dilemma. To obtain information on potential targets, the radar must emit electromagnetic (EM) energy that reflects off the target. Processing the reflected energy is then used to obtain range and bearing data. With repeated attempts, this range and bearing data provides estimated courses and speeds for those targets. The development of passive detection receiver technologies, however, allowed targets the potential to detect EM emission and obtain useful information from the signal. Direction and specific energy characteristic information allows the potential identification of emitter types and location. This information, when correlated with known data such as which ships currently carry such emitters and what those emitters are used for can be used to determine identification of the emitting vessel and possible intent. For example, if a certain emitter is known to be used as a fire-control radar for only a certain type of ship, and the emissions of the radar are detected, the illuminated vessel can obtain the information that that particular ship class, in a particular direction, is attempting to obtain a fire control solution. Once the particulars of an emitter are known then Electronic Attack (EA) measures such as jamming can be used to a greater effect. Obviously, this two-way flow of information is detrimental towards stealth and radar effectiveness, and thus has negative impacts on a variety of missions for the military. Thus, a desire grew to develop "stealthy" radars that do not reveal themselves as easily.

b. Low Probability of Intercept (LPI) Radars

LPI radar systems have become an important and developing tactical requirement [1]. In simplest terms, a method used to attempt to achieve LPI is spreading the emitted energy over a wider range of frequencies using various pulse compression techniques. This allows energies at specific frequencies to be lower and therefore harder

to detect. The ultimate goal of LPI emitter systems is to have the emitted energy become indistinguishable from noise for the target, while providing quality information to the emitter.

c. Potential Detection Methodology

A methodology for use in detection of LPI emitters is detailed by Professor Phillip E. Pace in Detecting and Classifying Low Probability of Intercept Radar [1]. To date, there have been two theses conducted by students at Naval Postgraduate School (NPS) that attempts to implement a portion of this method on the SRC-6 reconfigurable computer. The work by Captain Kevin Stoffel, United States Marine Corps (USMC), involves conversion of an outside signal into a frequency-time plot of data using an Analog to Digital converter connected into QMFBs on the SRC hardware [2]. A thesis by Ensign Dane Brown, United States Naval Reserve (USNR), details a method for preprocessing the initial frequency-time plot into a binary-pixel bitmap for classification on the SRC hardware as well [3].

2. Purpose of this Thesis

The development of the reconfigurable computer involves a compromise between two established and successful architectures. The common computer normally uses a general-purpose processor that computes sequentially, that is, executing specific instructions on the processor one at a time. Operating system developments such as multithreading may allow the appearance of multiple simultaneous operations but the hardware is typically running only one process at a time. The benefit of this format is that the general processor has a great flexibility in what it does, because the operations can cause various types of output from various types of input. The sequential nature of the processor, however, may result in time-delay of information, especially in situations that require large amounts of processing of the input.

A different architecture that has been explored is application-specific hardware in the form of Application Specific Integrated Circuits (ASICs). This architecture generally uses specific circuitry that conducts a single type or small range of processes on certain data types. An example of this type of circuit would be some of the commonly available DSP chips, that are designed specifically for particular kinds of communication processing. The benefit of this type of architecture is it can process at higher speeds,

because the nature of input, process, and output is usually a well-understood constant and thus the entire architecture is relatively static. The downside of this is the loss of versatility.

Reconfigurable computing uses Field-Programmable Gate Arrays (FPGAs) to provide process-specific circuits. In the case of the SRC-6 computer, functions called *macros* allow the user establish these circuits. In this way an ASIC-type architecture mimics the versatility of software running on a general purpose computer, while allowing potential gains in processing ability and speed due to the ability to shape the FPGA to efficiently process the data. This shaping includes parallel-processing hardware schemes, that have a potential for speed gains over sequential processors, in spite of the relatively low clock speeds of FPGA systems.

This thesis explores the use of a Feed-forward, Multi-Layer Perceptron (MLP) Artificial Neural Network (ANN) architecture to conduct image classification in a reconfigurable computing environment. A MLP ANN can be 'trained' to classify images from given inputs and, therefore, has the potential to assist in classifying the preprocessed data that arrives from the aforementioned QMFB array. Thus, this ANN has the potential to directly contribute towards detection and classification of LPI emitters. Realizing this network in a reconfigurable environment provides the potential to realize significant gains in the time required to effectively conduct classification.

B. DESIGN OVERVIEW

1. Overview of the SRC-6 Reconfigurable Computer

In 1996 SRC Computers Incorporated was established in Colorado Springs, Colorado, by the well known computer entrepreneur Seymour Cray. The company developed the IMPLICIT + EXPLICIT™ architecture, the Carte™ programming environment, and the MAP® reconfigurable processor, with the overall goal of increasing processor performance [4].

a. IMPLICIT + EXPLICIT™ Architecture

The SRC IMPLICIT + EXPLICIT™ architecture is the overarching system by which Dense Logic Devices (DLDs), such as microprocessor and ASIC

devices are coupled with Direct Execution Logic (DEL) such as the MAP® reconfigurable logic. A graphical representation of this architecture, taken from a SRC Computer white paper on the subject, is shown in Figure 1.

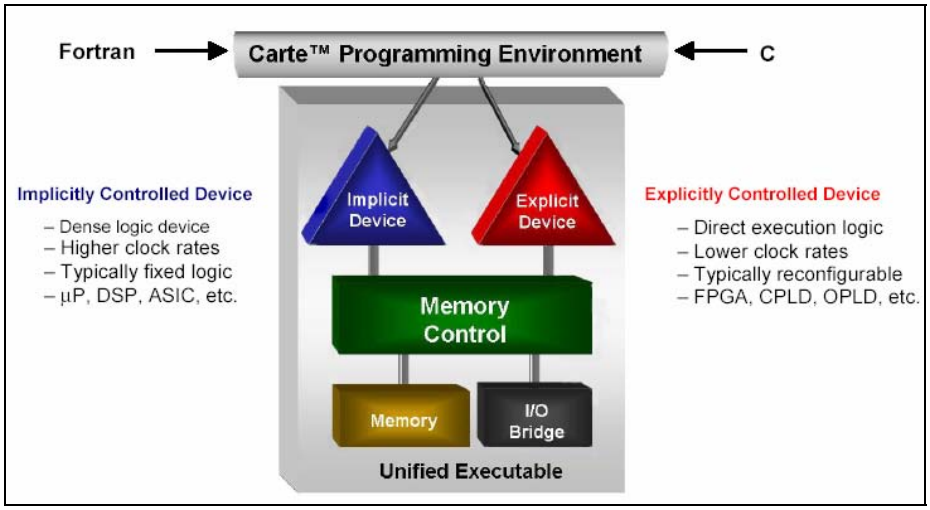


Figure 1. IMPLICIT + EXPLICIT™ Architecture (From [4])

The Carte™ Programming environment allows programmers to tailor previous C++ or Fortran code with minor modifications and execute in a reconfigurable environment. For example, the 'main.c' code will execute purely on the implicitly-controlled 2.8 GHz Intel Xeon microprocessor if that is the programmer's wish. If the programmer decides to execute code on the MAP® DEL processor, it is executed in the manner of a function call to a subroutine contained in a separate source-code file with a .mc suffix. These DEL-specific files can include user-generated macros developed in Verilog or Very High Speed Integrated Circtuit (VHSIC) Hardware Description Language (VHDL), augmenting the capabilities of the C++ language to deal with individual bits or optimizing speed by explicitly determining the DEL processes. The overarching nature of the Carte™ programming environment to handle DLD and DEL control is shown in Figure 1. At compile time, these files are combined in an executable along with C++ code. It is important to note that Verilog/VHDL macro programming is not essential to running most code on the SRC. Verilog/VHDL coding allows the user, however, to directly control the FPGA resources.

b. Hardware Environment

The MAP® DEL processor is the device that enables the reconfigurability of the SRC-6. The MAP® is comprised of 2 Xilinx XC2V6000 FPGAs for use as user logic, six banks of On-Board Memory (OBM) that provide 24MB of Random Access Memory (RAM) storage connected to the user logic with a 4800 MB/s bus, a 2400 MB/s General Purpose Input/Output (GPIO) connection that provides a communication channel directly off the MAP®, and another Xilinx XC2V6000, which acts as a controller. A graphical representation of the interfaces from SRC Computers, Incorporated is provided in Figure 2.

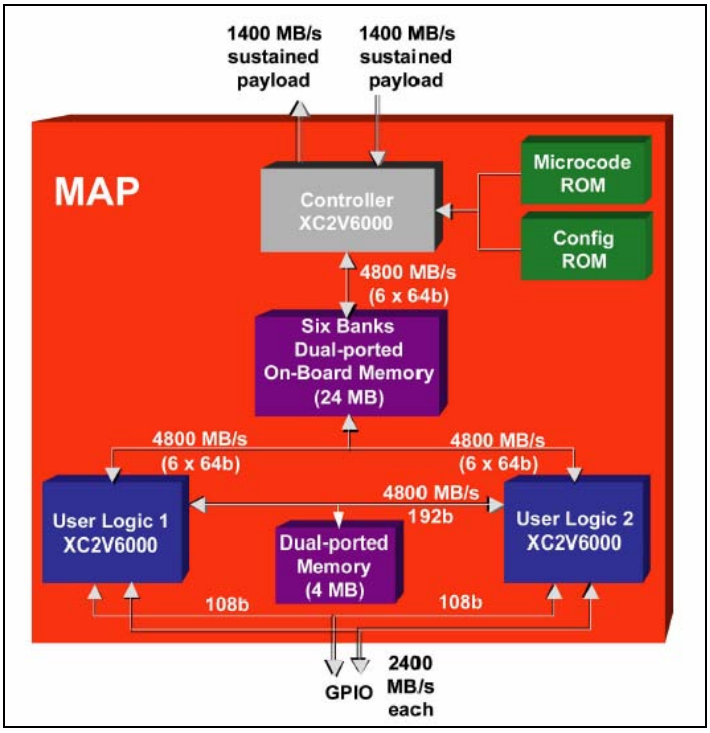


Figure 2. MAP® Direct Execution Logic (DEL) Processor (From [4])

OBM is not the only memory available to the user, because the FPGA itself holds 144 Block RAM (BRAM) units of 2048 bytes each. The way the Carte™ environment handles this distinction in code is by making use of OBM explicit in the .mc code, while variables and arrays locally called in the .mc code are stored in BRAM. One important item of note is that the user logic 18x18 multipliers share the same input lines as the BRAM. Attention must therefore be directed to resource allocation in the case

where a program will use large amounts of either multipliers or BRAM. While this was not a problem encountered for this project, expansions of the original ANN design may require designers to be aware of this potential conflict, especially if multiple BRAM banks are used to achieve simultaneous access for speed of execution. The FPGA itself can be configured to act as a RAM, in a form referred to as distributed select RAM. The distributed select RAM memory method was not pursued in this project.

2. Data Input and Preprocessing

The initial requirement for converting the LPI detection system specified in [1] to run in a reconfigurable computing environment was the development of a data input mechanism. The thesis work conducted by Kevin M. Stoffel describes a system comprising of an Analog-to-Digital Converter coupled with a hardware interface and SRC programming that inputs the data from the hardware through a QMFB. This combination of hardware and software allows the generation of 8-bit frequency-time plots whose size is constrained by current MAP® hardware limitations. While these constraints are discussed in much greater detail in [2], the end result is an eight-bit pixel bitmap that must be then preprocessed for ease of classification.

The preprocessing portion of the overall LPI detection system converts the eight-bit pixel bitmap to a single-bit pixel bitmap using a function to apply a threshold to the data. The end product of this code is a NxN square bitmap that is then used by the classifying portion to determine the nature of the input. Initial planning sessions envisioned the output of the preprocessing step to be a 32x32 single-bit pixel bitmap. A detailed discussion of the threshold function and preprocessing step is contained in [3].

3. ANN Image Classifier

To enable correct classification of potential LPI emitters, a Feed-forward MLP ANN was designed. Because the initial discussion agreed upon a 32x32 pixel bitmap image as the input source, this became a primary requirement for the initial design. The resulting architecture developed into a 1024-5-5 Feed-forward MLP ANN, which means that the network had 1024 inputs, 5 hidden layer nodes, and 5 outputs. The network architecture is displayed in Figure 3.

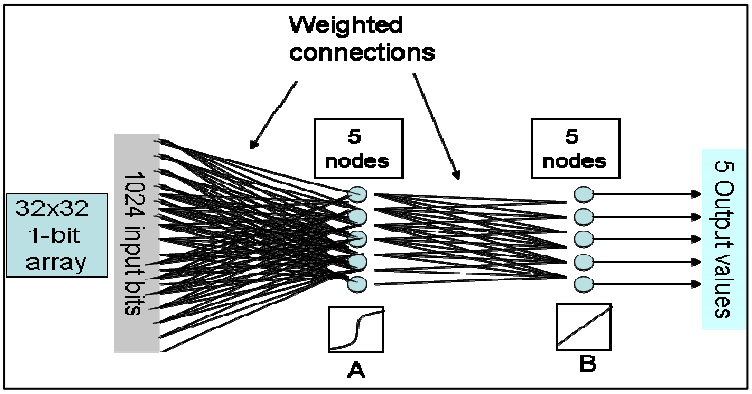


Figure 3. 1024-5-5 ANN Design

The above figure shows the Feed-forward nature of the design, which inputs the bitmap on the left to produce an output sequence on the right. The five hidden layer nodes at 'A' are coupled with a sigmoid transfer function, while the output layer nodes at 'B' are coupled with a pure linear transfer function. The ANN weights for each connection were generated in an off-chip modeling program that used sequential processing and floating-point accuracy for the common backpropagation algorithm to minimize Root Mean Squared (RMS) error. The weights were then converted into integer values with 3 decimal bits for use in the on-MAP® Reconfigurable-environment ANN (RANN). The ANN was trained on 5 different representations of preprocessed LPI signal bitmaps generated using the open-source Linux tools 'bitmap' and 'bmtoa'. These same bitmaps were used as testing data for the RANN to check for accuracy.

4. Alternative Image Classification Method

An alternative method of image classification is provided that uses Exclusive-Or (XOR) logic to directly compare stored images against the input. This method takes advantage of the ability of reconfigurable processors to conduct numerous parallel processes to achieve considerable speed gains. The five images previously used for the ANN training and testing are stored as sixteen lines of 64-bit data to maximize bandwidth use. Each line of the input image is XOR-compared with the respective lines of the stored images. The result of the comparison is then tallied to count the number of ones, which represent differences between the input and stored image. Because matching images produce zero ones, exact matches are quickly and easily found with this method.

A threshold is applied to ensure that a stored image is not paired when it differs more than ten percent of the total pixels from a stored image, thus providing an indication of uncertain output.

C. THESIS ORGANIZATION

The remainder of this thesis is organized as follows:

- Chapter II discusses previous work in ANNs, and a background in the requirement for image classification.
- Chapter III discusses the specifics of the images generated for this application and the sequential-processor ANN used to generate weights.
- Chapter IV examines the ANN design used on the SRC-6 reconfigurable computer.
- Chapter V displays the results of the SRC Neural Network against a similar network run on a sequential processor in floating-point arithmetic.
- Chapter VI examines the XOR comparison method of image comparison, and provides initial results.
- Chapter VI provides an overall summary of results, the conclusions drawn from those results, and potential future work.

II. BACKGROUND

A. NEURAL NETWORKS

1. History of Development

The ANN is inspired by the brain. Hermann von Helmholtz, Ernst Mach, and Ivan Pavlov made significant contributions to neural research at the beginning of the 20th Century that led to ANN development [5]. While non-mathematical in nature, the work done by these early pioneers was instrumental in development of the concepts used later in ANN development.

The models developed for the brain's data processing centered on the way that neurons are interconnected and communicate. The key concept developed that the connectedness of neurons allowed a large number of simple simultaneous processes that result in the complex processing capabilities of the brain. A typical processor of a home computer can conduct numerous sequential instructions per second, but the ability to do this processing in parallel is limited by the fundamental structure of the processor itself. The motivation for development of a neural processing model is probably best described in the 1988 Defense Advanced Research Projects Agency (DARPA) Neural Network Study:

> At its most fundamental level, interest in neural networks is prompted by two facts: (a) the nervous system function of even a 'lesser' animal can easily solve problems that are very difficult for conventional computers, including the best computers now available, and (b) the ability to model biological nervous system function using man-made machines increases understanding of that biological function [6].

Work in neural networks therefore seeks to accomplish with multiple complex connections and simple processes what cannot be done with complex processors with simpler connections. The goal is the construction of systems that can do the jobs that sequential processors historically did not usually do well, such as complex control problems, stock market prediction, and image classification. These systems are closely based on what we know about neurological function. An average biological neuron contains dendrites that accept input signals that are then processed in the cell body which transmits a single signal on an axon to synapses. Similarly, the average artificial neuron is

a simple processing element that contains weighted input connections to a summation/threshold node, providing a single output. Figure 4 shows the similarities inherent in this relationship.

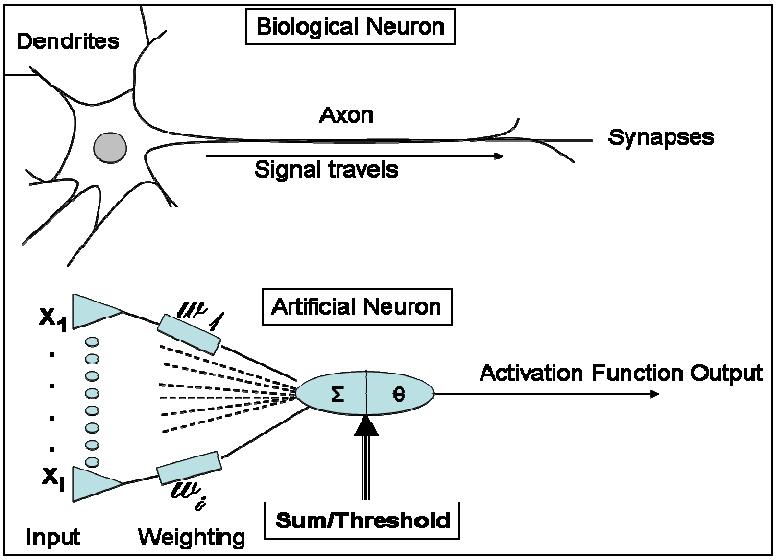


Figure 4. Biological vs. Artificial Neurons

An ANN is therefore an interconnected group of artificial neurons arranged in some type of architecture. A common architecture and the one used for this project is the Feed-Forward MLP, shown in Figure 3.

ANNs did not begin to thrive until the development of the backpropagation algorithm in the early 1980s, which seems to have happened simultaneously by different researchers [5]. This development was crucial because it allowed effective *training* of a neural network of increased complexity. This development, along with the availability of relatively cheap and powerful computers allowed the influence of neural networks to rise, gaining the prominence of neural networks seen today in everything from spam filters to speech recognition.

2. Multi-Layer Perceptron Networks

a. Basics of Multi-Layer Perceptron Design

While there are a number of different ANN architectures available, the Multi-Layer Perceptron (MLP) architecture was chosen for two primary reasons. First,

the MLP network is among the most popular applied networks available, and therefore is represented well in the available literature. Second, the MLP network is capable of handling a large number of inputs without extreme interference from the *curse of dimensionality* [7]. What this essentially means is that a MLP network scheme is better suited for handling large amounts of potentially redundant inputs without adding increased hidden layer requirements. This particular project required the ability to handle potentially large amounts of input since the assumed input was 1024 pixels in a 32x32 bitmap. A more detailed discussion of the particular problem of dimensionality is addressed in [7]. Finally, the output of these networks can allow for 'uncertainty' if the network is trained with "One-of-C" outputs. This assigns an active state to one of C different outputs only in the case of a correct classification. Thus, because only one output should signal due to a certain class of output, the presence of more than one signal can imply uncertainty of the network in classification. Human operators are therefore alerted to examine the image themselves and help protect against false classification.

The design of a MLP ANN incorporates a version of the artificial neuron shown in Figure 4. Inputs to the artificial neuron, or 'node', are typically multiplied by a weight specific to that input for that particular neuron. The weighted inputs are then summed together and the result applied to a transfer function. The transfer function can theoretically be of any type, from step functions to sinusoids. Experience with ANN use, along with the development of the backpropagation algorithm, tend to limit the useful transfer functions for a MLP into a few particular types. This is due to the desire to have outputs of a specific range in addition to having a transfer function that is differentiable. A differentiable transfer function is an essential component of the backpropagation algorithm. Transfer functions whose derivative function output is easily calculated without large amounts of arithmetic steps are valuable, as this capability aids in quicker calculation during backpropagation training. Some commonly used transfer functions are the linear, sigmoid, and hyperbolic tangent, shown in Figure 5.

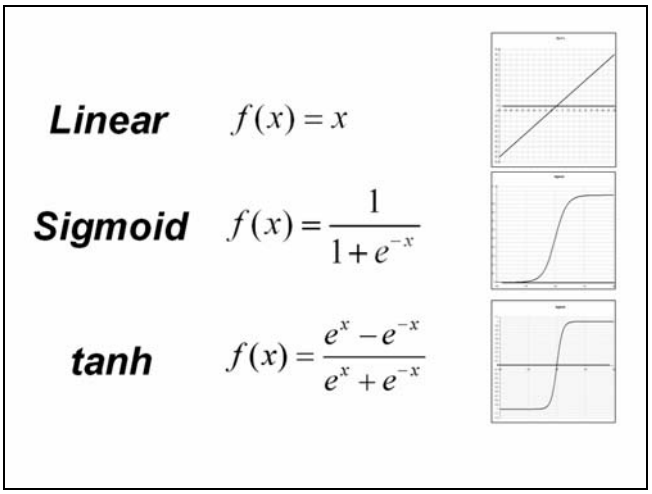


Figure 5. Common Transfer Functions

b. Backpropagation in Detail

Consider backpropagation. It is important to note that the standard backpropagation algorithm is a simplification of the Least Mean Square (LMS) algorithm developed by Bernard Widrow and Marcian Hoff for single-layer networks in 1960 [8]. The LMS algorithm represented a change in focus from selecting weights to achieve particular network outputs via the perceptron learning rule to incremental shifting the weights based on minimization of mean-squared error between desired and observed output. This is fundamental in that it shifts the decision boundaries in the network away from the training set output areas, therefore allowing greater generalization of the network and less susceptibility to noise [8]. The algorithms proved valuable for signal processing, but because they were designed for a single-layer network a generalization was required to adapt the algorithm for multi-layer network training [8].

