A MULTICHIP NEUROMORPHIC MOTION PROCESSOR FOR EXTRACTING EGOMOTION INFORMATION

by

Shaikh Arif Shams

A Thesis Submitted to the Faculty of the
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE

In the Graduate College
THE UNIVERSITY OF ARIZONA

2 0 0 0

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED:_____

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

_____ _____

Charles M. Higgins
Assistant Professor of
Electrical and Computer Engineering

Date

ACKNOWLEDGEMENTS

At first, I like to thank my advisor, Charles M. Higgins. His guidance throughout the project was invaluable to me.

This work was carried out at the National Science Foundation's State/Industry/University Cooperative Research Centers' (NSF-S/IUCRC) Center for Low Power Electronics (CLPE). CLPE is supported by the NSF (Grant #EEC-9523338), the State of Arizona, and the following companies and foundations: Conexant, Gain Technology, Intel Corporation, Medtronic Microelectronics Center, Microchip Technology, Motorola, Inc., The Motorola Foundation, ON Semiconductor, Philips Semiconductors, Raytheon, Syncron Technologies, LLT, Texas Instruments and Western Design Center.

I also like to thank all the members of our lab for good advice and personal help.

DEDICATION

To my parents.

Table of Contents

Table of Contents - *continued*

List of Figures

List of Figures - *continued*

List of Figures - *continued*

List of Figures - *continued*

List of Figures - *continued*

List of Figures - *continued*

ABSTRACT

In this thesis a hardware motion processor is presented that is sensitive to complex spatial patterns of motion in the visual field; for example, patterns of expansion, contraction, and rotation which might be encountered in the self-motion of a robotic system. Biological motion processing strategies use multiple stages of simple parallel processors. However, the extent of pixel-parallel focal plane image processing is limited by pixel area and imager fill factor. Addition of more processing dramatically reduces the number of pixels on a reasonable-sized die. An obvious solution to this dilemma is the use of a multiple-chip system. The motion computation is split into more than one chip: a photosensitive sender, motion processing transceiver chips and an integrating receiver. An asynchronous digital inter-chip communication protocol is used in communication between analog VLSI chips designed on neuromorphic principles. This multichip motion processor retains the primary advantages of focal plane neuromorphic image processing: low power consumption, continuous-time operation, and small size. The sender chip detects moving edges in the image focused onto it and transmits that information to the transceivers. Using the position and timing information of edges detected by the sender chip, each pixel of the transceiver chip computes the local one-dimensional velocity of moving edges and transmits that information to the receiver chip. The receiver chip spatially integrates the motion information coming from the transceiver chips to produce sensitivity to patterns of optical flow. Using EPROMs on the way from sender to transceivers, the visual field is rotated such that each identical transceiver chip detects a different direction of motion. The information from the transceivers is combined by a routing processor in prearranged patterns and transmitted to the receiver chip. This multichip neuromorphic motion processor is ideal for sophisticated, real-time onboard sensors for autonomous robotics applications.

1 Introduction

1.1 Background

Determination of self-motion is a useful computation for any entity that moves within and interacts with its environment. A passive (and biologically relevant) method for accomplishing this is by observing patterns of visual motion. When an observer moves through its environment, the images of objects move across its retina. This retinal image motion, or optic flow, contains important information about the observer's instantaneous rotation and translation and the relative distances of the points in the environment. Psychophysical studies show that by using this visual information an observer can determine its heading when it moves either in a straight line [1]-[6] or in a curved path [7],[8]. To determine curved path motion, the visual system should compute both translational and rotational components of the observer's motion from retinal image information. Gibson [9] originally noted that translation of an observer through a stationary environment generates a radial pattern of optical flow, in which the observer's heading direction is specified by the focus of expansion (FOE). This is true only for pure translational motion. The focus of expansion shifts or ceases to exist for any rotational motion of the observer. In general, the singular point location in the visual field contains important egomotion information.

Optic flow is computationally intensive to process in real time and since real time autonomous navigation is usually tightly constrained in power consumption, it requires power efficient computation of optic flow. Considering this, analog VLSI technology used in a neuromorphic approach has a number of advantages over traditional digital technology to process optic flow. Due to the use of focal plane computation, in the neuromorphic VLSI approach there is no image transfer bottleneck, which is present in the conventional approach using a CCD camera together with a DSP processor. To avoid temporal aliasing, a high frame rate is maintained in the conventional CCD-DSP

approach, which results in high power consumption. Frames from a neuromorphic VLSI CMOS imager, on the other hand, are sampled on demand, i.e., the output is data driven. This results in low power consumption and temporal aliasing-free computation. Despite the benefits of higher fill factor and reprogrammability, conventional real time computer vision systems are bulky and run in discrete time. Neuromorphic systems, on the other hand, are run in continuous time and are smaller and lighter weight. Also, standard CCD cameras do not compensate for local variations in illumination, as do neuromorphic VLSI image sensors.

Pixel parallel focal plane image processing requires larger pixel size as we incorporate more and more processing circuitry in the pixel. A CMOS imager performing no explicit image processing can be built using only one transistor along with the photosensitive element [10]. A few more transistors and capacitors need to be added to allow adaptation to the mean light level over a wide range [11]. Focal plane motion processing requires even more transistors and capacitors [12]-[14]. If more processing such as optical flow field smoothing [15] or discontinuity detection are desired, transistor count increases significantly. In redesigning a CMOS imager to be a focal plane motion processor, the imager fill factor (percentage of pixel area dedicated for collecting light) drops from above 80% to less than 5% while the pixel area grows by a factor of more than ten [16]. If an intermediate computation can be communicated off the photosensitive chip without losing the advantages of focal plane computation, the effective focal plane processing can be extended while retaining practical pixel resolutions.

Communication between chips must be done without incurring significant delays, dramatically increasing power consumption, or introducing temporal aliasing in order to retain the advantages of continuous time focal plane image processing in a single chip. One way to accomplish this task is to use a digital, asynchronous, low-latency interchip communication protocol [16]. Communication between chips takes place only when there is a change in visual input, resulting in low power consumption and utilization of maximum bus bandwidth.

In this work, a multichip motion processor capable of performing egomotion computation is built using three different kinds of CMOS VLSI chips which were custom designed using neuromorphic principles [17]. This motion processor computes real time optical flow using the interchip communication protocol mentioned above.

1.2 Related Work

Volumes of research (see [18] for a review) have been performed in extracting self-motion information. Although many algorithms exist for determining the singular points of the flow field in complex egomotion, only a few of them are well suited for real-time implementation. Fermuller et al. [18]-[20] bounds the location of FOE/AOR. Burger et al. [21] determines a 2-D region of possible FOE-locations. Barth et al. [22] uses motion parallax to calculate the FOE. Jain [23] uses vector back-projection to calculate the FOE.

Attempts to perform egomotion processing in integrated hardware have only begun recently. Indiveri et al. [24] uses the zero crossing in a 1-D array of CMOS velocity sensors to detect one component of the focus of expansion. In another chip, the sum of a radial array of velocity sensors is used to compute the rate of flow field expansion, from which the time-to-contact can be calculated. McQuirk [25] has built a CCD-based image processor in which an iterative algorithm is used to locate consistent stable points in the image, and the focus of expansion results. Deutschmann et al. [26] extend Indiveri et al.'s work to 2-D by summing rows and columns in a 2-D CMOS motion sensor array. Software is used to detect zero crossings and find the flow field singular point. Higgins et al. [27] compute the position of singular point and self-motion information using the sign of optical flow.

The interchip communication protocol used in this work was originally envisioned by Mahowald [28] as a circuit analogy to the optic nerve. This protocol was used to transmit visual signals out of a silicon retina. For the same purpose, Boahen [29]

strengthened and formalized this protocol. In the last few years, several variants of this scheme have emerged [30]-[32]. This asynchronous interchip communication protocol has been successfully used in neuromorphic sensory processing in recent years. Boahen [33] implemented binocular disparity-selective elements by interfacing two silicon retinas to three receiver chips. Vernier et al. [34] implemented orientation-selective receptive fields by using an asynchronous interface to a silicon retina. Andreou et al. [35] used EPROMs for linear or nonlinear address remapping in interchip communication. Kumar et al. [36] built an auditory front-end chip with an asynchronous interface for further off-chip processing. Higgins et al. [16] used this protocol to build three-chip motion processors to compute complex visual motion.

An asynchronous architecture for mapping address events, presented in [37], will be useful in making the motion processor, presented in this work, perform more efficiently.

1.3 Organization of the Thesis

Chapter 2 presents the background of the multichip neuromorphic motion processor. The AER communication protocol is discussed. The block diagram of the motion processor is presented. The neuromorphic hardware components used in this motion processor along with the algorithm used in computing motion are discussed. Chapter 3 discusses how this motion processor can be made sensitive to complex patterns of motion. Chapter 4 presents experimental results and comparison of the results with theoretical predictions. Finally, chapter 5 concludes the thesis with some recommendations for further research.

2 Hardware Architecture

2.1 Background

The number of input/output pins in chips is very limited. For this reason, time multiplexing of the communication channels is necessary in multichip systems [38]. Usually, pixel activity of vision chips with adaptive silicon retinas [39] is sparse. Sequential multiplexing techniques make inefficient use of the communication channel bandwidth [40] while transmitting data from these chips. On the other hand, asynchronous event-based communication protocol is very efficient for multichip neuromorphic systems [41], [42]. For such systems, the Address-Event Representation (AER), which has been used in this work for interchip communication, is the most well-developed method. In this representation, “events” are digital pulses with continuous interval. A silicon retina using AER sends digital pulses encoding the position of the activated pixel. The temporal pattern of these digital pulses carries the analog information of the signal.

Two digital control lines and several digital address lines are utilized to interface a sender chip to a receiver chip in the original form of AER, as shown in Figure 2.1. In this protocol, the occurrence of a binary event is transmitted from the sender to the receiver in continuous time. A four-phase asynchronous handshake between sender and receiver guarantees reliable communication between them. The spatial position of a requesting sender pixel is transmitted over the address lines to the receiver chip and the event is forwarded to the corresponding receiver pixel.

Any sender pixel can communicate digital spikes to the corresponding receiver pixel using this protocol. Requests can come from any pixel in the array at any instant. Therefore, an arbitration scheme on the sender is necessary to serialize simultaneous

events onto the single communications bus. Because the asynchronous protocol operates so quickly (on nanosecond scales) relative to the timescale of visual stimuli (on millisecond scales), the serialization caused by sharing of a single digital bus is not a serious problem in sensor applications. Various arbitration schemes exist [28], [31], [43]. A binary tree arbiter [44] has been used in this work. This scheme yields a quick decision and scales well to large array sizes. The hardware implementation of AER used in this chipset has been described in [44].

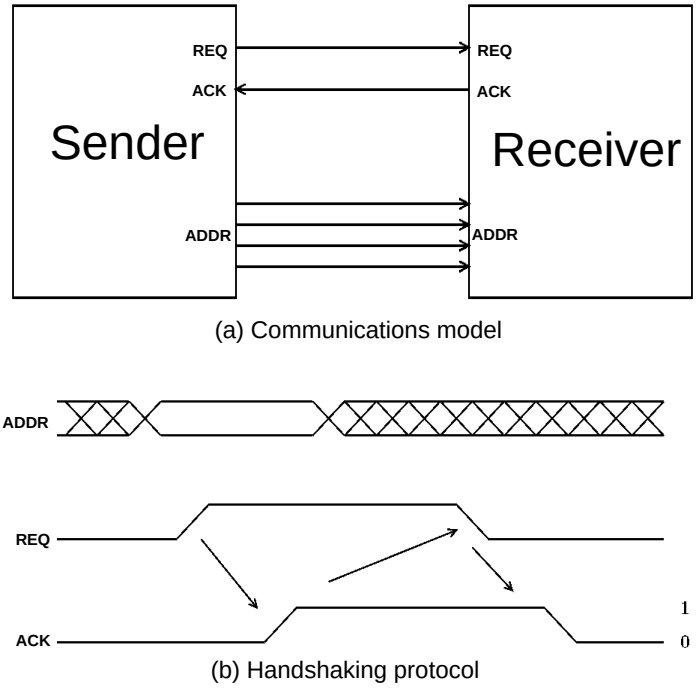


Figure 2.1: AER protocol summary. (a) the model for AER transmission: a sender chip communicates with a receiver chip via request, acknowledge and address lines. (b) the handshaking protocol for transmission using the above control and address lines: a request with a valid address leads to an acknowledgment, which in turn leads to falling request and falling acknowledge.

2.2 Multichip System

The hardware architecture of the multichip neuromorphic motion processor is described here using a block diagram.

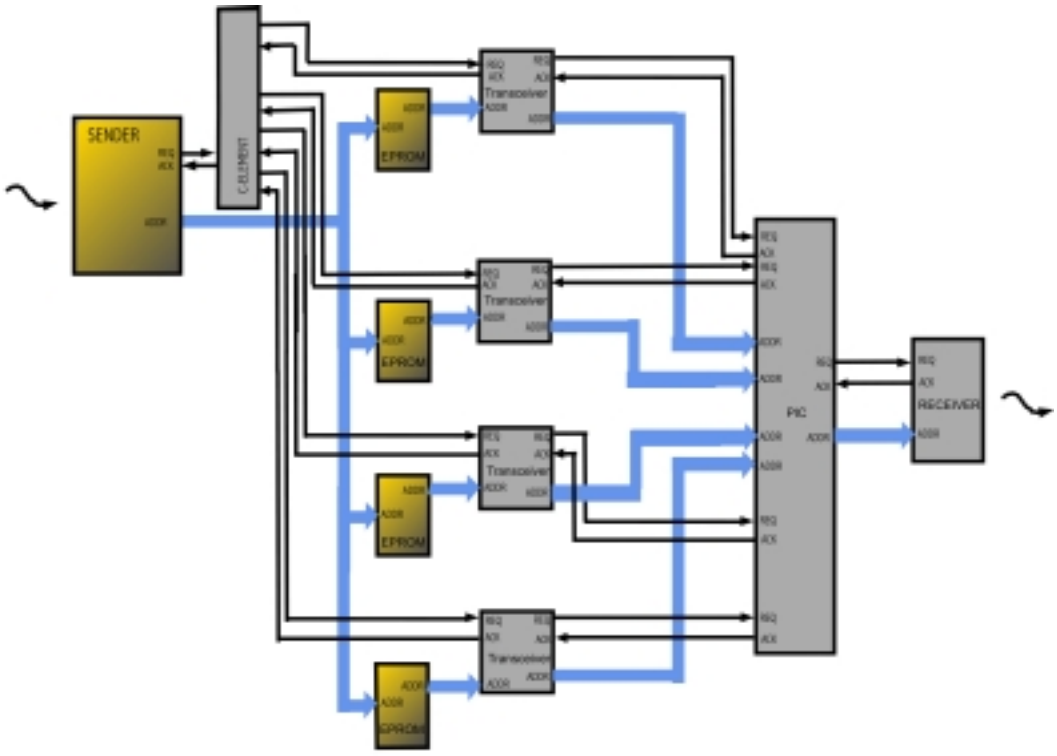


Figure 2.2: Block diagram of the multichip motion processor developed in this work.

The block diagram of the developed multichip neuromorphic motion processor is shown in Figure 2.2. The sender chip detects moving contrast edges in an image focused directly upon it. Edge information is transmitted to four transceiver chips via the AER bus. These transceiver chips perform motion computation. All transceivers are sensitive to the motion in a particular direction. In order to get sensitivity to the motion in four different directions by the four identical transceivers, the visual field is rotated using an EPROM on the way of the address lines from the sender to each transceiver chip. The EPROM can rotate the visual field to any desired direction. Four different EPROMs, corresponding to four different transceivers, are performing rotation to four different directions, 0°, 90°, 180° and 270°. Because four transceiver chips are present, circuitry is necessary to ensure that all four transceiver chips have acknowledged the single sender event before the system continues. This circuit is known as a C-element. The information from the transceiver chips is combined by a routing processor (PIC) in prearranged

patterns and transmitted to the receiver chip. The transceivers communicate with the PIC through the request, acknowledge and address lines. The PIC is programmed in such a way that when more than one transceiver is making requests then it selects an arbitrary transceiver among them. It uses the address bits coming from that transceiver and a tag representing the corresponding transceiver when it communicates with the receiver. Different mappings have been performed by the PIC for different experiments.

2.3 Hardware Components

Three custom VLSI chips have been designed on neuromorphic principles [17] and used in this work to form the multichip motion processor. They are

i) photosensitive sender

ii) motion processing transceiver

iii) integrating receiver

These chips compute real-time optical flow using the AER communication protocol.

2.3.1 Photosensitive Sender Chip

The sender detects moving contrast edges in the image directly focused onto it. This chip is the visual front end for all further processing. AER circuitry communicates the edge locations from the sender chip to the transceiver chips.

There is 14 X 12 array of pixels in the core of the sender chip. The layout diagram of this chip is shown in Figure 2.3. There are an adaptive photoreceptor [11], a non-linear differentiator circuit [45] and interchip communication circuitry in each sender pixel. The circuit diagram of a pixel is shown in Figure 2.4.

The photoreceptor output (V_{prout}) adapts to the local light intensity on slow time scales (a few seconds), providing high gain for transient signals that are centered on the adaptation point. It works over a wide range of illumination without a change in bias

settings. The adaptive photoreceptor has been designed using only four transistors, a photodiode and two explicit capacitors. The transistor M_2 has been used as the adaptive element. A feedback to the gate of the transistor M_1 helps in long term averaging the input signal. The inverting amplifier consisting of the transistors M_3 and M_4 amplifies the input signal. The output V_{prout} has been fed back through M_2 and the capacitor divider formed from C_1 and C_2. C_2 stores the adaptation state. [11] describes the operation of the adaptive photoreceptor in greater detail.

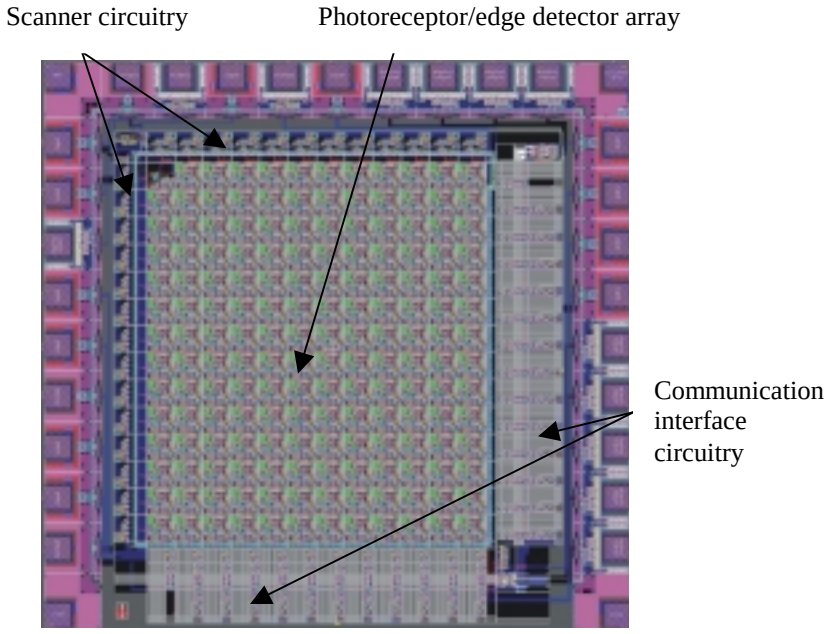


Figure 2.3: Layout of the sender chip fabricated on a MOSIS tiny chip in a 1.2 μm standard CMOS process.

The passage of a spatial edge causes a sudden change in local light intensity and causes a change in photoreceptor output. The nonlinear differentiator circuit responds with a current pulse (I_{out}) on any sudden change in the photoreceptor output. Therefore this combined circuit consisting of adaptive photoreceptor and nonlinear differentiator is termed a temporal edge detector (TED). An operational amplifier has been formed using the transistors M_5-M_9 with a bias $V_{hysbias}$. The transistors M_{10}-M_{15} and the capacitors C_3-C_5 form a nonlinear feedback configuration. The capacitor divider consisting of C_4 and

C_5 sets the feedback voltage gain. The adaptive element M_{14} is identical to that used in the photoreceptor. It helps the V- node of the amplifier adapt slowly to V_a. The nonlinear differentiator circuit of [45] has been modified in this chip to allow response to both signs of derivative. As a result, the TED responds to both bright-to-dark and dark-to-bright edges and the density of the produced flow fields increases greatly.

The V_{leak} bias sets a current threshold which transistor M_{16} must overcome to lower the voltage V_{mem} in the communication interface circuit. An event is communicated off-chip when V_{mem} is lowered. The communications interface circuitry of the sender pixel is shown in Figure 2.4. Before a request is generated from the pixel, R_{pix} is (inactive) low, A_{pix} is (inactive) high, and D_{pix} is (inactive) low. When sufficient current is integrated on node V_{mem} that it overcomes the threshold set by V_{thr}, wired-OR R_{pix} (shared by all pixels in a row) is pulled high. When the row arbiter makes A_{pix} low, V_{mem} gets reset to V_{dd} and as a result R_{pix} is released. It also pulls up wired-OR D_{pix} (shared by all pixels in a column). D_{pix} remains high until A_{pix} becomes (inactive) high. Only the change of illumination causes events to be communicated on the bus. This results in a very efficient use of bus bandwidth. Arbitration, address encoding, and other interface circuitry to support the AER protocol are located in the periphery of the sender chip and discussed in [44]. Serial scanner circuitry is also incorporated in this chip to get the raw data from the photoreceptor.

Multiple requests (“a burst of spikes”) from a single sender pixel are created on the AER bus during the period of time when the nonlinear differentiator’s current output is large enough to overcome the leakage current. Figure 2.5 shows a typical burst from a single pixel. Each spike in the burst is arbitrated independently to the bus. Therefore bursts from different sender pixels may be found interleaved on the interchip communication bus.

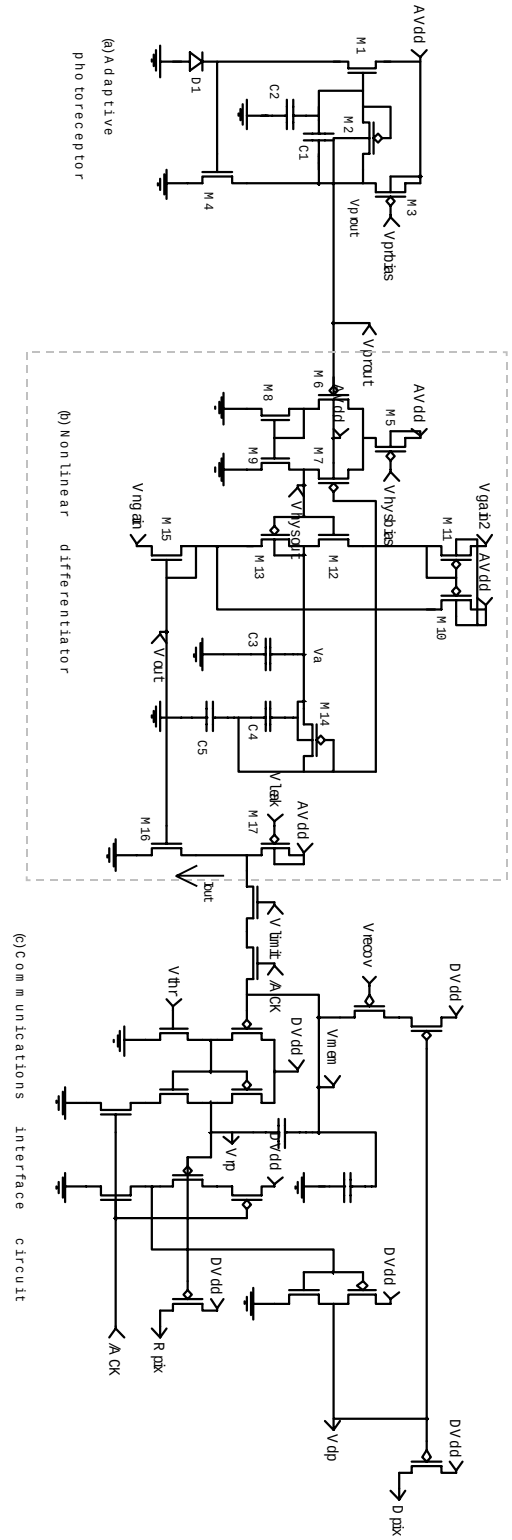


Figure 2.4: Sender pixel circuitry. (a) Adaptive photoreceptor, (b) Nonlinear differentiator and (c) Communications interface circuit.

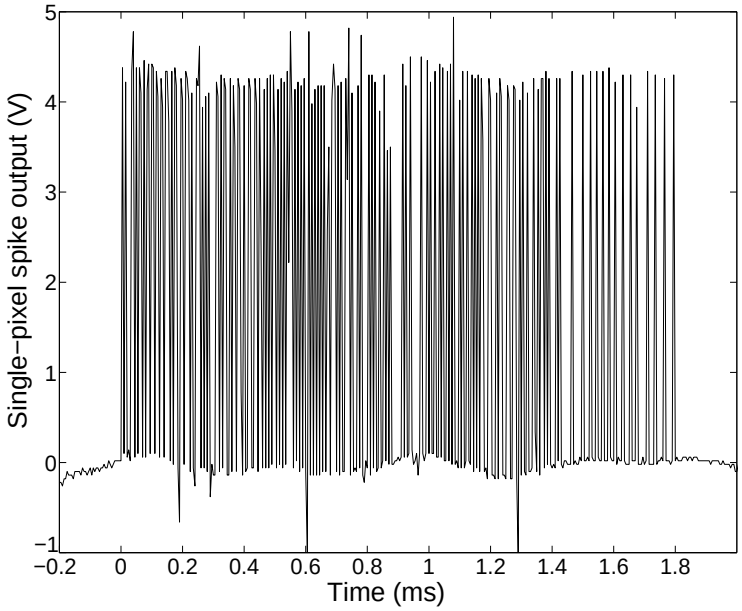


Figure 2.5: Sender pixel burst response; this spike train is the response of an individual pixel to a passing edge. Since true spike width is approximately 50 ns and spike separation is on the order of 6 μs, the spike duration has been lengthened to make individual spikes visible. This 1.8 ms burst peaks at a spike rate of approximately 160 kHz and encompasses around 200 individual spikes. The stimulus edge occurred at approximately t = -15 ms.

2.3.2 Motion Transceiver Chip

This chip calculates one-dimensional (1-D) optical flow vectors from the edge information on the AER bus.

There is a 12 X 12 array of transceiver pixels in the core of the transceiver chip. Figure 2.6 shows the layout diagram of this chip. Each transceiver pixel contains the communications interface circuitry to communicate both with the sender and with the receiver peripheral circuitry and a motion circuit implementing a 1-D version of the ITI (Inhibit, Trigger, and Inhibit) motion algorithm. Using only nearest-neighbor connectivity, this circuit uses the pulses coming from the interface circuit on the receiving side of the transceiver to produce voltages encoding the direction of motion. These voltages are then converted into a bi-directional current that drives the interface circuit on the sending side.

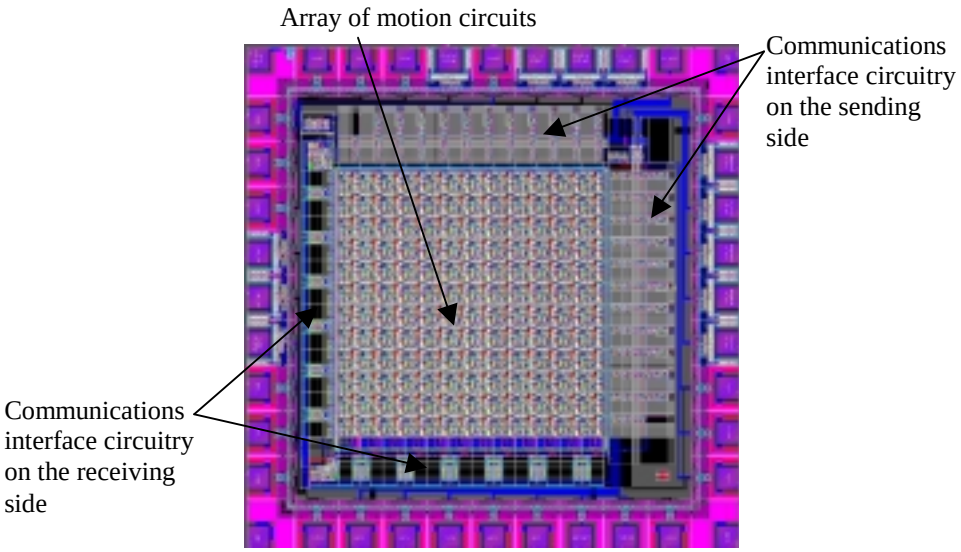


Figure 2.6: Layout of the transceiver chip fabricated on a MOSIS tiny chip in a 1.2µm standard CMOS process.