Backpropagation uses a variant of the LMS algorithm called steepest descent. While the details of the derivation of these variants can be found in available resources like [4], there are a couple of important details to discuss. Steepest descent seeks to alter weights so that the output moves in the direction of the gradient of the error function. The learning rate ά determines how far in that direction steepest descent will move in one training iteration. The primary result of this is that unless ά is within a correct range, steepest descent will probably not minimize the error to a global minimum. If ά is too small, then a global or local minimum may not even be found. If ά is too large,

the algorithm is unstable and will not converge at all. Two simplified contour maps that illustrate this concept are shown in Figure 6:

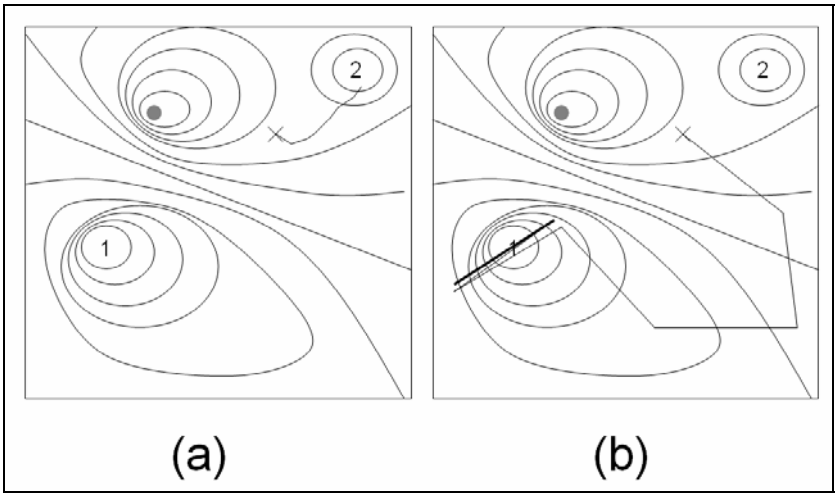


Figure 6. Learning Rate Effects with (a) smaller and (b) larger than desired rate

The above figure demonstrates an imaginary error function contour map, with the global maximum at the grey shaded circle, global minimum at '1', and local minimum at '2'. We see that when the learning rate is too small steepest descent/backpropagation tends to descend into a local minimum. A lower learning rate shortens the 'jumps' taken with every iteration, which increases the time it takes to train the network to achieve a minimum. In relatively 'flat' areas of the error surface, a low learning rate can stall without finding a minimum at all, due to the reliance on the gradient. When the learning rate is too large, the algorithm may never settle close enough to the global minimum to provide effective results, oscillating around the minimum but not reaching it.

The first step in backpropagation training is to propagate a set of inputs corresponding to a known output through a network beginning with random or pre-selected weights. The outputs obtained are then compared to those that are desired to obtain the error, that is then fed backwards through the differentiated transfer functions multiplied by a learning rate, as well as each connection weight to determine individual sensitivities for each node at each layer. These sensitivities are then used to update the

weights in an attempt to shift the output error to a global minimum. Each iteration of backpropagation training is referred to as an *epoch*. For a set number of epochs, different weight initializations can result in different RMS error at the output, depending on whether the backpropagation algorithm converged, encountered a local minima, or managed to reach the global minima. Exactly what constitutes an 'acceptable' RMS value depends on the consumer of the output. Increasing the number of training epochs can reduce the RMS error on trained values, with the additional increased probability of *overfitting* the network to the training data. What this means is that when the network is exposed to actual input after training, minor aberrations in the input from noise or other sources can result in very different output than expected, because generalization of the network was lowered by the increased amount of training to a specific type. The aforementioned information regarding ANNs is discussed with greater detail in [5], [6], and [7].

B. IMAGE CLASSIFICATION

1. Current Research

Image classification covers a wide range of applications currently used in business and government. For example, a demand exists for tumor detection in X-Ray and Magnetic Resonance Imaging (MRI) images, usually performed by doctors visually scanning images themselves. The demand for automatic classification in this example is for use as a pointer, aiding doctors to see potential trouble areas they might have otherwise missed due to the difficulty in visually searching tissue scans for cancerous growth [9]. Another example of an application for automatic image and pattern identification is in the field of biometrics, specifically fingerprint identification. In this field, automatic identification methods are used to save time, especially for the purpose of fingerprint matching in homeland security and police applications [10]. Detections of targets of interest in satellite imagery are yet another example where an automatic image classifier would help government and military users to make the best use of their available data.

2. An Application for Detection of LPI Emitters

The LPI Emitter detection method used in [1] requires some form of classification in order to make use of the output of the QMFB. As discussed by Professor Phillip E. Pace in *Detecting and Classifying Low Probability of Intercept Radar*:

> The presentation of the QMFB results to a trained operator will allow the signal parameters to be extracted, and can enable good classification results when the information from several layers is combined. [11]

Thus the classifications of QMFB results provide the most information when a trained human and time to extract information is present. As shown in Figure 7, the QMFB method can produce a contour image frequency-time plot:

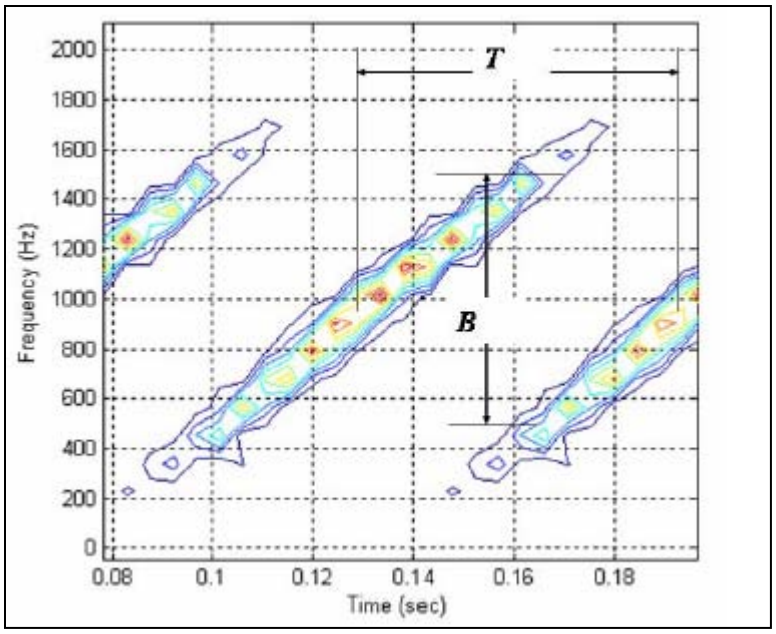


Figure 7. QMFB Contour Frequency-Time Image (From [1])

Professor Pace, however, predicts the future arrival of Anti-Ship Cruise Missiles (ASCMs) equipped with LPI seeker heads [12]. This development would dramatically reduce the time available for trained operators to extract information. This suggests a requirement for automatic classification of the QMFB output to reduce the time required to extract actionable information.

THIS PAGE INTENTIONALLY LEFT BLANK

III. IMAGE AND NETWORK WEIGHT GENERATION

A. IMAGE GENERATION

1. Image Source

Since the goal of this project is to develop an ANN capable of correctly classifying images that were extracted from a QMFB and run through a preprocessing step, it was essential to obtain images with which to train the network. This work was conducted simultaneously as the work on the data input, QMFB, and preprocessing steps. Thus, there was no immediate way to obtain actual preprocessed QMFB products from sample waveforms via the SRC hardware during the design timeframe of this project. While MATLAB code could be used to obtain values, one of the key benefits of a ANN is that it can be retrained on new data. With that in mind, the decision was made to generate 32x32 pixel images based on sample preprocessed QMFB outputs displayed in [1], using file formats directly compatible with the SRC-6 computer.

The program used to generate the images was the open-source Linux tool 'bitmap' written by Davor Matic, MIT X Consortium [13] and contained in the standard Red Hat Linux distributions. 'Bitmap' provides a simple interface that allows the user to expressly set grid widths and lengths and therefore was useful in producing an accurate canvas with which to create sample training images. The added benefit of using 'bitmap' was the use of the 'bmtoa' tool, also written by Davor Matic and included in the distribution that directly allowed conversion of 'bitmap'-created images into American Standard Code for Information Interchange (ASCII) files with characters that represent pixel color. With these two tools available free of charge and readily accessible on the computer, it was simple to design bitmap data files visually on a canvas and then convert the files to represent the planned output format from the preprocessing code.

2. Selection of Training Images

In order to provide for 'uncertainty' in the output as previously discussed in Chapter II, Section 2a, five outputs were selected for the neural network architecture with a "One-of-C" setup. Therefore there would be 5 categories that could be trained for selection by the network. In order to train the network to recognize 'no signal' as a valid category, only four actual patterns were generated. These were the P4, T4, T3, and T2 as

discussed in [1]. Representations of these signals were created using 'Bitmap' in order to test network training and response. These are shown in Appendix A. It is important to restate that superficial differences between the images generated for testing and actual output from the QMFB and threshold programs is irrelevant at this stage of research. The neural network is set up to accept new weights as a programmed-in requirement. The concept is that the actual images for certain patterns will be used to train these new weights in future applications.

B. WEIGHT TRAINING SEQUENTIAL-PROCESSOR NETWORK

1. Background

One of the strengths inherent in an ANN is the capacity for 'learning'. During supervised training of a MLP ANN, training inputs are paired with desired outputs and backpropagation, or some other training algorithm is used to adjust the connection weights until the network performs reliably. Thus, weight adjustment is a critical component of how the network will perform after training.

While the SRC has macros designed to handle floating point, a significant time savings can be realized by conducting all mathematical operations in fixed-point integer. In addition, floating point operations, if instantiated on the MAP®, can result in costly space allocation. For example, if we were to use the standard sigmoid presented earlier: $sig(x) = \frac{1}{1+e^{-x}}$, we see that there are three floating point operations that must be conducted on the MAP®. These are the exponential function, addition, and the division. The problem arises when we consider the amount of space required on the XC2V6000 FPGA for these operations. A single 64-bit floating-point divide occupies approximately 1/8 the entire FPGA logic. The exponential function occupies 3-8% of the FPGA space. This would place a severe constraint on each node in the network in just the sigmoid transfer function instantiation itself, let alone storage or connection weighting. While a single sigmoid function could be pipelined for use by every node, this would cost clocks and degrade from the objective of trying to make the classification run as close to real time as possible. An alternate solution is to create a LUT representation of the sigmoid function in fixed-point, providing the quantization error incurred is acceptable. While

this is the method eventually used in the network, it, and fixed-point calculation in general, presented a problem for effectively training the network. It is important to note here that research at NPS is being conducted by LCDR Tom Mack and Professor Jon T. Butler in creating high-precision function generators in the SRC-6 reconfigurable environment [14]. This methodology is discussed in more detail in [15]. Thus, the potential exists to further refine this network using the tools currently in development, because the sigmoid is one of the functions researched in this work.

Chapter II discussed the ramifications of ineffective learning rates on the network. Whenever fixed-point integers are used in place of floating-point, quantization error occurs. For example, if two bits of decimal point are used in fixed-point, the maximum quantization error is ±.125, because the two bits can only represent increments of .25: 0, .25, .50, .75. While more decimal bits can be used to gain greater precision, floating point notation is designed to handle precision. Training a network is inherently susceptible to errors in precision, because without a precise enough application of the learning rate the system may never converge to a minimum. In addition, errors in precision limit the effective calculations during each iteration, potentially increasing the amount of epochs required by a significant amount. For execution of a well-trained network, however, precision is less significant. In a network with average levels of generalization, quantization error will be treated as noise by the network and the network will produce the correct results. This presented a dilemma of whether to use a fixed-point system for a quickly-executing network with potential training problems, or use a floating-point system for a slower-executing network that may not fit on the MAP® but is able to train effectively. The solution to this dilemma was to incorporate the best aspects of both systems, and avoid the problems by separating the training network from the execution network.

2. Sequential Weight-Generation Program Design

A Feed-Forward MLP ANN is unique in that once the weights are set to acceptable execution levels by an effective training session backpropagation is no longer required. With this concept in mind the decision was made to separate the RANN training from the network envisioned to actually classify the data obtained from QMFB

preprocessing. This approach was based in part on inspiration derived from a graduate project by Steffan Nissen regarding a Fast ANN design [16].

The sacrifice made by this decision is the loss of real-time training of the network, because new desired image-output pairs would be required to run first in a C++ model of the execution network in order to allow for weight generation. Since the alternative was a network that potentially was unable to converge to minimum error or operate slower than conventional sequential-processor neural networks, this sacrifice was determined as acceptable.

To construct the weight-generation program, some public-domain neural network source code written by Dr. Phil Brierly was used as a base [17]. The original code is included in Appendix B, while the code specifically used for weight generation is included in Appendix C. The weight-generation code includes the training and testing bitmap arrays defined within the actual source, as 'trainInputs' and 'testInputs' respectively, for the sake of reproducibility and traceability. It is inferred that these arrays will actually be populated from data extracted from the preprocessing step on the SRC and thus a minor modification to the code will be required. Likewise, the selection of number of epochs and learning rate may have to be adjusted when new data is presented to the network to match desired RMS error and training time. A sample output from this program is included in Appendix D. The sample output has been truncated in several areas for the sake of brevity, because the weights are randomly initialized each time the weight-generation network runs. Therefore, the output will be unique each time and thus not reproducible. The purpose of Appendix D is to show an example of the data available after every run. One important item of note is the time required for training, that in the particular case of the run shown in Appendix D was 8.57 seconds for 1000 epochs. This large delay requirement is a major reason why the network training was shifted off-MAP®. With such a large delay, real-time computation on the SRC is impossible.

3. Program Operation

The first step in the program is the initialization of the weights with random numbers via the function call 'initWeights()'. The desired training outputs are then initialized via the 'initData()' function call. This is the function call that should contain

training image file accesses for loading the 'trainInputs' array in future designs of the weight training program. The program is designed to conduct all training of weights specifically to the 'trainInputs' array. Next, the program enters into the epoch loop for training. During each epoch, the program selects patterns at random and propagates them through the network via the 'calcNet()' function call. The output array, 'outPred' is then compared to the desired output array 'trainOutput' to obtain the error array for that particular pattern, 'errThisPat'. This error is first backpropagated through the 'WeightChangesHO' function call to adjust the hidden-to-output layer connection weights, then backpropagated through the 'WeightChangesIH' function call to adjust the input-to-hidden layer connection weights. Once a number of patterns equal to the array size have been randomly selected, propagated and backpropagated, the program calls 'calcOverallError' to calculate the overall RMS error for that epoch. An if-then statement is used after the 'calcOverallError' function call to determine whether to print the RMS error or not. This statement is user-adjustable by merely changing the modulus division divisor, currently set to print error every ten epochs. Note that all steps in this weight training process involve floating-point variables to maximize the precision of training. In addition, instead of using a for-loop linked to the number of desired epochs for training, a while loop can be substituted and linked to the overall RMS error. The training section of the program has clock reads before and after in order to provide timing data specific to the training process itself.

The program then converts the floating point weights to 3-decimal point integers using the function call 'Integerize', that simply multiplies the floating point values by eight and casts them as integers, discarding the remainder. This methodology incurs a maximum quantization error of .075 and if higher weight precision is later desired, this portion can be modified to produce a multiplication of 2^x in order to provide *x* integer decimal bits. Care should be taken to ensure against overflow, however, as these 32-bit weights will later be added on the MAP® and steps have not been taken to limit overflow other than to limit the amount of decimal bits in the integer values. In an effort to compare precision between the 'integerized' weights and pure-floating point operation, the integer weights are run through the network with the 'intcalcNet' call and results displayed with the 'calcIntError' call. As seen from the example in Appendix D, the

integer weights provide an overall RMS error of .25, while the thousandth epoch floating-point RMS error was 0.112453. The increase in RMS error from integer weights may be acceptable depending on the application. In this particular case which uses 'One-of-C' outputs, a classification is still visible in the output and thus was acceptable for these purposes. The effect of integer weights on the output of the ANN will be discussed more thoroughly in Chapter V.

The final section of the program prints the integer values in 64-bit hexadecimal format to a file called 'weightout'. This file is separate from the output contained in Appendix D, which nominally outputs to screen but can be redirected to a file in the execution call with the '>>' Linux redirector command. 32-bit weights for the nodes are paired together in a single 64-bit value to maximize the use of communication bandwidth into the MAP® from the OBM. Nodes zero and one for the input-to-hidden connection are paired together, followed by nodes zero and one weights for the hidden-to-output connection. Nodes two and three are likewise paired and follow immediately after. Finally, both sets of node four weights are padded with 32 zeros and written to the file. This padding can be removed and replaced with additional node weights if more nodes are added to the network in later work. Once trained, these weights are not envisioned to change, and thus the RANN can continuously run with the same weight file if optimal settings are determined and found.

IV. RECONFIGURABLE-ENVIRONMENT ARTIFICIAL NEURAL NETWORK (RANN) DESIGN AND OPERATION

A. DESIGN OVERVIEW

The design goals for the RANN code were speed of execution, reusability of code, and minimized use of MAP® resources. While programming in the Carte™ environment aided the implementation of some processes, incorporation of VHDL code was also required to meet these design goals. Thus, several design decisions were made in the process of creating the ANN architecture. These are discussed below.

1. Network Input

There are two required input sources for the RANN to enable execution. The first is the weight-generation output file 'weightout' from the program discussed in detail in Chapter III. The second is the image output from the QMFB-preprocessing steps.

a. Connection Weights File

While the RANN is currently designed to access this file in the same directory as the SRC executable, the main.c source can be altered if this setup proves unwieldy in the future. It is essential for the current software, however, to have a trained-weight file set up in the format described in Figure 8.

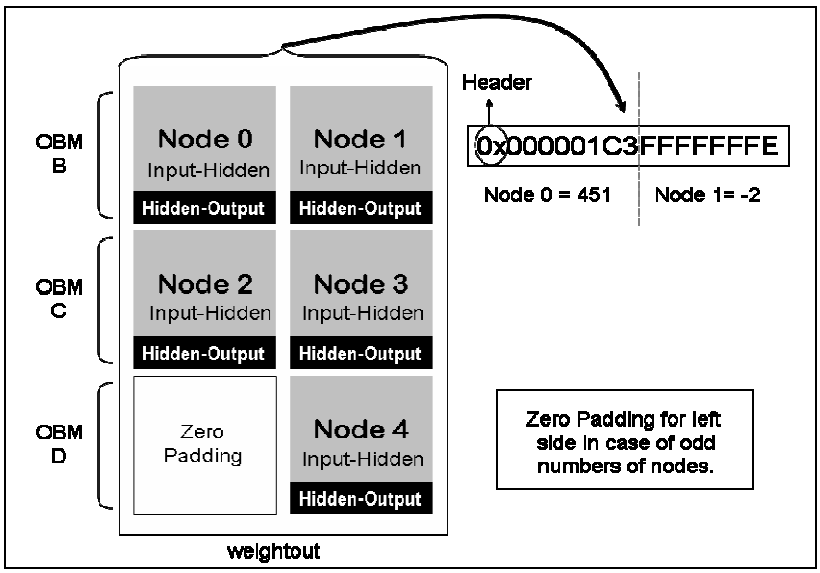


Figure 8. File Design Architecture for 'weightout' Program

The purpose of stacking nodes side by side is to take advantage of the maximum bandwidth available to OBM memory reads. Because one 64-bit long word can be read back from an OBM per clock, it makes sense to combine two 32-bit integers in the same read. These integers are then easily extracted using the user-callable macro 'split_64to32' on the MAP® processor as a matter of bit routing on the FPGA. It is important to restate that these weight values are the 3-decimal bit integer values produced by casting a floating-point value to integer that has been multiplied by 8.

Because the current architecture is designed with 5 hidden layer nodes and 5 output layer nodes, the 'weightout' file is set up with the input-to-hidden layer connection weights for a particularly numbered node to be followed by the hidden-to-output layer connection weights for the similarly-named node in the output. This was merely a convention in placement, because the weights are placed in OBM banks by main.c, and can theoretically be placed in any order providing the reconfigurable-specific code is designed to obtain them correctly. For example, the zero-padding area can be used for additional input-to-hidden layer connection weights if an additional hidden layer node is added, or filled with additional sets of hidden-to-output layer connection weights if multiple output nodes are added. The current iteration of the main.c program places the first 'set' of weights into OBM Bank B, the second in C, and the last in D, as shown in Figure 8. This was required to limit the number of accesses to OBM in an effort to speed network execution. The requirement for a larger number of network nodes can potentially increase clock speed, as OBM banks will incur multiple accesses. A possible solution to this is discussed in the 'Future Work' section of Chapter VII.

b. Preprocessed Image Input

The preprocessed image input from the QMFB is the data that the network will classify. As previously discussed, these files were self-generated due to inaccessibility to actual data at the time the network program was written, using the 'bitmap' and 'bmtoa' tools. The original 'bmtoa' output files were altered from a binary ASCII file to a 32-bit hex padded with 32 zero bits ASCII file to conform to the output format used by Ensign Brown and documented in [3]. This was accomplished with a simple conversion program contained in Appendix F.

The main.c file currently requires an argument consisting of the file name of the 64-bit hex ASCII image file. At execution, the implicit executable main.c places this file's data into OBM bank A. The explicit executable is thus able to strip the zero padding off in the same manner as separating the two 32-bit integer weights for the connection weight data using 'split_64to32' and discarding the padding. The image data is then available for propagation through the network.

2. Input-to-Hidden Layer Processing

The input-to-hidden layer processing consists of two distinct steps. The first is the connection weighting and summation of the input image data for each hidden layer node. The second step is the sigmoid transfer function processing, which is essential in introducing nonlinear response to the network.

a. Hidden-Layer Connection Weighting and Summation

A typical MLP ANN architecture, such as the weight-generation program, processes the input in a fairly standard manner. Each input is usually multiplied by a connection weight specific to a particular node and then the weighted inputs for each node are summed together to produce an input to the transfer function. Because the required input image was 32 bits in height and 32 bits in length, this would result in 1024 multiplies and indeed the weight-generation program accomplishes hidden-layer processing in this manner. The Carte™ environment, coupled with the fact that the input is binary, allows the multiplication and summation to take place in the same process, using an accumulator macro supplied by SRC. Because multiplication is irrelevant with a multiplicand of zero or one, the input image data is used as an enable for 5 separate accumulator macros. The output of each accumulator is designated for a particular hidden layer node. A graphical representation of this setup is shown in Figure 9. The use of this particular arrangement allowed complete weighting and summing for all five hidden layer nodes within 1067 clocks, primarily due to OBM memory data access timing for each input-to-hidden layer connection weight. This execution time has the potential to be halved in future work in a methodology discussed in Chapter VII.

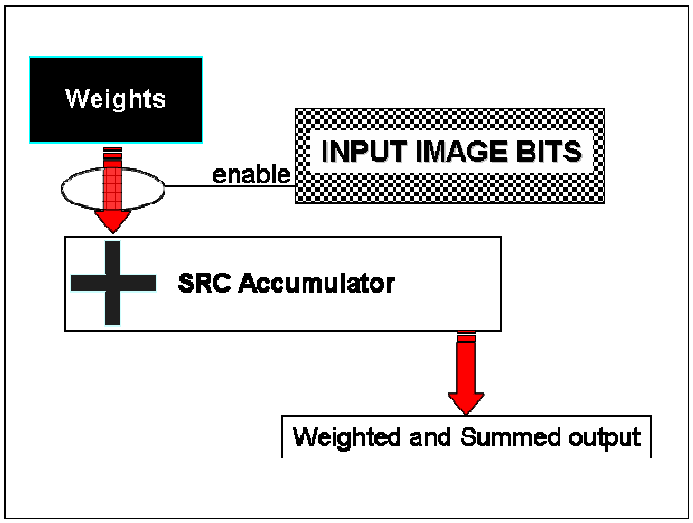


Figure 9. Hidden-Layer Weight Accumulator

b. Sigmoid Transfer Function Processing

The use of a sigmoid transfer function is an essential component of this ANN design. First, the transfer function provides the capability for nonlinear response, increasing the capability of the network. Second, the transfer function allows the hidden layer output to be bounded between zero and one, aiding the network in avoiding unintentional integer overflow. A potential detriment of using this function, however, was the potential loss of speed in terms of producing the output. Recall that the sigmoid function is $sig(x) = \frac{1}{1+e^{-x}}$. Realization of this functions output via mathematical processing would be complex and potentially costly in time. Thus, the decision was made to encapsulate this function via a VHDL macro that would act as a LUT. Because the sigmoid function is bounded between zero and one, a four decimal bit output is used as a compromise between greater precision and LUT size. With weight values incurring quantization error from 3 decimal bits anyway, increasing the sigmoid function past four decimal bits was also considered to have questionable benefits. The sigmoid function VHDL code, black box file, and info file are contained in Appendix G. The sigmoid function is referred to as "SIGFOUR" by the explicit program, and executes as a pipelined user macro with a latency of zero, significantly speeding the

process output. The function is able to be called with a latency of zero, as it is encapsulated as a VHDL 'process', its operation triggered by the change of the input variable.

3. Hidden-to-Output Layer Processing

The hidden-to-output layer weights were used to populate a two-dimensional array called 'wt2' in the explicit code, thus instantiating the array in BRAM. This array is populated with the hidden-to-output layer weights contained in OBM multiplied by the corresponding sigmoid function output for each hidden layer node. Once the array is populated with data, accumulators are again used to sum the five inputs to each output node, thus producing five outputs in a "One-of-C" configuration. This is shown in Figure 10:

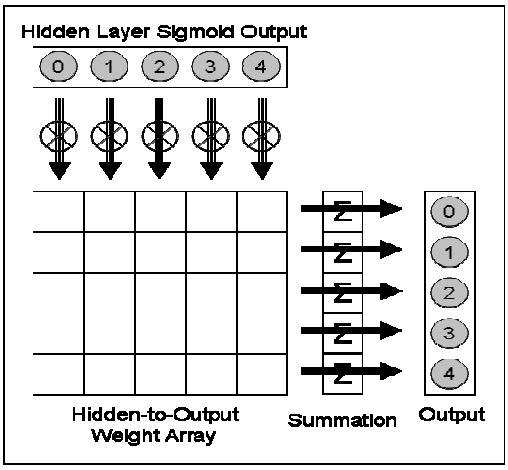


Figure 10. Hidden-to-Output Layer Processing

The figure shows the hidden-layer sigmoid output being individually multiplied with each particular row weight in the corresponding column. After multiplication is complete, each new individual column value in a particular row is summed to provide the output for each node. Thus, each output node receives an individually-weighted set of outputs from each of the hidden-layer nodes, maintaining the interconnectivity inherent to an ANN. An array is chosen to enhance reusability of code. The array must be changed if the number of hidden layer nodes or output layer nodes changes. For an array of weights, this involves changing the array values in the wt2 declaration, instead of adding additional individual variables that represent each of the

array squares shown above in Figure 10. Contained within the source code is a commented-out section that uses these individual variables instead of an array. For a 1024x5x5 network using individual variables, a savings of 69 MAP® clocks was realized. In the interest of reusability, the array is used but if future work uses the same 1024-5-5 architecture, reversion to individual Hidden-to-Output weight variables may provide better performance, as it separates the data into different BRAM blocks that allow for simultaneous access.

The final output from the RANN MAP® code is a 32-bit integer with seven decimal bits, resulting from the four decimal bit sigmoid outputs multiplied by the three decimal bit Hidden-to-Output layer connection weights. Thus, the output can be used by itself or converted to floating-point and divided by 128 to produce the 'actual' output. The current iteration converts to floating point on the sequential processor code in order to provide a base for comparison to the output of the pure floating-point network.

V. NETWORK PERFORMANCE COMPARISON

A. PERFORMANCE COMPARISON METHODOLOGY

To adequately estimate performance of the RANN, a specific methodology was devised in the interest of standardization to previous comparable work. In this vein, the decision was made to compare speed of execution similar to how Ensign Brown compared speeds in [3], with the exception of omitting MATLAB performance. The decision to omit MATLAB performance is due to the understanding that, as an interpreted language, the speed of execution was assumed to automatically be less than equivalent code in standard C++.

For the purpose of comparison, the weight generation C++ program was used as a basis for the sequential-processor timing. This was accomplished with the addition of a loop at the end that assigns a pattern number sequentially and then calls calcNet, the network floating-point propagation function. The use of the weight-generation program was due to the fact that essentially, the architecture is the same with the exception of the use of integers and features specific to the Carte™ programming environment. Thus, an accurate comparison can be made between a floating-point sequential neural network and the RANN.

Ten thousand floating-point propagation trials were run, which amount to two thousand of each of the five standard inputs sequentially. The timing before and after were made in a manner similar to that used by Ensign Brown in [3], in an effort to standardize the results observed from SRC conversions. The result from the sequential-processor network was 1.02 seconds for ten thousand runs, which equates to 102 μs per network execution. These results can be observed in the last line of Appendix D, the weight generation code output

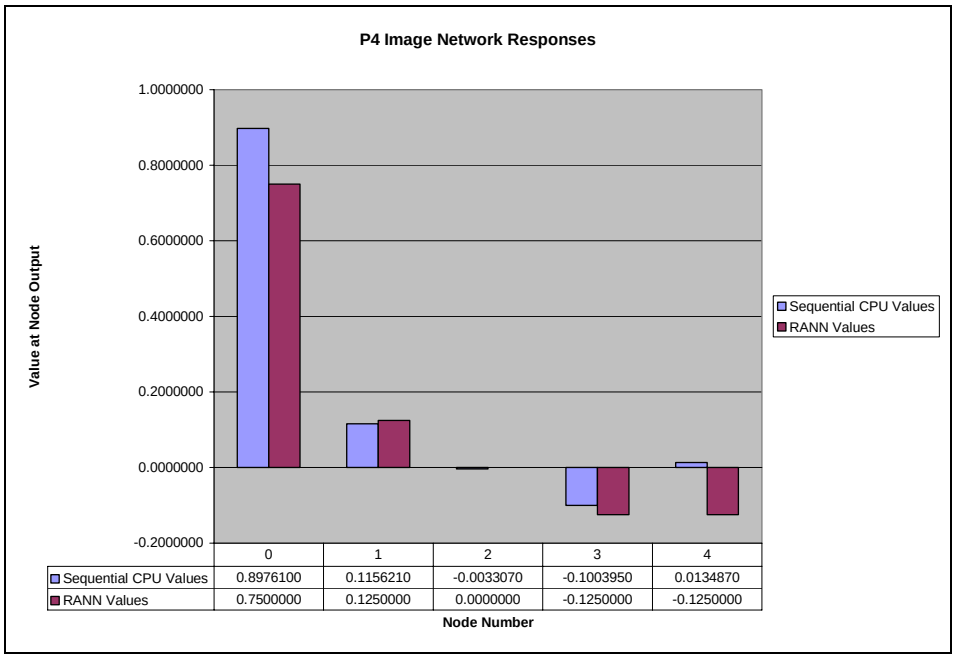
Conversely, the reconfigurable code runs at a standard 1149 clocks per iteration, which equates to 11.49 μs per network calculation given the 100 MHz clocking speed of the MAP®. In terms of speed of processing, execution of this RANN format takes 11.26% as much time to execute as the same network running on a sequential processor.

Please note that for different images of the same size these numbers would remain the same, because the amount of weights are fixed and thus the propagation merely becomes an issue of math processing timing.

Execution Hardware	Time (µs)
Sequential Processor	102.00
Reconfigurable MAP®	11.49

Table 1. Network Execution Times

From the data, the RANN outperformed the existing architecture by approximately a factor of ten on the basis of speed of processing. This is somewhat mitigated by the potential increase of RMS error incurred via the use of fixed-point variables on the reconfigurable hardware, but the comparisons of actual network output for the P4 image visually by bar graph in Figure 11 show that in both cases, the classification can be clearly discerned regardless of RMS error. All test images show comparable results and are available in Appendix H. The actual values of overall RMS error on each image for both types of network are shown in Table 2.



	0	1	2	3	4
Sequential CPU Values	0.8976100	0.1156210	-0.0033070	-0.1003950	0.0134870
RANN Values	0.7500000	0.1250000	0.0000000	-0.1250000	-0.1250000

Figure 11. P4 Image Network Output Comparison

	RMS Error Values for Images				
	P4	T4	T3	T2	NoInput
Sequential FP Proc.	0.0826122	0.1077632	0.0229733	0.1071012	0.052627
Reconfigurable MAP®	0.147902	0.125	0.0790569	0.1936492	0.11848

Table 2. RMS Error Value Comparison

B. SRC-SPECIFIC PERFORMANCE

The RANN code was designed with a goal of minimizing the demand on the MAP® hardware. The reason for this is that this project was envisioned to run as a parallel section simultaneously with the data input program created by Captain Stoffel [2], along with the preprocessing program created by Ensign Brown [3]. Because estimated hardware demands were initially envisioned by all three researchers as large, a necessary design goal that materialized was the minimization of those hardware demands so that each of the three sections could run simultaneously without impacting the operation of the others.

One of the products of compilation in the Carte™ environment is the creation of a log that summarizes the exact hardware demands that the program will incur. For the RANN, this summary provided the following data:

Logic Utilization:

Number of Slice Flip Flops: 8,858 out of 67,584 13%

Number of 4 input LUTs: 5,710 out of 67,584 8%

Logic Distribution:

Number of occupied Slices: 6,689 out of 33,792 19%

The use of 19 percent of the slices available on particular MAP® was acceptable, as it left a full four-fifths of the slice untouched for parallel code instantiation. While the demand on OBM is large for this particular program, OBM usage for Ensign Brown's code is merely as a means of input and output, which can and is intended for substitution with data streams from Captain Stoffel's code to the RANN [3]. The use of OBM by Captain Stoffel's code as an intermediary storage mechanism for data extraction [2] and can potentially be replaced with streams as well.

C. SUMMARY

The increased speed gain of the RANN is directly attributable to several factors. First, the use of fixed-point math within the MAP® greatly simplifies the hardware. This not only decreases logic demands on the FPGA but also decreases the time required to obtain output. Second, the use of LUT approximations of the sigmoid transfer function eliminate large calculation demands and instead replace them with what is essentially an on-chip memory access. A four-bit decimal approximation allows this table to be manageable, without incurring exorbitant RMS error in output calculation. The only cost of the VHDL sigmoid approximation approach is the requirement for the initial user to construct and link the initial source code, along with the increased place-and-routing time incurred when the explicit and implicit code is compiled on the SRC with the 'make hw' command. As an effective VHDL source code was created and linked for this work, that particular requirement is mitigated. This program is also envisioned to be compiled once and only recompiled once a new weight set is generated, thus mitigating the longer compile-time. Finally, the MAP® hardware allows the simultaneous execution of several processes. For example, each of the five hidden layer nodes conduct an accumulate operation, enabled by the input image, once per clock. This same operation on a sequential processor must be done separately at each node, in sequence. These three core factors enabled the approximately tenfold speed performance observed with the RANN, and suggestions are made in Chapter VII as to how to increase these gains even more. A set of simplified instructions in the use of the RANN is included in Appendix I.

VI. EXCLUSIVE-OR (XOR) IMAGE COMPARITOR

A. RECONFIGURABLE PROGRAM DESIGN

A distinctly different method of image classification is a brute-force method of direct bit-to-bit comparison. For sequential processors, this method lacks elegance as there may be numerous images to compare against in selecting the correct match. The SRC Carte™ programming environment, lends itself to easily accomplishing this method in parallel, achieving significant clock savings. In this method, the bits in the input image are XORed with the corresponding bits of the stored image. A 1 in the resulting image corresponds to a difference between the input and the stored images. The "popcount_64" pure functional macro provided by SRC allows the single-clock counting of 1 bits in an input, providing an integer sum of this count as an output. This macro applied to the output of an XOR comparison provides an index of difference for each output. This concept is pictured below in Figure 12, where the sum of differences in this example would equal '2' from the two dark pixels which represent ones.

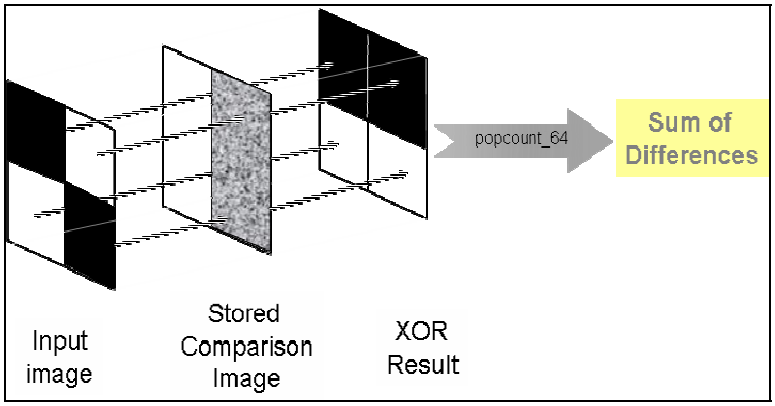


Figure 12. 'XOR-Mask' Comparator

To achieve the best possible speed in the Carte™ environment, the comparison images should each be stored as a 16-deep, 64-bit BRAM array instead of OBM memory. This configuration should allow simultaneous access per clock for each image with an incoming 64-bit preprocessed input, providing the input has been optimized by eliminating unnecessary bit padding, a recommendation that is discussed in detail in the

future work section. Declaring a constant BRAM array initialized with individual values, however, is a function recently introduced into the Carte™ 2.2 programming environment. During the course of program production, only the 2.1 programming environment was installed and therefore constant BRAM arrays were not used in this program.

In place of constant BRAM arrays, individual BRAM variables were used as shown in the source code, contained in Appendix J. The downside of not using arrays is the inability to loop the comparisons, as each individual variable must be called separately. This has resulted in particularly long code for a simple procedure.

The inherent advantage to the XOR comparison method is speed of execution, as potentially large numbers of images can be simultaneously compared and a result found in fewer clocks than that which is required by the RANN architecture. The disadvantage in the XOR comparison method involves demands on the MAP® hardware. Training a neural network with more images does not necessarily increase the amount of hidden layer weights, as it may only require more training epochs and the weights will be adjusted differently. The only reason more hardware demands will be made by the RANN architecture is if output response suffers and to compensate, a decision is made to increase hidden layer nodes, and thus input-to-hidden layer connection weights. Increasing the number of images for an XOR comparator will automatically require more BRAM, distributed Select RAM, or OBM memory to hold the comparison images and thus automatically places a larger burden on the hardware. It is important to note that only 5 comparison images were stored for this particular execution of the program

B. RECONFIGURABLE PROGRAM EXECUTION

1. Hardware Demands

The compilation log file for the XOR comparator shows an increased burden on the hardware, particularly in LUT usage:

<u>Logic Utilization:</u>

Number of Slice Flip Flops:	15,240	out of 67,584	22%
Number of 4 input LUTs:	10,337	out of 67,584	15%

Logic Distribution:

Number of occupied Slices: 9,151 out of 33,792 27%

This increase can be attributed the relatively larger amount of calculations taking place on the hardware compared to the RANN. There exist several ways to decrease this burden that are discussed in the future work section of Chapter VII.

2. Performance Gains:

The reconfigurable XOR comparator output for each of the five types of input images is provided in Appendix K. A sample output is provided below:

```
>./ex07 p4input64

65 clocks
Difference Output= Pattern 1(0)
Difference Output= Pattern 2(159)
Difference Output= Pattern 3(165)
Difference Output= Pattern 4(180)
Difference Output= Pattern 5(97)
Closest Match is Pattern 1
Which is: P4 Image
```

As shown above, this classification executes completely in 65 clocks, resulting in 650 nanoseconds per execution. This is significantly faster than the 1149 clocks required for RANN execution, and provides comparable output.

C. SEQUENTIAL COMPARISON PROGRAM

A standard C++ program was developed as a basis for comparison for the XOR comparator. The source code for this program is contained within Appendix L. The code for this program was written to achieve similar output to the reconfigurable comparator, as seen in the sample below:

```
>./xorcomp t4input

Time to complete 10000 trials (in seconds): 1.780t
Number of Different bits for P4 Image -->(160)
Number of Different bits for T4 Image -->(0)
Number of Different bits for T3 Image -->(224)
Number of Different bits for T2 Image -->(160)
Number of Different bits for No Image -->(160)
Since the lowest delta is 0, this image most closely resembles:
T4 Image
```

This program conducts 10,000 trials in 1.78 seconds, resulting in a timing of approximately 176 µs per run. This execution speed is actually 74 µs slower in execution than the ANN, potentially due to the time required to extract results from the XOR comparison. This program also uses a 32-bit image line width that increases the number of comparisons from 16 to 32. In this case, without the popcount_64 macro, the ones were extracted via modulus 2 executions followed by bit shifting by 1. This ones extraction method was thought to trivialize any gains obtained from using a 64-bit width image file, because the amount of iterations required to extract the ones would be the same. There seems to be a problem in the ones extraction as well. Although the program always selects the correct match, the ones values obtained from the sequential program for other images are not correct. While further refinement of the program could be conducted to reduce execution time, and ensure correct extraction, the point is that an architecture that excels in a reconfigurable environment does not necessarily do so in a sequential one.

VII. CONCLUSION

A. SUMMARY OF WORK

This thesis describes a proposed design for an ANN on the SRC-6 reconfigurable computer. Advantages inherent in this design are a tenfold speed increase, with limited and possibly insignificant increase in output error. There are several neural network architecture changes that enable these advantages. The first is separation of the weight training from network execution. The second is using LUT representations of the nonlinear sigmoid transfer function. The third is execution of the neural network in reconfigurable hardware to take advantage of parallel processing.

While ASIC components can and have been used to create neural networks, the potential speed increases are mitigated by the loss of flexibility. The RANN architecture lends itself to reusability and modification. In the case where an increase in the number of hidden layer nodes is required, the programs can be altered whereas a new ASIC would have to be commissioned. Therein lies the strength of the reconfigurable architecture, which is flexibility in response to changing demands. Increases that develop in the speed of FPGA clocking and the ability to conduct floating-point operations will only add to the strengths inherent in the reconfigurable computing domain.

The role of the RANN program as part of a comprehensive LPI detection system has been described. A discussion of LPI systems development and the requirement for detection capability is provided to show potential for practical value of the RANN in future military applications. The history and development of neural networks is given to provide background information for those unfamiliar with the technology. The science behind ANNs is provided to assist in understanding some of the difficult design decisions made in creating a network capable of being run in the SRC-6 reconfigurable environment. These design decisions have been discussed at length to provide understanding of the developed code, all of which is capable of being run in an open-source environment. Finally, performance data is provided that supports the conclusion

that neural networks can be run in a reconfigurable environment with substantial speed increases and comparable performance levels.

B. SUGGESTED FUTURE WORK

This thesis provides a number of different avenues for future work in the realm of signal processing, reconfigurable computing, and ANNs. These suggestions allow for the continuation of research in these areas.

1. Comprehensive Analysis of the SRC LPI Detection System

To date, the work conducted with regards to implementing the LPI Detection methodology outlined by Professor Phillip E. Pace in [1] has been separately conducted. Data input hardware and programming has been created by Captain Kevin Stoffel [2], preprocessing programming created by Ensign Dane Brown [3], and image classification programming via ANN is detailed in this work. The next step in creating and evaluating the complete system is joining all three programs to run jointly and in parallel. This was envisioned to be accomplished by having each program run in a parallel section, as described in section 5.9 of the SRC C Programming Environment v2.1 Guide [18]. Data would be transferred between sections with the use of streams, eliminating much of the use of OBM Memory Banks. Specific to this project, the use of OBM Bank A could be discarded and streams from the preprocessing step stored in BRAM for ease of access. OBM banks B, C, and D are still envisioned to be used to store weight values, because storing in BRAM may be precluded by the use of Multiplication blocks in Captain Stoffel's code [2]. A potential solution is the use of separate MAP® devices for the data input and preprocessing/classification codes, using the GPIO bandwidth to stream data.

The comprehensive analysis can also provide standardization of the bitmap image size based on constraints found in [2]. Because the input-to-hidden connection weighting drives the timing on the RANN, with 1024 accumulations costing approximately 1067 clocks, reduction in bitmap size may significantly increase network speed, at a potential cost in classification performance. Comprehensive analysis can also provide actual preprocessing images from simulated signals, that should result in a network that is trained to operate closer to real world data. As previously discussed, this implementation of RANN is trained on images approximated from waveforms contained in [1]. While

speed performance would not be expected to change, output performance would increase from the use of actual signal input in training.

2. Program Optimization

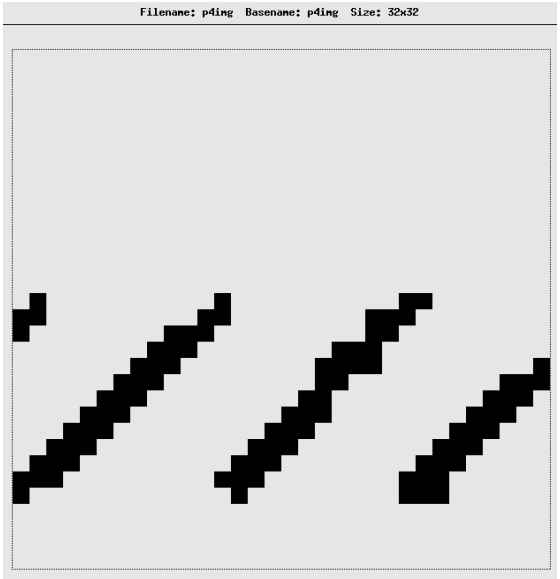
Another potential avenue for network performance becomes available in the case where all six banks of OBM are available for use by the RANN. The additional banks of memory can be used at maximum bandwidth by striping input-to-hidden weight values among the six banks, allowing more than 1 weight read per clock per hidden-layer node. This can potentially halve processing time of the network as a whole, because input-to-hidden layer weighting and summation currently is the largest boundary value in terms of processing time.

Another avenue for optimization involves the current output of Ensign Brown's code. Instead of using 32 64-bit words that are padded with 32 unnecessary bits the 32-bit outputs should be stacked, resulting in 16 64-bit outputs. These improved outputs maximize the use of streaming data bandwidth between parallel sections, and can easily be broken down into their constituent components with the 'split_64to32' macro. The reconfigurable XOR comparator was designed with this optimization in mind.