The pulses coming into each 1-D motion sensor represent the order in which the corresponding photoreceptors have been crossed by a moving spatial edge. This motion sensor suffers from the aperture problem like all local motion sensors. So, this can only respond to the component of optical flow normal to the local gradients of light intensity. This 1-D computation gives the sign of the projection of the normal flow vector onto the sender chip orientation.

The basis of the ITI motion algorithm is a voltage signal that is pulled high (triggered) by one TED of the sender chip and pulled low (inhibited) by a neighboring TED of the sender chip. The 1-D ITI motion algorithm is diagrammed in Figure 2.7. A moving edge must cross both TEDs of the sender chip in order to be detected. If the triggering TED is crossed last, the voltage signal remains high, representing the passing of an edge in the preferred direction. If the inhibiting TED is crossed last, the voltage goes high briefly and becomes low after the edge passes. The ITI motion circuit was inspired by the FTI sensor of Kramer [46].

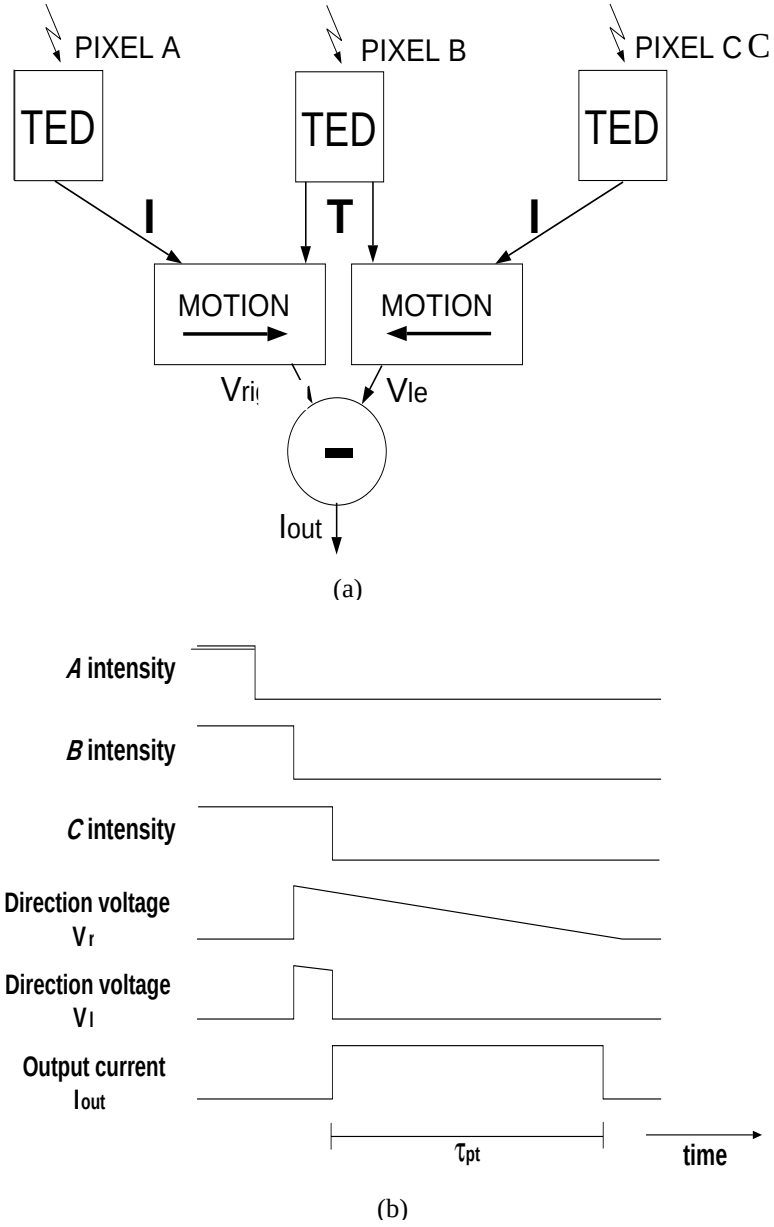


Figure 2.7: ITI motion algorithm; (a) Block diagram. (b) Simulated traces. An edge crossing from left to right pulls high (triggers) direction voltages for both directions V_r and V_l in pixel B. The same edge subsequently crossing pixel C pulls low (inhibits) the incorrect direction voltage V_l. The resulting positive current I_{out} indicates rightward motion. Pixels B and A interact similarly to detect leftward motion, resulting in a negative current I_{out}.

When an edge crosses any pixel of the sender chip, the AER pulse communicates to the corresponding pixel in the transceiver chip, thereby triggering the direction voltages for both directions. The neighboring pixel of the sender chip, crossed by the

edge next, communicates with the corresponding pixel of the transceiver chip over the AER interface and this pulse inhibits the direction voltage in the incorrect direction, leaving only the correct direction of motion activated. No signal current flows to or from the AER interface circuit to the receiver until one of the two direction voltages is inhibited, because the signal current is produced only when there is a difference between the two direction voltages.

A pair of neighboring 1-D pixels communicates with each other using only two wires. Inhibiting pulses flow in opposite directions through these wires. Since a pixel has two neighbors each 1-D pixel is connected to its neighbors through a total of four wires. Two of them are entering and the other two are leaving.

The communications interface circuit on the receiving side of the transceiver shown in Figure 2.8 is much simpler than the interface circuit on the sending side. When X_{sel} and Y_{sel} are both active high, R_{in} becomes low and V_{rmem} grows in magnitude depending on V_{qua} and V_{tau} bias settings. CMOS inverters are added to digitize the voltage level and generate V_{out} and V_{outbar} for use in the motion circuit.

Figure 2.8 also shows the schematic diagram of the ITI motion circuit together with the communications interface circuit on the sending and the receiving side. The voltage signal V_{outbar} results from the local TED of the sender chip and pulls up (triggers) both direction voltages V_l and V_r to V_{dd}. The signals V_{left} and V_{right} are the signals V_{out} coming from neighboring pixels to the left and right respectively, and are used to pull down (inhibit) the direction voltages. The capacitors are linearly discharged through the transistors with constant gate bias V_{leak}. This determines the persistence time τ_{pt} of the motion vectors. The current output circuit shown in Figure 2.8 computes the difference between the two direction voltages. This current is the input to the AER interface circuit of the transceiver on the sending side, which operates in the same way as the communications interface circuit of the sender chip described earlier. If one of the two direction voltages V_l and V_r is high then the sign of this current is determined by which of them is high. Only a small mismatch current is produced when both the direction voltages are high. The voltage V_{lim} sets the magnitude of the current.

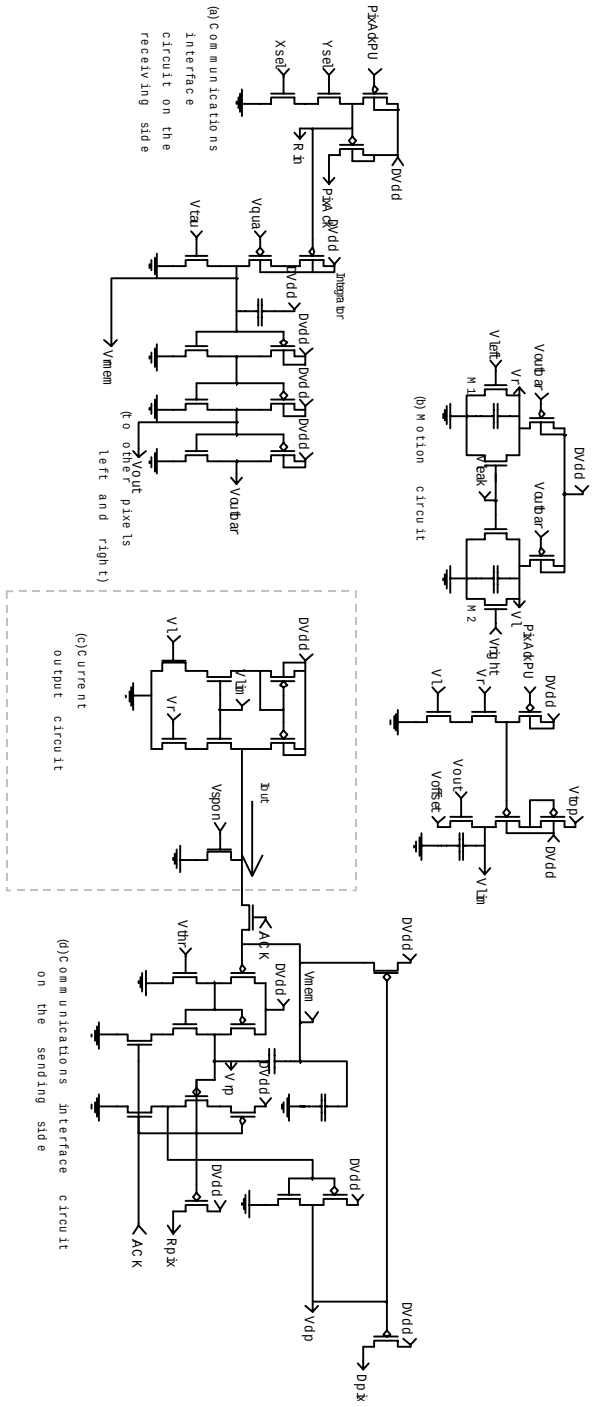


Figure 2.8: Transceiver pixel circuits. (a) Communications interface circuit on the receiving side. (b) Motion circuit. The local TED signal V_{outbar}, coming through AER interface circuit, is used to pull high the direction voltages V_r and V_l when an edge passes. V_{left} and V_{right} are the signals V_{out} coming from neighboring pixels, and are used to pull the direction voltages low. The bias V_{leak} sets the persistence time τ_{pt} of the motion signal. (c) Current output circuit. The state of the direction voltages determines the sign of the bi-directional output current I_{out}. V_{lim} sets its magnitude. (d) Communications interface circuit on the sending side.

2.3.3 Integrating Receiver Chip

There is a 27 X 29 array of receiver pixels in the core of the receiver chip. Figure 2.9 shows the layout diagram of this chip. Along with the communications interface circuit each receiver pixel contains a circuit to integrate spikes received from the interface circuitry. This circuit is shown in Figure 2.10 together with the communications interface circuit. Address decoding and interface circuitry to support the protocol are placed in the periphery and are discussed in [44]. A serial scanner has been incorporated in this chip for readout of the integrated output.

When X_{sel} and Y_{sel} are both active high, R_{pix} becomes low and V_{mem} grows in magnitude depending on V_{qua} and V_{tau} bias settings.

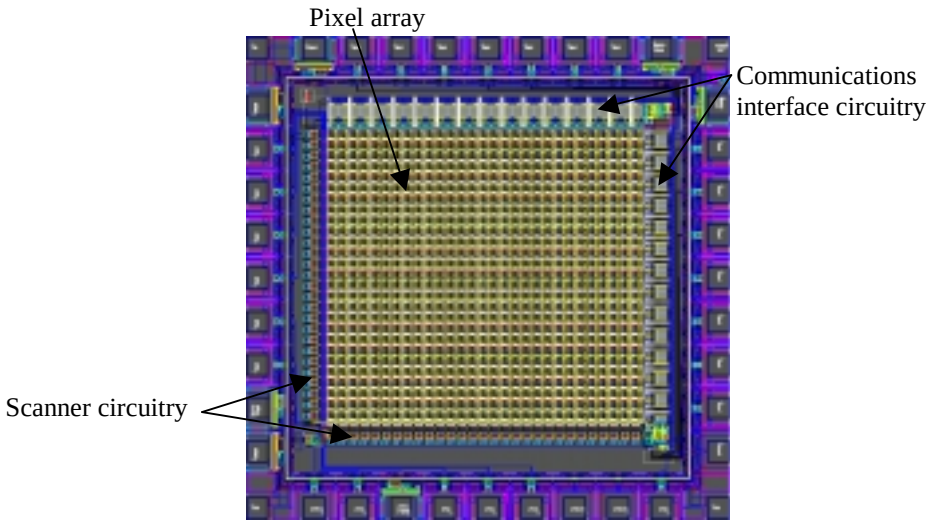


Figure 2.9: Layout of the receiver chip fabricated on a MOSIS tiny chip in a 1.2μm standard CMOS process.

Referring to Figure 2.10, each spike coming from the AER communications interface circuit to a particular receiver pixel pulls down R_{pix} of that pixel which in turn makes a charging current flowing through transistor M1 to charge the output capacitor. V_{qua} sets the magnitude of the charging current. There is a charging current whenever a spike comes in. The constant discharging current is set by V_{tau}. There is a particular spike frequency f_0 for which the charging and discharging current are the same on average

keeping V_{mem} at a steady state value. If the frequency is just above this equilibrium value V_{mem} does not discharge fully, resulting in a build up of voltage on the capacitor. If the frequency is just below this equilibrium value V_{mem} does not get charged as much as it gets discharged, making the voltage on the capacitor drop. The final value of the capacitor voltage depends on the number of spikes which came in. For an infinite number of spikes the voltage vs. frequency curve shows a sharp rise from V_{oL} to V_{oH}. However, for a smaller number of spikes this curve is smoother because for the frequencies near f_0 sufficient charge is not accumulated/discharged to reach the upper or lower voltage limits. The spike width, and thus the charge delivered, is almost the same for each spike. Figure 2.11 shows a sketch of V_{mem} versus frequency for different number of spikes. The equilibrium frequency can be calculated as follows.

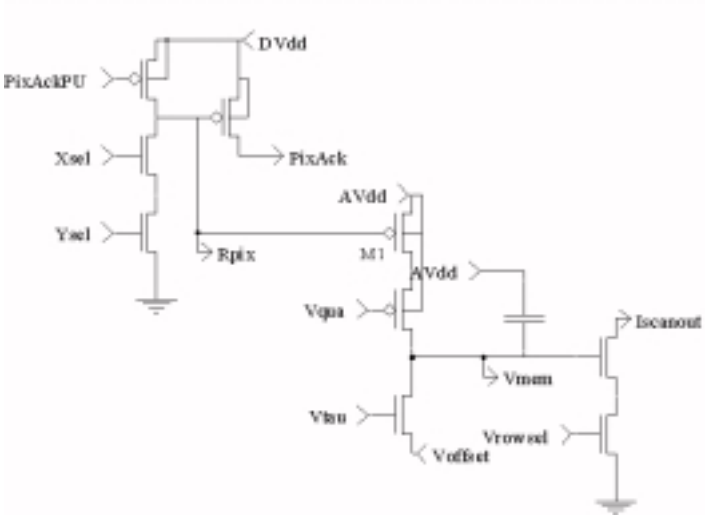


Figure 2.10: Receiver pixel consisting of an integrating circuit together with the communications interface circuit.

If the quantity of charge transferred to the capacitor per spike is Q, then the average charging current is fQ, where f = spike frequency. If the constant discharging current is I_τ, then at the steady state V_{mem} condition, $I_\tau = f_0 Q$. So the equilibrium frequency $f_0 = I_\tau/Q$. As I_τ is roughly proportional to $e^{V\tau}$ and Q is roughly proportional to e^{Vqua}, f_0 can be approximated as $f_0 = e^{V\tau}/Te^{Vqua}$, where T is the spike width. f_0 can thus be

set by adjusting the relative setting of the V_{qua} and V_τ biases. The absolute V_τ setting is also important in determining the decay time constant of the integration.

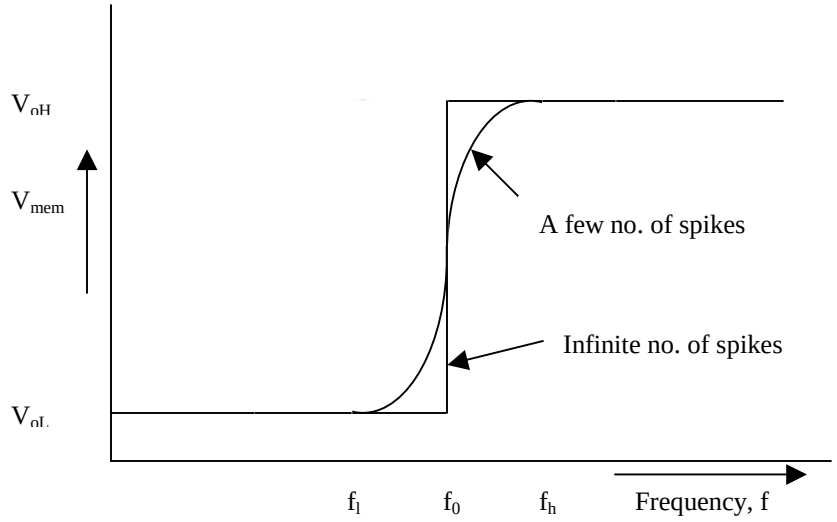


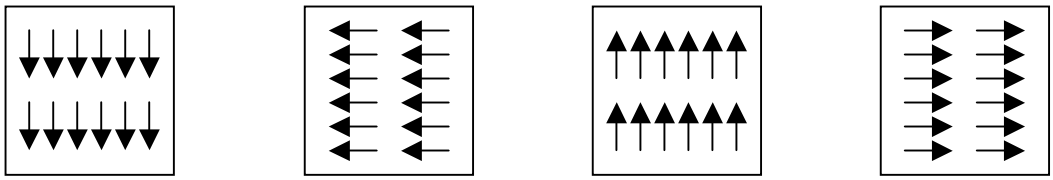
Figure 2.11: V_{mem} vs. frequency sketch.

Referring to the plot for a small number of spikes in Figure 2.11, we don't get any signal output for spike frequencies less than f_l. This is the lower limit of the spike frequency of this motion processor to have signal output. Output varies with frequency in the frequency range from f_l to f_h. f_h is the threshold frequency above which the output does not vary. So, each receiver pixel effectively performs a thresholding operation.

3 Synthesis of Complex Motion Units

3.1 Introduction

The four transceivers perform motion computation with the mapped address field received from four corresponding EPROMs. The combination of a transceiver and corresponding EPROM is termed a channel hereafter. So, we have four channels sensitive to four different directions of motion as shown in Figure 3.1. Downward motion is considered the reference 0°, because this is the preferred direction of motion for the motion chip without any rotation of address field. A channel is sensitive if there is any component of motion in its preferred direction. To accomplish the goal of this project, i.e. to build a motion processor with the ability to detect complex patterns of motion such as expansion, contraction and rotation, different parts of the flow field from different channels are combined.



(a) (b) (c) (d)

Figure 3.1: Four channels sensitive to four different direction of motion: (a) 0°, (b) 90°, (c) 180° and (d) 270°. Downward motion is considered the reference 0°, because this is the preferred direction of motion for the motion chip without any rotation of address field. Rotations are performed clock-wise. For example, 90° rotation makes leftward motion and so on.

3.2 Expansion-sensitive Unit

To make the motion processor sensitive to expanding patterns with the focus of expansion at the center of the field of view, the leading half of the flow fields from all

four channels are combined as shown in Figure 3.2. As a result of this combination, the motion processor becomes sensitive to the resultant flow field as shown in Figure 3.3, which has been obtained by adding the flow fields of Figure 3.2. The transceiver core has 12 X 12 pixels. In practical implementation, the leading 6 X 12 pixels are combined to have the required flow field combination as shown in Figure 3.4 where the pixels to be combined are colored black. An arrow is drawn at the top of the pixel representation of each channel to show its preferred direction. The reason behind this combination of pixels to implement an expansion-sensitive unit with FOE at the center of the visual field is obvious from the following analysis.

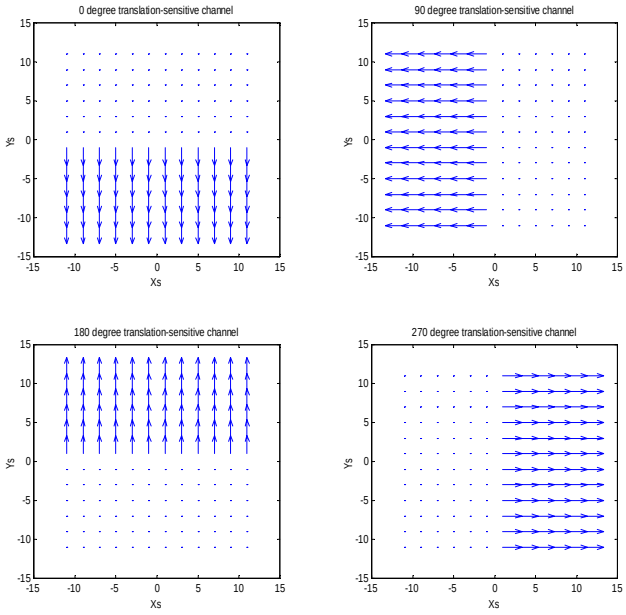


Figure 3.2: Flow fields from each channel are shown that are to be combined to synthesize an expansion-sensitive unit with the FOE at the center of the visual field.

Summing the responses of the black colored pixels (Figure 3.4) from only one of the four channels at a time gives rise to the output as shown in Figure 3.5 in response to the expanding stimulus for varying FOE positions. In calculating theoretical values, it is assumed that any pixel gives rise to a response 1 if it finds any component of the stimulus in the preferred direction. It contributes a value 0 otherwise. The contributions from all the pixels have been added to get the final values of Figure 3.5. Linear addition of these

plots results in the plot shown in Figure 3.6 that is exactly the response of an expansion-tuned unit (with the FOE at the center of the visual field) to the expanding stimulus for varying FOE positions showing the maximum response for the FOE position at the center of the field of view. All four active channels give rise to this plot in Figure 3.6. So, the combination of pixels mentioned above makes the motion processor sensitive to the expanding patterns with FOE at the center of the visual field.

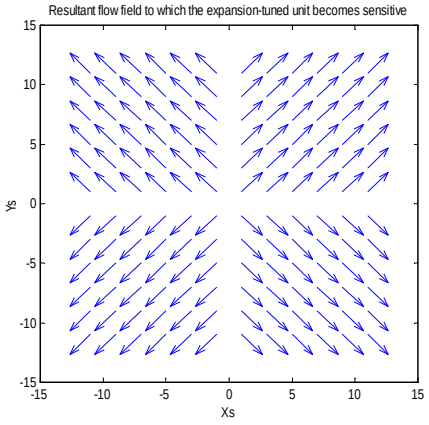


Figure 3.3: Resultant flow field to which the unit becomes sensitive as a result of the flow field combination specified in Figure 3.2.

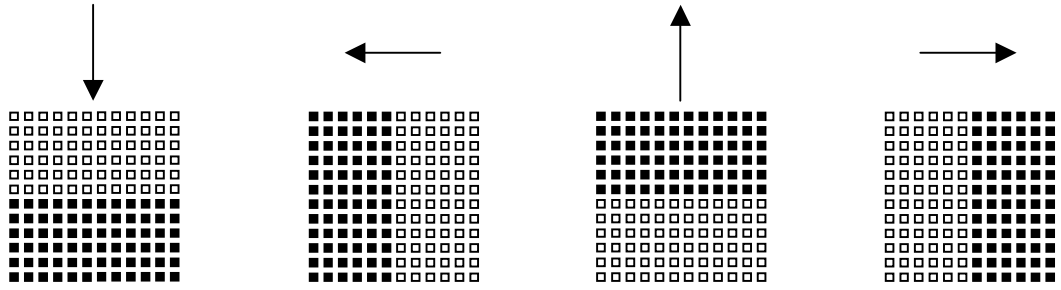


Figure 3.4: The pixels from each transceiver to be combined to make the processor sensitive to expanding patterns (with the FOE at the center of the visual field) are shown in black.

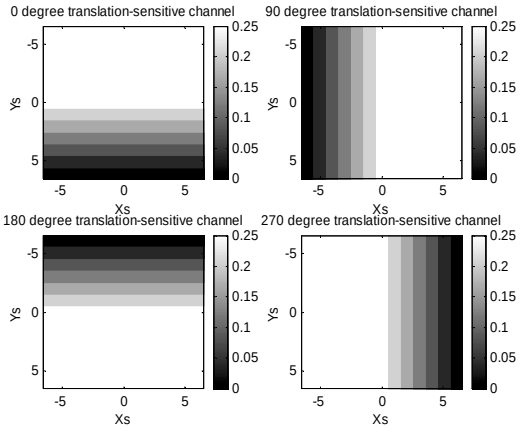


Figure 3.5: Theoretical prediction from individual channels in response to expanding patterns. (Xs, Ys) represents the coordinates of the FOE in pixels. Theoretically predicted value is represented by brightness.

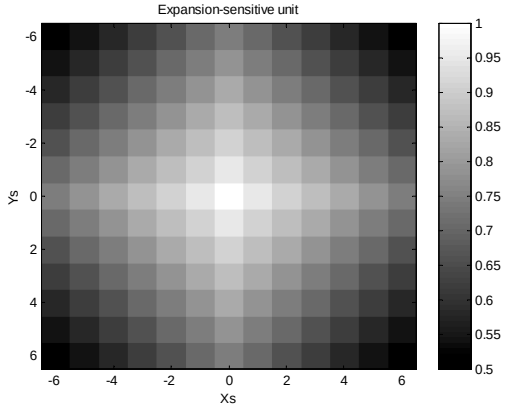


Figure 3.6: Theoretical prediction from expansion-sensitive unit with FOE at the center of the field of view in response to expanding patterns. (Xs, Ys) represents the coordinates of the FOE in pixels. Theoretically predicted value is represented by brightness.

3.3 Contraction-sensitive Unit

In order to make the processor sensitive to contracting patterns with the focus of contraction at the center of the field of view, the trailing half of the flow fields from all the channels have to be combined. As a result of this combination, the motion processor becomes sensitive to the resultant flow field as shown in Figure 3.7, which has been obtained by adding the component flow fields. The combination of the pixels of the

transceivers in practical implementation is shown in Figure 3.8 where the pixels to be combined are shown in black color.

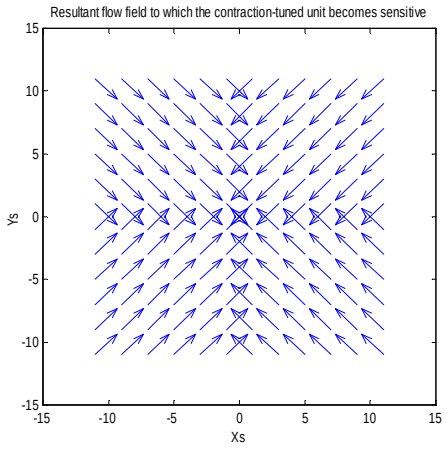


Figure 3.7: Resultant flow field to which the unit becomes sensitive as a result of the specified flow field combination for contraction.

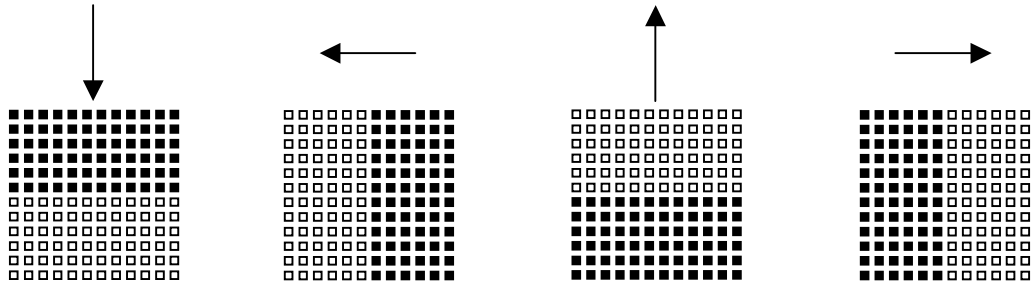


Figure 3.8: The pixels from each transceiver to be combined to make the processor sensitive to contracting patterns are shown in black.