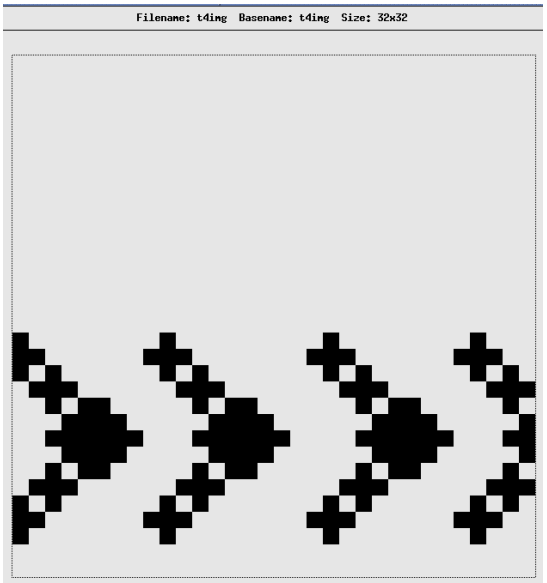
The large hardware requirements of the reconfigurable XOR comparator can be mitigated by a few optimizations. First, the planned upgrade of the NPS SRC-6 Carte™ programming environment to 2.2 will allow the use of constant BRAM arrays. Declaring a BRAM array with values already instantiated saves time since otherwise an array would have to be populated by OBM or streams from other parallel sections. Populating a BRAM array in this manner incurs a penalty as these values must be read from OBM or the stream. Arrays are valuable since loop variables can be used as indexes into the array, allowing looped reads from the array when several repetitious calculations are required. A replacement for using these arrays in this manner is declaring individual variables initialized with the desired values. Individual variables require loop unrolling, as there is no array to index using a loop counter. If the desired number of comparison images increases significantly, OBM storage of images is a potential alternative. While OBM use may sacrifice performance by limiting the number of data accesses per clock, the XOR-comparison methodology may still outperform the RANN.

THIS PAGE INTENTIONALLY LEFT BLANK

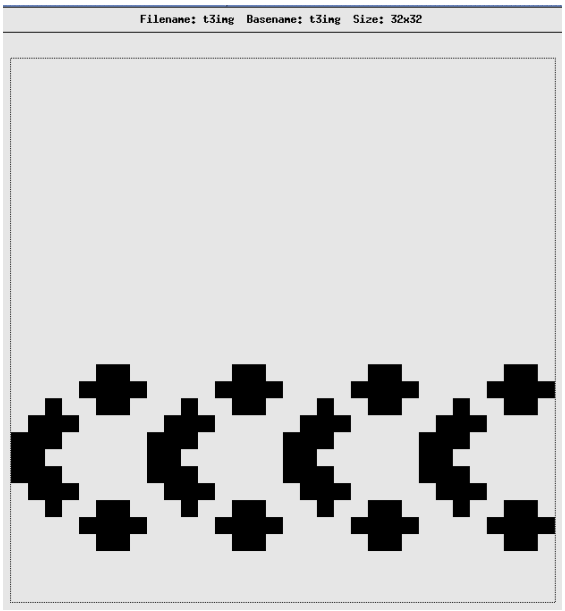
APPENDIX A. IMAGES CREATED FOR NETWORK TESTING



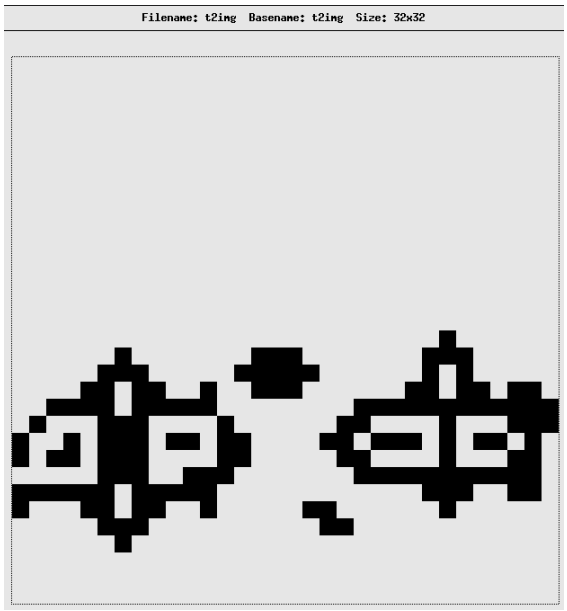
P4 Image



T4 Image



T3 Image



T2 Image

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. PUBLIC DOMAIN NEURAL NETWORK CODE

```
////////////////////////////////////////////////////////////////////////////
//MLP neural network in C++
//Original source code by Dr Phil Brierley
//www.philbrierley.com
//Translated to C++ - dspink Sep 2005
//This code may be freely used and modified at will
//C++ Compiled using Bloodshed Dev-C++ free compiler http://www.bloodshed.net/
//C Compiled using Pelles C free windows compiler http://smorgasbordet.com/
////////////////////////////////////////////////////////////////////////////

//#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>


//// Data dependent settings ////
#define numInputs  3
#define numPatterns  4


//// User defineable settings ////
#define numHidden 4
const int numEpochs = 500;
const double LR_IH = 0.7;
const double LR_HO = 0.07;


//// functions ////
void initWeights();
void initData();
void calcNet();
void WeightChangesHO();
void WeightChangesIH();
void calcOverallError();
void displayResults();
double getRand();


//// variables ////
int patNum = 0;
double errThisPat = 0.0;
double outPred = 0.0;
double RMSerror = 0.0;

// the outputs of the hidden neurons
double hiddenVal[numHidden];

// the weights
double weightsIH[numInputs][numHidden];
double weightsHO[numHidden];
```

```
// the data
int trainInputs[numPatterns][numInputs];
int trainOutput[numPatterns];


//==============================================================
//************* function definitions *************************
//==============================================================


//**********************************
// calculates the network output
void calcNet(void)
{
    //calculate the outputs of the hidden neurons
    //the hidden neurons are tanh
    int i = 0;
    for(i = 0;i<numHidden;i++)
    {
                                hiddenVal[i] = 0.0;

        for(int j = 0;j<numInputs;j++)
        {
                                 hiddenVal[i] = hiddenVal[i] + (trainInputs[patNum][j] * weightsIH[j][i]);
        }

        hiddenVal[i] = tanh(hiddenVal[i]);
    }

   //calculate the output of the network
   //the output neuron is linear
   outPred = 0.0;

   for(i = 0;i<numHidden;i++)
   {
    outPred = outPred + hiddenVal[i] * weightsHO[i];
   }
    //calculate the error
    errThisPat = outPred - trainOutput[patNum];

}


//************************************
//adjust the weights hidden-output
void WeightChangesHO(void)
{
   for(int k = 0;k<numHidden;k++)
   {
    double weightChange = LR_HO * errThisPat * hiddenVal[k];
    weightsHO[k] = weightsHO[k] - weightChange;

    //regularisation on the output weights
    if (weightsHO[k] < -5)
    {
```

```
     weightsHO[k] = -5;
    }
    else if (weightsHO[k] > 5)
    {
     weightsHO[k] = 5;
    }
   }

 }


//************************************
// adjust the weights input-hidden
void WeightChangesIH(void)
{

  for(int i = 0;i<numHidden;i++)
  {
   for(int k = 0;k<numInputs;k++)
   {
    double x = 1 - (hiddenVal[i] * hiddenVal[i]);
    x = x * weightsHO[i] * errThisPat * LR_IH;
    x = x * trainInputs[patNum][k];
    double weightChange = x;
    weightsIH[k][i] = weightsIH[k][i] - weightChange;
   }
  }

}


//************************************
// generates a random number
double getRand(void)
{
 return ((double)rand())/(double)RAND_MAX;
}



//************************************
// set weights to random numbers
void initWeights(void)
{

 for(int j = 0;j<numHidden;j++)
 {
    weightsHO[j] = (getRand() - 0.5)/2;
    for(int i = 0;i<numInputs;i++)
    {
     weightsIH[i][j] = (getRand() - 0.5)/5;
     printf("Weight = %f\n", weightsIH[i][j]);
    }
  }

}
```

```
//************************************
// read in the data
void initData(void)
{
   printf("initialising data\n");

   // the data here is the XOR data
   // it has been rescaled to the range
   // [-1][1]
   // an extra input valued 1 is also added
   // to act as the bias
   // the output must lie in the range -1 to 1

   trainInputs[0][0]  = 1;
   trainInputs[0][1]  = -1;
   trainInputs[0][2]  = 1;    //bias
   trainOutput[0] = 1;

   trainInputs[1][0]  = -1;
   trainInputs[1][1]  = 1;
   trainInputs[1][2]  = 1;       //bias
   trainOutput[1] = 1;

   trainInputs[2][0]  = 1;
   trainInputs[2][1]  = 1;
   trainInputs[2][2]  = 1;        //bias
   trainOutput[2] = -1;

   trainInputs[3][0]  = -1;
   trainInputs[3][1]  = -1;
   trainInputs[3][2]  = 1;     //bias
   trainOutput[3] = -1;

}


//************************************
// display results
void displayResults(void)
{
 for(int i = 0;i<numPatterns;i++)
 {
  patNum = i;
  calcNet();
  printf("pat = %d actual = %d neural model = %f\n",patNum+1,trainOutput[patNum],outPred);
 }
}


//************************************
// calculate the overall error
void calcOverallError(void)
{
    RMSerror = 0.0;
```

```
    for(int i = 0;i<numPatterns;i++)
      {
       patNum = i;
       calcNet();
       RMSerror = RMSerror + (errThisPat * errThisPat);
      }
    RMSerror = RMSerror/numPatterns;
    RMSerror = sqrt(RMSerror);
}



//=================================================================
//********** THIS IS THE MAIN PROGRAM **************************
//=================================================================


int main(void)
{
 // seed random number function
 srand ( time(NULL) );

 // initiate the weights
 initWeights();

 // load in the data
 initData();

 // train the network
   for(int j = 0;j <= numEpochs;j++)
   {   for(int i = 0;i<numPatterns;i++)
       { //select a pattern at random
         patNum = rand()%numPatterns;

         //calculate the current network output
         //and error for this pattern
         calcNet();

         //change network weights
         WeightChangesHO();
         WeightChangesIH();
       }

       //display the overall network error
       //after each epoch
       calcOverallError();
       printf("epoch = %d RMS Error = %f\n",j,RMSerror);
   }

 //training has finished
 //display the results
 displayResults();


 system("PAUSE");
 return 0;}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. WEIGHT GENERATION NEURAL NETWORK CODE

```
////////////////////////////////////////////////////////////////////////////
//Reconfigurable Neural Network Weight Generation and
//Comparison Basis code.
//Based off Original source code by Dr Phil Brierley
//www.philbrierley.com
//Modifications made by LT Scott P. Bailey, USN
//This code may be freely used and modified at will
//C++ Compiled using g++ GNU C++ compiler/
////////////////////////////////////////////////////////////////////////////

#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
using std::cout;
using std::endl;


//// Data dependent settings ////
#define numInputs  1024 //representing the 32x32 input image spread across
// 1024 input 'neurons'.  Intention is to treat input weights as a memory
// access in the SRC hardware, selected by a 1 or not with a 0.
#define numPatterns  5 //Network is trained on 5 'images' approximated from
//Prof. Pace's book 'Low Probability of Intercept Radar':  The order of
//images in trainInputs array is:  P4, T4, T3, T2, and NoInput, an array of
//zeros.  The testInputs array currently holds two sets of the trainInputs
//data, for use in future comparitive testing.
#define numOutputs 5  //Number of Outputs is five.
#define numTESTPatterns 10 //Used for future comparitive testing.


//// User defineable settings ////
#define numHidden 5
const int numEpochs = 1000;
const double LR_IH = 0.7;
const double LR_HO = 0.07;


//// functions ////
void initWeights();
void initData();
void calcNet();
void WeightChangesHO();
void WeightChangesIH();
void calcOverallError();
void displayResults();
double getRand();


//// variables ////
```

```
int patNum = 0;
//double errThisPat = 0.0; **shifted from variable to array
//double outPred = 0.0;    *shifted from variable to array
double RMSerror = 0.0;

// the outputs of the hidden neurons
double hiddenVal[numHidden];

// the output of output neurons
double outPred[numOutputs];
double errThisPat[numOutputs];

// the weights
double weightsIH[numInputs][numHidden];
int intweightsIH[numInputs][numHidden];
int posmax = 0;
int negmax = 0;
//double weightsHO[numHidden];
double weightsHO[numHidden][numOutputs];
int intweightsHO[numHidden][numOutputs];

//the output file
FILE * outfile;
// the data
int trainInputs[numPatterns][numInputs] = { { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0 } , { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0,
0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0,
0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1,
1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1,
1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0,
1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0,
0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1,
0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0,
0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } , {
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0,
0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0,
1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1,
1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0,
1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1,
1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1,
1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },{ 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1,
0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1,
1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0,
1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1,
1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0,
0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } , { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } } ;
int testInputs[numTESTPatterns][numInputs] = { { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0,
1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0,
0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1,
1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0,
0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,
1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } , { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
```

1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0,
0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0,
0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1,
0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1,
1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0,
0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1,
1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0,
0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1,
0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0 } , { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0,
0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0,
0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0,
0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0,
0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0,
0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0,
0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0,
0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } , { 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0,
0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1,
0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0,
0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1,
0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0,
0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } , { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } , { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0,
0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0,
0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0,
1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0,
0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1,
1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } , {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0,
0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0,
0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0,

0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1,
1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0,
0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0,
1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0,
0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0,
0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0 } , { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1,
0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1,
1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0,
0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0,
0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1,
0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0,
1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0,
0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } , { 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1,
1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1,
1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1,
1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1,
0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1,
1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1,
0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } , { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } } ;
int trainOutput[numPatterns][numOutputs];


//==============================================================
//************** function definitions **************************
//==============================================================


//***********************************
// calculates the network output
void calcNet(void)
{
   //calculate the outputs of the hidden neurons
   //the hidden neurons have a sigmoid transfer function
   //equal to (1/1+exp(-x)).
   int i = 0;
   for(i = 0;i<numHidden;i++)
   {
              hiddenVal[i] = 0.0;

      for(int j = 0;j<numInputs;j++)
      {
               hiddenVal[i] = hiddenVal[i] + (trainInputs[patNum][j] * weightsIH[j][i]);
      }

      hiddenVal[i] =(1/(1+exp(-(hiddenVal[i])))); //sigmoid
   }

  //calculate the output of the network
  //the output neurons have a pure linear transfer function,
  //which means summed and weighted input = output for this layers' nodes

  for(int i = 0;i<numOutputs;i++)
  {
    outPred[i] = 0.0;

   for(int j = 0;j<numHidden;j++)
    {
      outPred[i] = outPred[i] + ( hiddenVal[j] * weightsHO[j][i]);
```

```
      errThisPat[i] = outPred[i] - trainOutput[patNum][i];
    }
  }
   //calculate the error

}
//***********************************
// calculates the network output using integer weights
void intcalcNet(void)
{
   //calculate the outputs of the hidden neurons
   //the hidden neurons have sigmoid activation function
   int i = 0;
   for(i = 0;i<numHidden;i++)
   {
               hiddenVal[i] = 0.0;

      for(int j = 0;j<numInputs;j++)
      {
               hiddenVal[i] = hiddenVal[i] + (trainInputs[patNum][j] * ((static_cast<double>(intweightsIH[j][i]))/8));
      }

      hiddenVal[i] =(1/(1+exp(-(hiddenVal[i]))));
   }

  //calculate the output of the network
  //the output neurons have pure linear activation functions

  for(int i = 0;i<numOutputs;i++)
  {
    outPred[i] = 0.0;

   for(int j = 0;j<numHidden;j++)
    {
      outPred[i] = outPred[i] + ( hiddenVal[j] * ((static_cast<double>(intweightsHO[j][i]))/8));
      errThisPat[i] = outPred[i] - trainOutput[patNum][i];
    }
  }
   //calculate the error

}

//***********************************
//adjust the weights hidden-output
void WeightChangesHO(void)
{
 for(int i = 0;i<numHidden;i++)
   {
  for(int k = 0;k<numOutputs;k++)
   {
    double weightChange = LR_HO * errThisPat[k] * hiddenVal[i];
    weightsHO[i][k] = weightsHO[i][k] - weightChange;

    //regularisation on the output weights
    if (weightsHO[i][k] < -5.0)
    {
```

```
     weightsHO[i][k] = -5.0;
    }
    else if (weightsHO[i][k] > 5.0)
    {
     weightsHO[i][k] = 5.0;
    }
   }
   }

}

//************************************
// adjust the weights input-hidden
void WeightChangesIH(void)
{

  for(int i = 0;i<numOutputs;i++)
  {
    for(int j = 0;j<numInputs;j++)
   {
    for(int k = 0;k<numHidden;k++)
   {
    double x = 1 - (hiddenVal[k] * hiddenVal[k]);
    x = x * weightsHO[k][i] * errThisPat[i] * LR_IH;
    x = x * trainInputs[patNum][j];
    double weightChange = x;
    weightsIH[j][k] = weightsIH[j][k] - weightChange;
   }
  }
  }
}


//************************************
// generates a random number
double getRand(void)
{
 return ((double)rand())/(double)RAND_MAX;
}



//************************************
// set weights to random numbers
void initWeights(void)
{
  for(int j = 0;j<numHidden;j++)
 {
    for(int i = 0;i<numInputs;i++)
    {
     weightsIH[i][j] = (getRand() - 0.5)/5;
    }
    for(int i = 0;i<numOutputs;i++)
    {
     weightsHO[j][i] = (getRand() - 0.5)/2;
    }
```

```
  }

}
//************************************
// set weights to random numbers
void Integerize(void)
{

 for(int j = 0;j<numHidden;j++)
 {
    for(int i = 0;i<numInputs;i++)
    {
      double zed = weightsIH[i][j];
     intweightsIH[i][j] = static_cast<int>(zed * 8);
    }
    for(int i = 0;i<numOutputs;i++)
    {
      double zod = weightsHO[j][i];
     intweightsHO[j][i] = static_cast<int>(zod * 8);
    }
  }

}

//************************************
// set weights to random numbers
void printWeights(void)
{

 for(int j = 0;j<numHidden;j++)
 {
    for(int i = 0;i<numInputs;i++)
    {
      cout << "WeightIH [" << i << "][" << j << "] =" << weightsIH[i][j] << "\t | IntWeightIH [" << i << "][" << j << "] =" <<
((static_cast<float>(intweightsIH[i][j]))/8) << endl;
    }
    for(int i = 0;i<numOutputs;i++)
    {
      cout << "WeightHO [" << j << "][" << i << "] =" << weightsHO[j][i] << "\t | IntWeightHO [" << j << "][" << i << "] =" <<
((static_cast<float>(intweightsHO[j][i]))/8) << endl;
    }
  }
 // output weights in integer form for utilization on SRC
 for(int j = 0;j<numHidden;j++)
 {
    for(int i = 0;i<numInputs;i++)
    {
      printf("True INTWeightIH [%d][%d] = %i \n", i,j,intweightsIH[i][j]);
      if (intweightsIH[i][j] > posmax) {
              posmax = intweightsIH[i][j];}
      else if (intweightsIH[i][j] < negmax) {
              negmax = intweightsIH[i][j];}
    }
    for(int i = 0;i<numOutputs;i++)
    {
      printf("True IntWeightHO [%d][%d] = %i \n",j,i,intweightsHO[j][i]);
```

```
    }
  }
 printf("Posmax = %i       Negmax = %i",posmax,negmax);
}


//************************************
// read in the data
void initData(void)
{
    cout << "initializing data" << endl;

    // the data here is the output setup for each pattern
    // Node 0 should fire only for P4 pattern (Pattern 0)
    // Node 1 should fire only for T4 pattern (Pattern 1)
    // Node 2 should fire only for T3 pattern (Pattern 2)
    // Node 3 should fire only for T2 pattern (Pattern 3)
    // Node 4 should fire only for no input   (Pattern 4)


    trainOutput[0][0] = 1;
    trainOutput[0][1] = 0;
    trainOutput[0][2] = 0;
    trainOutput[0][3] = 0;
    trainOutput[0][4] = 0;

    trainOutput[1][0] = 0;
    trainOutput[1][1] = 1;
    trainOutput[1][2] = 0;
    trainOutput[1][3] = 0;
    trainOutput[1][4] = 0;

    trainOutput[2][0] = 0;
    trainOutput[2][1] = 0;
    trainOutput[2][2] = 1;
    trainOutput[2][3] = 0;
    trainOutput[2][4] = 0;

    trainOutput[3][0] = 0;
    trainOutput[3][1] = 0;
    trainOutput[3][2] = 0;
    trainOutput[3][3] = 1;
    trainOutput[3][4] = 0;

    trainOutput[4][0] = 0;
    trainOutput[4][1] = 0;
    trainOutput[4][2] = 0;
    trainOutput[4][3] = 0;
    trainOutput[4][4] = 1;

    cout << "Data Initialization complete" << endl;
}


//************************************
// display results
```

```
void displayResults(void)
{
 for(int i = 0;i<numTESTPatterns;i++)
 {
   for(int j = 0;j<numOutputs;j++)
     {
  patNum = i;
  calcNet();
  printf("pat = %d output neuron = %d actual = %d neural model =
%f\n",patNum+1,j+1,trainOutput[patNum][j],outPred[j]);
 }
   cout << endl;
 }
}


//************************************
// calculate the overall error
void calcOverallError(void)
{
     RMSerror = 0.0;
     for(int i = 0;i<numPatterns;i++)
       {
        patNum = i;
        calcNet();
        RMSerror = RMSerror + (errThisPat[i] * errThisPat[i]);
       }
     RMSerror = RMSerror/numPatterns;
     RMSerror = sqrt(RMSerror);
}
//************************************
// calculate the overall error
void calcINTError(void)
{
     RMSerror = 0.0;
     for(int i = 0;i<numPatterns;i++)
       {
        patNum = i;
        intcalcNet();
        RMSerror = RMSerror + (errThisPat[i] * errThisPat[i]);
       }
     RMSerror = RMSerror/numPatterns;
     RMSerror = sqrt(RMSerror);
     cout << "Integerized RMS error:" << RMSerror << endl;
}



//==============================================================
//********** THIS IS THE MAIN PROGRAM *************************
//==============================================================


int main(void)
{
 // seed random number function
```

```
 time_t start, finish; //variables for timing calculations
 double timediff; //difference holder for timing output

start = clock();
srand ( time(NULL) );
outfile = fopen ("weightout","w");

// initiate the weights
initWeights();

// load in the data
initData();

// train the network
   for(int j = 0;j <= numEpochs;j++)
   {
       for(int i = 0;i<numPatterns;i++)
       {
         //select a pattern at random
         patNum = rand()%numPatterns;

         //calculate the current network output
         //and error for this pattern
         calcNet();

         //change network weights
         WeightChangesHO();
         WeightChangesIH();
       }

       //display the overall network error
       //after each epoch
       calcOverallError();

       if (!(j%10)){
       printf("epoch = %d RMS Error = %f\n",j,RMSerror);
             }
}
   finish = clock();
//training has finished
//display the results
   Integerize();
intcalcNet();
displayResults();
calcINTError();
printWeights();
for(int i = 0;i<numInputs;i++)
       {
             fprintf (outfile, "%08X%08X\n", intweightsIH[i][0], intweightsIH[i][1]);
       }
for(int i = 0;i<numOutputs;i++)
       {
             fprintf (outfile, "%08X%08X\n", intweightsHO[0][i], intweightsHO[1][i]);
       }
for(int i = 0;i<numInputs;i++)
       {
```

```
            fprintf (outfile, "%08X%08X\n", intweightsIH[i][2], intweightsIH[i][3]);
        }
 for(int i = 0;i<numOutputs;i++)
        {
            fprintf (outfile, "%08X%08X\n", intweightsHO[2][i], intweightsHO[3][i]);
        }
for(int i = 0;i<numInputs;i++)
        {
            fprintf (outfile, "00000000%08X\n", intweightsIH[i][4]);
        }
for(int i = 0;i<numOutputs;i++)
        {
            fprintf (outfile, "00000000%08X\n", intweightsHO[4][i]);
        }

    cout << "\nTime required for network training (seconds): "
         << ((double)(finish - start))/CLOCKS_PER_SEC << "\n";
 start = clock();
 for(int k = 0;k<10000;k++)
   {
     patNum = k%numPatterns;  //sequentially run through all patterns 2000
                              //times each for timing test.
     calcNet();               //calculate and discard output since we are only
                              //obtaining timing data here.
   }
 finish = clock();
 timediff = ((double)(finish-start))/CLOCKS_PER_SEC;
 printf ( "Time in seconds required for 10000 network runs, patterns in sequential order: %.3f\t",timediff);
 system("PAUSE");
 return 0;
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. OUTPUT OF WEIGHT GENERATION NEURAL NETWORK CODE