Combining the black colored pixels (Figure 3.8) from only one of the four channels at a time gives rise to the output as shown in Figure 3.9 in response to the contracting stimulus for varying FOC positions. Linear addition of these plots results in the same plot shown in Figure 3.6 that is exactly the response of a contraction-sensitive unit (with the FOC at the center of the visual field) to the contracting stimulus for varying FOC positions showing the maximum response for the FOC position at the center of the field of view. All four active channels give rise to this plot in Figure 3.6. So, the

combination of pixels mentioned above makes the motion processor sensitive to the contracting patterns with FOC at the center of the visual field.

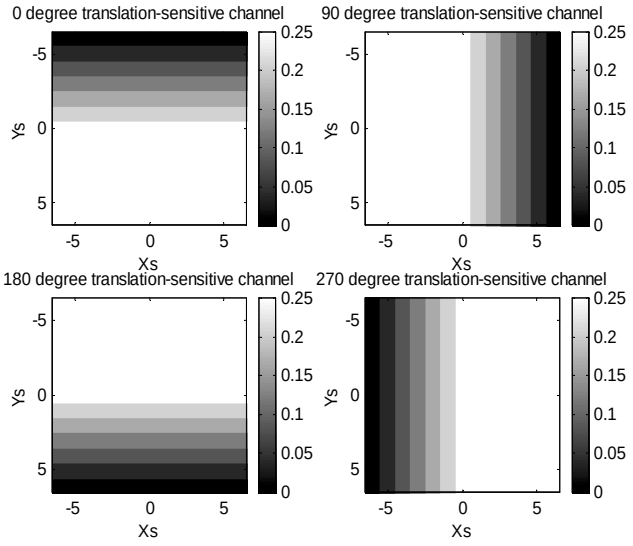


Figure 3.9: Theoretical prediction from individual channels in response to contracting patterns. (Xs, Ys) represents the coordinates of the FOC in pixels. Theoretically predicted value is represented by brightness.

3.4 Rotation-sensitive Units

To make the motion processor sensitive to CCW and CW rotating patterns with the axis of rotation at the center of the visual field the right and left half of the flow fields from all the channels should be combined respectively. As a result of these combinations, the motion processor becomes sensitive to the resultant flow fields as shown in Figure 3.10 and Figure 3.11 respectively which have been obtained by adding the respective component flow fields. The required pixels of the transceivers to combine in practical implementation of the CCW and CW rotation-tuned units are shown in black color in Figure 3.12 and Figure 3.13 respectively. The theoretical prediction from CCW/CW rotation-sensitive units with AOR at the center of the field of view in response to CCW/CW rotating patterns is also same as that shown in Figure 3.6.

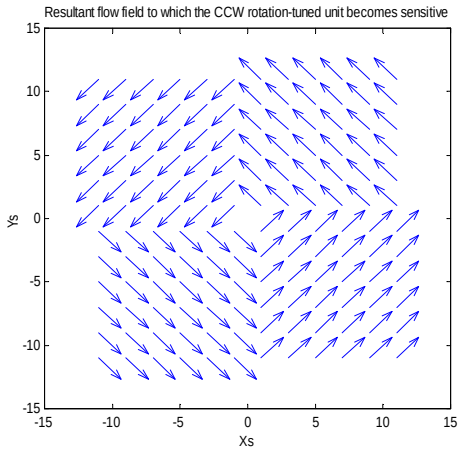


Figure 3.10: Resultant flow field to which the unit becomes sensitive as a result of the specified flow field combination for CCW rotation.

Although the four units above (expansion, contraction, CCW rotation and CW rotation) have been implemented individually all of them can be implemented simultaneously by combining the appropriate transceiver pixels and mapping them to four different receiver pixels simultaneously. In fact, in this mapping every transceiver pixel is used exactly twice, i.e. each transceiver pixel is mapped to exactly two receiver pixels. For each request coming from a transceiver, the routing processor sends two requests to the receiver.

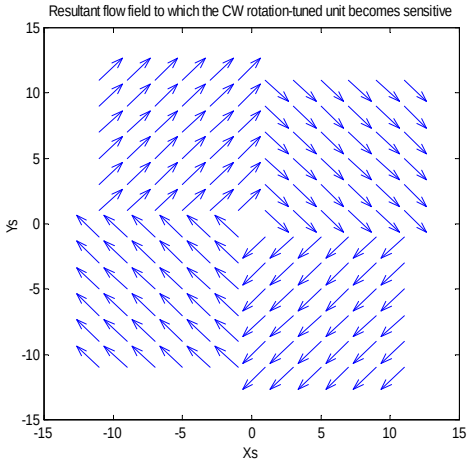


Figure 3.11: Resultant flow field to which the unit becomes sensitive as a result of the specified flow field combination for CW rotation.

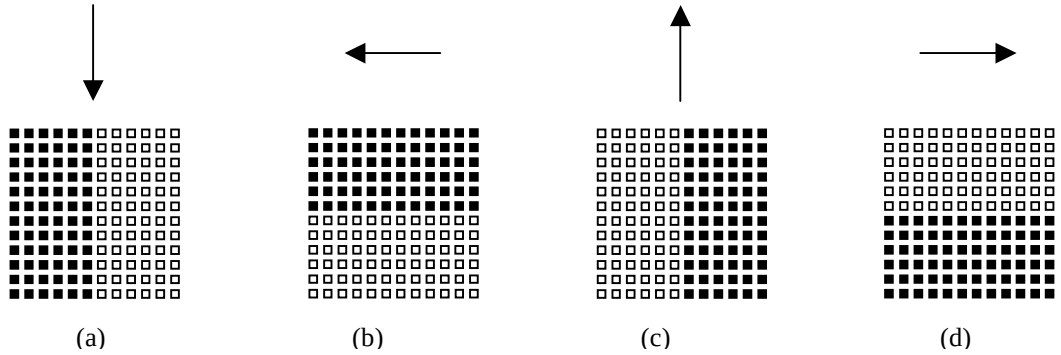


Figure 3.12: The pixels from each transceiver to be combined to make the processor sensitive to CCW rotating patterns are shown in black.

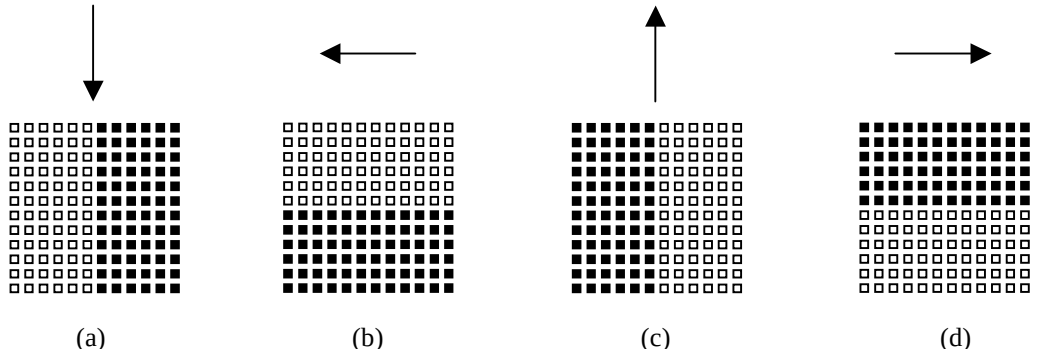


Figure 3.13: The pixels from each transceiver to be combined to make the processor sensitive to CW rotating patterns are shown in black.

3.5 Varying Receptive Field

The receptive field of a unit is the portion of the visual field used in synthesizing that unit. In order to implement the desired patterns of connection, the routing processor (PIC) used in this motion processor sends, in general, a number of requests to the receiver for each request coming from any transceiver. The next incoming request can be processed only after all the requests to the receiver (generated by the PIC for the earlier transceiver request) have been acknowledged. This slows the system down. This overhead can be reduced by using a partial receptive field of the system instead of the full

receptive field. Using partial receptive fields allows the routing processor to make less requests to perform the same tasks in many applications.

In previous sections, full receptive fields of the chips have been used to derive the units sensitive to expansion, contraction, CCW rotation and CW rotation with the FOE/FOC/AOR at the center of the field of view, because the entire half of the pixels of each transceiver chip is combined to synthesize them. It is also possible to use reduced receptive field or, in other words, to combine less than the entire half of the pixels to synthesize these units. For example, we can combine 6 X 3 pixels from each transceiver as shown in Figure 3.14 to have a receptive field of 6 X 6 pixels to synthesize expansion-tuned unit with FOE at the center of the visual field. Some performance will obviously be lost in these combinations. Figure 3.15 shows the theoretical prediction of output from this expansion-tuned unit in response to the expanding stimulus for varying positions of FOE over the field of view of the chip. This plot can be compared to Figure 3.6 to see the performance loss in using the partial receptive field.

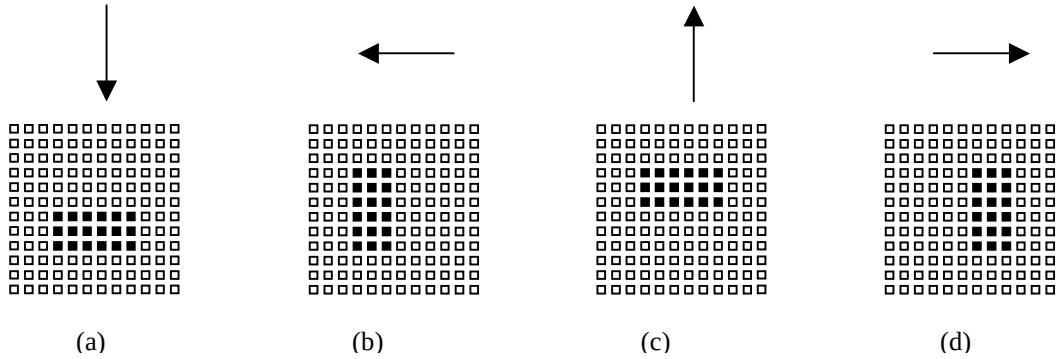


Figure 3.14: The pixels from each transceiver to be combined to make the processor sensitive to expanding patterns using partial receptive field (6X6 in this case) are shown in black.

Although we are losing some performance in using the partial receptive field, the motion processor can be made to perform more tasks by using the unused pixels of the reduced receptive field implementation. For example, we can also combine the unused pixels from each transceiver of Figure 3.14 to perform some more tasks. As shown in Figure 3.16, by mapping the black colored pixels to one receiver pixel and the gray colored pixels to another, the motion processor can be made sensitive to large and small expanding fields at the same time. Figure 3.17 shows the theoretical prediction from the

unit tuned to large expanding fields while Figure 3.15 is the theoretical prediction from the unit tuned to small expanding fields.

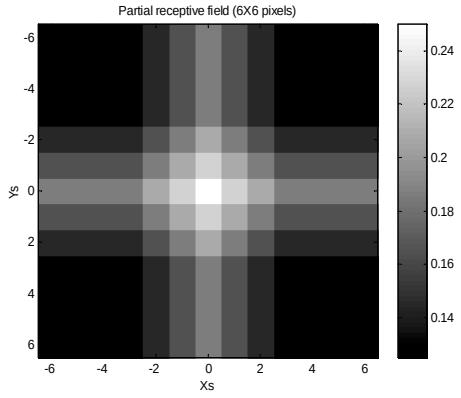
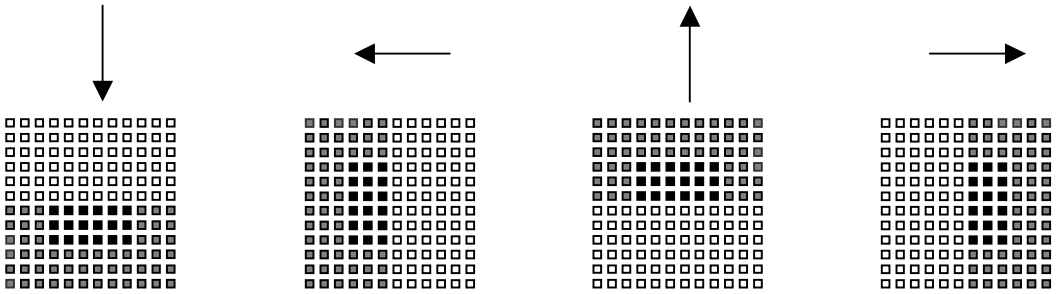


Figure 3.15: Theoretical prediction from expansion-tuned unit using partial receptive field (6X6 in this case) in response to expanding patterns. (Xs, Ys) represents the coordinates of the FOE in pixels. Theoretically predicted value is represented by brightness.



(a) (b) (c) (d)

Figure 3.16: The pixels from each transceiver to be combined to make the processor sensitive to large and small expanding fields simultaneously are shown in black and gray color respectively.

3.6 Off-Centered FOE Tuning

The motion processor can also be made sensitive to expanding patterns with FOE not centered in the visual field. Figure 3.18 shows the pixels to be combined to make a unit sensitive to expanding patterns with an off-center FOE. Figure 3.19 shows the corresponding theoretical prediction.

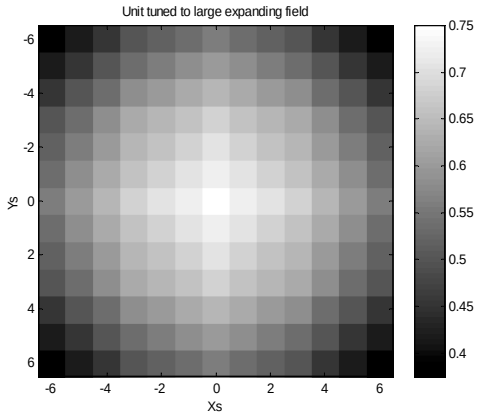


Figure 3.17: Theoretical prediction from expansion-sensitive unit tuned to large expanding field. (Xs, Ys) represents the coordinates of the FOE in pixels. Theoretically predicted value is represented by brightness.

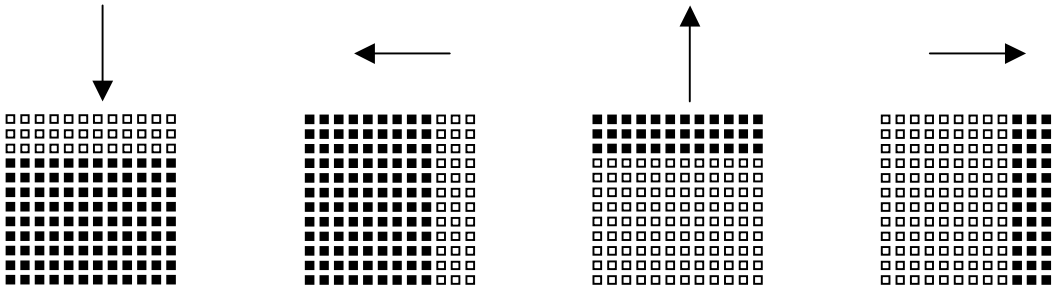


Figure 3.18: The pixels from each transceiver to be combined to make the processor sensitive to expanding patterns with an off-centered FOE are shown in black.

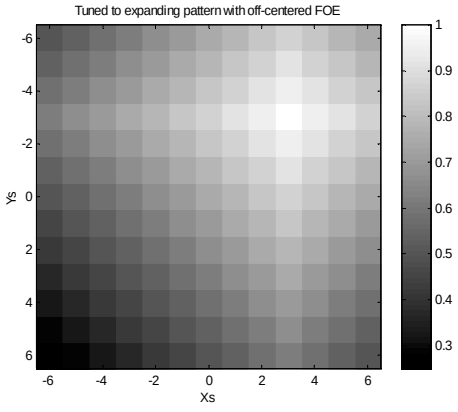


Figure 3.19: Theoretical prediction from expansion-sensitive unit tuned to expanding patterns with an off-centered FOE. (Xs, Ys) represents the coordinates of the FOE in pixels. Theoretically predicted value is represented by brightness.

Figure 3.20 shows the pixels to be combined from each transceiver with different symbols to make the motion processor sensitive to expanding patterns with four different positions of FOE simultaneously. This implementation utilizes the benefit from using the reduced receptive field. Using the reduced receptive field, we get four units simultaneously tuned to expanding patterns with four different FOE positions without slowing down the system from the implementation of Figure 3.18. The pixels of four different symbols from each transceiver are mapped to four different pixels of the receiver to have four units tuned to expanding patterns with four different positions of FOE. Figure 3.21 shows the theoretical predictions for these four units. To make the system tuned to four FOE positions simultaneously using the mapping like Figure 3.18, four requests must go out of the PIC for each input request, resulting in a slower response. In the reduced receptive field implementation of these simultaneous tuned units, only one request goes out of the PIC for each input request.

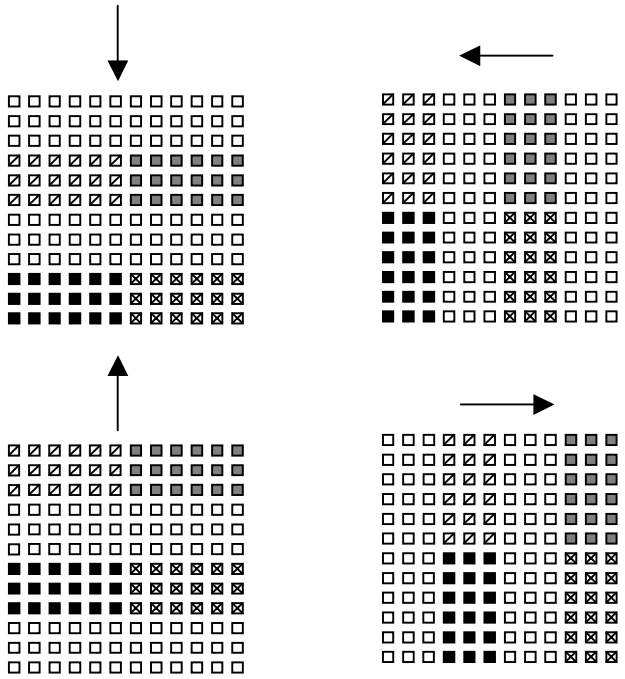


Figure 3.20: The pixels from each transceiver to be combined to make the processor sensitive to expanding patterns with different positions of FOE simultaneously are shown with different symbols.

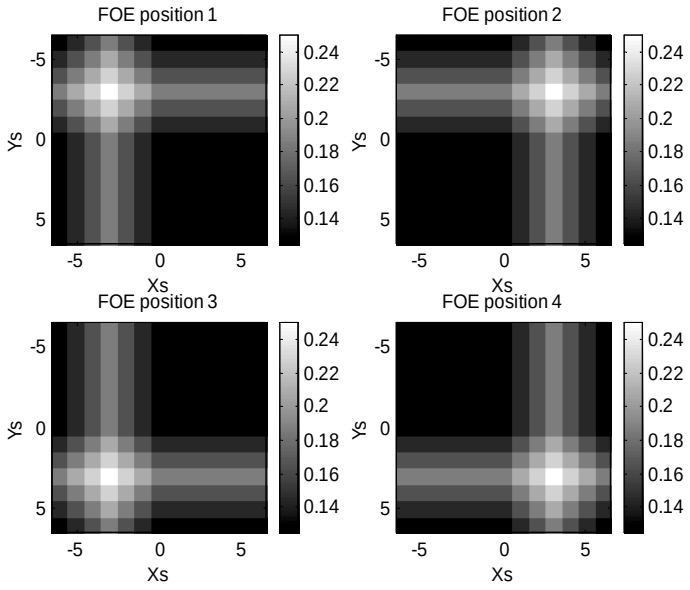


Figure 3.21: Theoretical outputs from four expansion-sensitive units tuned to four different positions of FOE. (Xs, Ys) represents the coordinates of the FOE in pixels. Theoretically predicted value is represented by brightness.

3.7 Larger FOE Region

The heading direction of an observer is specified by the FOE only for a purely translational motion of the observer [9]. Slight deviation from pure translational motion shifts the position of the singular point. Sensitivity to the expanding patterns with a larger region of FOE might be helpful in extracting motion information in such cases.

All the expansion-sensitive units discussed so far are sensitive to expanding patterns with FOE within a very small region. The theoretical predictions having sharp peak drawn so far show this tiny region.

For the expansion-sensitive unit tuned to expanding patterns with FOE at the central region of the visual field, the distances between the four central pixels determine this small central region. These units can be made sensitive to expanding patterns with a larger region of FOE positions by combining less or more than the half of the pixels from each transceiver as shown in Figure 3.22 and Figure 3.23 respectively. Figure 3.24 and

Figure 3.25 show the theoretical predictions from the units corresponding to the mapping in Figure 3.22 and Figure 3.23 respectively in response to expanding stimulus with varying FOE positions.

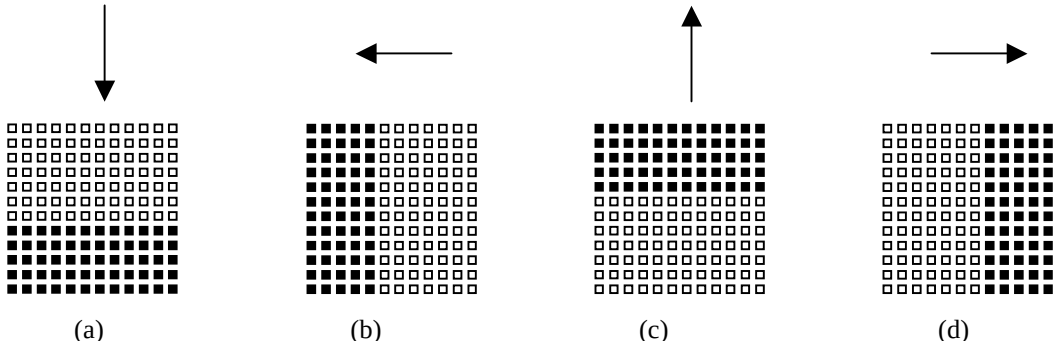


Figure 3.22: The pixels from each transceiver (less than the entire half) to be combined to make the processor sensitive to expanding patterns with a larger region of FOE position are shown in black.

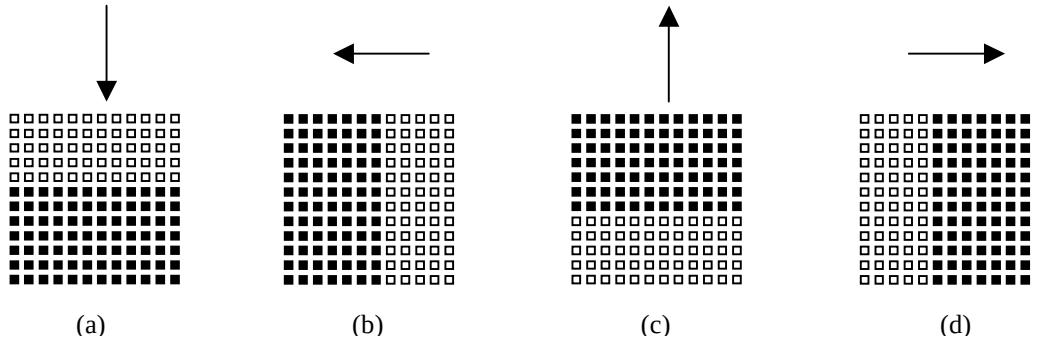


Figure 3.23: The pixels from each transceiver (more than the entire half) to be combined to make the processor sensitive to expanding patterns with a larger region of FOE position are shown in black.

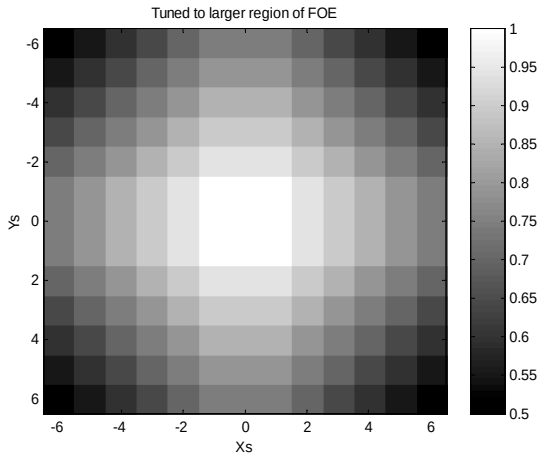


Figure 3.24: Output from expansion-tuned unit tuned to expanding patterns with larger region of FOE (corresponding to the mapping in Figure 3.22). (Xs, Ys) represents the coordinates of the FOE in pixels. Theoretically predicted value is represented by brightness.

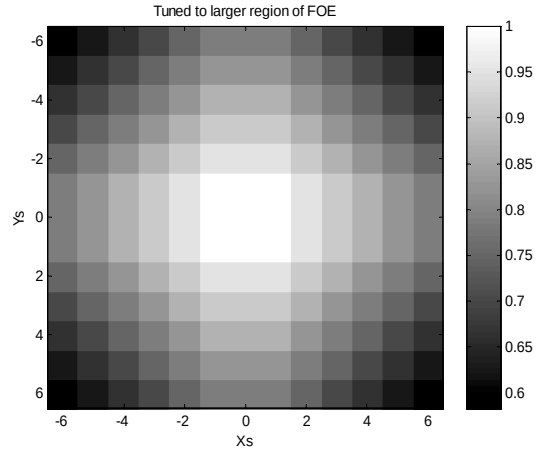


Figure 3.25: Output from expansion-tuned unit tuned to expanding patterns with larger region of FOE (corresponding to the mapping in Figure 3.23). (Xs, Ys) represents the coordinates of the FOE in pixels. Theoretically predicted value is represented by brightness.

4 Experimental results

4.1 Introduction

This chapter details all the experiments performed on the developed multichip motion processor. A metric ("performance") is introduced to help compare different results. Experimental results are compared with the theoretical predictions where appropriate. The cause of any difference between the experimental result and the theoretical prediction is discussed. The circuit conditions have been kept the same whenever the results of different experiments have been compared. The experimental setup is presented before all the experiments are described.

4.2 Experimental Setup

Figure 4.1 shows the experimental setup. The multichip motion processor has been built on a wire-wrap board. While taking data, the board has been kept upright using a vise. An LCD monitor is placed in front of the board for low flicker stimulation. The relative position of the board and the LCD monitor has been calibrated, so that the photosensitive sender chip can see the full view of the stimulus on the LCD monitor. The scanning circuits of the chips are connected to the computer through an interface card. A photograph of this setup is shown in Figure 4.2.

Different kinds of stimuli have been presented on the LCD monitor for different experiments. Expanding and contracting patterns have been generated using a circle, approximated by a polygon with twenty sides for speed of graphics. A photograph of the expanding/contracting circle is shown in Figure 4.3. CW and CCW rotating wagon wheel patterns have four spokes as shown in Figure 4.4. A square wave grating moving in a fixed direction is the presented moving bar stimulus, the photograph of which is shown in

Figure 4.5. The direction of bar movement is specified in the stimulus generating program.

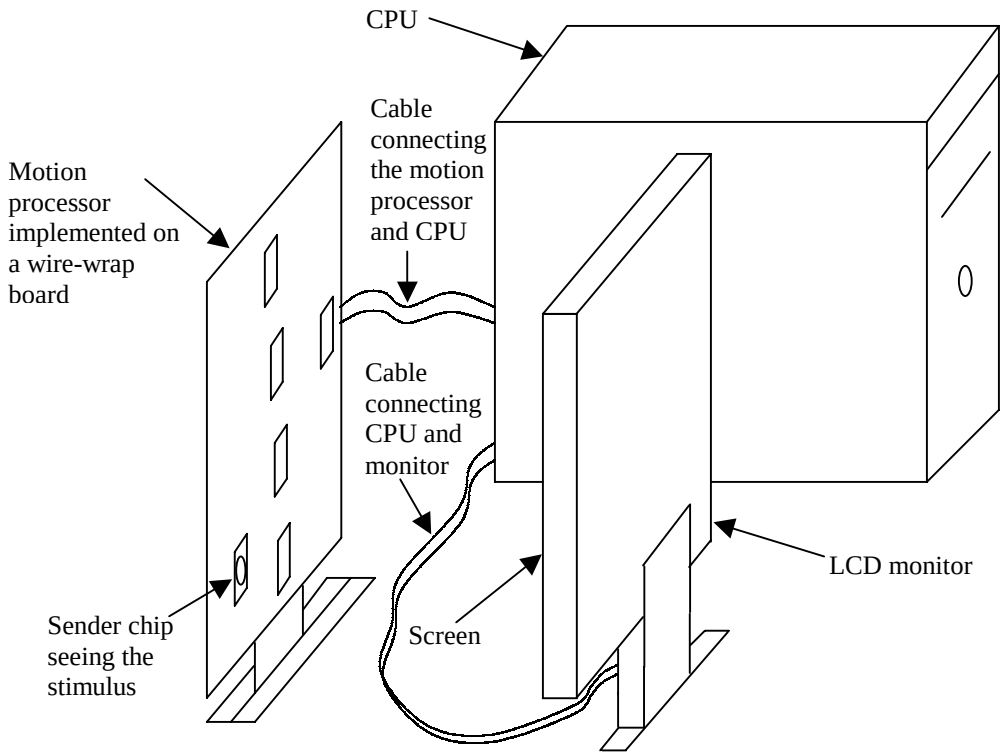


Figure 4.1: Experimental setup.