```
initializing data
Data Initialization complete
epoch = 0 RMS Error = 0.870113
epoch = 10 RMS Error = 0.565932
epoch = 20 RMS Error = 0.461639
epoch = 30 RMS Error = 0.337236
epoch = 40 RMS Error = 0.360158
epoch = 50 RMS Error = 0.352713
.
.
.
epoch = 950 RMS Error = 0.152628
epoch = 960 RMS Error = 0.130989
epoch = 970 RMS Error = 0.122706
epoch = 980 RMS Error = 0.123740
epoch = 990 RMS Error = 0.159130
epoch = 1000 RMS Error = 0.112453
pat = 1 output neuron = 1 actual = 1 neural model = 0.897610
pat = 1 output neuron = 2 actual = 0 neural model = 0.115621
pat = 1 output neuron = 3 actual = 0 neural model = -0.003307
pat = 1 output neuron = 4 actual = 0 neural model = -0.100395
pat = 1 output neuron = 5 actual = 0 neural model = 0.013487

pat = 2 output neuron = 1 actual = 0 neural model = 0.117319
pat = 2 output neuron = 2 actual = 1 neural model = 0.860009
pat = 2 output neuron = 3 actual = 0 neural model = -0.036769
pat = 2 output neuron = 4 actual = 0 neural model = 0.131088
pat = 2 output neuron = 5 actual = 0 neural model = 0.078532

pat = 3 output neuron = 1 actual = 0 neural model = -0.018217
pat = 3 output neuron = 2 actual = 0 neural model = 0.002024
pat = 3 output neuron = 3 actual = 1 neural model = 0.983433
pat = 3 output neuron = 4 actual = 0 neural model = -0.015625
pat = 3 output neuron = 5 actual = 0 neural model = 0.042241

pat = 4 output neuron = 1 actual = 0 neural model = -0.087029
pat = 4 output neuron = 2 actual = 0 neural model = 0.168297
pat = 4 output neuron = 3 actual = 0 neural model = -0.005189
pat = 4 output neuron = 4 actual = 1 neural model = 0.854516
pat = 4 output neuron = 5 actual = 0 neural model = 0.016214

pat = 5 output neuron = 1 actual = 0 neural model = 0.005419
pat = 5 output neuron = 2 actual = 0 neural model = 0.006125
pat = 5 output neuron = 3 actual = 0 neural model = 0.045536
pat = 5 output neuron = 4 actual = 0 neural model = 0.000751
pat = 5 output neuron = 5 actual = 1 neural model = 0.891801

pat = 6 output neuron = 1 actual = 0 neural model = 0.897610
pat = 6 output neuron = 2 actual = 0 neural model = 0.115621
pat = 6 output neuron = 3 actual = 0 neural model = -0.003307
pat = 6 output neuron = 4 actual = 0 neural model = -0.100395
```

```
pat = 6 output neuron = 5 actual = 0 neural model = 0.013487

pat = 7 output neuron = 1 actual = 0 neural model = 0.117319
pat = 7 output neuron = 2 actual = 0 neural model = 0.860009
pat = 7 output neuron = 3 actual = 0 neural model = -0.036769
pat = 7 output neuron = 4 actual = 0 neural model = 0.131088
pat = 7 output neuron = 5 actual = 0 neural model = 0.078532

pat = 8 output neuron = 1 actual = 0 neural model = -0.018217
pat = 8 output neuron = 2 actual = 0 neural model = 0.002024
pat = 8 output neuron = 3 actual = 0 neural model = 0.983433
pat = 8 output neuron = 4 actual = 0 neural model = -0.015625
pat = 8 output neuron = 5 actual = 0 neural model = 0.042241

pat = 9 output neuron = 1 actual = 0 neural model = -0.087029
pat = 9 output neuron = 2 actual = 0 neural model = 0.168297
pat = 9 output neuron = 3 actual = 0 neural model = -0.005189
pat = 9 output neuron = 4 actual = 0 neural model = 0.854516
pat = 9 output neuron = 5 actual = 0 neural model = 0.016214

pat = 10 output neuron = 1 actual = 0 neural model = 0.005419
pat = 10 output neuron = 2 actual = 0 neural model = 0.006125
pat = 10 output neuron = 3 actual = 0 neural model = 0.045536
pat = 10 output neuron = 4 actual = 0 neural model = 0.000751
pat = 10 output neuron = 5 actual = 0 neural model = 0.891801

Integerized RMS error:0.256174
WeightIH [0][0] =-0.0949626   | IntWeightIH [0][0] =0
WeightIH [1][0] =0.0454202    | IntWeightIH [1][0] =0
WeightIH [2][0] =0.0849273    | IntWeightIH [2][0] =0
WeightIH [3][0] =0.0420867    | IntWeightIH [3][0] =0
WeightIH [4][0] =-0.0554202   | IntWeightIH [4][0] =0
WeightIH [5][0] =-0.072371    | IntWeightIH [5][0] =0
.
.
.
WeightIH [1018][0] =-0.0511559          | IntWeightIH [1018][0] =0
WeightIH [1019][0] =-0.053143           | IntWeightIH [1019][0] =0
WeightIH [1020][0] =0.000278342         | IntWeightIH [1020][0] =0
WeightIH [1021][0] =-0.0561117          | IntWeightIH [1021][0] =0
WeightIH [1022][0] =-0.0502371          | IntWeightIH [1022][0] =0
WeightIH [1023][0] =0.0803187           | IntWeightIH [1023][0] =0
WeightHO [0][0] =0.0290877   | IntWeightHO [0][0] =0
WeightHO [0][1] =0.0101948   | IntWeightHO [0][1] =0
WeightHO [0][2] =-0.89236    | IntWeightHO [0][2] =-0.875
WeightHO [0][3] =0.0171173   | IntWeightHO [0][3] =0
WeightHO [0][4] =1.74136     | IntWeightHO [0][4] =1.625
WeightIH [0][1] =-0.0583113  | IntWeightIH [0][1] =0
WeightIH [1][1] =0.000955512            | IntWeightIH [1][1] =0
WeightIH [2][1] =0.0899366   | IntWeightIH [2][1] =0
WeightIH [3][1] =-0.00610633 | IntWeightIH [3][1] =0
WeightIH [4][1] =-0.0765403  | IntWeightIH [4][1] =0
WeightIH [5][1] =-0.00479997 | IntWeightIH [5][1] =0
.
.
.
```

WeightIH [1018][1] =0.0830811 | IntWeightIH [1018][1] =0
WeightIH [1019][1] =-0.0938855 | IntWeightIH [1019][1] =0
WeightIH [1020][1] =-0.0358995 | IntWeightIH [1020][1] =0
WeightIH [1021][1] =-0.0402071 | IntWeightIH [1021][1] =0
WeightIH [1022][1] =0.0763507 | IntWeightIH [1022][1] =0
WeightIH [1023][1] =-0.0714589 | IntWeightIH [1023][1] =0
WeightHO [1][0] =-0.106482 | IntWeightHO [1][0] =0
WeightHO [1][1] =-0.847758 | IntWeightHO [1][1] =-0.75
WeightHO [1][2] =0.127842 | IntWeightHO [1][2] =0.125
WeightHO [1][3] =-0.129586 | IntWeightHO [1][3] =-0.125
WeightHO [1][4] =1.70507 | IntWeightHO [1][4] =1.625
WeightIH [0][2] =-0.0876872 | IntWeightIH [0][2] =0
WeightIH [1][2] =0.0127844 | IntWeightIH [1][2] =0
WeightIH [2][2] =0.0153877 | IntWeightIH [2][2] =0
WeightIH [3][2] =0.0286805 | IntWeightIH [3][2] =0
WeightIH [4][2] =-0.00752517 | IntWeightIH [4][2] =0
WeightIH [5][2] =0.0157781 | IntWeightIH [5][2] =0
.
.
.
WeightIH [1018][2] =0.0147297 | IntWeightIH [1018][2] =0
WeightIH [1019][2] =-0.0822293 | IntWeightIH [1019][2] =0
WeightIH [1020][2] =-0.0118917 | IntWeightIH [1020][2] =0
WeightIH [1021][2] =-0.0643692 | IntWeightIH [1021][2] =0
WeightIH [1022][2] =0.0601095 | IntWeightIH [1022][2] =0
WeightIH [1023][2] =-0.0802205 | IntWeightIH [1023][2] =0
WeightHO [2][0] =0.204348 | IntWeightHO [2][0] =0.125
WeightHO [2][1] =0.691712 | IntWeightHO [2][1] =0.625
WeightHO [2][2] =-0.0315801 | IntWeightHO [2][2] =0
WeightHO [2][3] =-0.723427 | IntWeightHO [2][3] =-0.625
WeightHO [2][4] =0.0623189 | IntWeightHO [2][4] =0
WeightIH [0][3] =-0.0978023 | IntWeightIH [0][3] =0
WeightIH [1][3] =0.0632836 | IntWeightIH [1][3] =0
WeightIH [2][3] =-0.0772838 | IntWeightIH [2][3] =0
WeightIH [3][3] =-0.0270558 | IntWeightIH [3][3] =0
WeightIH [4][3] =0.00305276 | IntWeightIH [4][3] =0
WeightIH [5][3] =-0.0413171 | IntWeightIH [5][3] =0
.
.
.
WeightIH [1018][3] =-0.0974996 | IntWeightIH [1018][3] =0
WeightIH [1019][3] =-0.0168094 | IntWeightIH [1019][3] =0
WeightIH [1020][3] =-0.00281798 | IntWeightIH [1020][3] =0
WeightIH [1021][3] =-0.0130025 | IntWeightIH [1021][3] =0
WeightIH [1022][3] =0.0801024 | IntWeightIH [1022][3] =0
WeightIH [1023][3] =0.0837144 | IntWeightIH [1023][3] =0
WeightHO [3][0] =0.664175 | IntWeightHO [3][0] =0.625
WeightHO [3][1] =-0.586286 | IntWeightHO [3][1] =-0.5
WeightHO [3][2] =0.920633 | IntWeightHO [3][2] =0.875
WeightHO [3][3] =0.605915 | IntWeightHO [3][3] =0.5
WeightHO [3][4] =-1.79019 | IntWeightHO [3][4] =-1.75
WeightIH [0][4] =-0.0793014 | IntWeightIH [0][4] =0
WeightIH [1][4] =0.0429598 | IntWeightIH [1][4] =0
WeightIH [2][4] =-0.0605446 | IntWeightIH [2][4] =0
WeightIH [3][4] =-0.0926724 | IntWeightIH [3][4] =0
WeightIH [4][4] =0.0719323 | IntWeightIH [4][4] =0

```
WeightIH [5][4] =-0.0639899   | IntWeightIH [5][4] =0
.
.
.
WeightIH [1018][4] =-0.0386649          | IntWeightIH [1018][4] =0
WeightIH [1019][4] =-0.0958416          | IntWeightIH [1019][4] =0
WeightIH [1020][4] =0.0242519           | IntWeightIH [1020][4] =0
WeightIH [1021][4] =-0.0815059          | IntWeightIH [1021][4] =0
WeightIH [1022][4] =-0.00462537         | IntWeightIH [1022][4] =0
WeightIH [1023][4] =0.0869094           | IntWeightIH [1023][4] =0
WeightHO [4][0] =-0.780292   | IntWeightHO [4][0] =-0.75
WeightHO [4][1] =0.744388    | IntWeightHO [4][1] =0.625
WeightHO [4][2] =-0.0334624  | IntWeightHO [4][2] =0
WeightHO [4][3] =0.231484    | IntWeightHO [4][3] =0.125
WeightHO [4][4] =0.0650451   | IntWeightHO [4][4] =0
True INTWeightIH [0][0] = 0
True INTWeightIH [1][0] = 0
True INTWeightIH [2][0] = 0
True INTWeightIH [3][0] = 0
True INTWeightIH [4][0] = 0
True INTWeightIH [5][0] = 0
.
.
.
True INTWeightIH [1018][4] = 0
True INTWeightIH [1019][4] = 0
True INTWeightIH [1020][4] = 0
True INTWeightIH [1021][4] = 0
True INTWeightIH [1022][4] = 0
True INTWeightIH [1023][4] = 0
True IntWeightHO [4][0] = -6
True IntWeightHO [4][1] = 5
True IntWeightHO [4][2] = 0
True IntWeightHO [4][3] = 1
True IntWeightHO [4][4] = 0
Posmax = 428      Negmax = -727
Time required for network training (seconds): 8.57
Time in seconds required for 10000 network runs, patterns in sequential order: 1.020
```

APPENDIX E. MLP NEURAL NETWORK CODE FOR THE SRC

MAIN.C CODE:

```
static char const cvsid[] = "$Id: main.c,v 2.1 2005/06/14 22:16:48 jls Exp $";

#include <libmap.h>
#include <stdlib.h>

void subr (int64_t I0[], int64_t I1[], int64_t I2[], int64_t I3[], int *Out0, int *Out1, int *Out2, int *Out3, int *Out4, int64_t
*time, int mapnum);

int main (int argc, char *argv[]) {
   FILE *res_map, *res_cpu, *inweight, *inimage;
//    int i = 0;
//    int j = 0;
//    int nog = 0;
   int64_t *A;
   int64_t *B;
   int64_t *C;
   int64_t *D;
   int64_t atmp = 0;
   int64_t btmp1 = 0;
   int64_t ctmp1 = 0;
   int64_t dtmp1 = 0;
//    int64_t btmp2 = 0;
//    int64_t ctmp2 = 0;
//    int64_t dtmp2 = 0;
   int sum0 = 0;
   int sum1 = 0;
   int sum2 = 0;
   int sum3 = 0;
   int sum4 = 0;
   int64_t tm;
//    int64_t pooky;
//    int64_t adata;
   int mapnum = 0;

   if ((res_map = fopen ("res_map", "w")) == NULL) {
      fprintf (stderr, "failed to open file 'res_map'\n");
      exit (1);
      }

   if ((res_cpu = fopen ("res_cpu", "w")) == NULL) {
      fprintf (stderr, "failed to open file 'res_cpu'\n");
      exit (1);
      }

   if (argc < 2) {
            fprintf (stderr, "Usage: ./ex07 imagefile\n");
            exit (1);
            }

   inimage = fopen (argv[ 1 ],"rt"); //input of image data- data must be 64-bit hex value array
```

```
   if ( !inimage ) {
     fprintf (stderr, "%s could not be opened./n", argv[ 1 ]);
            exit (1);
            }

   A = (int64_t*) malloc (32 * sizeof (int64_t));
   B = (int64_t*) malloc (1029 * sizeof (int64_t));
   C = (int64_t*) malloc (1029 * sizeof (int64_t));
   D = (int64_t*) malloc (1029 * sizeof (int64_t));
   srandom (99);
   inweight = fopen ("weightout","rt"); //change weightout to any weight file
// NOTE:  This program is set up to accept weights ONLY in the current order of
// Neuron 0 and 1 input to hidden weights in hex right next to each other (x1024),
// followed by Neuron 0 and 1 hidden-to-output weights in hex (x5).  Neuron 2 + 3
// follows in a similar manner (x1029), and finally Neuron 4 weights with zero padding
// (x1029), allowing maximum use of bandwidth.

   for (int j=0; j<(1029); j++) {  //this inputs Neuron 0 and 1 weights (first and second layer)
            fscanf (inweight,"%llx",&btmp1);  //into array 'B'.
            B[j] = btmp1;
            }
    for (int j=0; j<(1029); j++) {//This inputs Neuron 2 and 3 weights (first and second layer)
            fscanf (inweight,"%llx",&ctmp1);  //into array 'C'.
            C[j] = ctmp1;
            }
   for (int j=0; j<(1029); j++) {  //This inputs Neuron 4 weights (first and second layer) into
            fscanf (inweight,"%llx",&dtmp1);            //array 'D'.
            D[j] = dtmp1;
            }

//    This was an old (and failed) way I was trying to initially input weights
//    if we were dealing in pure data it may have worked.
//    fread (B, 8, 1024, inweight);  //read weight 0 and 1 data
//    fread (C, 8, 1024, inweight);  //read weight 2 and 3 data
//    fread (D, 4, 1024, inweight);  //read weight 4 data
   fclose (inweight);

   for (int j=0; j<(32); j++) {
      fscanf (inimage,"%qi",&atmp);    //loading A with image data
            A[j] = atmp;
            }
   fclose (inimage);

//*****************************************************************************************
//                          TESTING ROUTINES ONLY - USED IN PROGRAM DEVELOPMENT
//*****************************************************************************************
//   for (i=0; i<32; i++) {
//      A[i] = 613566756; //setting up a series of 32 0's, followed by 16 '100' patterns
//            }
//
//    nog = 32*32;
//    for (j=0; j<=(nog-1); j++) {
//            B[j] = j;
//            if (!(j%32)) {
//            adata = 613566756;
//            }
```

```
//      printf ("B[%d] = %d ", j, B[j]);
//      pooky = adata % 2;
//      adata = adata>>1;
//              printf ("Enabler is %d \n", pooky);
//              }
//*****************************************************************************************
//  TESTING ROUTINE TO DETERMINE FSCANF INPUT CORRECTNESS - DETERMINATION OF '7FFFFFFF'
//  PROBLEM.
//*****************************************************************************************
//    for (int j=0; j<(32); j++) {
//      printf ("A[%d] = %llx ", j, A[j]);
//      atmp = A[j] >> 32;
//      printf (" which is %ld", atmp);
//      atmp = A[j] << 32;
//      atmp = atmp >> 32;
//      printf (" and %d \n", atmp);
//              }
//    for (int j=0; j<(1024); j++) {
//      printf ("B[%d] = %llx", j, B[j]);
//      btmp1 = B[j] >> 32;
//      printf (" which is %ld", btmp1);
//      btmp1 = B[j] << 32;
//      btmp1 = btmp1 >> 32;
//      printf (" and %d \n", btmp1);
//              }
//    for (int j=0; j<(1024); j++) {
//      printf ("C[%d] = %llx", j, C[j]);
//      ctmp1 = C[j] >> 32;
//      printf (" which is %ld", ctmp1);
//      ctmp1 = C[j] << 32;
//      ctmp1 = ctmp1 >> 32;
//      printf (" and %d \n", ctmp1);
//              }
//    for (int j=0; j<(1024); j++) {
//      printf ("D[%d] = %llx", j, D[j]);
//      dtmp1 = D[j] >> 32;
//      printf (" which is %ld", dtmp1);
//      dtmp1 = D[j] << 32;
//      dtmp1 = dtmp1 >> 32;
//      printf (" and %d \n", dtmp1);
//              }
//*****************************************************************************************
//                              END OF TESTING ROUTINES
//*****************************************************************************************
   map_allocate (1);

    subr (A, B, C, D, &sum0, &sum1, &sum2, &sum3, &sum4, &tm, mapnum);

    printf ("%lld clocks\n", tm);

    printf ("Outputs= Neuron 0(%d) \n", sum0);
    printf ("Outputs= Neuron 1(%d) \n", sum1);
    printf ("Outputs= Neuron 2(%d) \n", sum2);
    printf ("Outputs= Neuron 3(%d) \n", sum3);
    printf ("Outputs= Neuron 4(%d) \n", sum4);
```

```
   map_free (1);

   exit(0);
   }
```

EX07.MC CODE:

```
/* $Id: ex07.mc,v 2.1 2005/06/14 22:16:48 jls Exp $ */


#include <libmap.h>

void subr (int64_t I0[], int64_t I1[], int64_t I2[], int64_t I3[], int *Out0, int *Out1, int *Out2, int *Out3, int *Out4,
int64_t *Out5, int64_t *Out6, int64_t *Out7, int64_t *Out8, int64_t *Out9, int *Out10, int *Out11, int *Out12, int *Out13,
int *Out14, int64_t *time, int mapnum) {
   OBM_BANK_A (AL, int64_t, MAX_OBM_SIZE)
   OBM_BANK_B (BL, int64_t, MAX_OBM_SIZE)
   OBM_BANK_C (CL, int64_t, MAX_OBM_SIZE)
   OBM_BANK_D (DL, int64_t, MAX_OBM_SIZE)
   int64_t t0, t1;
   int i = 0;
   int num2 = 1024;  //number of inputs
   int num3 = 1029;  //number of inputs + number of outputs
   int aodd = 0;
   int aeven = 0;
   int bodd = 0;
   int beven = 0;
   int codd = 0;
   int ceven = 0;
   int dodd = 0;
   int deven = 0;
   int wt2[5][5]; //node 0 2nd layer weight array
//    int wt15; //node 0 2nd layer weight array
//    int wt25; //node 0 2nd layer weight array
//    int wt35; //node 0 2nd layer weight array
//    int wt45; //node 0 2nd layer weight array
   int ptr1 = 0; //pointer to OBM array values in 2nd layer
   int ptr2 = 0; //pointer to BRAM array values in 2nd layer
   int j = 0;
   int k = 0;
   int ctr = 0;
   int hold0 = 0;
   int hold1 = 0;
   int hold2 = 0;
   int hold3 = 0;
   int hold4 = 0;
   int sum0 = 0;
   int sum1 = 0;
   int sum2 = 0;
   int sum3 = 0;
   int sum4 = 0;
   int sum5 = 0;
   int sum6 = 0;
   int sum7 = 0;
```

```
int sum8 = 0;
int sum9 = 0;
int sig0 = 0;
int sig1 = 0;
int sig2 = 0;
int sig3 = 0;
int sig4 = 0;
int enable = 0;
int upgrade = 0;
int image = 0;

DMA_CPU (CM2OBM, AL, MAP_OBM_stripe(1,"A"), I0, 1, 32*sizeof(int64_t), 0);
wait_DMA (0);
DMA_CPU (CM2OBM, BL, MAP_OBM_stripe(1,"B"), I1, 1, 1029*sizeof(int64_t), 0);
wait_DMA (0);
DMA_CPU (CM2OBM, CL, MAP_OBM_stripe(1,"C"), I2, 1, 1029*sizeof(int64_t), 0);
wait_DMA (0);
DMA_CPU (CM2OBM, DL, MAP_OBM_stripe(1,"D"), I3, 1, 1029*sizeof(int64_t), 0);
wait_DMA (0);

read_timer (&t0);

for (i=0; i<num2; i++) {
        cg_count_ceil_32(1, 0, i==0, 31, &k);
        cg_count_ceil_32(k==0, 0, i==0, 32767, &j);
        split_64to32 (AL[j], &aodd, &aeven);
        if (k==0) {             //if then to allow loop unrolling method
        image = aeven;                  //must only update image when j increases
        }                               //otherwise shift will not matter
        upgrade = j<<5;
        ctr=((31 - k) + upgrade); //have to match array input
                                        //up with image input
        enable = image%2; //save on modulus calculations
        split_64to32 (BL[ctr], &bodd, &beven);
        cg_accum_add_32 (bodd,(enable),0,(i==0),&sum0);
        cg_accum_add_32 (beven,(enable),0,(i==0),&sum1);
        split_64to32 (CL[ctr], &codd, &ceven);
        cg_accum_add_32 (codd,(enable),0,(i==0),&sum2);
        cg_accum_add_32 (ceven,(enable),0,(i==0),&sum3);
        split_64to32 (DL[ctr], &dodd, &deven);
        cg_accum_add_32 (deven,(enable),0,(i==0),&sum4);
        image=image>>1;
        }
        SIGFOUR (sum0, &sig0);
        SIGFOUR (sum1, &sig1);
        SIGFOUR (sum2, &sig2);
        SIGFOUR (sum3, &sig3);
        SIGFOUR (sum4, &sig4);
for (i=0; i<5; i++) {
        ptr1 = (num2 + i);
        split_64to32 (BL[ptr1], &bodd, &beven);
        split_64to32 (CL[ptr1], &codd, &ceven);
        split_64to32 (DL[ptr1], &dodd, &deven);
        wt2[0][i] = (bodd * sig0);
        wt2[1][i] = (beven * sig1);
        wt2[2][i] = (codd * sig2);
```

```
            wt2[3][i] = (ceven * sig3);          //arrays prior to use.
            wt2[4][i] = (deven * sig4);
            }
//*****************************************************************************************************************
// ***        NOTE:  THIS CODE SECTION WAS OPTIMIZED FOR A 5-NODE HIDDEN LAYER WITH            ****
// ***        5 OUTPUTS.  DUE TO CONCERNS REGARDING REUSABILITY OF CODE, THIS SECTION          ****
// ***        WAS NOT UTILIZED THOUGH IT ALLOWS A REDUCTION IN ABOUT 50 CLOCKS OF              ****
// ***        PROCESSING TIME.                                                                 ****
// *****************************************************************************************************************
//          split_64to32 (BL[1024], &bodd, &beven);
//      split_64to32 (CL[1024], &codd, &ceven);
//      split_64to32 (DL[1024], &dodd, &deven);
//          wt00 = bodd;
//          wt01 = beven;
//          wt02 = codd;
//          wt03 = ceven;
//          wt04 = deven;
//          split_64to32 (BL[1025], &bodd, &beven);
//      split_64to32 (CL[1025], &codd, &ceven);
//      split_64to32 (DL[1025], &dodd, &deven);
//          wt10 = bodd;
//          wt11 = beven;
//      wt12 = codd;
//      wt13 = ceven;
//      wt14 = deven;
//          split_64to32 (BL[1026], &bodd, &beven);
//      split_64to32 (CL[1026], &codd, &ceven);
//      split_64to32 (DL[1026], &dodd, &deven);
//          wt20 = bodd;
//          wt21 = beven;
//      wt22 = codd;
//      wt23 = ceven;
//      wt24 = deven;
//          split_64to32 (BL[1027], &bodd, &beven);
//      split_64to32 (CL[1027], &codd, &ceven);
//      split_64to32 (DL[1027], &dodd, &deven);
//          wt30 = bodd;
//          wt31 = beven;
//      wt32 = codd;
//      wt33 = ceven;
//      wt34 = deven;
//          split_64to32 (BL[1028], &bodd, &beven);
//      split_64to32 (CL[1028], &codd, &ceven);
//      split_64to32 (DL[1028], &dodd, &deven);
//          wt40 = bodd;
//          wt41 = beven;
//      wt42 = codd;
//      wt43 = ceven;
//      wt44 = deven;
//          wt00 = wt00 * sig0;
//          wt01 = wt01 * sig1;
//          wt02 = wt02 * sig2;
//          wt03 = wt03 * sig3;
//          wt04 = wt04 * sig4;
//          wt10 = wt10 * sig0;
//          wt11 = wt11 * sig1;
```

```
//          wt12 = wt12 * sig2;
//          wt13 = wt13 * sig3;
//          wt14 = wt14 * sig4;
//          wt20 = wt20 * sig0;
//          wt21 = wt21 * sig1;
//          wt22 = wt22 * sig2;
//          wt23 = wt23 * sig3;
//          wt24 = wt24 * sig4;
//          wt30 = wt30 * sig0;
//          wt31 = wt31 * sig1;
//          wt32 = wt32 * sig2;
//          wt33 = wt33 * sig3;
//          wt34 = wt34 * sig4;
//          wt40 = wt40 * sig0;
//          wt41 = wt41 * sig1;
//          wt42 = wt42 * sig2;
//          wt43 = wt43 * sig3;
//          wt44 = wt44 * sig4;
//
    for (i=0; i<5; i++) {
//    hold0 = (wt0[i] * sig0);
    cg_accum_add_32 (wt2[i][0],1,0,(i==0),&sum5);
//    hold1 = (wt1[i] * sig1);
    cg_accum_add_32 (wt2[i][1],1,0,(i==0),&sum6);
//    hold2 = (wt1[i] * sig2);
    cg_accum_add_32 (wt2[i][2],1,0,(i==0),&sum7);
//    hold3 = (wt3[i] * sig3);
    cg_accum_add_32 (wt2[i][3],1,0,(i==0),&sum8);
//    hold4 = (wt4[i] * sig4);
    cg_accum_add_32 (wt2[i][4],1,0,(i==0),&sum9);
    }
//          sum5 = (wt0[0] + wt1[0] + wt2[0] + wt3[0] + wt4[0]);
//          sum6 = (wt0[1] + wt1[1] + wt2[1] + wt3[1] + wt4[1]);
//          sum7 = (wt0[2] + wt1[2] + wt2[2] + wt3[2] + wt4[2]);
//          sum8 = (wt0[3] + wt1[3] + wt2[3] + wt3[3] + wt4[3]);
//          sum9 = (wt0[4] + wt1[4] + wt2[4] + wt3[4] + wt4[4]);
  *Out0 = sum5;
  *Out1 = sum6;
  *Out2 = sum7;
  *Out3 = sum8;
  *Out4 = sum9;
  *Out5= sum0;
  *Out6= sum1;
  *Out7= sum2;
  *Out8= sum3;
  *Out9= sum4;
  *Out10 = sig0;
  *Out11 = sig1;
  *Out12 = sig2;
  *Out13 = sig3;
  *Out14 = sig4;
   read_timer (&t1);
   *time = t1 - t0;

   }
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX F. IMAGE CONVERSION PROGRAM

```
//  conv.c - A program written to convert the binary bitmap files into hex
//  representation for the SRC-6 Reconfigurable Neural Network Program.
//  Written by LT Scott Bailey, Naval Postgraduate School. 2006
//  Usage:  ./conv [name of file to convert] >> [name of output file]
//*************************************************************************
//*  NOTE:  This simple program will only convert pure binary ascii files
//*  which are exactly 32 characters wide (33 with an endl), and 32 lines
//*  in length.  In short, only a 32x32 bitmap generated with the
//*  'bitmap' command and converted via the 'bmtoa -chars 01' line
//*  can be successfully converted with this program, unless the format
//*  is mimiced exactly.  This program places data into a format
//*  utilized from Ensign Dane Brown's Thesis to pass preprocessed data
//*  to the RANN.  If there are alterations to the bitmap size, the
//*  'len' and 'wid' constants must be altered to match.
//*************************************************************************
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
using namespace std;

int main (int argc, char *argv[]) {
        ifstream infile;
        const int wid = 8;     //width of bitmap in 4-bit sections
        const int len = 32;  //length of bitmap in lines
        int cnt1,cnt2;
        char tmp = 48;
        int output;
        if (argc < 2) {
                fprintf (stderr, "Usage: ./conv inputfile\n");
                exit (1);
                }
        infile.open(argv[ 1 ],ios::in);  //input of image data, 32 lines of 32-bit binary ascii each.

        if ( !infile ) {
                fprintf (stderr, "%s could not be opened.\n", argv[ 1 ]);
                exit (1);
                }

        for (cnt1 = 0; cnt1 < len; cnt1++) {
                cout << "0x00000000";
                for (cnt2 = 0; cnt2 < wid; cnt2++) {
                output = 0;                     //clears output
                tmp = infile.get();
                if (tmp == ('1')) {      //obtain 2^3 value
                output = 8; }
                tmp = infile.get();
                if (tmp == ('1')) {      //obtain 2^2 value
                output = output + 4; }
                tmp = infile.get();
                if (tmp == ('1')) {      //obtain 2^1 value
                output = output + 2; }
```

```
                tmp = infile.get();
                if (tmp == ('1')) {      //obtain 2^0 value
                output = output + 1; }
                cout << hex << output;
                } // end inner for loop and one line of code
        tmp == infile.get(); //clears the endl from the input ASCII file
        cout << endl;  //and sends it back out again.
        }       //end outer loop and should have complete hex bitmap
                //ready for use in the SRC
        infile.close();
return 0;
        } //end main
```

APPENDIX G. SIGMOID FUNCTION VHDL FILES

SIGFOUR.VHD:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity SIGFOUR is
    Port ( A : in std_logic_vector(31 downto 0);
           Q : out std_logic_vector(31 downto 0));
end SIGFOUR;

architecture Behavioral of SIGFOUR is

begin
   process(A)
   begin
   Q(31 downto 5) <= "00000000000000000000000000";
   if (A(31 downto 0) = "00000000000000000000000000000000" or
   A(31 downto 0) = "00000000000000000000000000000001") then
   Q(4 downto 0) <= "01000"; elsif
   (A(31 downto 0) = "00000000000000000000000000000010" or
   A(31 downto 0) = "00000000000000000000000000000011") then
   Q(4 downto 0) <= "01001"; elsif
   (A(31 downto 0) = "00000000000000000000000000000100" or
   A(31 downto 0) = "00000000000000000000000000000101") then
   Q(4 downto 0) <= "01010"; elsif
   (A(31 downto 0) = "00000000000000000000000000000110" or
   A(31 downto 0) = "00000000000000000000000000000111") then
   Q(4 downto 0) <= "01011"; elsif
   (A(31 downto 0) = "00000000000000000000000000001000" or
   A(31 downto 0) = "00000000000000000000000000001001" or
   A(31 downto 0) = "00000000000000000000000000001010") then
   Q(4 downto 0) <= "01100"; elsif
   (A(31 downto 0) = "00000000000000000000000000001011" or
   A(31 downto 0) = "00000000000000000000000000001100" or
   A(31 downto 0) = "00000000000000000000000000001101") then
   Q(4 downto 0) <= "01101"; elsif
   (A(31 downto 0) = "00000000000000000000000000001110" or
   A(31 downto 0) = "00000000000000000000000000001111" or
   A(31 downto 0) = "00000000000000000000000000010000" or
   A(31 downto 0) = "00000000000000000000000000010001" or
   A(31 downto 0) = "00000000000000000000000000010010") then
   Q(4 downto 0) <= "01110"; elsif
   (A(31 downto 0) = "00000000000000000000000000010011" or
   A(31 downto 0) = "00000000000000000000000000010100" or
   A(31 downto 0) = "00000000000000000000000000010101" or
```

```
  A(31 downto 0) = "00000000000000000000000000010110" or
  A(31 downto 0) = "00000000000000000000000000010111" or
  A(31 downto 0) = "00000000000000000000000000011000" or
  A(31 downto 0) = "00000000000000000000000000011001" or
  A(31 downto 0) = "00000000000000000000000000011010") then
  Q(4 downto 0) <= "01111"; elsif
  (A(31) = '0' and
  A(31 downto 0) > "00000000000000000000000000011010") then
  Q(4 downto 0) <= "10000"; elsif
  A(31 downto 0) = "11111111111111111111111111111111" then
  Q(4 downto 0) <= "01000"; elsif
  (A(31 downto 0) = "11111111111111111111111111111101" or
  A(31 downto 0) = "11111111111111111111111111111110") then
  Q(4 downto 0) <= "00111"; elsif
  (A(31 downto 0) = "11111111111111111111111111111011" or
  A(31 downto 0) = "11111111111111111111111111111100") then
  Q(4 downto 0) <= "00110"; elsif
  (A(31 downto 0) = "11111111111111111111111111111001" or
  A(31 downto 0) = "11111111111111111111111111111010") then
  Q(4 downto 0) <= "00101"; elsif
  (A(31 downto 0) = "11111111111111111111111111110110" or
  A(31 downto 0) = "11111111111111111111111111110111" or
  A(31 downto 0) = "11111111111111111111111111111000") then
  Q(4 downto 0) <= "00100"; elsif
  (A(31 downto 0) = "11111111111111111111111111110011" or
  A(31 downto 0) = "11111111111111111111111111110100" or
  A(31 downto 0) = "11111111111111111111111111110101") then
  Q(4 downto 0) <= "00011"; elsif
  (A(31 downto 0) = "11111111111111111111111111101110" or
  A(31 downto 0) = "11111111111111111111111111101111" or
  A(31 downto 0) = "11111111111111111111111111110000" or
  A(31 downto 0) = "11111111111111111111111111110001" or
  A(31 downto 0) = "11111111111111111111111111110010") then
  Q(4 downto 0) <= "00010"; elsif
  (A(31 downto 0) = "11111111111111111111111111100101" or
  A(31 downto 0) = "11111111111111111111111111100110" or
  A(31 downto 0) = "11111111111111111111111111100111" or
  A(31 downto 0) = "11111111111111111111111111101000" or
  A(31 downto 0) = "11111111111111111111111111101001" or
  A(31 downto 0) = "11111111111111111111111111101010" or
  A(31 downto 0) = "11111111111111111111111111101011" or
  A(31 downto 0) = "11111111111111111111111111101100" or
  A(31 downto 0) = "11111111111111111111111111101101") then
  Q(4 downto 0) <= "00001"; else
  Q(4 downto 0) <= "00000";
  end if;
  end process;

end Behavioral;
```

BLK.V:

```
module SIGFOUR (A,Q) /* synthesis syn_black_box  */  ;
   input [31:0] A;
   output [31:0] Q;
endmodule
```

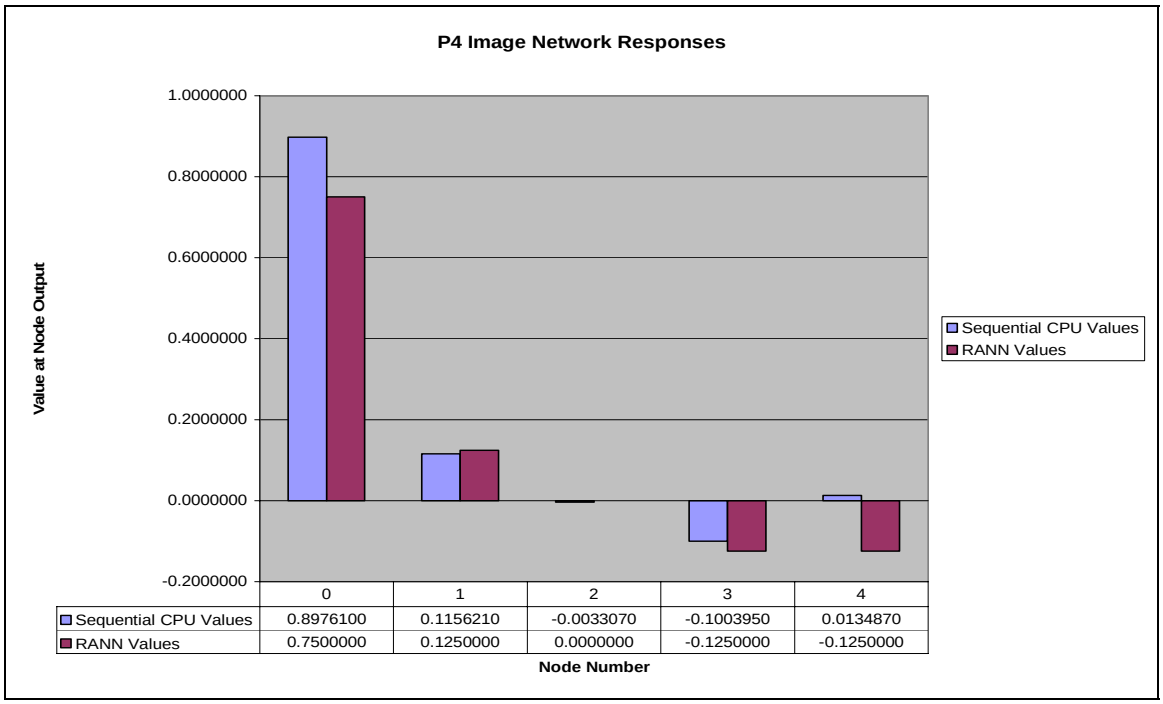
INFO FILE:

```
BEGIN_DEF "SIGFOUR"
   MACRO = "SIGFOUR";
   STATEFUL = NO;
   EXTERNAL = NO;
   PIPELINED = YES;
   LATENCY = 0;