Figure 4.2: Photograph of the experimental setup.

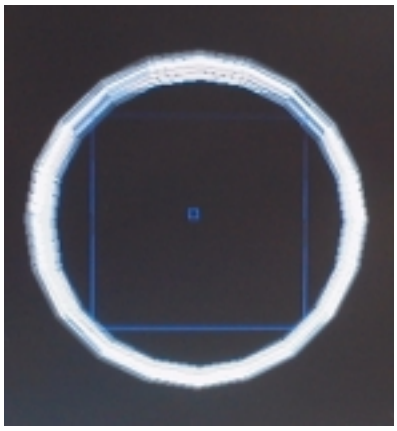


Figure 4.3: Photograph of the expanding/contracting stimulus.

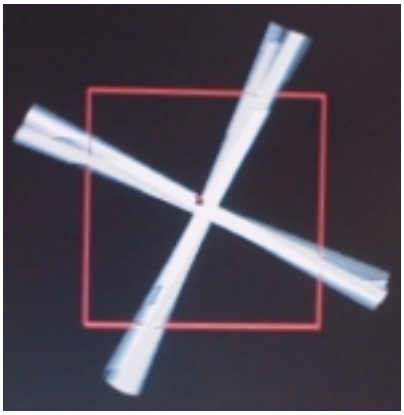


Figure 4.4: Photograph of the CCW/CW rotating stimulus.

Figure 4.5: Photograph of the moving bar stimulus.

4.3 Power Consumption

The developed multichip motion processor consumes a power of 1.76 Watts on average. The microcontroller and the oscillator to generate its clock consume 208 mW on average. The average power consumption by the neuromorphic VLSI chips is about 5 mW each [16]. Some power is consumed by the supporting digital chips, i.e. inverters, NOR gates, NAND gates, SR flip-flops and multiplexers. The rest of the power is consumed by the bias circuits, i.e. voltage regulators, biasing potentiometers and buffers.

4.4 Performance

For most of the experiments of this chapter, output has been recorded for 33X33 screen positions of FOE/FOC/AOR of expanding/contracting/rotating stimuli. In these experiments, performance is calculated as a ratio of the average output over the central 17X17 region of FOE/FOC/AOR to the average output over the entire 33X33 region of FOE/FOC/AOR.

Because there are 11 inter-pixel regions in each row or column of pixels in the motion chip, the field of view of the chip has been considered a 13X13 chip pixel region (including the outside regions of the chip) in calculating theoretical values. Theoretical performance is calculated as a ratio of the average theoretical output value for the central region of FOE/FOC/AOR over the chip to the average theoretical output value for the entire 13X13 region of FOE/FOC/AOR over the chip. The performance varies depending on the portion of the entire region considered as the central region, i.e. 1X1, 3X3, 5X5, 7X7, 9X9 and 11X11. Figure 4.6 shows the performance of an expansion-sensitive unit with FOE at the center of the visual field in response to the expanding patterns with varying FOE positions. It clearly shows that the performance depends on the area considered as the central region. Clearly, the narrower the central region is defined, the higher the performance is. The narrowest central region gives the theoretical value of the performance of about 1.37, which will be considered the standard performance when practical performances are discussed.

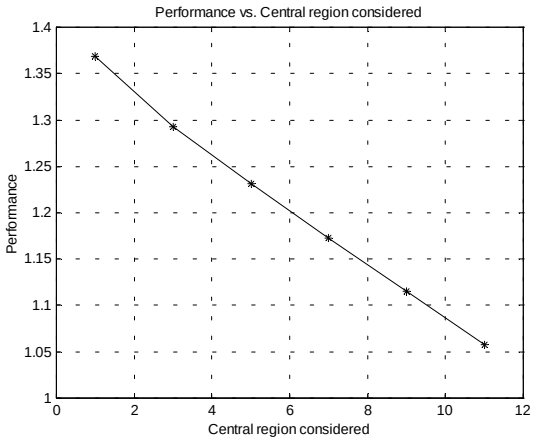


Figure 4.6: Theoretical performance as a function of the considered central region.

4.5 Experiment 1: Simultaneous tuning to expansion, contraction, CCW and CW rotation and four directions of translational motion

4.5.1 Setup

To perform this experiment, the PIC has been programmed to map the pixels from each transceiver to eight different receiver pixels to make the circuit sensitive to expanding patterns with the FOE at the center of the visual field of the chip, contracting patterns with the FOC at the center of the visual field of the chip, CCW and CW rotating patterns with the AOR at the center of the visual field of the chip, and patterns of translation motion in four different directions: 0°, 90°, 180° and 270°, all simultaneously. Therefore, the motion processor has been made to detect any of the eight different kinds of motion whenever they appear in front of the sender chip. The PIC combines different parts of different transceivers as described in Chapter 3 to make the circuit sensitive to expansion, contraction, and CCW and CW rotation. The information from each entire transceiver has been combined to make the circuit sensitive to four directions of translation motion. These eight different mappings are performed by the PIC and the information is passed to eight different pixels of the integrating receiver, one pixel per

tuning. The PIC makes exactly four requests to the receiver for every request it receives from any transceiver. Two requests are required to implement four complex units simultaneously as mentioned in section 3.4. One more request is necessary to implement the units tuned to four directions of translational motion. But this difference in the number of requests (spike frequencies) to the receiver pixels corresponding to four complex units and four translational units require different receiver (V_{qua}) bias settings for complex and translational units. This problem has been resolved by sending an additional request to the receiver pixels corresponding to translational units. This makes the spike frequencies same for complex units and translational units and thus makes the system work for a single receiver bias setting.

A few bursts of these four requests going out of the PIC, viewed on the logic analyzer, are shown in Figure 4.7. The time span between sending the last request of a burst to the first request of the next burst is significant due to the execution of large number of PIC instructions. The same amount of time is required between two successive requests if the PIC would be sending one request out for each input request. The time spans between the requests of a burst are much smaller because they are generated one after another without executing other PIC instructions in between. Therefore, sending four requests out for each input request does not make the system four times slower. The slowdown is much less. For simplicity, we will term this as 4 to 1 slowdown hereafter.

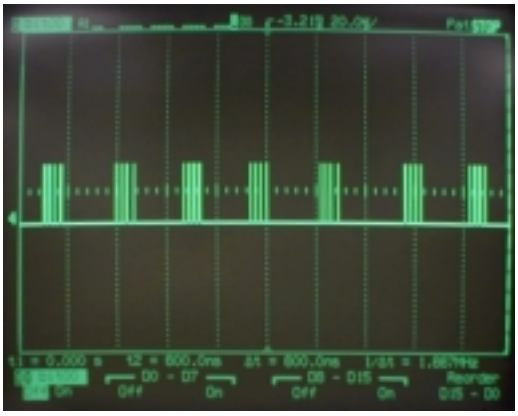


Figure 4.7: Bursts of requests coming out of the PIC.

Five different kinds of stimuli have been presented in this experiment. Expanding, contracting, CCW rotating and CW rotating stimuli have been presented and output has been recorded from each of the eight different tuned units for 33X33 screen positions of FOE/FOC/AOR over the visual field of the chip. Moving bar stimuli have been presented and output has been recorded from each of the eight different tuned units for different angles of the direction of moving bars varied at a step of 15°. The scanner circuit of the receiver chip is clocked as the outputs are recorded from eight pixels of the chip.

4.5.2 Result

Figure 4.8 shows the gray scale plots of outputs from eight different tuned units in response to the contracting stimulus. Similar plots are shown in Figure 4.9, Figure 4.10 and Figure 4.11 for expanding, CW rotating and CCW rotating stimuli respectively. Polar plots of outputs from all eight units in response to the moving bar stimulus are shown in Figure 4.12.

The gray scale plots show that the contraction-tuned unit is sensitive only to the contracting stimulus with the FOC at the center of the field of view, the expansion-tuned unit is sensitive only to the expanding stimulus with the FOE at the center of the field of view, the CW rotation-tuned unit is sensitive only to the CW rotating stimulus with the AOR at the center of the field of view and the CCW rotation-tuned unit is sensitive only to the CCW rotating stimulus with the AOR at the center of the field of view. The translation-sensitive units also respond to these four stimuli for some positions of FOC/FOE/AOR, because for these positions of FOC/FOE/AOR the stimuli present translation motions to the chip. For example, for an expanding stimulus when the FOE is outside the field of view of the chip on the left, the right-sensitive translation unit responds because the stimulus looks like full field rightward motion. The polar plots show that the translation-tuned units also detect the right direction of motion. Obviously, all the output plots differ from their theoretical counterparts shown in Figure 4.13, Figure 4.14, Figure 4.15, Figure 4.16 and Figure 4.17 respectively for valid reasons that are discussed below.

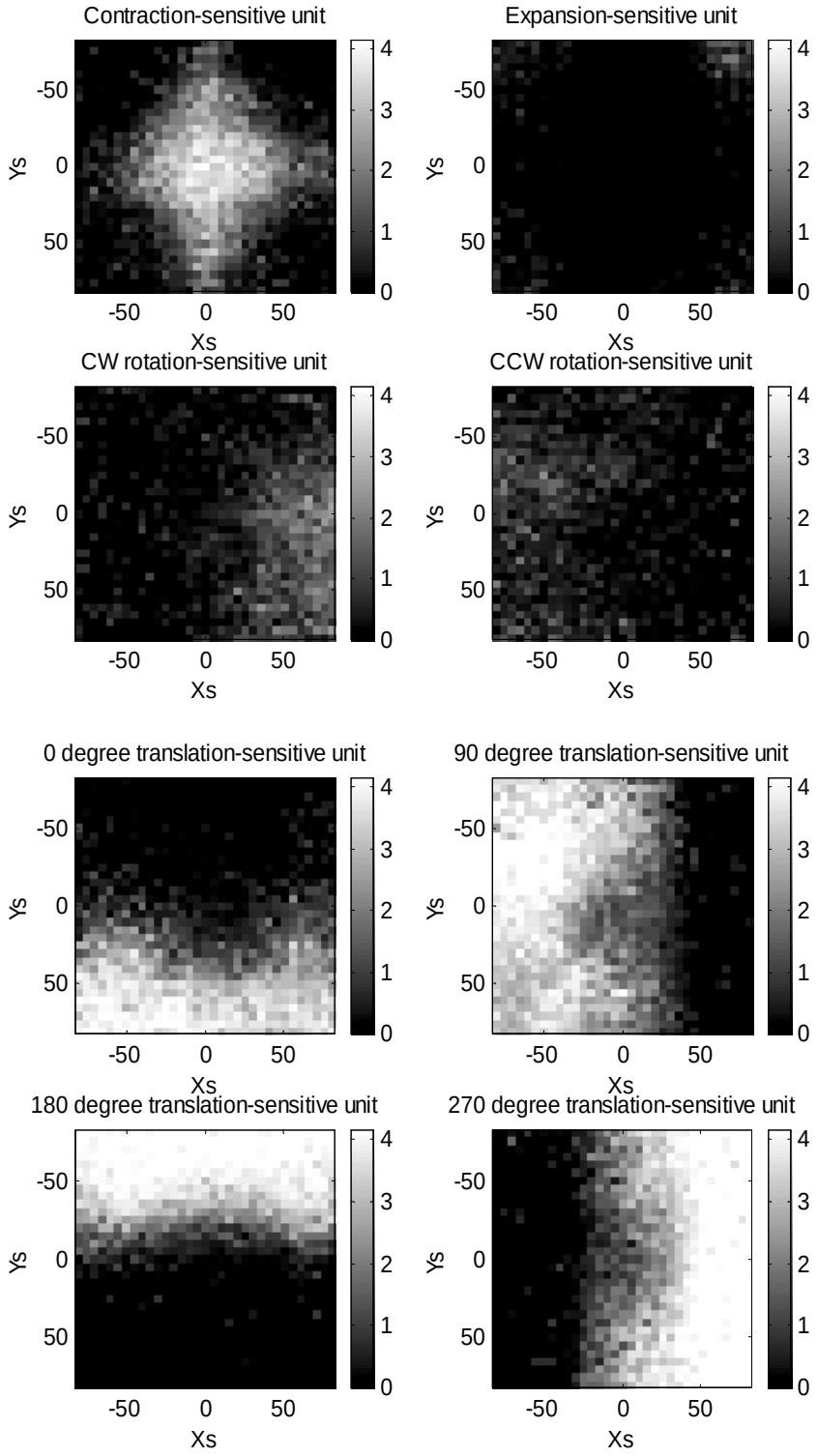


Figure 4.8: Outputs from eight different tuned units in response to the contracting stimulus. (Xs, Ys) represents the coordinates of the FOC in screen pixels. Output is represented by brightness.

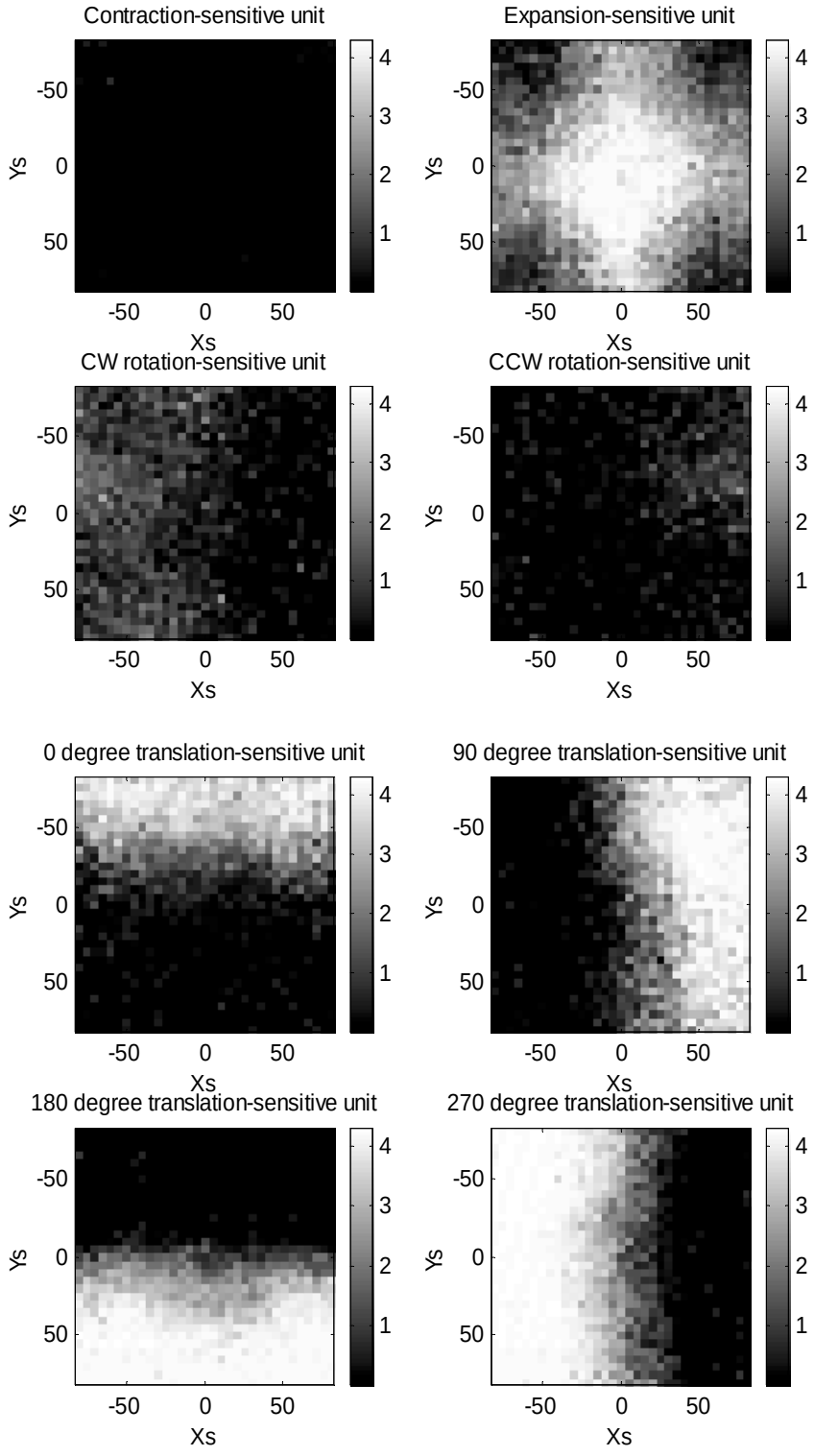


Figure 4.9: Outputs from eight different tuned units in response to the expanding stimulus. (Xs, Ys) represents the coordinates of the FOE in screen pixels. Output is represented by brightness.

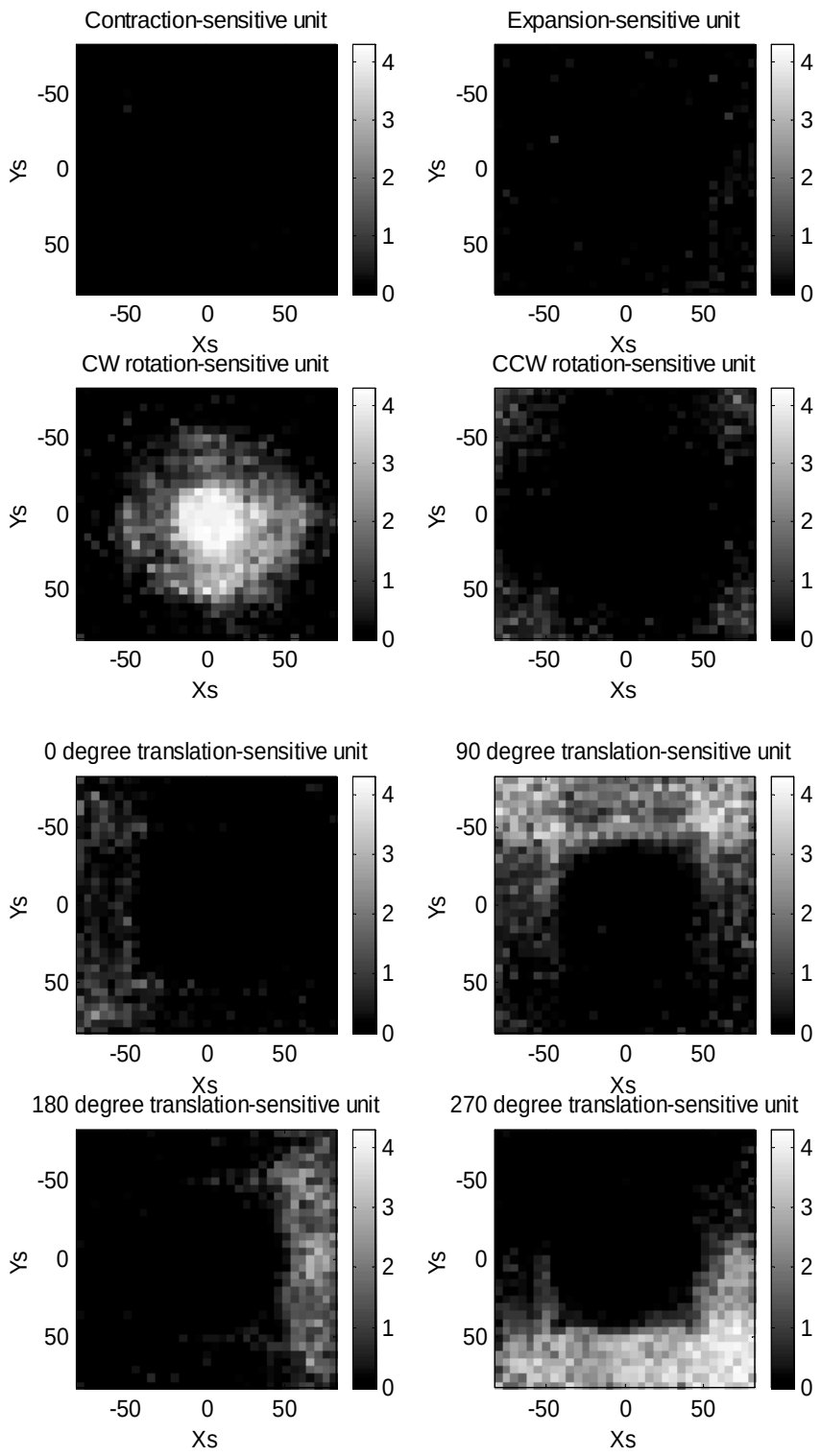


Figure 4.10: Outputs from eight different tuned units in response to the CW rotating stimulus. (Xs, Ys) represents the coordinates of the AOR in screen pixels. Output is represented by brightness.

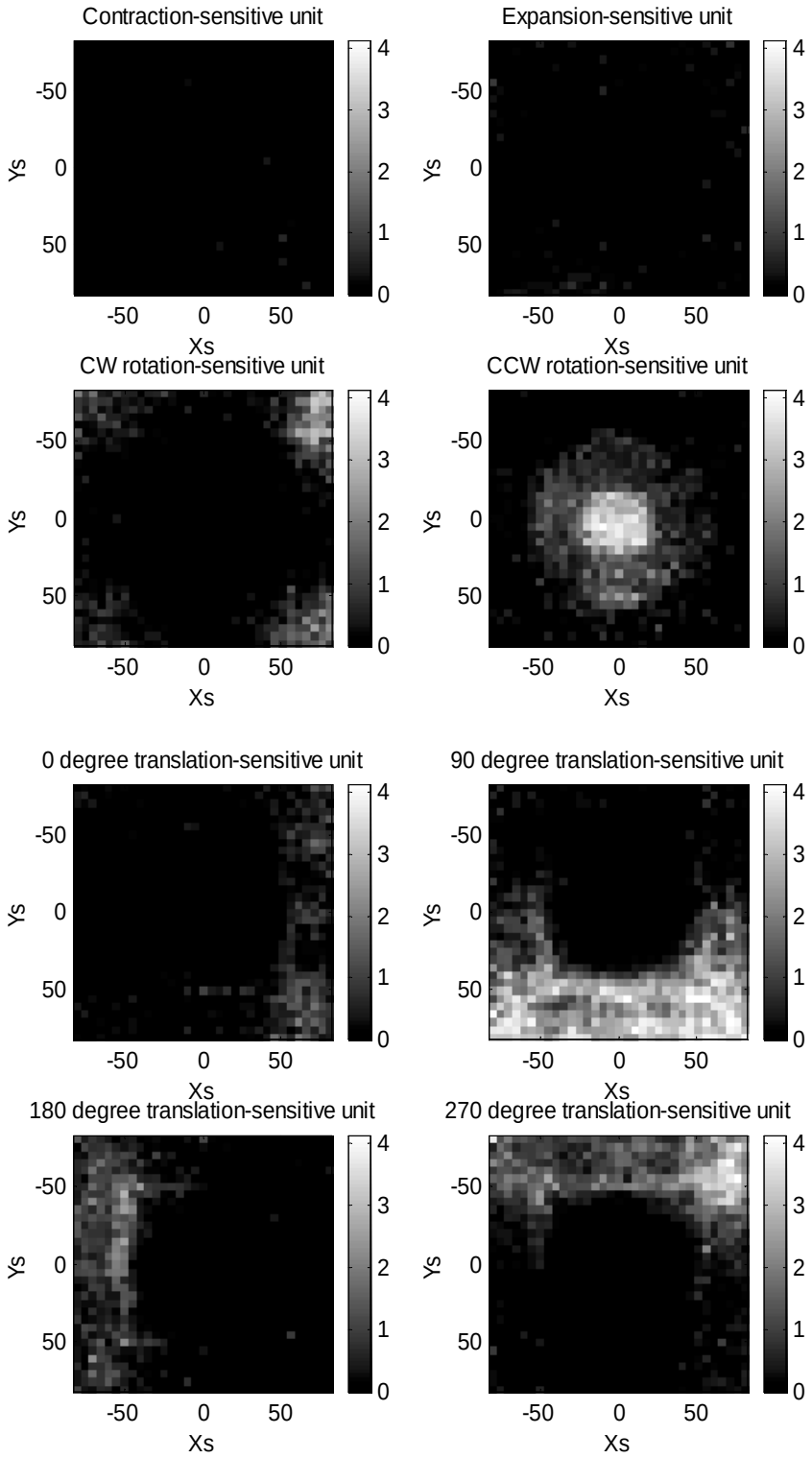


Figure 4.11: Outputs from eight different tuned units in response to the CCW rotating stimulus. (Xs, Ys) represents the coordinates of the AOR in screen pixels. Output is represented by brightness.

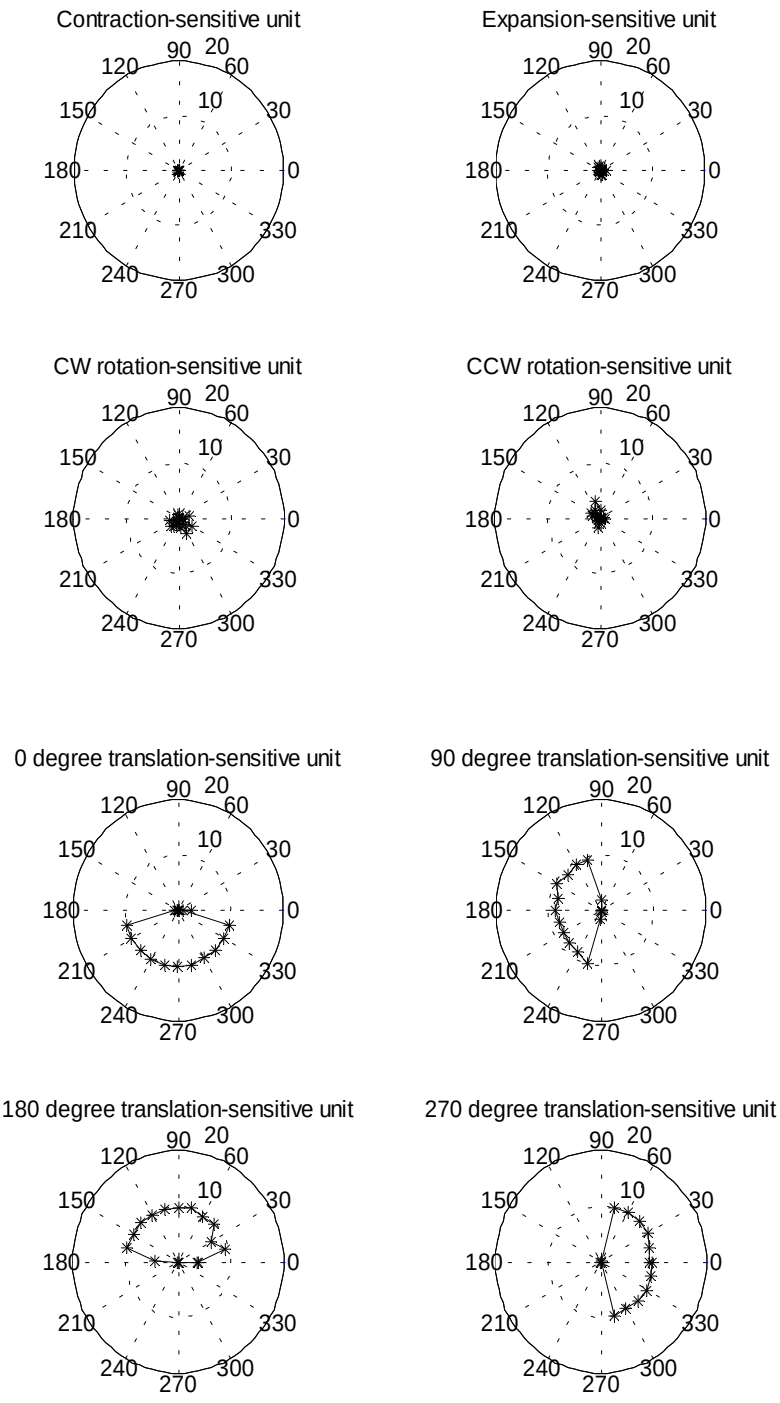


Figure 4.12: Outputs from eight different tuned units in response to the translational stimulus. The direction of bar movement of the moving bar stimulus has been changed from 0° to 360° at a step of 15°.