   INPUTS = 1:
     I0 = INT 32 BITS (A[31:0])  // explicit input
     ;
   OUTPUTS = 1:
     O0 = INT 32 BITS (Q[31:0])  // explicit output
     ;
   DEBUG_HEADER = #
      void SIGFOUR__dbg (int A, int Q);
   #;
   DEBUG_FUNC = #
      void SIGFOUR__dbg (int A, int Q ) {
            if (A <= -28)
                        Q = 0; else
            if (-27 <= A <= -19)
                        Q = 1; else
            if (-18 <= A <= -14)
                        Q = 2; else
            if (-13 <= A <= -11)
                        Q = 3; else
            if (-10 <= A <= -8)
                        Q = 4; else
            if (-7 <= A <= -6)
                        Q = 5; else
            if (-5 <= A <= -4)
                        Q = 6; else
            if (-3 <= A <= -2)
                        Q = 7; else
            if (-1 <= A <= 1)
                        Q = 8; else
            if (2 <= A <= 3)
                        Q = 9; else
            if (4 <= A <= 5)
                        Q = 10; else
            if (6 <= A <= 7)
                        Q = 11; else
            if (8 <= A <= 10)
                        Q = 12; else
            if (11 <= A <= 13)
                        Q = 13; else
            if (14 <= A <= 18)
                        Q = 14; else
            if (19 <= A <= 26)
                        Q = 15; else
                        Q = 16;
            }
   #;
END_DEF
```

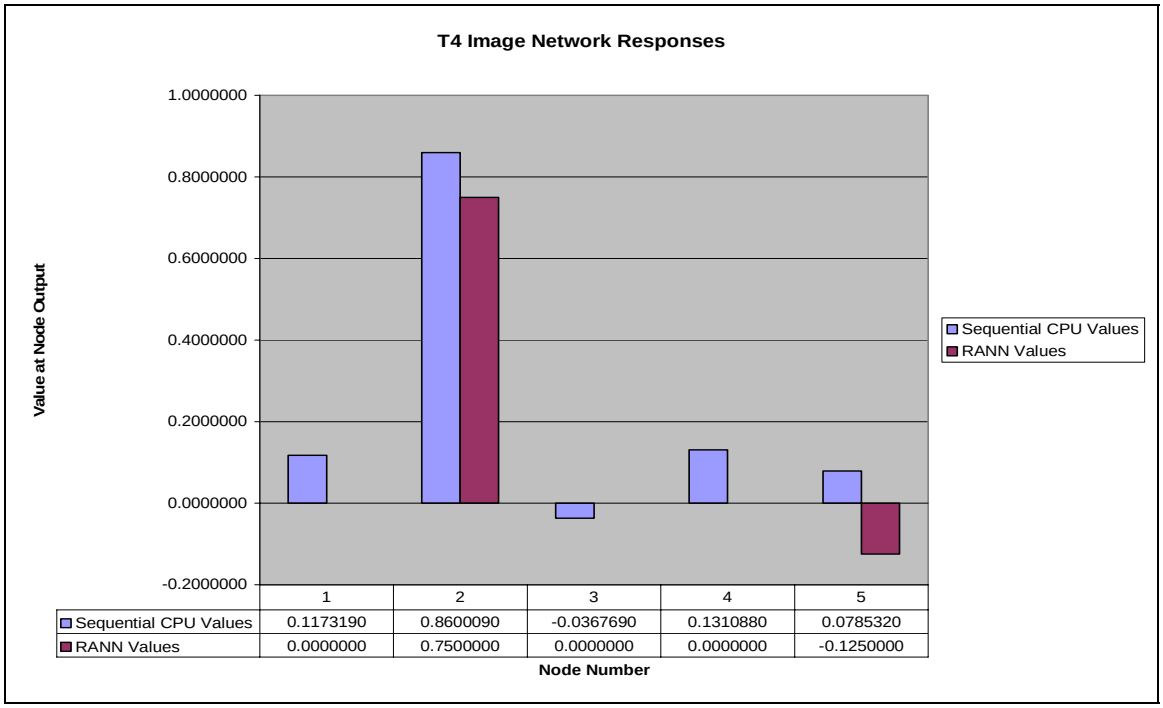
THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX H. NETWORK OUTPUT GRAPHS



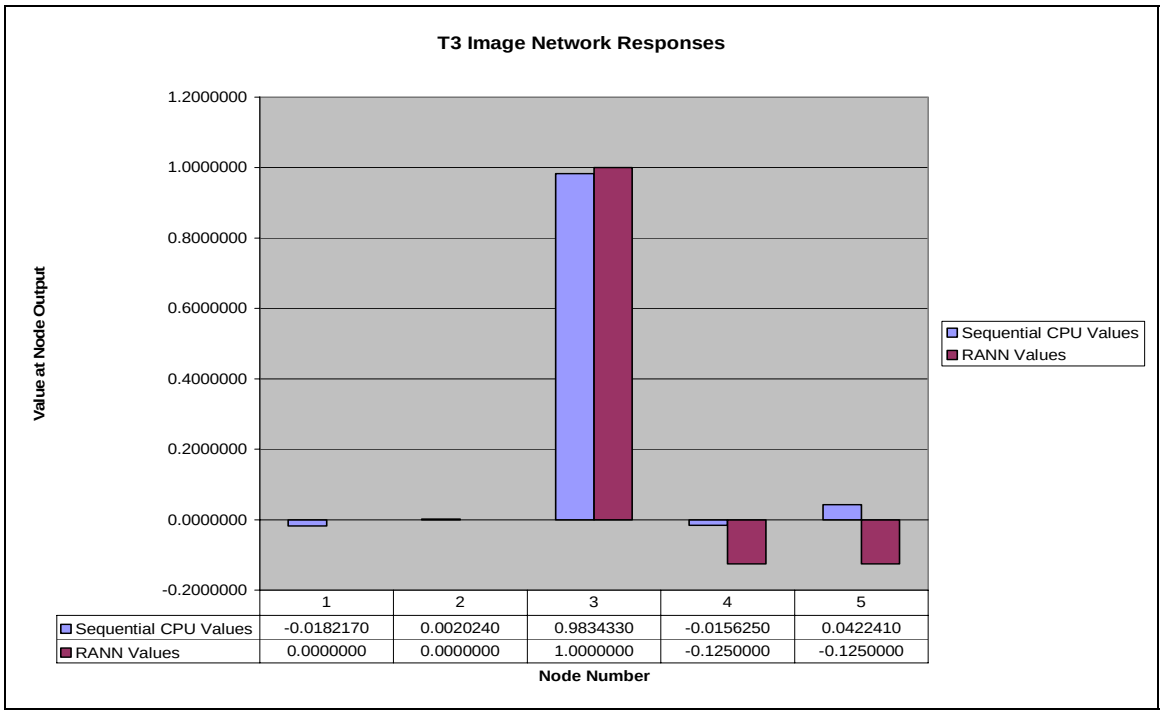
	0	1	2	3	4
Sequential CPU Values	0.8976100	0.1156210	-0.0033070	-0.1003950	0.0134870
RANN Values	0.7500000	0.1250000	0.0000000	-0.1250000	-0.1250000

Figure 13. P4 Image Network Output Comparison



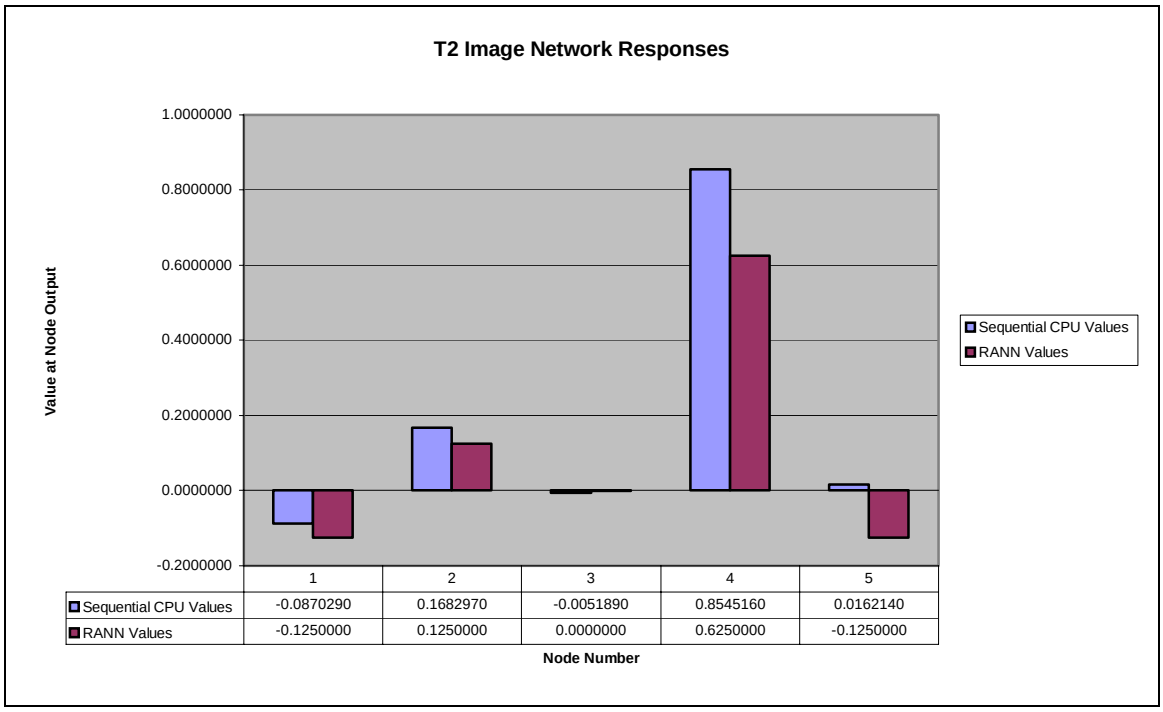
	1	2	3	4	5
Sequential CPU Values	0.1173190	0.8600090	-0.0367690	0.1310880	0.0785320
RANN Values	0.0000000	0.7500000	0.0000000	0.0000000	-0.1250000

Figure 14. T4 Image Network Output Comparison



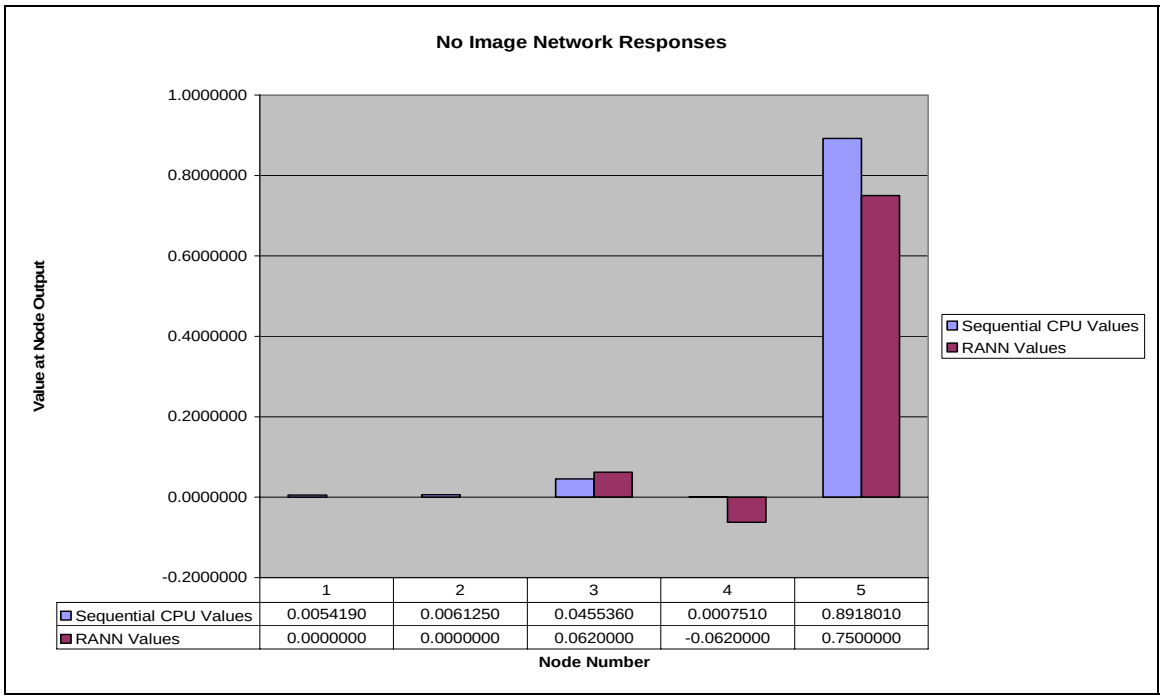
	1	2	3	4	5
Sequential CPU Values	-0.0182170	0.0020240	0.9834330	-0.0156250	0.0422410
RANN Values	0.0000000	0.0000000	1.0000000	-0.1250000	-0.1250000

Figure 15. T3 Image Network Output Comparison



	1	2	3	4	5
Sequential CPU Values	-0.0870290	0.1682970	-0.0051890	0.8545160	0.0162140
RANN Values	-0.1250000	0.1250000	0.0000000	0.6250000	-0.1250000

Figure 16. T2 Image Network Output Comparison



	1	2	3	4	5
Sequential CPU Values	0.0054190	0.0061250	0.0455360	0.0007510	0.8918010
RANN Values	0.0000000	0.0000000	0.0620000	-0.0620000	0.7500000

Figure 17. No Image Network Output Comparison

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX I. RECONFIGURABLE-ENVIRONMENT ARTIFICIAL NEURAL NETWORK (RANN) INSTRUCTION GUIDE

1. If new classification images are not being used, this step can be skipped. With new images, input-output pairing must be set by the user. The use of 'One-of-C' output is recommended for best results. Multiple similar images can be used within the same classification for training, as long as the numPatterns variable is updated with the increased images, and the trainOutput array has the increased pattern classification data. The trainOutput classifications can be reused on similar patterns that are desired to be grouped together, but the output will probably not discern between different inputs in the same classification. A change in the number of output classifications will require more output nodes to be added and changes made to the weight generation code, as well as the RANN code to support this. Similarly, increasing hidden layer nodes or changing input image sizes will require modification of both programs.

2. To test generalization, the images in testInput can be manipulated as the user desires. It is recommended to leave the first five sections of the array the same, as these are identical to the training data and can serve as a basis for comparison.

3. The weight generation program is then compiled. To compile this program in a linux environment with executable name [DEFAULT] the command is: g++ newint8.c –o [DEFAULT]. The compiled code is executed with the command: ./[DEFAULT] >> file, where 'file' is the name of the file desired for output as seen in Appendix D. The code will also create a file called 'weightout' in the directory of the executable that will contain the weight values of the trained network.

4. The 'weightout' file must be in the directory of the RANN code. The RANN code is then compiled on the SRC with the command 'make hw'. Please note that while code was added to the info file to simulate

execution of the VHDL sigmoid units, the ‘make debug’ SRC feature will not provide accurate output.

5. Upon successful completion of ‘make hw’, the RANN can be executed with ./ex07 [INPUT] >> ‘file’, where [INPUT] is the input image in hex form, and ‘file’ is the desired name of the file to which output is redirected. The redirection can be eliminated to view output on the screen. Alterations to the code will be required to stream input images in from other parallel code blocks, and thus the [INPUT] argument code would be required to be removed.

6. It is strongly recommended to either have experience in the SRC-6 programming environment, attend the 3-day Carte™ Workshop training session available by the company (see http://www.srccomp.com/TrainingSupport.htm) , or take EC 4820 prior to use of this code in order to have familiarity in its use. This set of instructions is provided as a means of enabling recreation of results.

APPENDIX J. SRC-6 EXCLUSIVE-OR COMPARITOR CODE

MAIN.C CODE:

```
static char const cvsid[] = "$Id: main.c,v 2.1 2005/06/14 22:16:48 jls Exp $";

#include <libmap.h>
#include <stdlib.h>

void subr (int64_t I0[], int *Out0, int *Out1, int *Out2, int *Out3, int *Out4, int *Out5, int64_t *time, int mapnum);

int main (int argc, char *argv[]) {
   FILE *res_map, *res_cpu, *inimage;
   int64_t *A;
   int64_t atmp = 0;
   int sum0 = 0;
   int sum1 = 0;
   int sum2 = 0;
   int sum3 = 0;
   int sum4 = 0;
   int64_t tm;
   int mapnum = 0;
   int patnum = 0;
   char patname [6][20] = { "Error Output" , "P4 Image" , "T4 Image" , "T3 Image" , "T2 Image" , "No Image" };

   if ((res_map = fopen ("res_map", "w")) == NULL) {
      fprintf (stderr, "failed to open file 'res_map'\n");
      exit (1);
      }

   if ((res_cpu = fopen ("res_cpu", "w")) == NULL) {
      fprintf (stderr, "failed to open file 'res_cpu'\n");
      exit (1);
      }

   if (argc < 2) {
            fprintf (stderr, "Usage: ./ex07 imagefile\n");
            exit (1);
            }

   inimage = fopen (argv[ 1 ],"rt"); //input of image data- data must be 64-bit hex value array
   if ( !inimage ) {
     fprintf (stderr, "%s could not be opened./n", argv[ 1 ]);
            exit (1);
            }

   A = (int64_t*) malloc (16 * sizeof (int64_t));
   srandom (99);


   for (int j=0; j<(16); j++) {
      fscanf (inimage,"%llx",&atmp);    //loading A with image data
            A[j] = atmp;
            }
```

```
   fclose (inimage);

  map_allocate (1);

   subr (A, &sum0, &sum1, &sum2, &sum3, &sum4, &patnum, &tm, mapnum);

   printf ("%lld clocks\n", tm);

   printf ("Difference Output= Pattern 1(%d) \n", sum0);
   printf ("Difference Output= Pattern 2(%d) \n", sum1);
   printf ("Difference Output= Pattern 3(%d) \n", sum2);
   printf ("Difference Output= Pattern 4(%d) \n", sum3);
   printf ("Difference Output= Pattern 5(%d) \n", sum4);
   printf ("Closest Match is Pattern %d \n", patnum);
   printf ("Which is: %s \n", patname[patnum]);

   map_free (1);

   exit(0);
   }
```

EX07.MC CODE:

```
/* $Id: ex07.mc,v 2.1 2005/06/14 22:16:48 jls Exp $ */


#include <libmap.h>

void subr (int64_t I0[], int *Out0, int *Out1, int *Out2, int *Out3, int *Out4,
int *Out5, int64_t *time, int mapnum) {
   OBM_BANK_A (AL, int64_t, MAX_OBM_SIZE)
   int64_t t0, t1, pat1xor, pat2xor, pat3xor, pat4xor, pat5xor, patholder;
   int num1 = 16; //this is the main loop counter - 16 64-bit image values
   int i = 0;
   int sum0 = 0;
   int sum1 = 0;
   int sum2 = 0;
   int sum3 = 0;
   int sum4 = 0;
   int patnum = 0;
   int pat1pop = 0; //results from 1pat popcount
   int pat1cntr = 0; //counter for 1pat
   int pat1name = 1; //must link 1 to name "P4 Input" in main.c
   int pat2pop = 0; //results from 1pat popcount
   int pat2cntr = 0; //counter for 1pat
   int pat2name = 2; //must link 1 to name "T4 Input" in main.c
   int pat3pop = 0; //results from 1pat popcount
   int pat3cntr = 0; //counter for 1pat
   int pat3name = 3; //must link 1 to name "T3 Input" in main.c
   int pat4pop = 0; //results from 1pat popcount
   int pat4cntr = 0; //counter for 1pat
   int pat4name = 4; //must link 1 to name "T2 Input" in main.c
   int pat5pop = 0; //results from 1pat popcount
   int pat5cntr = 0; //counter for 1pat
   int pat5name = 5; //must link 1 to name "No Input" in main.c
//*****************************************************************************************************************
```

```
//*  THIS IS THE IMAGE VARIABLE SPACE
//*  Note that this is not a very useful way to store the data.  The CARTE 2.2 Programming Environment allows for
//*  const BRAM arrays which would make this storage easier to use.

   int64_t pat10 = 0x0000000000000000;
   int64_t pat11 = 0x0000000000000000;
   int64_t pat12 = 0x0000000000000000;
   int64_t pat13 = 0x0000000000000000;
   int64_t pat14 = 0x0000000000000000;
   int64_t pat15 = 0x0000000000000000;
   int64_t pat16 = 0x0000000000000000;
   int64_t pat17 = 0x0000000040080180;
   int64_t pat18 = 0xc018070080700600;
   int64_t pat19 = 0x00e01c0001c03c01;
   int64_t pat1a = 0x038030070700600e;
   int64_t pat1b = 0x0e00e01c1c01c038;
   int64_t pat1c = 0x38038070700700e0;
   int64_t pat1d = 0xe00e01c0800401c0;
   int64_t pat1e = 0x0000000000000000;
   int64_t pat1f = 0x0000000000000000;

   int64_t pat20 = 0x0000000000000000;
   int64_t pat21 = 0x0000000000000000;
   int64_t pat22 = 0x0000000000000000;
   int64_t pat23 = 0x0000000000000000;
   int64_t pat24 = 0x0000000000000000;
   int64_t pat25 = 0x0000000000000000;
   int64_t pat26 = 0x0000000000000000;
   int64_t pat27 = 0x0000000000000000;
   int64_t pat28 = 0x0000000080401008;
   int64_t pat29 = 0xc0e0381ca050140a;
   int64_t pat2a = 0x70380e072c160582;
   int64_t pat2b = 0x1e0f03c13f1f87e3;
   int64_t pat2c = 0x1e0f03c12c160582;
   int64_t pat2d = 0x70380e07a050140a;
   int64_t pat2e = 0xc0e0381c80401008;
   int64_t pat2f = 0x0000000000000000;

   int64_t pat30 = 0x0000000000000000;
   int64_t pat31 = 0x0000000000000000;
   int64_t pat32 = 0x0000000000000000;
   int64_t pat33 = 0x0000000000000000;
   int64_t pat34 = 0x0000000000000000;
   int64_t pat35 = 0x0000000000000000;
   int64_t pat36 = 0x0000000000000000;
   int64_t pat37 = 0x0000000000000000;
   int64_t pat38 = 0x0000000000000000;
   int64_t pat39 = 0x060606060f0f0f0f;
   int64_t pat3a = 0x2626262670707070;
   int64_t pat3b = 0xe0e0e0e0c0c0c0c0;
   int64_t pat3c = 0xe0e0e0e070707070;
   int64_t pat3d = 0x262626260f0f0f0f;
   int64_t pat3e = 0x0606060600000000;
   int64_t pat3f = 0x0000000000000000;

   int64_t pat40 = 0x0000000000000000;
```

```
int64_t pat41 = 0x0000000000000000;
int64_t pat42 = 0x0000000000000000;
int64_t pat43 = 0x0000000000000000;
int64_t pat44 = 0x0000000000000000;
int64_t pat45 = 0x0000000000000000;
int64_t pat46 = 0x0000000000000000;
int64_t pat47 = 0x0000000000000000;
int64_t pat48 = 0x00000040020380e0;
int64_t pat49 = 0x0707c0a00d9381b6;
int64_t pat4a = 0x3df00fff47081847;
int64_t pat4b = 0x976c375ab70c1846;
int64_t pat4c = 0x07380ffefdf000e6;
int64_t pat4d = 0x8d90604007003000;
int64_t pat4e = 0x0200000000000000;
int64_t pat4f = 0x0000000000000000;

int64_t pat50 = 0x0000000000000000;
int64_t pat51 = 0x0000000000000000;
int64_t pat52 = 0x0000000000000000;
int64_t pat53 = 0x0000000000000000;
int64_t pat54 = 0x0000000000000000;
int64_t pat55 = 0x0000000000000000;
int64_t pat56 = 0x0000000000000000;
int64_t pat57 = 0x0000000000000000;
int64_t pat58 = 0x0000000000000000;
int64_t pat59 = 0x0000000000000000;
int64_t pat5a = 0x0000000000000000;
int64_t pat5b = 0x0000000000000000;
int64_t pat5c = 0x0000000000000000;
int64_t pat5d = 0x0000000000000000;
int64_t pat5e = 0x0000000000000000;
int64_t pat5f = 0x0000000000000000;



//* END OF IMAGE VARIABLE SPACE
//*****************************************************************************************************************


DMA_CPU (CM2OBM, AL, MAP_OBM_stripe(1,"A"), I0, 1, 32*sizeof(int64_t), 0);
wait_DMA (0);


read_timer (&t0);
pat1cntr = 0;
pat2cntr = 0;
pat3cntr = 0;
pat4cntr = 0;
pat5cntr = 0;
pat1xor = AL[0] ^ pat10; //xor comparison
popcount_64 (pat1xor, &pat1pop);
pat1cntr = pat1cntr + pat1pop;
pat2xor = AL[0] ^ pat20; //xor comparison
popcount_64 (pat2xor, &pat2pop);
```

```
pat2cntr = pat2cntr + pat2pop;
pat3xor = AL[0] ^ pat30; //xor comparison
popcount_64 (pat3xor, &pat3pop);
pat3cntr = pat3cntr + pat3pop;
pat4xor = AL[0] ^ pat40; //xor comparison
popcount_64 (pat4xor, &pat4pop);
pat4cntr = pat4cntr + pat4pop;
pat5xor = AL[0] ^ pat50; //xor comparison
popcount_64 (pat5xor, &pat5pop);
pat5cntr = pat5cntr + pat5pop;
pat1xor = AL[1] ^ pat11; //xor comparison
popcount_64 (pat1xor, &pat1pop);
pat1cntr = pat1cntr + pat1pop;
pat2xor = AL[1] ^ pat21; //xor comparison
popcount_64 (pat2xor, &pat2pop);
pat2cntr = pat2cntr + pat2pop;
pat3xor = AL[1] ^ pat31; //xor comparison
popcount_64 (pat3xor, &pat3pop);
pat3cntr = pat3cntr + pat3pop;
pat4xor = AL[1] ^ pat41; //xor comparison
popcount_64 (pat4xor, &pat4pop);
pat4cntr = pat4cntr + pat4pop;
pat5xor = AL[1] ^ pat51; //xor comparison
popcount_64 (pat5xor, &pat5pop);
pat5cntr = pat5cntr + pat5pop;
pat1xor = AL[2] ^ pat12; //xor comparison
popcount_64 (pat1xor, &pat1pop);
pat1cntr = pat1cntr + pat1pop;
pat2xor = AL[2] ^ pat22; //xor comparison
popcount_64 (pat2xor, &pat2pop);
pat2cntr = pat2cntr + pat2pop;
pat3xor = AL[2] ^ pat32; //xor comparison
popcount_64 (pat3xor, &pat3pop);
pat3cntr = pat3cntr + pat3pop;
pat4xor = AL[2] ^ pat42; //xor comparison
popcount_64 (pat4xor, &pat4pop);
pat4cntr = pat4cntr + pat4pop;
pat5xor = AL[2] ^ pat52; //xor comparison
popcount_64 (pat5xor, &pat5pop);
pat5cntr = pat5cntr + pat5pop;
pat1xor = AL[3] ^ pat13; //xor comparison
popcount_64 (pat1xor, &pat1pop);
pat1cntr = pat1cntr + pat1pop;
pat2xor = AL[3] ^ pat23; //xor comparison
popcount_64 (pat2xor, &pat2pop);
pat2cntr = pat2cntr + pat2pop;
pat3xor = AL[3] ^ pat33; //xor comparison
popcount_64 (pat3xor, &pat3pop);
pat3cntr = pat3cntr + pat3pop;
pat4xor = AL[3] ^ pat43; //xor comparison
popcount_64 (pat4xor, &pat4pop);
pat4cntr = pat4cntr + pat4pop;
pat5xor = AL[3] ^ pat53; //xor comparison
popcount_64 (pat5xor, &pat5pop);
pat5cntr = pat5cntr + pat5pop;
pat1xor = AL[4] ^ pat14; //xor comparison
```