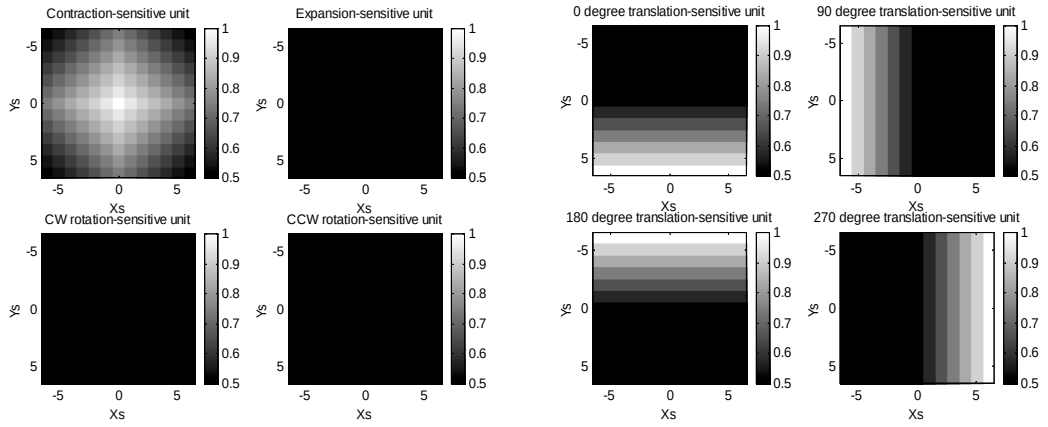


Figure 4.13: Theoretical prediction of outputs from eight different tuned units in response to the contracting stimulus. (Xs, Ys) represents the coordinates of the FOC in pixels. Theoretically predicted value is represented by brightness.

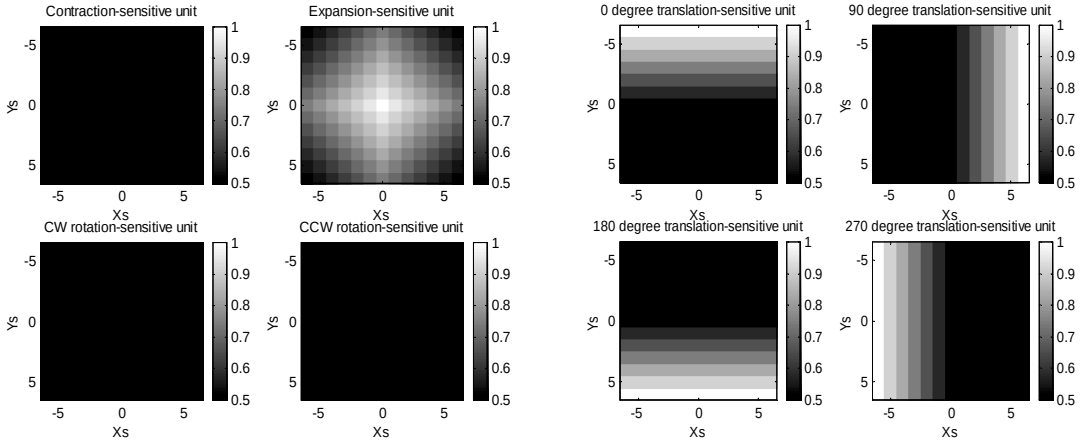


Figure 4.14: Theoretical prediction of outputs from eight different tuned units in response to the expanding stimulus. (Xs, Ys) represents the coordinates of the FOE in pixels. Theoretically predicted value is represented by brightness.

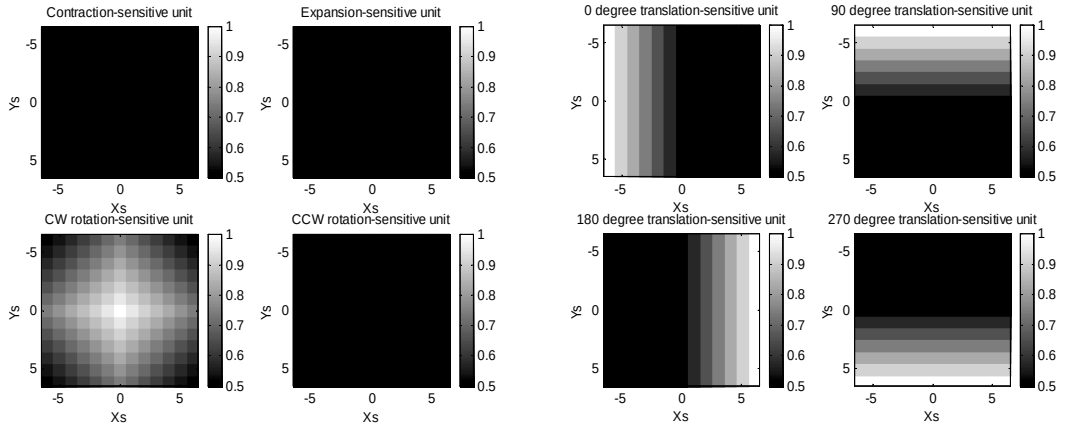


Figure 4.15: Theoretical prediction of outputs from eight different tuned units in response to the CW rotating stimulus. (Xs, Ys) represents the coordinates of the AOR in pixels. Theoretically predicted value is represented by brightness.

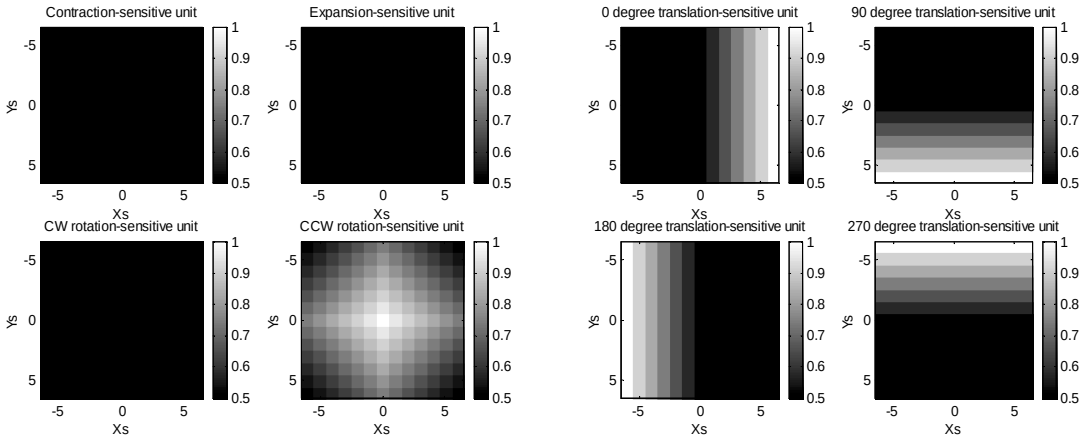


Figure 4.16: Theoretical prediction of outputs from eight different tuned units in response to the CCW rotating stimulus. (Xs, Ys) represents the coordinates of the AOR in pixels. Theoretically predicted value is represented by brightness.

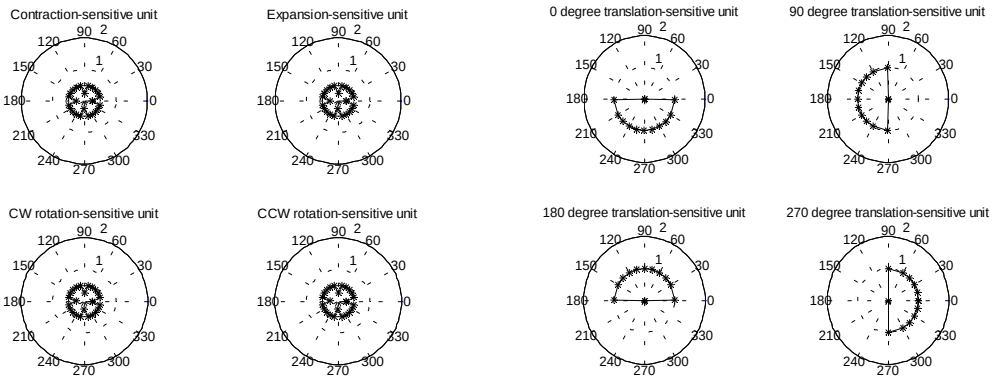


Figure 4.17: Theoretical prediction of outputs from eight different tuned units in response to the translational stimulus.

In calculating theoretical values, it is assumed that any transceiver pixel gives rise to a response 1 if there is any component of the stimulus in the preferred direction of that pixel. It contributes a value 0 otherwise. The contributions from all the pixels have been added to get the final value. The theoretical plots have not taken the effect of the thresholding circuit (described in Chapter 2) into account, which is the main reason behind the difference between the practical outputs and the theoretical predictions. The outputs of complex units in Figure 4.8, Figure 4.9, Figure 4.10 and Figure 4.11 do not show as sharp a peak as their counterpart in the theoretical plots because of this thresholding effects of the receiver pixels. Another reason for not having a sharp peak lies in the stimulus shape. For an expanding stimulus, the expanding circle starts with a non-zero bar-width defined in the stimulus generating function. So, some pixels near the FOE do not see a passing edge. Moreover, each expanding circle does not start exactly the same. Similarly, for a contracting stimulus, the contracting circle has a non-zero bar-width when it disappears in the center. So, some pixels near the FOC do not see a passing edge. Moreover each contracting circle does not disappear exactly the same. The intersection of the rotating bars in CCW and CW rotational stimuli is not a point, but a square region with non-zero length of sides. So, some pixels near the AOR in both

rotational stimuli do not see a passing edge. The reason for the difference in the output of contraction-sensitive unit in Figure 4.8 and the output of expansion-sensitive unit in Figure 4.9 is that although the same circular stimulus has been used to generate expanding and contracting patterns, one is not necessarily the reverse playback of the other. This difference in stimuli causes the difference between these two plots. Any difference between the output of CW rotation-sensitive unit in Figure 4.10 and the output of CCW rotation-sensitive unit in Figure 4.11 has also been arisen from a difference in the stimulus. Although theoretically, the range of translational motion having a component in preferred direction of any channel extends 90° both ways of the preferred direction, practically it extends less than 90°. This is because the transceiver cannot detect motion above a certain velocity and at some angles near 90° from the preferred direction, an edge crosses two neighboring pixels so fast that the speed limit for motion detection is exceeded. The polar plots of the translational-sensitive units of Figure 4.12 show this fact. This contributes to making the output plots differ from their theoretical counterparts.

4.6 Experiment 2: Expansion sensitivity while changing receiver threshold

This experiment shows the dependence on V_{qua} (receiver threshold) of the sensitivity of the system to expanding patterns. The effects are similar for contracting, CCW rotating and CW rotating patterns.

4.6.1 Setup

The motion processor has been made sensitive to expanding patterns with the FOE at the center of the visual field only. An expanding stimulus has been presented on the LCD monitor and the FOE is varied over 33X33 screen positions over the field of view of the chip. Output has been recorded for every position of FOE. The V_{qua} bias has been changed for each run.

4.6.2 Result

The gray scale plots of outputs are shown in Figure 4.18 for different V_{qua} biases. From these plots, we can see that the output varies with changing V_{qua} bias as expected from the analysis in chapter 2. Because the charging current of the capacitor changes with V_{qua} bias, the output voltage V_{mem} changes accordingly (Figure 2.10). Figure 4.19 shows a plot of performance vs. V_{qua} bias. Performance varies for different V_{qua}. The plot shows that V_{qua}=4.76 gives us the best performance. This performance is even higher than the standard performance shown in section 4.3 because of the thresholding effect of the receiver pixel discussed in chapter 2.

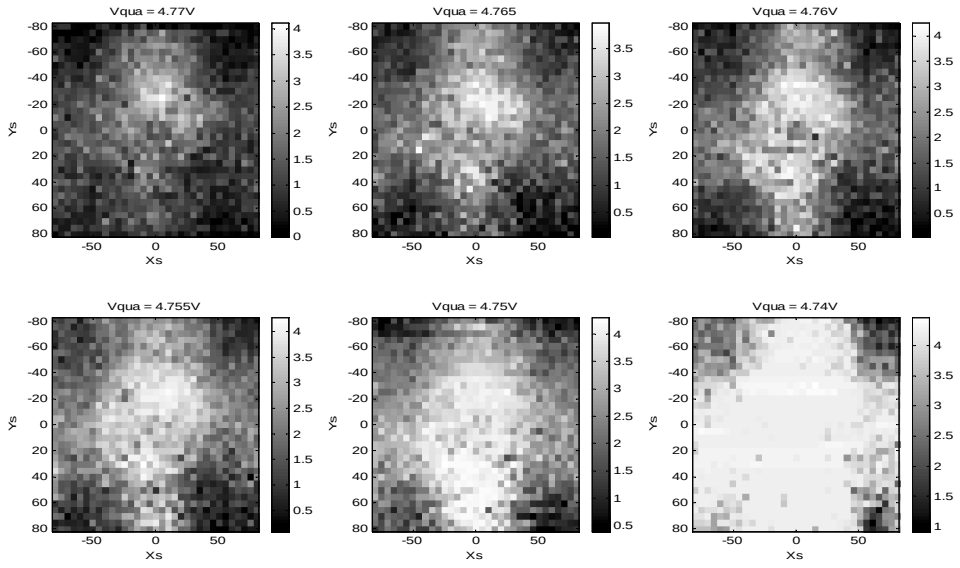


Figure 4.18: Output from expansion-tuned unit for different V_{qua} bias settings. (Xs, Ys) represents the coordinates of the FOE in screen pixels. Output is represented by brightness.

The performance calculated from the output of the expansion-sensitive unit in response to the expanding stimulus in experiment 1 is 1.5. From the performance vs. V_{qua} bias plot shown in Figure 4.19, we can see that performance varies with changing V_{qua} and that this bias can be set such that the same performance as experiment 1 (1.5) can be found from the expansion-tuned unit if it is synthesized alone. Therefore, we are not losing performance by synthesizing all eight units at once in experiment 1.

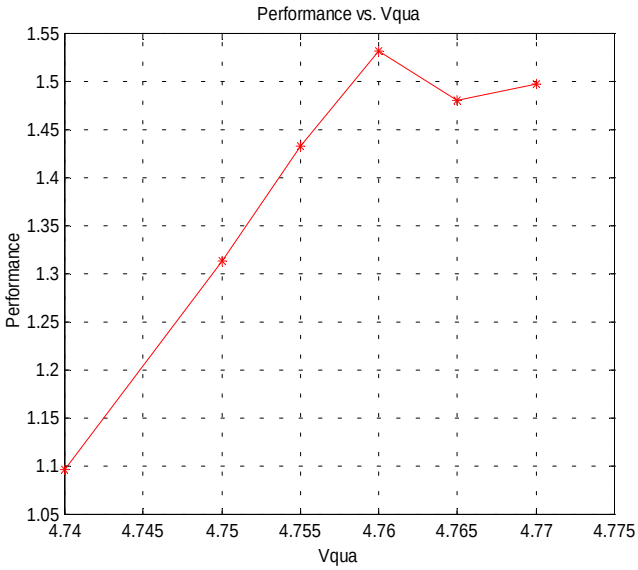


Figure 4.19: Performance vs. V_{qua} bias plot.

4.7 Experiment 3: Varying receptive field size

This experiment investigates if it is possible to get good performance without using the full receptive field. If good performance can be achieved using only a part of the receptive field, then it will be possible to do multiple tasks simultaneously dedicating different partial receptive fields for different tasks. In that way, we can do more tasks in a smaller amount of time. It is obvious that this more efficient processing can come only at the cost of performance.

4.7.1 Setup

In this experiment, the motion processor has been made sensitive to expanding patterns with the FOE at the center of the visual field of the chip. The PIC has been programmed with varying receptive fields (12 X 12, 10 X 10, 8 X 8, 6 X 6, 4 X 4 and 2 X 2 pixels) for different runs. An expanding stimulus has been presented on the LCD monitor and the FOE is varied over 33X33 screen positions over the field of view of the chip. Output has been recorded for every position of FOE.

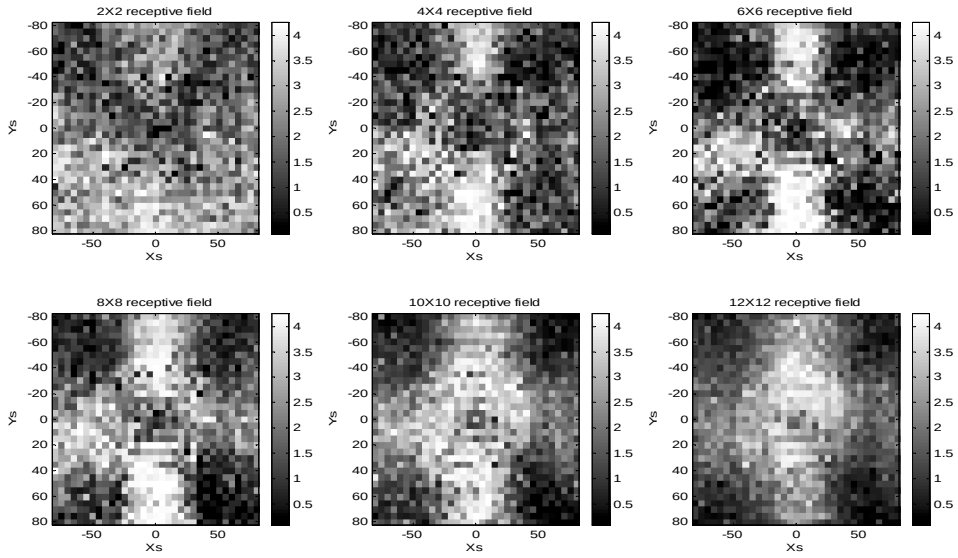


Figure 4.20: Outputs from expansion-tuned units synthesized using different receptive fields. (Xs, Ys) represents the coordinates of the FOE in screen pixels. Output is represented by brightness.

4.7.2 Result

Figure 4.20 shows the output plots for different receptive fields. Corresponding theoretical predictions are shown in Figure 4.21 without considering the threshold effect of the receiver. The V_{qua} bias has been changed to account for the change in spike frequency for different receptive fields. In each case, the V_{qua} bias has been set to a value so that the average outputs over the 33X33 screen positions are same for all receptive fields. This gives rise to four distinct screen pixel regions that are blurred for some reduced receptive fields to compensate for worse response, i.e. lower output in the central region for reduced receptive field. The reason behind this lower output in the central region is that the expanding circle starts with a non-zero width and the percentage of the total pixels (used in synthesizing the expansion-sensitive units) not seeing any motion increases for reduced receptive field when the FOE is at the center. Performance is plotted vs. receptive field in Figure 4.22, which shows that a receptive field of as little as 10 X10 pixels can be used without lowering the performance below the standard. If we reduce the receptive field more the performance drops below the standard. As the

transceiver core in use has only 12X12 pixels, there is not much benefit in reducing this receptive field (from 12X12 to 10X10) in the system in hand. But for a transceiver core of 128X128, the system will be able to perform much more tasks by using this reduced receptive field instead of the full receptive field.

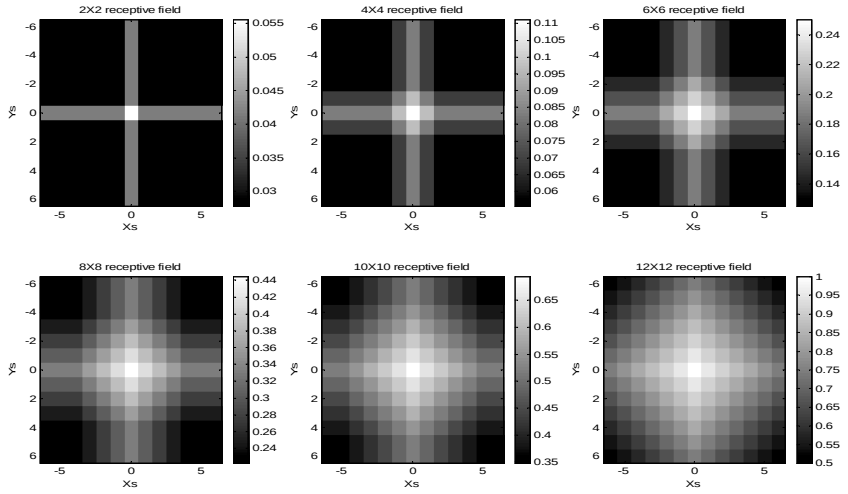


Figure 4.21: Theoretical predictions from expansion-tuned units synthesized using different receptive fields. (Xs, Ys) represents the coordinates of the FOE in pixels. Theoretically predicted value is represented by brightness.

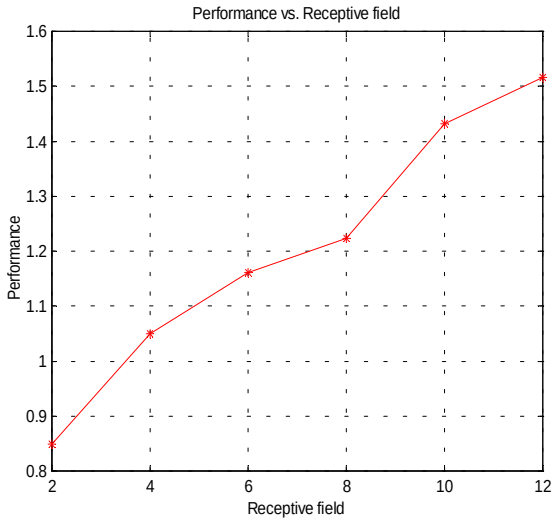


Figure 4.22: Performance vs. Receptive field plot. The numbers in receptive field axis represent a square region. For example the number 12 represents a receptive field of 12X12 pixels.

4.8 Experiment 4: Effects of PIC speed and sequential mapping

This experiment shows the effects of PIC speed and the sequential mapping by the routing processor to several receiver pixels on output values.

4.8.1 Setup

The transceiver pixels have been mapped to nine different receiver pixels to make all nine units of the motion processor tuned to expanding patterns with the FOE at the center of the visual field of the chip. An expanding stimulus has been presented on the LCD monitor and the FOE is varied over 33X33 screen positions over the field of view of the chip. Output has been recorded from each of the nine receiver pixels for every position of FOE.

4.8.2 Result

The output plots of nine units are shown in Figure 4.23. Although all these plots should look the same theoretically, a difference in output has arisen from the low speed of the PIC. The PIC sends nine requests sequentially to the receiver for each request coming from a transceiver. So, there is a 9 to 1 slowdown in this implementation. In this implementation, receiver pixel 1 receives the request first and pixel 9 receives it last, mapping in a sequence in ascending pixel numbers. The time span from one request to the other (sent by this slow PIC) is so large that the integrated output of the earlier mapped pixels gets significantly more time to decay than that of the later mapped pixels before the output values from all nine pixels are scanned out. So, a gradual increase in output values is seen from the earlier mapped pixels to the later mapped pixels. Pixel 9 has the most recent output value and pixel 1 has the oldest. Figure 4.23 shows that only four pixels (pixel 4 through pixel 7) are giving useful outputs in this implementation. So, we can say that this system gives useful output up to 4 to 1 slowdown by the PIC. This sets the limit of this system. Experiment 1 was successful, because we had 4 to 1 slowdown there though eight units were synthesized simultaneously. Changing the V_{qua}

bias changes the outputs in all nine receiver pixels. Therefore, the V_{qua} bias can be set such that any four pixels in a row among these nine pixels give useful outputs.

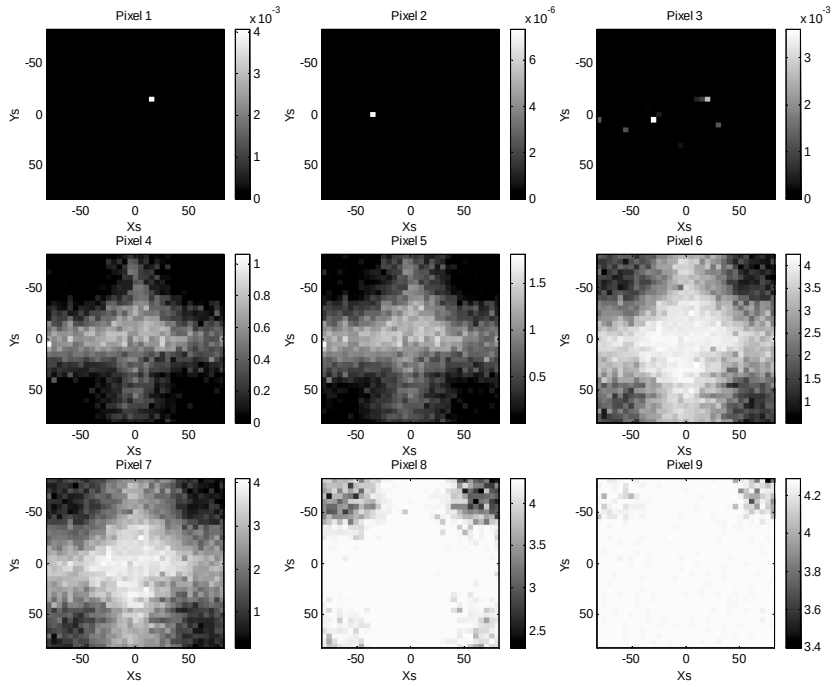


Figure 4.23: Outputs from nine units tuned to expanding patterns with the FOE position at the center. (Xs, Ys) represents the coordinates of the FOE in screen pixels. Output is represented by brightness.

4.9 Experiment 5: Expansion-sensitive units for 3X3 FOE positions

This experiment shows how this motion processor performs in simultaneous detection of expanding patterns with different positions of FOE over the field of view of the chip.

4.9.1 Setup

The address lines from transceivers have been mapped to nine receiver pixels using the PIC in such a way that the motion processor becomes sensitive to expanding

patterns with nine different positions of FOE. An expanding stimulus has been presented on the LCD monitor and the FOE has been varied over 33X33 screen positions over the field of view of the chip. Output has been recorded from each of the nine receiver pixels for every position of FOE.