```
popcount_64 (pat1xor, &pat1pop);
pat1cntr = pat1cntr + pat1pop;
pat2xor = AL[4] ^ pat24; //xor comparison
popcount_64 (pat2xor, &pat2pop);
pat2cntr = pat2cntr + pat2pop;
pat3xor = AL[4] ^ pat34; //xor comparison
popcount_64 (pat3xor, &pat3pop);
pat3cntr = pat3cntr + pat3pop;
pat4xor = AL[4] ^ pat44; //xor comparison
popcount_64 (pat4xor, &pat4pop);
pat4cntr = pat4cntr + pat4pop;
pat5xor = AL[4] ^ pat54; //xor comparison
popcount_64 (pat5xor, &pat5pop);
pat5cntr = pat5cntr + pat5pop;
pat1xor = AL[5] ^ pat15; //xor comparison
popcount_64 (pat1xor, &pat1pop);
pat1cntr = pat1cntr + pat1pop;
pat2xor = AL[5] ^ pat25; //xor comparison
popcount_64 (pat2xor, &pat2pop);
pat2cntr = pat2cntr + pat2pop;
pat3xor = AL[5] ^ pat35; //xor comparison
popcount_64 (pat3xor, &pat3pop);
pat3cntr = pat3cntr + pat3pop;
pat4xor = AL[5] ^ pat45; //xor comparison
popcount_64 (pat4xor, &pat4pop);
pat4cntr = pat4cntr + pat4pop;
pat5xor = AL[5] ^ pat55; //xor comparison
popcount_64 (pat5xor, &pat5pop);
pat5cntr = pat5cntr + pat5pop;
pat1xor = AL[6] ^ pat16; //xor comparison
popcount_64 (pat1xor, &pat1pop);
pat1cntr = pat1cntr + pat1pop;
pat2xor = AL[6] ^ pat26; //xor comparison
popcount_64 (pat2xor, &pat2pop);
pat2cntr = pat2cntr + pat2pop;
pat3xor = AL[6] ^ pat36; //xor comparison
popcount_64 (pat3xor, &pat3pop);
pat3cntr = pat3cntr + pat3pop;
pat4xor = AL[6] ^ pat46; //xor comparison
popcount_64 (pat4xor, &pat4pop);
pat4cntr = pat4cntr + pat4pop;
pat5xor = AL[6] ^ pat56; //xor comparison
popcount_64 (pat5xor, &pat5pop);
pat5cntr = pat5cntr + pat5pop;
pat1xor = AL[7] ^ pat17; //xor comparison
popcount_64 (pat1xor, &pat1pop);
pat1cntr = pat1cntr + pat1pop;
pat2xor = AL[7] ^ pat27; //xor comparison
popcount_64 (pat2xor, &pat2pop);
pat2cntr = pat2cntr + pat2pop;
pat3xor = AL[7] ^ pat37; //xor comparison
popcount_64 (pat3xor, &pat3pop);
pat3cntr = pat3cntr + pat3pop;
pat4xor = AL[7] ^ pat47; //xor comparison
popcount_64 (pat4xor, &pat4pop);
pat4cntr = pat4cntr + pat4pop;
```

```
pat5xor = AL[7] ^ pat57; //xor comparison
popcount_64 (pat5xor, &pat5pop);
pat5cntr = pat5cntr + pat5pop;
pat1xor = AL[8] ^ pat18; //xor comparison
popcount_64 (pat1xor, &pat1pop);
pat1cntr = pat1cntr + pat1pop;
pat2xor = AL[8] ^ pat28; //xor comparison
popcount_64 (pat2xor, &pat2pop);
pat2cntr = pat2cntr + pat2pop;
pat3xor = AL[8] ^ pat38; //xor comparison
popcount_64 (pat3xor, &pat3pop);
pat3cntr = pat3cntr + pat3pop;
pat4xor = AL[8] ^ pat48; //xor comparison
popcount_64 (pat4xor, &pat4pop);
pat4cntr = pat4cntr + pat4pop;
pat5xor = AL[8] ^ pat58; //xor comparison
popcount_64 (pat5xor, &pat5pop);
pat5cntr = pat5cntr + pat5pop;
pat1xor = AL[9] ^ pat19; //xor comparison
popcount_64 (pat1xor, &pat1pop);
pat1cntr = pat1cntr + pat1pop;
pat2xor = AL[9] ^ pat29; //xor comparison
popcount_64 (pat2xor, &pat2pop);
pat2cntr = pat2cntr + pat2pop;
pat3xor = AL[9] ^ pat39; //xor comparison
popcount_64 (pat3xor, &pat3pop);
pat3cntr = pat3cntr + pat3pop;
pat4xor = AL[9] ^ pat49; //xor comparison
popcount_64 (pat4xor, &pat4pop);
pat4cntr = pat4cntr + pat4pop;
pat5xor = AL[9] ^ pat59; //xor comparison
popcount_64 (pat5xor, &pat5pop);
pat5cntr = pat5cntr + pat5pop;
pat1xor = AL[10] ^ pat1a; //xor comparison
popcount_64 (pat1xor, &pat1pop);
pat1cntr = pat1cntr + pat1pop;
pat2xor = AL[10] ^ pat2a; //xor comparison
popcount_64 (pat2xor, &pat2pop);
pat2cntr = pat2cntr + pat2pop;
pat3xor = AL[10] ^ pat3a; //xor comparison
popcount_64 (pat3xor, &pat3pop);
pat3cntr = pat3cntr + pat3pop;
pat4xor = AL[10] ^ pat4a; //xor comparison
popcount_64 (pat4xor, &pat4pop);
pat4cntr = pat4cntr + pat4pop;
pat5xor = AL[10] ^ pat5a; //xor comparison
popcount_64 (pat5xor, &pat5pop);
pat5cntr = pat5cntr + pat5pop;
pat1xor = AL[11] ^ pat1b; //xor comparison
popcount_64 (pat1xor, &pat1pop);
pat1cntr = pat1cntr + pat1pop;
pat2xor = AL[11] ^ pat2b; //xor comparison
popcount_64 (pat2xor, &pat2pop);
pat2cntr = pat2cntr + pat2pop;
pat3xor = AL[11] ^ pat3b; //xor comparison
popcount_64 (pat3xor, &pat3pop);
```

```
pat3cntr = pat3cntr + pat3pop;
pat4xor = AL[11] ^ pat4b; //xor comparison
popcount_64 (pat4xor, &pat4pop);
pat4cntr = pat4cntr + pat4pop;
pat5xor = AL[11] ^ pat5b; //xor comparison
popcount_64 (pat5xor, &pat5pop);
pat5cntr = pat5cntr + pat5pop;
pat1xor = AL[12] ^ pat1c; //xor comparison
popcount_64 (pat1xor, &pat1pop);
pat1cntr = pat1cntr + pat1pop;
pat2xor = AL[12] ^ pat2c; //xor comparison
popcount_64 (pat2xor, &pat2pop);
pat2cntr = pat2cntr + pat2pop;
pat3xor = AL[12] ^ pat3c; //xor comparison
popcount_64 (pat3xor, &pat3pop);
pat3cntr = pat3cntr + pat3pop;
pat4xor = AL[12] ^ pat4c; //xor comparison
popcount_64 (pat4xor, &pat4pop);
pat4cntr = pat4cntr + pat4pop;
pat5xor = AL[12] ^ pat5c; //xor comparison
popcount_64 (pat5xor, &pat5pop);
pat5cntr = pat5cntr + pat5pop;
pat1xor = AL[13] ^ pat1d; //xor comparison
popcount_64 (pat1xor, &pat1pop);
pat1cntr = pat1cntr + pat1pop;
pat2xor = AL[13] ^ pat2d; //xor comparison
popcount_64 (pat2xor, &pat2pop);
pat2cntr = pat2cntr + pat2pop;
pat3xor = AL[13] ^ pat3d; //xor comparison
popcount_64 (pat3xor, &pat3pop);
pat3cntr = pat3cntr + pat3pop;
pat4xor = AL[13] ^ pat4d; //xor comparison
popcount_64 (pat4xor, &pat4pop);
pat4cntr = pat4cntr + pat4pop;
pat5xor = AL[13] ^ pat5d; //xor comparison
popcount_64 (pat5xor, &pat5pop);
pat5cntr = pat5cntr + pat5pop;
pat1xor = AL[14] ^ pat1e; //xor comparison
popcount_64 (pat1xor, &pat1pop);
pat1cntr = pat1cntr + pat1pop;
pat2xor = AL[14] ^ pat2e; //xor comparison
popcount_64 (pat2xor, &pat2pop);
pat2cntr = pat2cntr + pat2pop;
pat3xor = AL[14] ^ pat3e; //xor comparison
popcount_64 (pat3xor, &pat3pop);
pat3cntr = pat3cntr + pat3pop;
pat4xor = AL[14] ^ pat4e; //xor comparison
popcount_64 (pat4xor, &pat4pop);
pat4cntr = pat4cntr + pat4pop;
pat5xor = AL[14] ^ pat5e; //xor comparison
popcount_64 (pat5xor, &pat5pop);
pat5cntr = pat5cntr + pat5pop;
pat1xor = AL[15] ^ pat1f; //xor comparison
popcount_64 (pat1xor, &pat1pop);
pat1cntr = pat1cntr + pat1pop;
pat2xor = AL[15] ^ pat2f; //xor comparison
```

```
popcount_64 (pat2xor, &pat2pop);
pat2cntr = pat2cntr + pat2pop;
pat3xor = AL[15] ^ pat3f; //xor comparison
popcount_64 (pat3xor, &pat3pop);
pat3cntr = pat3cntr + pat3pop;
pat4xor = AL[15] ^ pat4f; //xor comparison
popcount_64 (pat4xor, &pat4pop);
pat4cntr = pat4cntr + pat4pop;
pat5xor = AL[15] ^ pat5f; //xor comparison
popcount_64 (pat5xor, &pat5pop);
pat5cntr = pat5cntr + pat5pop;
patnum = pat5name;
patholder = pat5cntr;
if (pat4cntr < patholder){
        patnum = pat4name;
        patholder = pat4cntr;}
if (pat3cntr < patholder){
        patnum = pat3name;
        patholder = pat3cntr;}
if (pat2cntr < patholder){
        patnum = pat2name;
        patholder = pat2cntr;}
if (pat1cntr < patholder){
        patnum = pat1name;
        patholder = pat1cntr;}

if (patholder > 102){
        patnum = 0; }        //Threshold for "no match" criteria is 103 pixels off, 10% discrepancy

*Out0 = pat1cntr;
*Out1 = pat2cntr;
*Out2 = pat3cntr;
*Out3 = pat4cntr;
*Out4 = pat5cntr;
*Out5= patnum;

read_timer (&t1);

*time = t1 - t0;

}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX K. OUTPUT OF RECONFIGURABLE XOR COMPARITOR

```
./ex07 p4input64
65 clocks
Difference Output= Pattern 1(0)
Difference Output= Pattern 2(159)
Difference Output= Pattern 3(165)
Difference Output= Pattern 4(180)
Difference Output= Pattern 5(97)
Closest Match is Pattern 1
Which is: P4 Image

./ex07 t4input64
65 clocks
Difference Output= Pattern 1(159)
Difference Output= Pattern 2(0)
Difference Output= Pattern 3(202)
Difference Output= Pattern 4(187)
Difference Output= Pattern 5(136)
Closest Match is Pattern 2
Which is: T4 Image

./ex07 t3input64
65 clocks
Difference Output= Pattern 1(165)
Difference Output= Pattern 2(202)
Difference Output= Pattern 3(0)
Difference Output= Pattern 4(175)
Difference Output= Pattern 5(128)
Closest Match is Pattern 3
Which is: T3 Image

./ex07 t2input64
65 clocks
Difference Output= Pattern 1(180)
Difference Output= Pattern 2(187)
Difference Output= Pattern 3(175)
Difference Output= Pattern 4(0)
Difference Output= Pattern 5(143)
Closest Match is Pattern 4
Which is: T2 Image

./ex07 noinput64
60 clocks
Difference Output= Pattern 1(97)
Difference Output= Pattern 2(136)
Difference Output= Pattern 3(128)
Difference Output= Pattern 4(143)
Difference Output= Pattern 5(0)
Closest Match is Pattern 5
Which is: No Image
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX L. SEQUENTIAL-PROCESSOR EXCLUSIVE-OR (XOR) COMPARITOR CODE

```
////////////////////////////////////////////////////////////////////////////
//xorcomp.c
//Exclusive-Or Comparitor program for Image Classification in C++
//Written by LT Scott P. Bailey, USN
//NOV 2006
//This code may be freely used and modified at will
////////////////////////////////////////////////////////////////////////////

#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
using std::cout;
using std::endl;


//// Data dependent settings ////
#define numPatterns  5 //This program X-OR compares 5 'images' approximated from
//Prof. Pace's book 'Low Probability of Intercept Radar':  The order of
//images in the patimg array is:  P4, T4, T3, T2, and NoInput, an array of
//zeros.  Additional images can be appended to the end as long as 'numPatterns'
//is revised.  Note that these images are the same as those used for the neural
//network program

//// global variables ////
int patnum = 0;
int pattmp = 0; //pattern line holder
int imgtmp = 0;       //input image line holder
int xortmp = 0; //xor result line holder
int patresult[numPatterns] = {0,0,0,0,0};
FILE *inimage;

//the data
int patimg[numPatterns][32] = { { 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
0x00000000 , 0x40080180 , 0xc0180700 , 0x80700600 , 0x00e01c00 , 0x01c03c01 , 0x03803007 , 0x0700600e ,
0x0e00e01c , 0x1c01c038 , 0x38038070 , 0x700700e0 , 0xe00e01c0 , 0x800401c0 , 0x00000000 , 0x00000000 ,
0x00000000 , 0x00000000 } , { 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
0x00000000 , 0x00000000 , 0x00000000 , 0x80401008 , 0xc0e0381c , 0xa050140a , 0x70380e07 , 0x2c160582 ,
0x1e0f03c1 , 0x3f1f87e3 , 0x1e0f03c1 , 0x2c160582 , 0x70380e07 , 0xa050140a , 0xc0e0381c , 0x80401008 ,
0x00000000 , 0x00000000 } , { 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x06060606 , 0x0f0f0f0f , 0x26262626 , 0x70707070 ,
0xe0e0e0e0 , 0xc0c0c0c0 , 0xe0e0e0e0 , 0x70707070 , 0x26262626 , 0x0f0f0f0f , 0x06060606 , 0x00000000 ,
0x00000000 , 0x00000000 } , { 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
0x00000000 , 0x00000000 , 0x00000040 , 0x020380e0 , 0x0707c0a0 , 0x0d9381b6 , 0x3df00fff , 0x47081847 ,
0x976c375a , 0xb70c1846 , 0x07380ffe , 0xfdf000e6 , 0x8d906040 , 0x07003000 , 0x02000000 , 0x00000000 ,
0x00000000 , 0x00000000 } , { 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
```

```
0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 , 0x00000000 ,
0x00000000 , 0x00000000 } };
//These are the five pattern images in a multiple-subscripted array.  Additional images can be appended to the end  ^
//where the carat is as long as the const 'numPatterns' is updated to reflect the change.
char patname[numPatterns][20] = {"P4 Image" , "T4 Image" , "T3 Image" , "T2 Image" , "No Image" };
int imagearr[32];

int main (int argc, char *argv[])
{
	const int wid = 32;   //width of bitmap in bits
	const int len = 32;  //length of bitmap in lines
	int cnt1,cnt2,cnt3;
	int sumtmp = 0;
	int lowdelta = 0;
	int nameindex = 0;
	int trials = 0;  //placeholder for 10000 trials loop
	time_t start, finish; //timing variables
	double timediff = 0; //difference calc
	if (argc < 2) {
		fprintf (stderr, "Usage: ./xorcomp inputfile\n");
		exit (1);
		}
	inimage = fopen (argv[1],"rt");  //input of image data, 32 lines of 32-bit hex ascii (8 char) each.

	if ( !inimage ) {
		fprintf (stderr, "%s could not be opened.\n", argv[ 1 ]);
		exit (1);
		}
	for (cnt1 = 0; cnt1 < len; cnt1++)
	{
	fscanf(inimage,"%lx",&imgtmp);
	imagearr[cnt1] = imgtmp;
	}

	fclose(inimage);

	start = clock();

	for (trials = 0; trials < 10000; trials++)
	{
	patresult[0] = 0;
	patresult[1] = 0;
	patresult[2] = 0;
	patresult[3] = 0;
	patresult[4] = 0;


	for (cnt1 = 0; cnt1 < len; cnt1++)
	{
		for (cnt2 = 0; cnt2 < numPatterns; cnt2++)
		{
		pattmp = patimg[cnt2][cnt1];
		xortmp = imagearr[cnt1] ^ pattmp;
		for (cnt3 = 0; cnt3 < wid; cnt3++)
```

```
                        {
                        sumtmp = sumtmp + (abs(xortmp%2)); //counts ones in the least significant value
position
                        xortmp >> 1;                    //shifts to move ones to the right.
                        } // end inner for loop and one line of code
                patresult[cnt2] = patresult[cnt2] + sumtmp;
                sumtmp = 0;
                } //end middle loop, resetting sumtmp and updating patresult
        }       //end outer loop and should have complete hex bitmap
                //ready for use in the SRC
        }       //end of trials loop
        finish = clock();
        timediff = ((double)(finish - start))/CLOCKS_PER_SEC;
        printf("Time to complete 10000 trials (in seconds): %.3ft \n",timediff);
        printf("Number of Different bits for P4 Image -->(%d) \n",patresult[0]);
        printf("Number of Different bits for T4 Image -->(%d) \n",patresult[1]);
        printf("Number of Different bits for T3 Image -->(%d) \n",patresult[2]);
        printf("Number of Different bits for T2 Image -->(%d) \n",patresult[3]);
        printf("Number of Different bits for No Image -->(%d) \n",patresult[4]);
        lowdelta = patresult[0];
        for (cnt1 = 1; cnt1 < numPatterns; cnt1++)
        {
                if (lowdelta > patresult[cnt1]) {
                        lowdelta = patresult[cnt1];
                        nameindex = cnt1;
                        }
        }
        printf("Since the lowest delta is %d, this image most closely resembles: \n",lowdelta);
        cout << patname[nameindex] << endl;


 return 0;
}//end main
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] Pace, P. E., *Detecting and Classifying Low Probability of Intercept Radar*, Boston Massachusetts: Artech House Inc., 2004.

[2] Stoffel, K. M., "Implementation of a Quadrature Mirror Filter Bank on an SRC Reconfigurable Computer for Real-Time Signal Processing," M.S. thesis, Naval Postgraduate School, Monterey, California, 2006.

[3] Brown, D. A., "ELINT Signal Processing on Reconfigurable Computers for Detection and Classification of LPI Emitters," M.S. thesis, Naval Postgraduate School, Monterey, California, 2006.

[4] SRC Computers, Inc., *IMPLICIT + EXPLICIT™ Architecture* White Paper, April 2005.

[5] Hagan, M.T., Demuth, H.B., and Beale, M.H., *Neural Network Design*, Boulder, Colorado: University of Colorado at Boulder, 2002.

[6] DARPA Neural Network Study, AFCEA International Press, November 1988

[7] Sarle, W. S., *Neural Networks FAQ*, Posting to USENET newsgroup comp.ai.neural-nets. Retrieved November 8, 2006.

[8] Hagan, M.T., Demuth, H.B., and Beale, M.H., *Neural Network Design*, Boulder, Colorado: University of Colorado at Boulder, 2002. pp.10-2, 11-2

[9] Filippas, J., Arochena, H., Amin, S.A., Naguib, R.N.G., and Bennett, M.K., "Comparison of two AI methods for colonic tissue image classification," *Engineering in Medicine and Biology Society, 2003. Proceedings of the 25th Annual International Conference of the IEEE* , vol.2, pp. 1323-1326, 17-21 September 2003

[10] Nishimura, K., Kishida, S., and Watanabe, T., "Effects of preprocessing and layered neural networks on individual identification," *Circuits and Systems, 2004. MWSCAS '04. The 2004 47th Midwest Symposium*, vol.3, pp. iii- 73-6 vol.3, 25-28, July 2004.

[11] Pace, P. E., *Detecting and Classifying Low Probability of Intercept Radar*, Boston Massachusetts: Artech House Inc., 2004. p.302

[12] Pace, P. E., *Detecting and Classifying Low Probability of Intercept Radar*, Boston Massachusetts: Artech House Inc., 2004. p.52

[13] 'Bitmap' program manual page, bundled with Linux Operating System version 2.6.9-1.667 by Red Hat Software, Inc. (Fedora Core 2).

[14] Mack, T. J., "Implementation of a high-speed numeric function generator on a COTS reconfigurable computer," M.S. Thesis, Naval Postgraduate School, Monterey, CA, June 2007, preprint.

[15] Sasao, T., Butler, J. T., and Riedel, M. D., "Application of LUT cascades to numerical function generators," *The 12th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI2004)*, Kanazawa, Japan, pp.422-429, October 18-19, 2004.

[16] Nissen, S., *Implementation of a Fast Artificial Neural Network Library (FANN)*, Graduate Project Report, Department of Computer Science, University of Copenhagen (DIKU), October, 2003. Retrieved from <http://prdownloads.sourceforge.net/fann/fann_doc_complete_1.0.pdf?download> [12NOV06]

[17] Brierley, P., *MLP Neural Network Source code in C++*. Retrieved from <http://www.philbrierley.com/code.html> [5DEC05]

[18] SRC Computers Inc., "SRC-6 C Programming Environment V2.1 Guide," SRC-007-16, SRC Computers Inc., Colorado Springs, Colorado, August 31, 2005.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chairman, Code EC
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

4. Alan Hunsberger
   National Security Agency
   Ft. Meade, Maryland

5. Douglas J. Fouts
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

6. Jon T. Butler
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

7. Phillip E. Pace
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, Califonia

8. David Caliga
   SRC Computers, Inc.
   Colorado Springs, Colorado

9. Jon Huppenthal
   SRC Computers, Inc.
   Colorado Springs, Colorado

10. Tom Mack
    Naval Postgraduate School
    Monterey, Califonia

11. Njuguna Macaria
Naval Postgraduate School
Monterey, Califonia