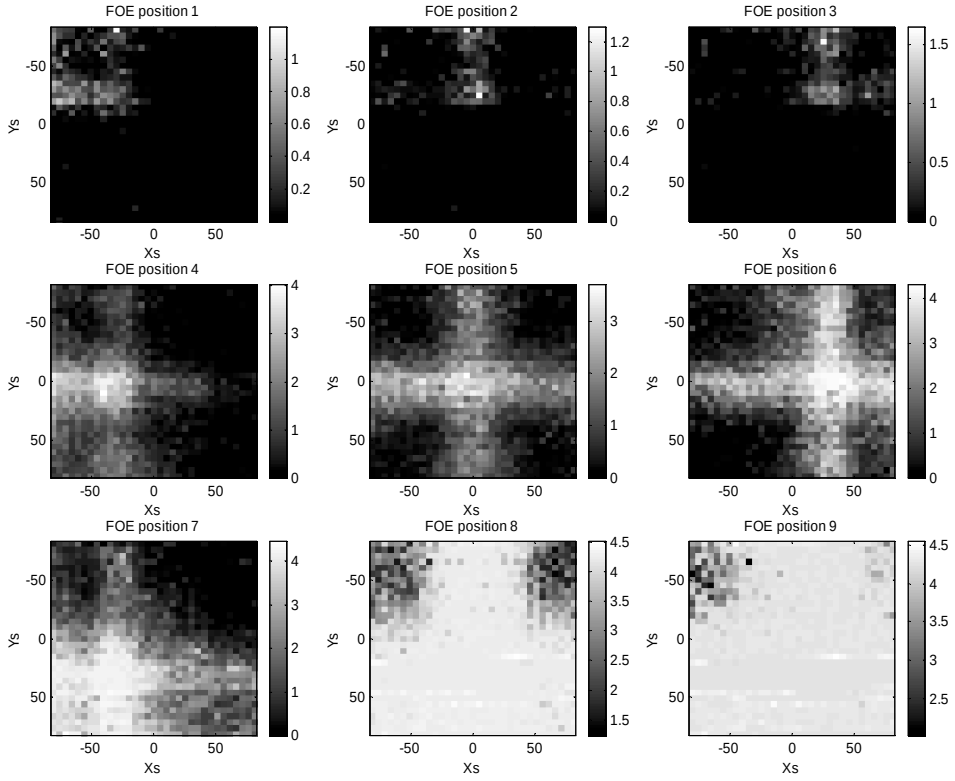


Figure 4.24: Outputs from nine units tuned to nine different FOE positions. (Xs, Ys) represents the coordinates of the FOE in screen pixels. Output is represented by brightness.

4.9.2 Result

The outputs from nine receiver pixels are shown in Figure 4.24. The corresponding theoretical plots are shown in Figure 4.25 without considering the threshold effect of the receiver. Figure 4.24 shows a large difference in the range of output values from nine different units because of the low speed of the PIC and the effects of sequential mapping by the PIC described in experiment 4. From the plots, it is

clearly seen that the motion processor works very well to detect expanding patterns with FOE positions four through seven (four FOE positions). Although the other five FOE positions could not be detected so strongly, these plots of Figure 4.24 shows some trends of the sensitivity of those units to the respective FOE positions. This experiment supports the fact that 4 to 1 slowdown by the PIC is the limit of this system to have useful outputs. Changing the V_{qua} bias changes the outputs from nine receiver pixels corresponding to nine FOE positions. Therefore, the V_{qua} bias can be set such that any four pixels in a row among these nine pixels corresponding to respective FOE positions give useful outputs.

Although this experiment synthesizes units only to the expanding patterns, synthesizing units for contracting, CCW rotating and CW rotating patterns with different positions of FOC/AOR can be similarly done by combining appropriate region from each transceiver. So, this motion processor can be used in determining singular points over the field of view of the chip.

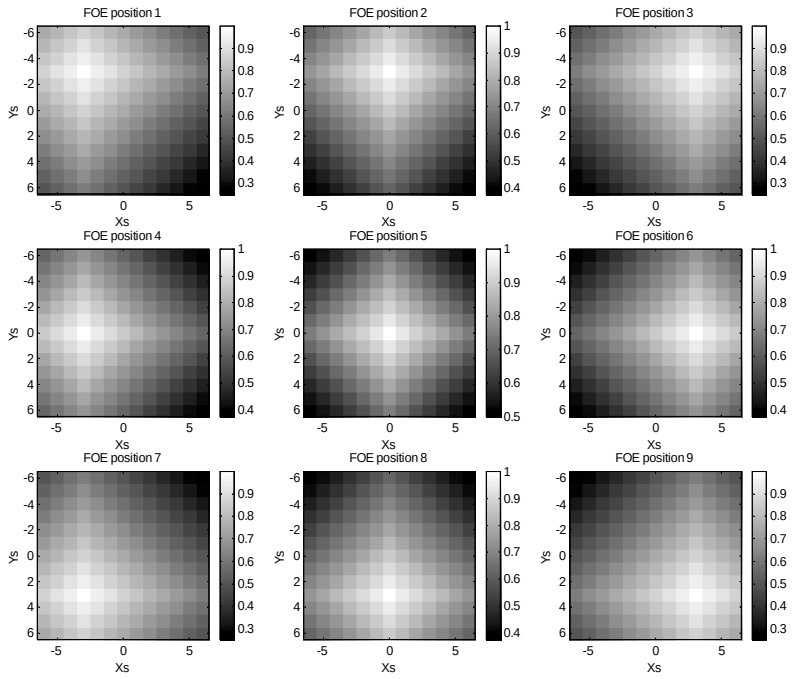


Figure 4.25: Theoretical prediction of outputs from nine units tuned to nine different FOE positions. (Xs, Ys) represents the coordinates of the FOE in pixels. Theoretically predicted value is represented by brightness.

4.10 Experiment 6: Large and small expanding field

This experiment shows how this motion processor can be made sensitive to large and small expanding fields at the same time.

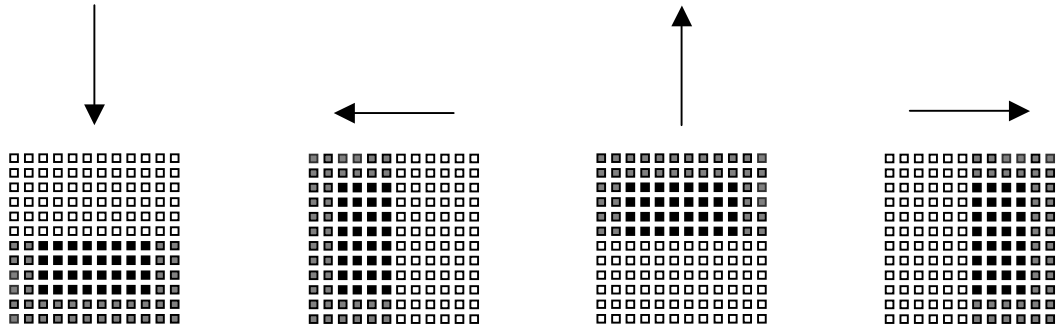


Figure 4.26: Mapping of transceiver pixels to the receiver pixels. Black and gray colored pixels are mapped to two different receiver pixels to make one tuned to small expansion and the other tuned to large expansion.

4.10.1 Setup

The address lines from the transceivers have been mapped to two different pixels of the receiver. Black colored transceiver pixels of Figure 4.26 have been mapped to one receiver pixel and the gray colored pixels have been mapped to the other. One of these two units, representing black transceiver pixels, is tuned to small expanding field and the other, representing gray transceiver pixels, is tuned to large expanding field. An expanding stimulus with the FOE at the center of the visual field has been presented on the LCD monitor and the maximum radius of the expanding circle, i.e. the magnitude of the expanding field is varied so that the largest circle covers the field of view of the chip. Outputs have been recorded from both of the receiver pixels for every magnitude of the expanding field.

4.10.2 Result

Figure 4.27 shows the plots of outputs from both receiver pixels vs. the maximum radius of the expanding field for each expanding field. From the plots, it is clearly seen that the unit tuned to small expanding field is sensitive to small expanding patterns and the unit sensitive to large expanding field is sensitive to large expanding patterns. The unit tuned to large expanding field does not respond for small expanding patterns. The unit tuned to small expanding field does not respond for very small expanding radius, because the expanding patterns start with non-zero bar-widths and so, this unit does not see any motion for very small values of radii of the expanding patterns. The larger value of output from the small expansion-tuned unit than that of the large expansion-tuned unit is due to the fact that four rows of pixels are mapped to one receiver pixel to synthesize the small expansion-tuned unit and only two rows of pixels are mapped to one receiver pixel to synthesize the large expansion-tuned unit. The output of the small expansion-tuned unit drops for radius greater than about sixty five screen pixels because the inter-stimulus interval increases significantly for these radii.

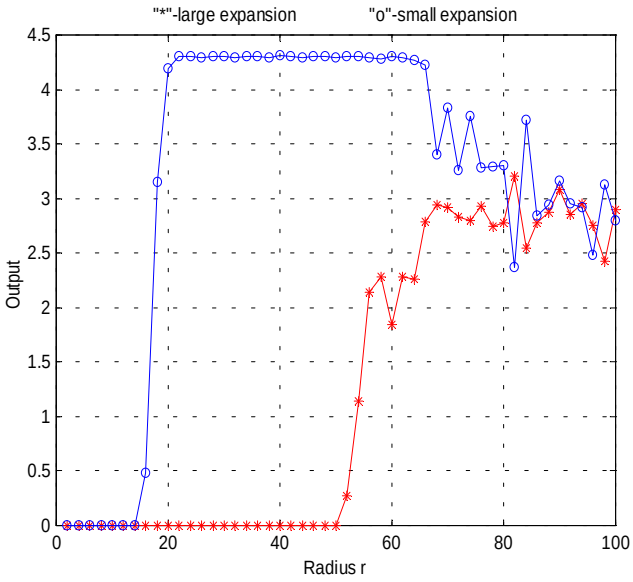


Figure 4.27: Output vs. Maximum expanding circle radius plot. Radius is measured in screen pixels.

4.11 Experiment 7: Effects of receiver pixel mismatch

This experiment demonstrates the effect of receiver pixel mismatch on the output. Signals coming from different pixels of different transceivers are directed to only one receiver pixel to have one particular tuning. Therefore, receiver pixel mismatch will result in different output values for different receiver pixels used.

4.11.1 Setup

The motion processor has been made sensitive only to the expanding patterns with FOE at the center of the visual field of the chip. Two receiver pixels have been used to have this tuning in two different runs. An expanding pattern is presented on the LCD monitor. The FOE of the patterns has been varied over 33X33 screen positions over the field of view of the chip. Output has been recorded for every position of FOE of the stimulus.

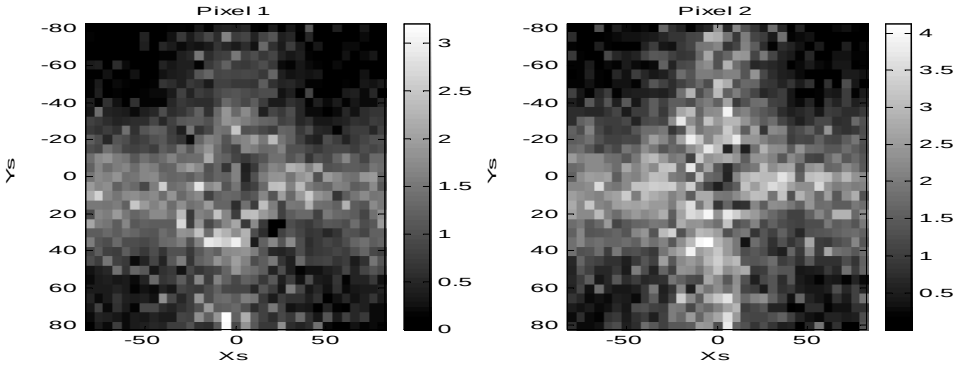


Figure 4.28: Outputs from expansion-sensitive units implemented on two different receiver pixels. (Xs, Ys) represents the coordinates of the FOE in screen pixels. Output is represented by brightness.

4.11.2 Result

Figure 4.28 shows the outputs from the expansion-sensitive units implemented on two different receiver pixels. The difference in outputs from two different receiver pixels is evident from these plots. The average outputs from pixel 1 and pixel 2 are 0.9 and 1.49

respectively. The performances are 1.46 and 1.45 respectively, indicating that these two pixels do not suffer much from mismatch for this particular bias setting. However, other bias settings might cause a larger difference in performance from these two pixels, as indicated from the difference in output levels.

4.12 Experiment 8: Variation of expanding stimulus speed

This experiment demonstrates the effect of variation in speed of expanding stimulus on performance.

4.12.1 Setup

The motion processor has been made sensitive only to expanding patterns with FOE at the center of the visual field of the chip. An expanding pattern is presented on the LCD monitor. The FOE of the patterns has been varied over 33X33 screen positions over the field of view of the chip. Output has been recorded for every position of FOE of the stimulus. The speed of the stimulus has been changed for each run.

4.12.2 Result

The output plots are shown in Figure 4.29. As the speed of stimulus is increased beyond a particular speed the motion processor is no longer sensitive to the expanding patterns with a centered FOE, because beyond that speed the simple sensors of this motion processor cannot detect motion. Figure 4.30 shows the performance vs. stimulus speed plot. It shows that the performance decreases below the standard value beyond a certain speed as expected.

4.13 Experiment 9: Variation of expanding stimulus width

This experiment demonstrates the effect of variation in width of expanding stimulus on performance.

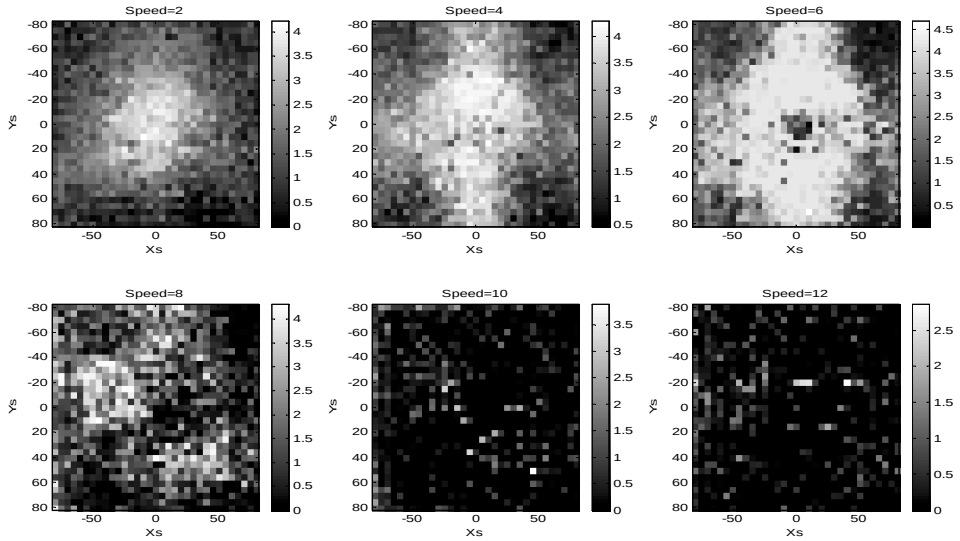


Figure 4.29: Outputs from expansion-tuned unit for different expanding stimulus speed. (Xs, Ys) represents the coordinates of the FOE in screen pixels. Output is represented by brightness.

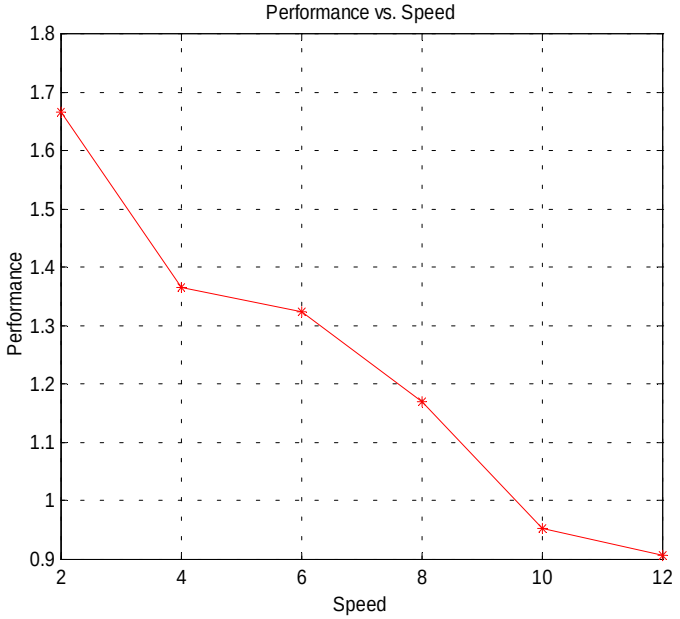


Figure 4.30: Performance vs. expanding stimulus speed plot.

4.13.1 Setup

The motion processor has been made sensitive only to expanding patterns with FOE at the center of the visual field of the chip. An expanding pattern is presented on the LCD monitor. The FOE of the patterns has been varied over 33X33 screen positions over the field of view of the chip. Output has been recorded for every position of FOE of the stimulus. The width of the stimulus has been changed for each run.

4.13.2 Result

The output plots are shown in Figure 4.31, which shows that the motion processor cannot detect expanding patterns of bar-width of 1 through 5 screen pixels because, for these small bar-widths, the TEDs used in the sender chip do not have enough variation in photoreceptor signal to detect an edge. Figure 4.32 shows the performance vs. stimulus width plot from which we can see that bar-width of values from 9 to 11 screen pixels results in a performance near the standard value.

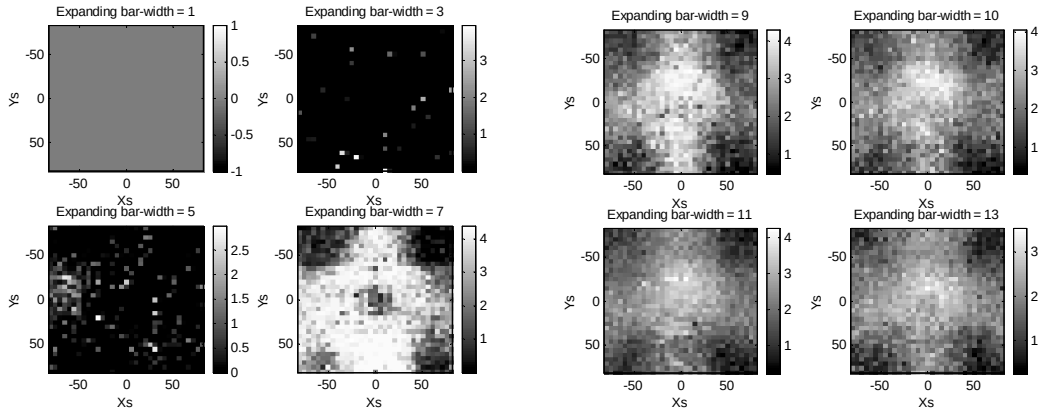


Figure 4.31: Outputs from expansion-tuned unit for different expanding stimulus width. (Xs, Ys) represents the coordinates of the FOE in screen pixels. Output is represented by brightness.

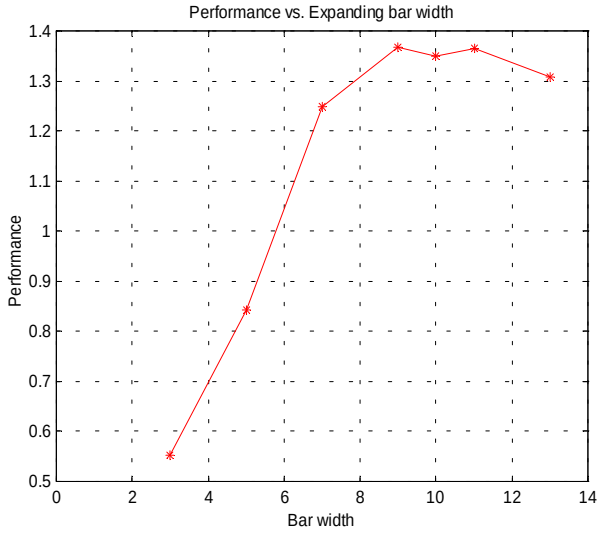


Figure 4.32: Performance vs. expanding stimulus width plot. Bar width is measured in screen pixels.

4.14 Experiment 10: Variation of CCW rotating stimulus speed

This experiment demonstrates the effect of variation in speed of a rotating stimulus on performance.

4.14.1 Setup

The motion processor has been made sensitive only to CCW rotating patterns with axis of rotation (AOR) at the center of the visual field of the chip. A CCW rotating pattern is presented on the LCD monitor. The AOR of the patterns has been varied over 33X33 screen positions over the field of view of the chip. Output has been recorded for every position of AOR of the stimulus. The speed of the stimulus has been changed for each run.

4.14.2 Result

The output plots are shown in Figure 4.33. All of these plots show low output values instead of strong responses in the central regions. This anomalous behavior has

arisen from a stimulus artifact. A slowly moving black region appears superimposed on the stimulus when the AOR is in the central region as shown in Figure 4.34. This has arisen from a beat frequency between the refresh rate of the monitor and the update of the program; the program is still updating the graphics memory when the monitor displays it. The stimulus is faster in the central region because smaller bars are drawn in this region of AOR to cover the visual field. Experiment 1 does not suffer from this stimulus artifact because the time span between drawing bars is greater there than that of this experiment. This is because eight pixels were scanned out in experiment 1 while drawing the stimulus, resulting in a slower stimulus.

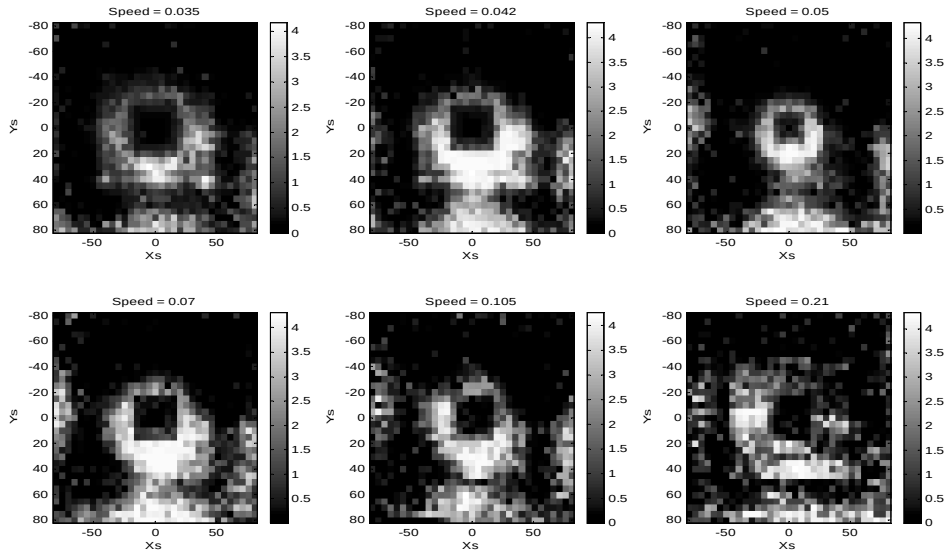


Figure 4.33: Outputs from CCW rotation-sensitive unit for different CCW rotating stimulus speed. (Xs, Ys) represents the coordinates of the AOR in screen pixels. Output is represented by brightness.

The stimulus artifact has been eliminated by incorporating delays in the stimulus generating/data recording program and the new results are plotted in Figure 4.35. The corresponding performance versus stimulus speed plot is shown in Figure 4.36 which shows good performance over a certain range of stimulus speed as expected because the simple motion sensors used to compute motion in this motion processor are sensitive to a fixed range of speed of motion.

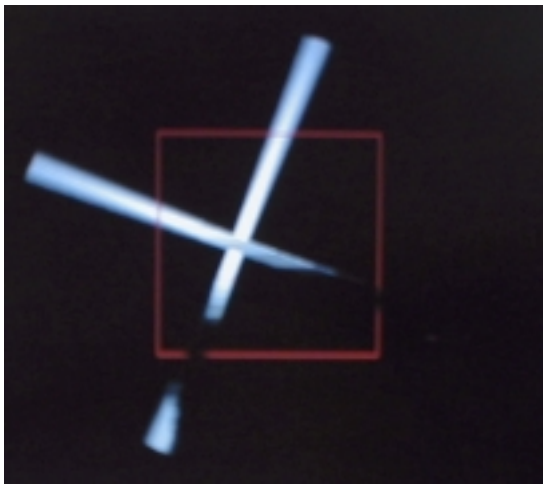


Figure 4.34: Photograph of the CCW rotating stimulus with an artifact when the AOR is near the central region.

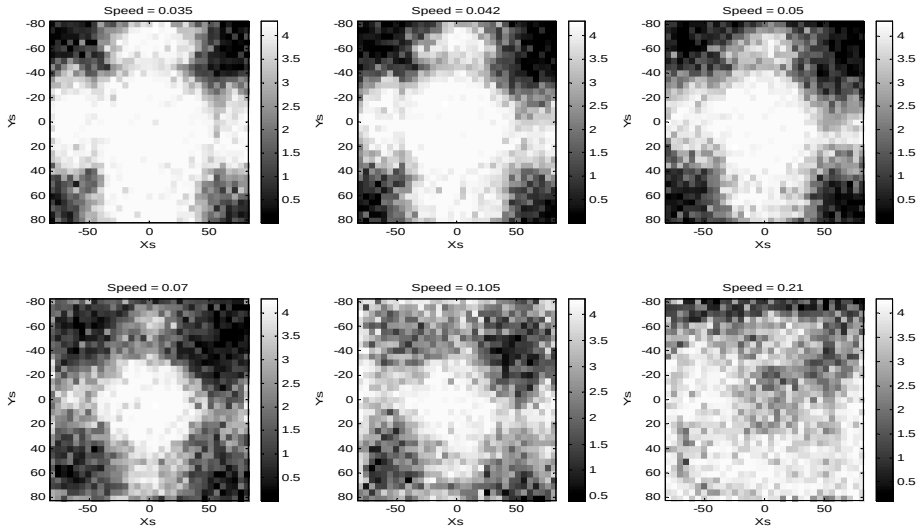


Figure 4.35: Outputs from CCW rotation-sensitive unit for different CCW rotating stimulus speed. (Xs, Ys) represents the coordinates of the AOR in screen pixels. Output is represented by brightness.

4.15 Experiment 11: Variation of CCW rotating stimulus width

This experiment demonstrates the effect of variation in the width of rotating stimulus on performance.

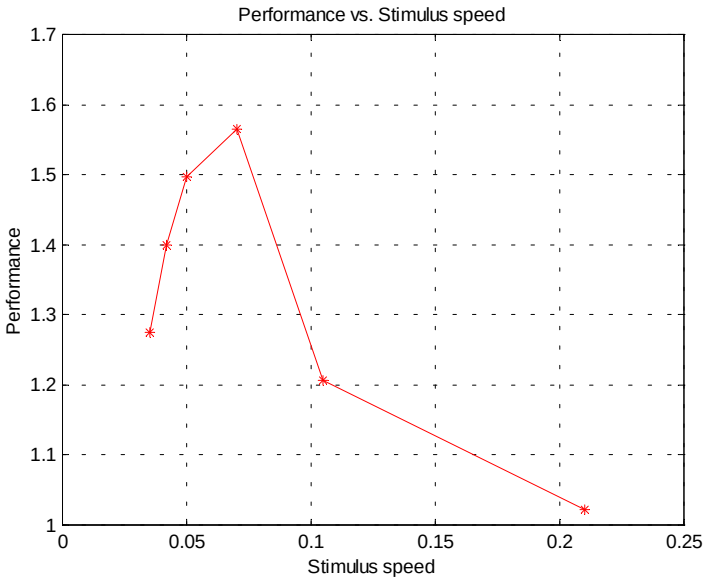


Figure 4.36: Performance vs. CCW rotating stimulus speed plot.

4.15.1 Setup

The motion processor has been made sensitive only to CCW rotating patterns with axis of rotation (AOR) at the center of the visual field of the chip. A CCW rotating pattern is presented on the LCD monitor. The AOR of the patterns has been varied over 33X33 screen positions over the field of view of the chip. Output has been recorded for every position of AOR of the stimulus. The width of the stimulus has been changed for each run.

4.15.2 Result

The output plots are shown in Figure 4.37. These outputs also suffer from the stimulus artifact (Figure 4.34) mentioned in the previous experiment.

The stimulus artifact has been eliminated by incorporating delays in the stimulus generating/data recording program and the new results are plotted in Figure 4.38. The corresponding performance versus stimulus width plot is shown in Figure 4.39 which

shows good performance for the stimulus width of 3 screen pixels or more. For bar-width of 1 screen pixel, the TEDs used in the sender chip do not have enough variation in photoreceptor signal to detect an edge.

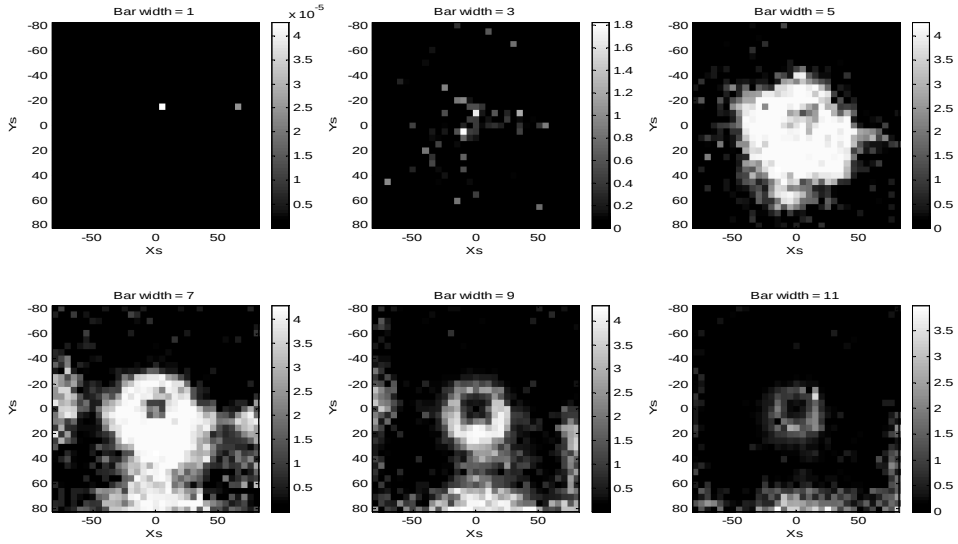


Figure 4.37: Outputs from CCW rotation-sensitive unit for different CCW rotating stimulus width. (Xs, Ys) represents the coordinates of the AOR in screen pixels. Output is represented by brightness.

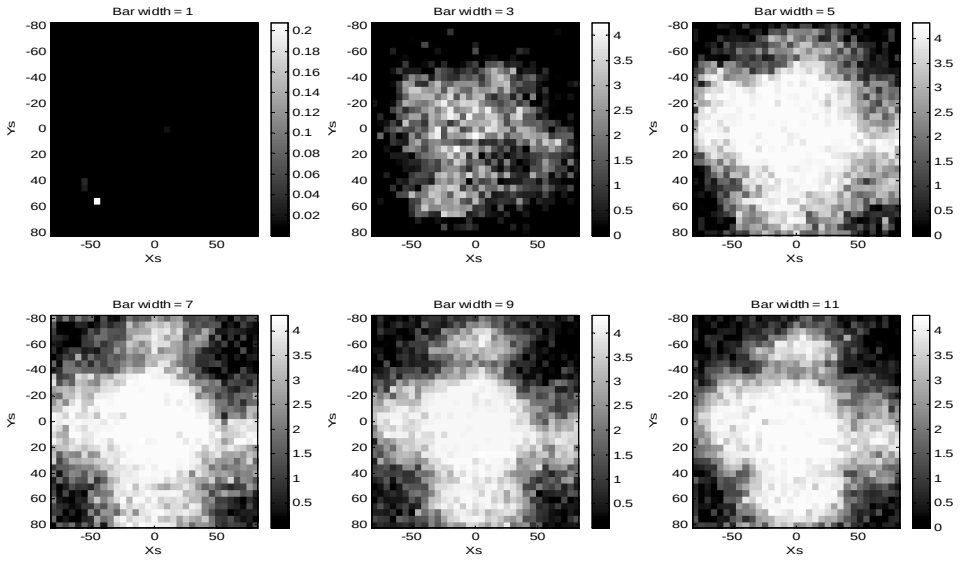


Figure 4.38: Outputs from CCW rotation-sensitive unit for different CCW rotating stimulus width. (Xs, Ys) represents the coordinates of the AOR in screen pixels. Output is represented by brightness.

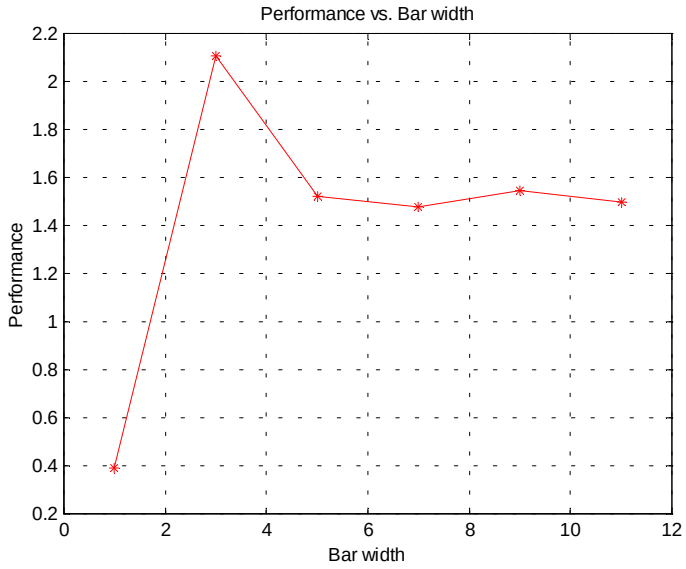


Figure 4.39: Performance vs. CCW rotating stimulus width plot. Bar width is measured in screen pixels.

5 Conclusions

In this work, a multichip neuromorphic motion processor has been presented. This motion processor has been made sensitive to complex spatial patterns of motion in the visual field. Patterns of expansion, contraction, rotation and translation and combinations of these might be encountered in the self-motion of a robotic system. Although a chip can be fabricated to detect any one of these complex patterns on a die of reasonable size, it increases die size drastically to make the circuit tuned to all of these complex patterns. A multichip system is well suited for this. This motion processor has been implemented on biological motion processing strategies that make use of multiple stages of simple parallel processors. This system retains the primary advantages of focal plane neuromorphic image processors: low power consumption, continuous-time operation and small size. So, this multichip motion processing system is ideal for sophisticated, real-time onboard sensors for autonomous robotics applications.

5.1 Limitations

However, there are some limitations to this motion processor. Although all the neuromorphic chips used in this motion processor can communicate spikes at much higher frequencies in the absence of the PIC, the frequency drops drastically after incorporating the PIC. The PIC used in this work runs at a clock rate of 33 MHz and each instruction takes four clock cycles to complete, taking the pipeline structure in consideration. Many instructions are executed in the PIC between the occurrences of two spikes. Therefore, the PIC is limiting the spike frequency in this motion processor.

In synthesizing different units simultaneously, the PIC sends a number of requests (N) sequentially to the receiver for each request coming from a transceiver, resulting in N

to 1 slowdown. Using this PIC, we do not get useful output with a slowdown of more than 4 to 1. The PIC speed and sequential mapping limits this number.

Although we are using an asynchronous digital bus for communicating address events (to and from neuromorphic chips) the routing processor (PIC) operates on clocks, which effectively discretize time resulting in interference with the functions of neuromorphic chips. This introduces discretization errors in the time domain for continuous time operation.

Another limitation of this motion processor is the manual biasing of the equilibrium frequency f_0, by the V_{qua} bias. The range of incoming spike frequencies to the receiver varies for different mappings of the units in motion processor. Thus the V_{qua} bias has to be changed manually, so that f_0 matches with the incoming spike frequency range.

Because there are only 12X12 pixels in each transceiver, benefits from using reduced receptive fields could not be used fully.

Finally, receiver pixel mismatch makes the output different for the same unit synthesized on different receiver pixels, although this was not seen to be a major problem.

5.2 Future Work

The limitations discussed in the previous section suggest to build a system which does not need to be biased manually for the synthesis of different units. We cannot get rid of sequential mapping as the neuromorphic chips in use also work on requests in a sequential manner. So, we should use a routing processor as fast as possible. It will help in synthesizing more units simultaneously by supporting more slowdown (N to 1). Replacing the PIC microcontroller with asynchronous FPGA hardware [37] for address event mapping will make the motion processor faster and more energy efficient at the cost of flexibility. In redesigning the integrating receiver, efforts in automatically biasing the thresholding effect and minimizing pixel mismatch will help to achieve outputs that

are more closely matched to the theoretical predictions. Increasing the array size of pixels will help in using reduced receptive field effectively.

This work shows the feasibility of the multichip system to perform complex tasks using simple units. Although this multichip system uses pulse based motion chips, multichip systems can also be built using motion chips that use other algorithms to compute motion. Multichip systems can be developed to perform other sensory tasks such as in auditory sensing, disparity in vision and so forth. Some of them, for example motion and disparity, can be combined in one multichip system.

6 Appendix A

6.1 Detailed Schematic

The schematic diagram of the motion processor is shown in Figure 6.1. All pin connections of the circuit are shown in this schematic diagram.

The circuit diagram of the C-element is shown in Figure 6.2. The C-element is designed using inverters, 3-input NAND gates, 3-input NOR gates and SR flip-flops. Three C-element circuits are used to design the C-element4 (Figure 6.3) for asynchronous interfacing between the sender and four transceiver chips. A delay element is connected at the R_{in} pin of the C-element4 on the way of the REQ line to allow EPROM output lines to become stable before the REQ signal arrives at the transceiver chips. The delay element consists of two inverters in series. An RC circuit is added in between the two inverters as shown in Figure 6.4. Voltage regulators are used to have the power supply voltage of 8.5 volts down to about 5 volts before applying it to the circuit. Buffers are used to stabilize the bias voltages for the pins that draw current. The REQ and X-address outputs give inverted values out of the chips. So, these signals are inverted by using inverters in these lines. There are 12 X 14 pixels in the sender chip and 12 x 12 pixels in the transceiver chips. The Integrating receiver has a 27 x 29 array of pixels. Address mismatch between the sender and the transceivers and between the transceivers and the receiver might cause lock-up of the system. So, reset circuitry has been added to each of the transceivers and the receiver in order to handle these address mismatches. There are only 33 I/O pins available from the PIC, which is not enough to accommodate eight address lines from each of the four transceivers. So, an 8 bit-wide 4/1 multiplexer is used to reduce the I/O pins needed for the address lines from the transceivers. Only two I/O pins are used to select the address lines from one of the four transceivers. The 8 bit-wide 4/1 multiplexer has been built using six 4 bit-wide 2/1 multiplexers (IC 74157) as shown in Figure 6.5. Incorporation of the multiplexer obviously introduces some extra delay.

The communications interface circuit of the sender and on the sending side of the transceiver suffers from a race condition (referring to Figure 2.4 and Figure 2.8, if V_{mem} is reset by ACK before D_{pix} goes completely high then D_{pix} will go low again) as a result of excessive requests on the bus and the circuit can get stuck in that condition. The PIC has been made to watch for stuck transceivers and one pin of the PIC has been used to send reset signal to the transceivers. The stuck condition in the sender chip has been eliminated by proper biasing. The receiver is connected to an interface card on the computer to allow display on the computer screen.

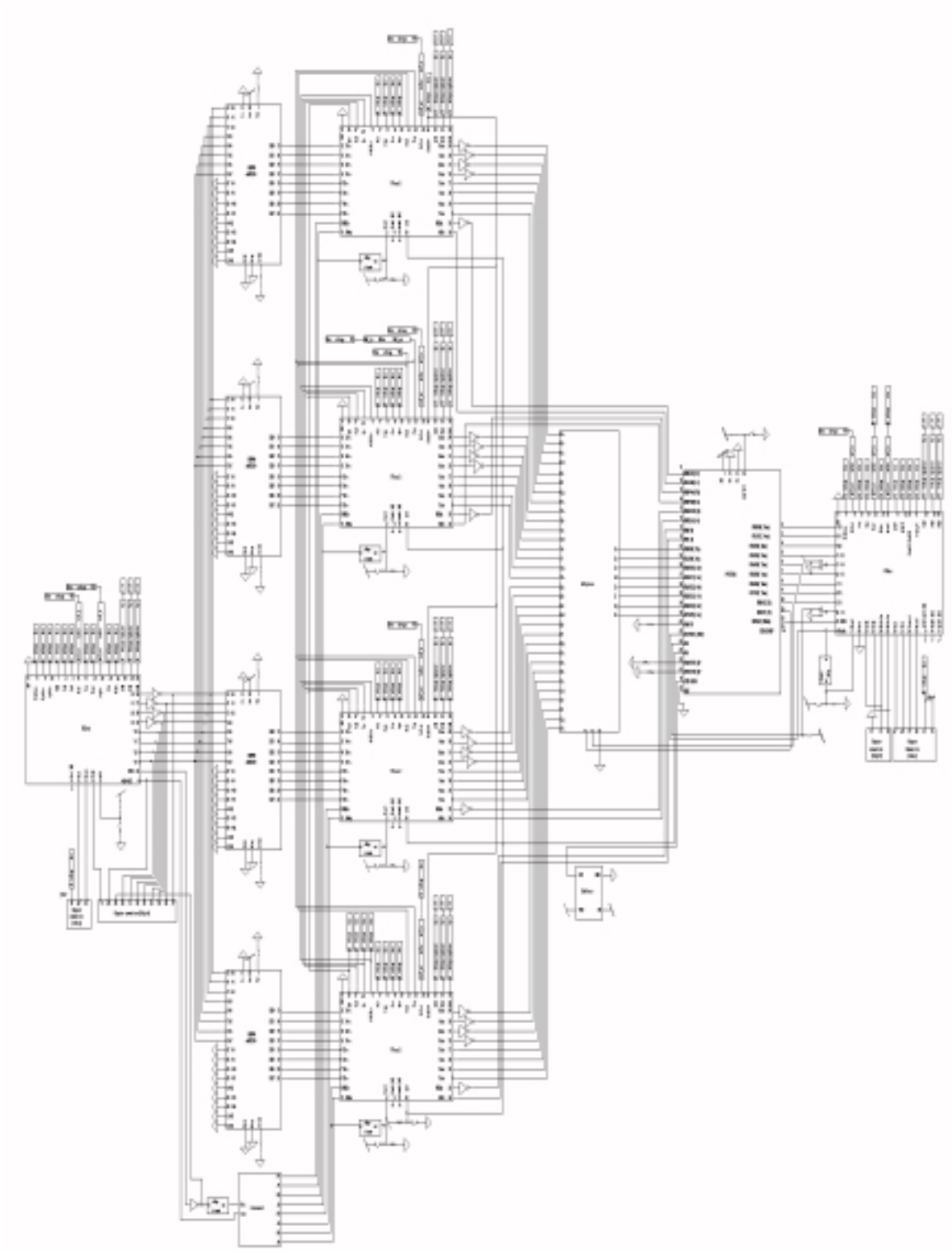


Figure 6.1: Schematic diagram of the developed multichip motion processor.

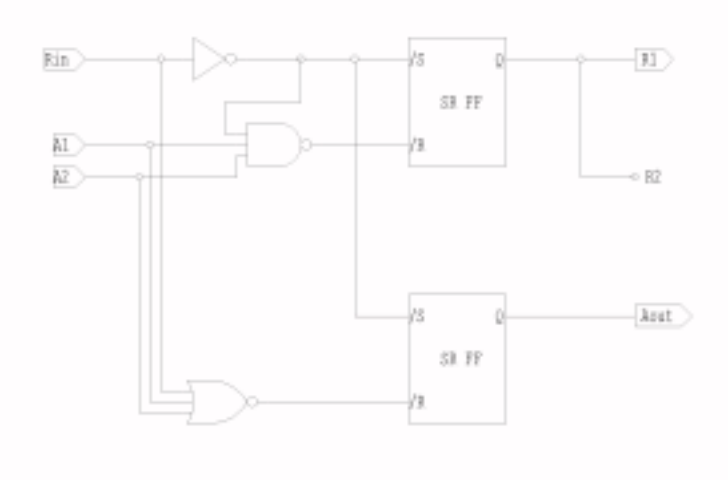


Figure 6.2: C-element.

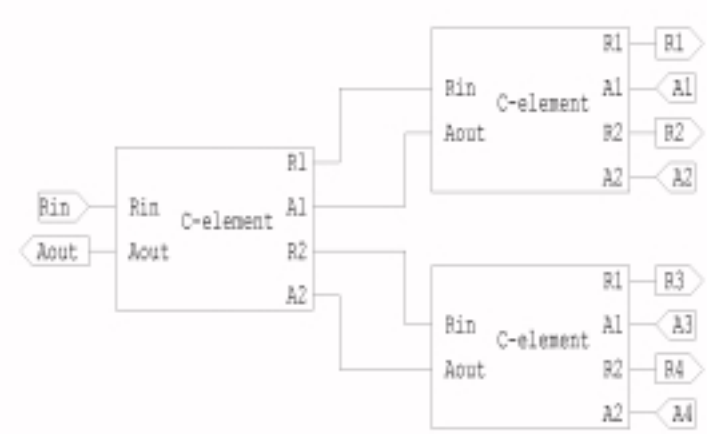


Figure 6.3: C-element4 designed with C-element.

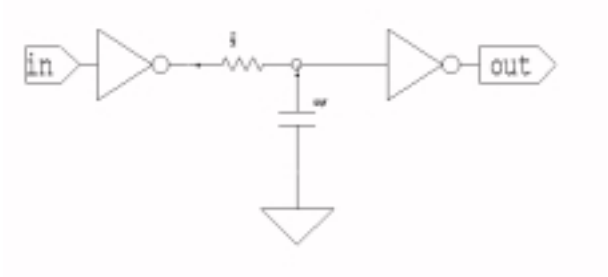


Figure 6.4: Delay element.

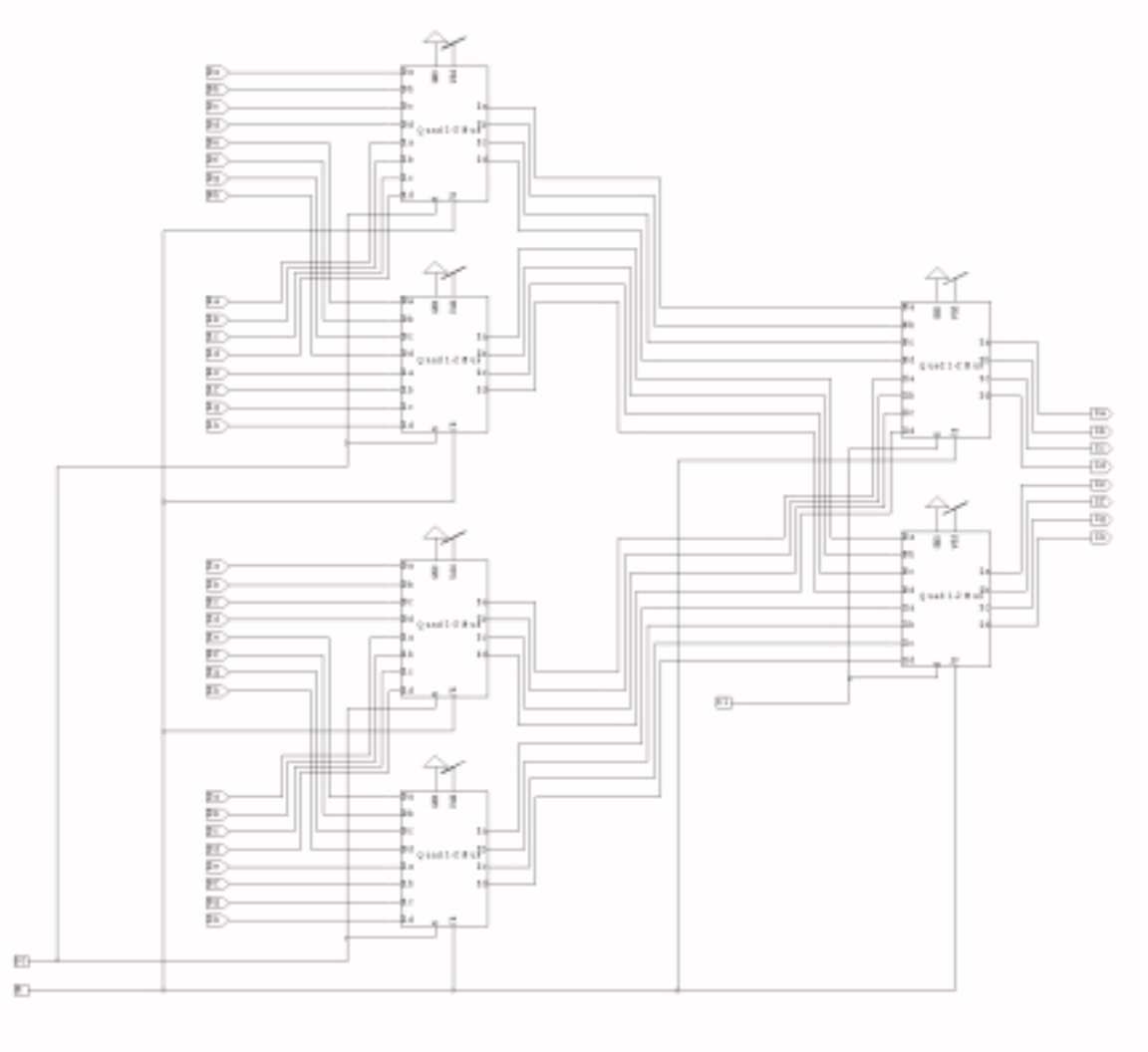


Figure 6.5: 8 bit wide 4 to 1 multiplexer designed with 4 bit wide 2 to 1 multiplexers.

7 Appendix B

7.1 PIC Programming

7.1.1 Simultaneous Synthesis of Complex and Translational Units

The PIC makes exactly four requests to the receiver for every request it receives from any transceiver. Two requests are required to implement four complex units simultaneously. One more request is necessary to implement the units tuned to four directions of translational motion. But this difference in the number of requests (spike frequencies) to the receiver pixels corresponding to four complex units and four translational units require different receiver (V_{qua}) bias settings for complex and translational units. This problem has been resolved by sending an additional request to the receiver pixels corresponding to translational units. This makes the spike frequencies same for complex units and translational units and thus makes the system work for a single receiver bias setting.

7.1.1.1 Header File: cmtrhf2.h

```
#ifndef CMTRHF2_H
#define CMTRHF2_H

#define TRUE  1
#define FALSE 0

void __STARTUP(void);
void _INT(void);
void _TMR0(void);
void _T0CKI(void);
void _PIV(void);
void main(void);
#endif
```

7.1.1.2 C File:cmtrhf2.c

```
#include <P17Cxx.H>

#include "CMTRHF2.H
#include <STDDEF.H
#define Device_CLK  33000000    // Device Oscillator is 33 MHz

unsigned int cnt = 0;
unsigned int ranreq;
unsigned int ran;
char done = 0;
unsigned int cnt0, cnt1, cnt2, cnt3;
/*
33      CONTRACTION
34      EXPANSION
35      CW ROTATION
36      CCW ROTATION
*/

//Address mapping
rom unsigned exp1[192] = {
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF
};
rom unsigned exp2[192] = {
0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF
};
rom unsigned *pexp1;
rom unsigned *pexp2;
```

```
//initialization
void __STARTUP(void)
{
   DDRB = 0x55;
   PORTB = 0;
   PORTA = 0;
   DDRC = 0;
   PORTC = 0;
   DDRD = 0xFF;
   PORTD = 0;
   DDRE = 0;
   PORTE = 0;
}
// INT interrupt handler
void _INT(void)
{
     while (1)
  {
  }
}

// TMR0 interrupt handler
void _TMR0(void)
{
  while (1)
  {
  }
}

// T0CKI interrupt handler
void _T0CKI(void)
{
  while (1)
  {
  }
}

// PIV interrupt handler
void _PIV(void)
{
    while (1)
  {
  }
}
void main(void)
{
   Install_INT(_INT);
   Install_TMR0(_TMR0);
   Install_T0CKI(_T0CKI);
   Install_PIV(_PIV);

   PORTAbits.RA2 = 0;    // MSB of the receiver address
   PORTAbits.RA3 = 1;    // reset transceivers for stuck condition
```

```
while(1)                    {
        while (done == 0)           //look for a transceiver request
        {
           cnt++;
           if (cnt > 3) cnt = 0;
           ranreq = cnt * 2;
           switch (ranreq)
           {
                   case 0:
                      if (PORTBbits.RB0 == 1)
                      {
                              PORTBbits.RB1 = 1;
                              done =1;
                              cnt0 = 0;
                              PORTBbits.RB1 = 0;
                      }
                      else
                              cnt0++;
                      break;
                   case 2:
                      if (PORTBbits.RB2 == 1)
                      {
                              PORTBbits.RB3 = 1;
                              done =1;
                              cnt1 = 0;
                              PORTBbits.RB3 = 0;
                      }
                      else
                              cnt1++;
                      break;
                   case 4:
                      if (PORTBbits.RB4 == 1)
                      {
                              PORTBbits.RB5 = 1;
                              done =1;
                              cnt2 = 0;
                              PORTBbits.RB5 = 0;
                      }
                      else
                              cnt2++;
                      break;
                   case 6:
                      if (PORTBbits.RB6 == 1)
                      {
                              PORTBbits.RB7 = 1;
                              done =1;
                              cnt3 = 0;
                              PORTBbits.RB7 = 0;
                      }
                      else
                              cnt3++;
                      break;
           }
```

```
        }

// Reset transceivers at stuck condition
        if ((cnt0 > 4999) || (cnt1 > 4999) || (cnt2 > 4999) || (cnt3 > 4999))
        {
          PORTAbits.RA3 = 0;
          cnt0 = 0;
          cnt1 = 0;
          cnt2 = 0;
          cnt3 = 0;
          PORTAbits.RA3 = 1;
        }
        done = 0;
        ran = ranreq / 2;
        if (ran < 2) PORTEbits.RE0 = ran;
        else PORTEbits.RE0 = ran - 2;
      PORTEbits.RE1 = ran / 2;
        switch (ranreq)
        {
          case 0:
            PORTC = 0x3A;
            break;
          case 2:
            PORTC = 0x39;
            break;
          case 4:
            PORTC = 0x38;
            break;
          case 6:
            PORTC = 0x37;
            break;
        }
        PORTEbits.RE2 = 1; // The first request to the translation-sensitive units
      PORTEbits.RE2 = 0;
        pexp1 = exp1 + PORTD;
        PORTC = *pexp1;
      PORTEbits.RE2 = 1; // The first request to the complex units
      PORTEbits.RE2 = 0;
        pexp2 = exp2 + PORTD;
        PORTC = *pexp2;
      PORTEbits.RE2 = 1; // The second request to the complex units
      PORTEbits.RE2 = 0;
        switch (ranreq)
        {
          case 0:
            PORTC = 0x3A;
            break;
          case 2:
            PORTC = 0x39;
            break;
          case 4:
            PORTC = 0x38;
            break;
```

```
                case 6:
                    PORTC = 0x37;
                    break;
            }
            PORTEbits.RE2 = 1; // The second request to the translation-sensitive units
        PORTEbits.RE2 = 0;
    }
}
```

7.1.2 Synthesis of the Expansion-sensitive Unit Only

7.1.2.1 Header File: onlyexp1.h

```
#ifndef ONLYEXP1_H
#define ONLYEXP1_H

#define TRUE  1
#define FALSE 0

void __STARTUP(void);
void _INT(void);
void _TMR0(void);
void _T0CKI(void);
void _PIV(void);
void main(void);
#endif
```

7.1.2.2 C File: onlyexp1.c

```
#include <P17Cxx.H>

#include "ONLYEXP1.H"
#include <STDDEF.H>
#define Device_CLK  33000000    // Device Oscillator is 33 MHz

unsigned int cnt = 0;           // example 16-bit integer global variable
unsigned int ranreq;
unsigned int ran;
char done = 0;
unsigned int cnt0, cnt1, cnt2, cnt3;

// Address mapping
rom unsigned exp1[192] = {
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
```

```
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF
};
rom unsigned *pexp1;

//initialization
void __STARTUP(void)
{
    DDRB = 0x55;
    PORTB = 0;
    PORTA = 0;
    DDRC = 0;
    PORTC = 0;
    DDRD = 0xFF;
    PORTD = 0;
    DDRE = 0;
    PORTE = 0;
}
// INT interrupt handler
void _INT(void)
{
      while (1)
   {
   }
}

// TMR0 interrupt handler
void _TMR0(void)
{
   while (1)
   {
   }
}

// T0CKI interrupt handler
void _T0CKI(void)
{
   while (1)
   {
   }
}

// PIV interrupt handler
void _PIV(void)
{
     while (1)
   {
   }
```

```
}
void main(void)
{
   Install_INT(_INT);
   Install_TMR0(_TMR0);
   Install_T0CKI(_T0CKI);
   Install_PIV(_PIV);

   PORTAbits.RA2 = 0;    // MSB of the receiver address
   PORTAbits.RA3 = 1;    // reset transceivers for stuck condition

   while(1)                    {
         while (done == 0)            // Look for a transceiver request
         {
            cnt++;
            if (cnt > 3) cnt = 0;
            ranreq = cnt * 2;
            switch (ranreq)
            {
                  case 0:
                     if (PORTBbits.RB0 == 1)
                     {
                           PORTBbits.RB1 = 1;
                           done =1;
                           cnt0 = 0;
                           PORTBbits.RB1 = 0;
                     }
                     else
                           cnt0++;
                     break;
                  case 2:
                     if (PORTBbits.RB2 == 1)
                     {
                           PORTBbits.RB3 = 1;
                           done =1;
                           cnt1 = 0;
                           PORTBbits.RB3 = 0;
                     }
                     else
                           cnt1++;
                     break;
                  case 4:
                     if (PORTBbits.RB4 == 1)
                     {
                           PORTBbits.RB5 = 1;
                           done =1;
                           cnt2 = 0;
                           PORTBbits.RB5 = 0;
                     }
                     else
                           cnt2++;
                     break;
                  case 6:
```

```
                    if (PORTBbits.RB6 == 1)
                    {
                        PORTBbits.RB7 = 1;
                        done =1;
                        cnt3 = 0;
                        PORTBbits.RB7 = 0;
                    }
                    else
                        cnt3++;
                    break;
                }
            }

// Reset transceivers at stuck condition
            if ((cnt0 > 4999) || (cnt1 > 4999) || (cnt2 > 4999) || (cnt3 > 4999))
            {
               PORTAbits.RA3 = 0;
               cnt0 = 0;
               cnt1 = 0;
               cnt2 = 0;
               cnt3 = 0;
               PORTAbits.RA3 = 1;
}
done = 0;

ran = ranreq / 2;

if (ran < 2) PORTEbits.RE0 = ran;

else PORTEbits.RE0 = ran - 2;
      PORTEbits.RE1 = ran / 2;

pexp1 = exp1 + PORTD;

PORTC = *pexp1;

if (PORTC == 0x34) // 0x33 for contraction-sensitive unit only

{
         PORTEbits.RE2 = 1;
         PORTEbits.RE2 = 0;

}
   }
}
```

7.1.3 3X3 FOE Positions

7.1.3.1 Header File:foe9ps1.h

```
#ifndef FOE9PS1_H
#define FOE9PS1_H

#define TRUE  1
#define FALSE 0

void __STARTUP(void);
void _INT(void);
void _TMR0(void);
void _T0CKI(void);
void _PIV(void);
void main(void);
#endif
```

7.1.3.2 C File: foe9ps1.c

```
#include <P17Cxx.H>
#include "FOE9PS1.H"
#include <STDDEF.H>
#define Device_CLK  33000000    // Device Oscillator is 33 MHz

unsigned int cnt = 0
unsigned int ranreq;
unsigned int ran;
char done = 0;
unsigned int cnt0, cnt1, cnt2, cnt3;
/*
FOE position 1: 0x33
FOE position 2: 0x34
FOE position 3: 0x35
FOE position 4: 0x36
FOE position 5: 0x37
FOE position 6: 0x38
FOE position 7: 0x39
FOE position 8: 0x3A
FOE position 9: 0x3B
*/

// Address mapping
rom unsigned exp1a[192] = {
0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
```

```
0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF
};
rom unsigned exp1b[192] = {
0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF,
0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x83, 0x33, 0x33, 0x33, 0xFF, 0xFF, 0xFF, 0xFF
};

rom unsigned exp2a[192] = {
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF
};
rom unsigned exp2b[192] = {
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF
};

rom unsigned exp2d[192] = {
0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
```

```
0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF,
0x84, 0x84, 0x84, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0x34, 0xFF, 0xFF, 0xFF, 0xFF
};
rom unsigned exp3a[192] = {
0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF
};

rom unsigned exp3d[192] = {
0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF,
0x85, 0x85, 0x85, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0x35, 0xFF, 0xFF, 0xFF, 0xFF
};
rom unsigned exp4a[192] = {
0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
```

```
0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF
};
rom unsigned exp4b[192] = {
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF
};
rom unsigned exp4c[192] = {
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF,
0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x86, 0x36, 0x36, 0x36, 0xFF, 0xFF, 0xFF, 0xFF
};

rom unsigned exp5a[192] = {
0x87, 0x87, 0x87, 0x87, 0x87, 0x87, 0x37, 0x37, 0x37, 0x37, 0x37, 0x37, 0xFF, 0xFF, 0xFF, 0xFF,
0x87, 0x87, 0x87, 0x87, 0x87, 0x87, 0x37, 0x37, 0x37, 0x37, 0x37, 0x37, 0xFF, 0xFF, 0xFF, 0xFF,
0x87, 0x87, 0x87, 0x87, 0x87, 0x87, 0x37, 0x37, 0x37, 0x37, 0x37, 0x37, 0xFF, 0xFF, 0xFF, 0xFF,
0x87, 0x87, 0x87, 0x87, 0x87, 0x87, 0x37, 0x37, 0x37, 0x37, 0x37, 0x37, 0xFF, 0xFF, 0xFF, 0xFF,
0x87, 0x87, 0x87, 0x87, 0x87, 0x87, 0x37, 0x37, 0x37, 0x37, 0x37, 0x37, 0xFF, 0xFF, 0xFF, 0xFF,
0x87, 0x87, 0x87, 0x87, 0x87, 0x87, 0x37, 0x37, 0x37, 0x37, 0x37, 0x37, 0xFF, 0xFF, 0xFF, 0xFF,
0x87, 0x87, 0x87, 0x87, 0x87, 0x87, 0x37, 0x37, 0x37, 0x37, 0x37, 0x37, 0xFF, 0xFF, 0xFF, 0xFF,
0x87, 0x87, 0x87, 0x87, 0x87, 0x87, 0x37, 0x37, 0x37, 0x37, 0x37, 0x37, 0xFF, 0xFF, 0xFF, 0xFF,
0x87, 0x87, 0x87, 0x87, 0x87, 0x87, 0x37, 0x37, 0x37, 0x37, 0x37, 0x37, 0xFF, 0xFF, 0xFF, 0xFF,
0x87, 0x87, 0x87, 0x87, 0x87, 0x87, 0x37, 0x37, 0x37, 0x37, 0x37, 0x37, 0xFF, 0xFF, 0xFF, 0xFF,
0x87, 0x87, 0x87, 0x87, 0x87, 0x87, 0x37, 0x37, 0x37, 0x37, 0x37, 0x37, 0xFF, 0xFF, 0xFF, 0xFF,
0x87, 0x87, 0x87, 0x87, 0x87, 0x87, 0x37, 0x37, 0x37, 0x37, 0x37, 0x37, 0xFF, 0xFF, 0xFF, 0xFF
};


rom unsigned exp6a[192] = {
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
```

```
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF
};
rom unsigned exp6b[192] = {
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF
};
rom unsigned exp6c[192] = {
0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF,
0x88, 0x88, 0x88, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0x38, 0xFF, 0xFF, 0xFF, 0xFF
};

rom unsigned exp7a[192] = {
0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF
};

rom unsigned exp7d[192] = {
```

```
0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF,
0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x89, 0x39, 0x39, 0x39, 0xFF, 0xFF, 0xFF, 0xFF
};
rom unsigned exp8a[192] = {
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF
};
rom unsigned exp8b[192] = {
0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF
};

rom unsigned exp8d[192] = {
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
```

```
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x8A, 0x3A, 0x3A, 0x3A, 0xFF, 0xFF, 0xFF, 0xFF,
};
rom unsigned exp9a[192] = {
0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF
};
rom unsigned exp9b[192] = {
0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF,
0x8B, 0x8B, 0x8B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0x3B, 0xFF, 0xFF, 0xFF, 0xFF
};

rom unsigned *pexp1a;
rom unsigned *pexp1b;
rom unsigned *pexp2a;
rom unsigned *pexp2b;
rom unsigned *pexp2d;
rom unsigned *pexp3a;
rom unsigned *pexp3d;
rom unsigned *pexp4a;
rom unsigned *pexp4b;
rom unsigned *pexp4c;
rom unsigned *pexp5a;
rom unsigned *pexp6a;
rom unsigned *pexp6b;
rom unsigned *pexp6c;
rom unsigned *pexp7a;
rom unsigned *pexp7d;
rom unsigned *pexp8a;
rom unsigned *pexp8b;
rom unsigned *pexp8d;
rom unsigned *pexp9a;
rom unsigned *pexp9b;
```

```
//initialization
void __STARTUP(void)
{
   DDRB = 0x55;
   PORTB = 0;
   PORTA = 0;
   DDRC = 0;
   PORTC = 0;
   DDRD = 0xFF;
   PORTD = 0;
   DDRE = 0;
   PORTE = 0;
}

// INT interrupt handler
void _INT(void)
{
     while (1)
  {
  }
}

// TMR0 interrupt handler
void _TMR0(void)
{
  while (1)
  {
  }
}

// T0CKI interrupt handler
void _T0CKI(void)
{
  while (1)
  {
  }
}

// PIV interrupt handler
void _PIV(void)
{
    while (1)
  {
  }
}
void main(void)
{
   Install_INT(_INT);
   Install_TMR0(_TMR0);
   Install_T0CKI(_T0CKI);
   Install_PIV(_PIV);
```

```
PORTAbits.RA2 = 0;    // MSB of the receiver address
PORTAbits.RA3 = 1;    // reset transceivers for stuck condition

while(1)                    {
        while (done == 0)           // Look for a transceiver request
        {
           cnt++;
           if (cnt > 3) cnt = 0;
           ranreq = cnt * 2;
           switch (ranreq)
           {
                   case 0:
                      if (PORTBbits.RB0 == 1)
                      {
                              PORTBbits.RB1 = 1;
                              done =1;
                              cnt0 = 0;
                              PORTBbits.RB1 = 0;
                      }
                      else
                              cnt0++;
                      break;
                   case 2:
                      if (PORTBbits.RB2 == 1)
                      {
                              PORTBbits.RB3 = 1;
                              done =1;
                              cnt1 = 0;
                              PORTBbits.RB3 = 0;
                      }
                      else
                              cnt1++;
                      break;
                   case 4:
                      if (PORTBbits.RB4 == 1)
                      {
                              PORTBbits.RB5 = 1;
                              done =1;
                              cnt2 = 0;
                              PORTBbits.RB5 = 0;
                      }
                      else
                              cnt2++;
                      break;
                   case 6:
                      if (PORTBbits.RB6 == 1)
                      {
                              PORTBbits.RB7 = 1;
                              done =1;
                              cnt3 = 0;
                              PORTBbits.RB7 = 0;
                      }
                      else
```

```
                        cnt3++;
                    break;
                }
            }

//Reset transceivers at stuck condition
            if ((cnt0 > 4999) || (cnt1 > 4999) || (cnt2 > 4999) || (cnt3 > 4999))
            {
               PORTAbits.RA3 = 0;
               cnt0 = 0;
               cnt1 = 0;
               cnt2 = 0;
               cnt3 = 0;
               PORTAbits.RA3 = 1;
            }
            done = 0;
            ran = ranreq / 2;
            if (ran < 2) PORTEbits.RE0 = ran;
            else PORTEbits.RE0 = ran - 2;
          PORTEbits.RE1 = ran / 2;
            switch (ranreq)
            {
               case 0:
               case 6:
                        pexp1a = exp1a + PORTD;
                        PORTC = *pexp1a;
                  break;
               case 2:
               case 4:
                        pexp1b = exp1b + PORTD;
                        PORTC = *pexp1b;
                        break;
            }
            if (PORTC == 0x33)
            {
               PORTEbits.RE2 = 1;
             PORTEbits.RE2 = 0;
            }
            switch (ranreq)
            {
               case 0:
               case 4:
                        pexp2a = exp2a + PORTD;
                        PORTC = *pexp2a;
                  break;
               case 2:
                        pexp2b = exp2b + PORTD;
                        PORTC = *pexp2b;
                        break;
               case 6:
                        pexp2d = exp2d + PORTD;
                        PORTC = *pexp2d;
                        break;
```

```
  }
  if (PORTC == 0x34)
  {
     PORTEbits.RE2 = 1;
PORTEbits.RE2 = 0;
  }
  switch (ranreq)
  {
     case 0:
     case 2:
              pexp3a = exp3a + PORTD;
              PORTC = *pexp3a;
        break;
     case 4:
     case 6:
              pexp3d = exp3d + PORTD;
              PORTC = *pexp3d;
              break;
  }
  if (PORTC == 0x35)
  {
     PORTEbits.RE2 = 1;
PORTEbits.RE2 = 0;
  }
  switch (ranreq)
  {
     case 0:
              pexp4a = exp4a + PORTD;
              PORTC = *pexp4a;
        break;
     case 2:
     case 6:
              pexp4b = exp4b + PORTD;
              PORTC = *pexp4b;
              break;
     case 4:
              pexp4c = exp4c + PORTD;
              PORTC = *pexp4c;
              break;
  }
  if (PORTC == 0x36)
  {
     PORTEbits.RE2 = 1;
PORTEbits.RE2 = 0;
  }

  pexp5a = exp5a + PORTD;
  PORTC = *pexp5a;
  if (PORTC == 0x37)
  {
     PORTEbits.RE2 = 1;
PORTEbits.RE2 = 0;
  }
```

```
  switch (ranreq)
  {
     case 0:
              pexp6a = exp6a + PORTD;
              PORTC = *pexp6a;
        break;
     case 2:
     case 6:
              pexp6b = exp6b + PORTD;
              PORTC = *pexp6b;
              break;
     case 4:
              pexp6c = exp6c + PORTD;
              PORTC = *pexp6c;
              break;
  }
  if (PORTC == 0x38)
  {
     PORTEbits.RE2 = 1;
PORTEbits.RE2 = 0;
  }
  switch (ranreq)
  {
     case 0:
     case 2:
              pexp7a = exp7a + PORTD;
              PORTC = *pexp7a;
        break;
     case 4:
     case 6:
              pexp7d = exp7d + PORTD;
              PORTC = *pexp7d;
              break;
  }
  if (PORTC == 0x39)
  {
     PORTEbits.RE2 = 1;
PORTEbits.RE2 = 0;
  }
  switch (ranreq)
  {
     case 0:
     case 4:
              pexp8a = exp8a + PORTD;
              PORTC = *pexp8a;
        break;
     case 2:
              pexp8b = exp8b + PORTD;
              PORTC = *pexp8b;
              break;
     case 6:
              pexp8d = exp8d + PORTD;
```

```
                PORTC = *pexp8d;
                break;
          }
          if (PORTC == 0x3A)
          {
             PORTEbits.RE2 = 1;
        PORTEbits.RE2 = 0;
          }
          switch (ranreq)
          {
             case 0:
             case 6:
                pexp9a = exp9a + PORTD;
                PORTC = *pexp9a;
               break;
             case 2:
             case 4:
                pexp9b = exp9b + PORTD;
                PORTC = *pexp9b;
                break;
          }
          if (PORTC == 0x3B)
          {
             PORTEbits.RE2 = 1;
        PORTEbits.RE2 = 0;
          }
   }
}
```

8 Appendix C

8.1 Circuit Biasing

8.1.1 Simultaneous Synthesis of Complex and Translational Units

TEDsndr3 (Volts)	ITIxcvr2 (Volts)	INTrcvr (Volts)
$V_{prbias} = 3.006$	$V_{spon} = 0$	$V_{qua} = 4.51$
$V_{hysbias} = 4.66$	$V_{qua} = 3.48$	$V_{op} = 0.165$
$V_{thr} = 2.493$	$V_{tau} = 1.016$	$V_{off} = 0$
$V_{ngain} = 0.181$	$V_{leak} = 0.091$	$V_{follbias} = 0.808$
$V_{limit} = 1.785$	$V_{top} = 0$	AckPD = 0.996
$V_{leak} = 4.17$	$V_{offset} = 0.3$	AckOrPU = 3.8
$V_{nin} = 0$	$V_{thr} = 1.572$	AckPU = 3.97
$V_{follbias} = 0$	$V_{pd} = 1.27$	$V_{sync\_pd} = 1.672$
$V_{rpd} = 5.09$	$V_{rpd} = 5.06$	$V_{lref} = 1.187$
$V_{dpd} = 1.988$	$V_{pu} = 4.3$	$V_{lbias} = 0.987$
$V_{pub} = 3.244$	$V_{follbias} = 0$	$V_{outref} = 3.231$
$V_{lref} = 0.209$		
$V_{prref} = 2.55$		

The bias setting for V_{qua} of INTrcvr might have to be changed for other mappings to account for the change in incoming spike frequency range.

9 Appendix D

All files are in the directory c:\users\skarif\research\thesis9_28.

9.1 Matlab Files to Generate Figures

9.1.1 Figures of Results

Matlab file name	Generated Figures
result4	Figure 4.8 - Figure 4.11
result4p	Figure 4.12
rvvqua1	Figure 4.18
rvrf1	Figure 4.20, Figure 4.22
rvpixel3	Figure 4.23
rfoe9	Figure 4.24
rvesz5	Figure 4.27
rvpixel1	Figure 4.28
rves1	Figure 4.29, Figure 4.30
rvew1	Figure 4.31, Figure 4.32
rvws1	Figure 4.33
rvws3	Figure 4.35, Figure 4.36
rvww1	Figure 4.37
rvww3	Figure 4.38, Figure 4.39

9.1.2 Figures of Theoretical Predictions

Matlab file name	Generated Figures
thvecade	Figure 3.2, Figure 3.3
plotthte	Figure 3.5, Figure 3.6
thvecadn	Figure 3.7
plotthtn	Figure 3.9
thvecadw	Figure 3.10
thvecadc	Figure 3.11
plotthrf6	Figure 3.15
plotthtep1	Figure 3.17
plotth1foe	Figure 3.19
plotth4foe	Figure 3.21
plotthte3	Figure 3.24
plotthte5	Figure 3.25
thperf1	Figure 4.6
plotthn	Figure 4.13
plotthe	Figure 4.14
plotthc	Figure 4.15
plotthw	Figure 4.16
plotthtr	Figure 4.17
plotthrf	Figure 4.21
plotth9foe	Figure 4.25

9.2 Files of Figures

File name	Figure
aer_modprot.eps	Figure 2.1
block3.jpg	Figure 2.2
ts3.jpg	Figure 2.3
ts3pix5.ps	Figure 2.4
trace.eps	Figure 2.5
itixcvr2.jpg	Figure 2.6
iti_blockdia1d.eps	Figure 2.7 (a)
iti_trace.eps	Figure 2.7 (b)
itipix5.ps	Figure 2.8
intlay.jpg	Figure 2.9
intrcvr.jpg	Figure 2.10
setup.jpg	Figure 4.2
exp1.jpg	Figure 4.3
rot1.jpg	Figure 4.4
movbar1.jpg	Figure 4.5
req4_1.jpg	Figure 4.7
rotart1.jpg	Figure 4.34
schematic3.sdb	Figure 6.1
celement.jpg	Figure 6.2
celemen4.jpg	Figure 6.3
delayelm.jpg	Figure 6.4
mux.jpg	Figure 6.5

10 References

[1] J. H. Rieger and L. Toet, “Human visual navigation in the presence of 3D rotations,” Biol. Cybern. 52, 377-381 (1985).

[2] W. H. Warren and D. J. Hannon, “Direction of self-motion is perceived from optical flow,” Nature (London) 336, 162-163 (1988).

[3] W. H. Warren and D. J. Hannon, “Eye movements and optical flow,” J. Opt. Soc. Am. A 7, 160-169 (1990).

[4] J. A. Crowell, C. S. Royden, M. S. Banks, K. H. Swenson, and A. B. Sekuler, “Optic flow and heading judgments,” Invest. Ophthalmol. Visual Sci. Suppl. 31, 522 (1992).

[5] A. V. van den Berg, “Robustness of perception of heading from optic flow,” Vision Res. 32, 1285-1296 (1992).

[6] J. A. Crowell and M. S. Banks, “Perceiving heading with different retinal regions and types of optic flow,” Percept. Psychophys. 53, 325-337 (1993).

[7] W. H. Warren, D. R. Mestre, A. W. Blackwell, and M. W. Morris, “Perception of circular heading from optical flow,” J. Exp. Psychol. Hum. Percept. Perform. 17, 28-43 (1991).

[8] K. T. Turano and X. Wang, “Visual discrimination between a curved and straight path of self motion: effects of forward speed,” Vision Res. 24, 107-114 (1994).

[9] J. J. Gibson, The Perception of the Visual World (Houghton Miffin, Boston, Mass., 1950), Chap. 7, pp. 117-144.

[10] E. R. Fossum, “CMOS image sensors: Electronic camera-on-a-chip,” IEEE Trans. Electron Devices, vol. 44, no. 10, pp. 1689-1698, 1997.

[11] T. Delbruck and C. Mead, “Analog VLSI phototransduction by continuous-time, adaptive, logarithmic photoreceptor circuits,” Tech. Rep. 30, Department of Computation and Neural Systems, California Institute of Technology, 1993.

[12] R. Etienne-Cummings, J. Van der Spiegel, and P. Mueller, “A focal plane visual motion measurement sensor,” IEEE Trans. On Circuit and Systems I, vol. 44, no. 1, pp. 55-56, 1997.

[13] R. A. Deutschmann and C. Koch, “Compact real-time 2-D gradient based analog VLSI motion sensor,” in Proceedings of the Int. Conf. On Advanced Focal Plane Arrays and Electronic Cameras, Zurich/Switzerland, 1998.

[14] C. M. Higgins, R. A. Deutschmann, and C. Koch, “Pulse-based 2-D motion sensors,” IEEE Trans. On Circuits and Systems II, vol. 46, no. 6, pp. 677-687, 1999.

[15] A. Stocker and R. Douglas, “Computation of smooth optical flow in a feedback connected analog network,” in Advances in Neural Information Processing Systems, M. S. Kearns, S. A. Solla, and D. A. Cohn, Eds., Cambridge, MA, 1999, vol. 11, MIT Press.

[16] C. M. Higgins and C. Koch, “A Modular Multichip Neuromorphic Architecture for Real-Time Visual Motion Processing,” Analog Integrated Circuits and Signal Processing, 24(3), September 2000.

[17] C. A. Mead, “Neuromorphic electronic systems,” Proceedings of the IEEE, vol. 78, pp. 1629-1636, 1990.

[18] C. Fermuller and Y. Aloimonos, “On the geometry of visual correspondence,” International Journal of Computer Vision, vol. 21, no. 3, pp. 233-247, 1997.

[19] C. Fermuller and Y. Aloimonos, “Direct perception of three-dimensional motion from patterns of visual motion,” Science, vol. 270, pp. 1973-1976, 22 December 1995.

[20] C. Fermuller and Y. Aloimonos, “Qualitative egomotion,” International Journal of Computer Vision, vol. 15, pp. 7-29, 1995.

[21] W. Burger and B. Bhanu, “Estimating 3-D egomotion from perspective image sequences,” IEEE Trans. Pattern Analysis and Machine Intelligence, vol. 12, no. 11, pp. 1040-1058, 1990.

[22] M. J. Barth and S. Tsuji, “Egomotion determination through an intelligent gaze control strategy,” IEEE Trans. Syst. Man Cybern., vol. 23, no. 5, pp. 1424-1432, 1993.

[23] R. Jain, “Direct computation of the focus of expansion,” IEEE Trans. Pattern Analysis and Machine Intelligence, vol. 5, pp. 58-64, 1983.

[24] G. Indiveri, J. Kramer, and C. Koch, “Parallel analog VLSI architectures for computation of heading direction and time-to-contact,” in Advances in Neural Information Processing Systems, D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, Eds., Cambridge, MA, 1996, vol. 8, pp. 720-726, MIT.

[25] I. McQuirk, “An analog VLSI chip for estimating the focus of expansion,” Tech. Rep. 1577, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1996.

[26] R. A. Deutschmann and O. G. Wenisch, “Compressive computation in analog VLSI motion sensors,” in Proceedings of Deutsche Arbeitsgemeinschaft fur Mustererkennung, 1998.

[27] C. M. Higgins and C. Koch, “An integrated Vision Sensor for the Computation of Optical Flow Singular Points,” Advances in Neural Information Processing Systems (NIPS) 11.

[28] M. A. Mahowald, VLSI analogs of neuronal visual processing: a synthesis of form and function, Ph.D. thesis, Department of Computation and Neural Systems, California Institute of Technology, Pasadena, CA., 1992.

[29] K. Boahen, “Retinomorphic vision systems,” in proceedings of the International Conference on Microelectronics for Neural Networks and Fuzzy Systems. IEEE, 1996.

[30] J. Lazzaro, J. Wawrzynek, M. Mahowald, M. Sivilotti, and D. Gillespie, “Silicon auditory processors as computer peripherals,” IEEE Trans. Neural Networks, vol. 4, no. 3, May 1993.

[31] A. Mortara, E. Vittoz, and P. Verier, “A communications scheme for analog VLSI perceptive systems,” IEEE Journal of Solid State Circuits, vol. 30, no. 6, June 1995.

[32] S. DeWeerth, G. Patel, M. Simoni, D. Schimmel, and R. Calabrese, “A VLSI architecture for modeling intersegmental coordination,” in Proc. Of the 17th conference on Advanced Research in VLSI, Ann Arbor, MI, 1997.

[33] K. Boahen, NSF Neuromorphic Engineering Workshop Report, Telluride, CO, 1996.

[34] P. Venier, A. Mortara, X. Arreguit, and E. Vittoz, “An integrated cortical layer for orientation enhancement,” IEEE Journal of Solid State Circuits, vol. 32, no. 2, pp. 177-186, February 1997.

[35] S. Grossberg, G. Carpenter, E. Schwartz, E. Mingolla, D. Bullock, P. Gaudiano, A. Andreou, G. Cauwenberghs, and A. Hubbard, “Automated vision and sensing systems at Boston University,” in Proc. Of the DARPA Image Understanding Workshop, New Orleans, LA, 1997.

[36] N. Kumar, W. Himmelbauer, G. Cauwenberghs, and A. G. Andreou, “An analog VLSI chip with asynchronous interface for auditory feature extraction,” IEEE Trans. On Circuit and Systems II, vol. 45, no. 5, pp. 600-606, May 1998.

[37] Ph. Hafliger, “An Asynchronous Address Event Mapping Architecture,” Submitted.

[38] C. M. Higgins and C. Koch, “Multichip Neuromorphic Motion Processing,” 1999 Conference on Advanced Research in VLSI, Atlanta, GA, 1998.

[39] J. Kramer and G. Indiveri, “Neuromorphic vision sensors and preprocessors in system application,” In Advanced Focal Plane Arrays and Electronic Cameras (AFPAEC’98), Zurich, Switzerland, May 1998.

[40] A. Mortara, “A pulsed communication/computation framework for analog VLSI perceptive systems,” In T. S. Lande, editor, Neuromorphic Systems Engineering, pp. 217-228. Kluwer Academic, Norwell, MA, 1998.

[41] K. Boahen, “A retinomorphic vision system,” IEEE Micro, 16(5):30-39, Oct. 1996.

[42] M. Mahowald, “An Analog VLSI System for Stereoscopic Vision,” Kluwer, Boston, 1994.

[43] Z. Kalayjian, J. Waskiewicz, D. Yochelson, and A. Andreou, “Asynchronous sampling of 2-D arrays using winner-takes-all arbitration,” in IEEE International Symposium on Circuits and Systems, Atlanta, GA, 1996.

[44] K. Boahen, “A throughput-on-demand 2-D address-event transmitter for neuromorphic chips,” in Proc. Of the 20th Conference on Advanced Research in VLSI, Atlanta, GA, 1999.

[45] J. Kramer, R. Sarpeshkar, and C. Koch, “Pulse-based analog VLSI velocity sensors,” IEEE Trans. Circuits and Systems II, vol. 44, pp. 86-101, 1997.

[46] J. Kramer, “Compact integrated motion sensor with three-pixel interaction,” IEEE Trans. Pattern Analysis and Machine Intelligence, vol. 18, pp. 455-460, 1996.