DESIGN OF A VIDEO HISTOGRAMMER USING AUTOMATED LAYOUT TOOLS

by

Brian C. Richards

Memorandum No. UCB/ERL M86/38

2 May 1986

DESIGN OF A VIDEO HISTOGRAMMER USING AUTOMATED LAYOUT TOOLS

by

Brian C. Richards

Memorandum No. UCB/ERL M86/38

2 May 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

DESIGN OF A VIDEO HISTOGRAMMER USING AUTOMATED LAYOUT TOOLS

by

Brian C. Richards

Memorandum No. UCB/ERL M86/38

2 May 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Table of Contents

1. Introduction.

Fully automated layout of VLSI circuits has been demonstrated as an effective method for designing custom digital signal processing circuits for a variety of audio applications. This works well for audio applications where one basic architecture can be used as a framework for many different tasks. For video rate applications, with sample rates around 5 to 10 MHz, a different architecture may be needed for each job, to take advantage of parallel processing. As a result, video rate processors are often assembled largely by hand. The purpose of this paper is to describe an attempt to design a video rate processor with minimal manual layout effort, using tools in LAGER, an automated layout system developed for audio DSP applications [5].

For audio applications, using LAGER can be compared to using an off the shelf DSP chip. Both have ROM and RAM for programs and data, and each has a pipelined ALU. To use them, a machine-language program can be written, assembled and debugged on a host system. For an off the shelf part, this program is then loaded into the chip, and the application is tested. LAGER, however, allows the user to specify characteristics of the architecture, such as the size of RAM and ROM, or whether a sub-program counter is needed. In both cases, a single ALU is used to perform all of the computations. With audio sample rates from 8 KHz to 40 KHz, and a clock rate of 2 MHz, the processor can perform from 50 to over 200 instructions per sample. Thus a single basic architecture can be used for many audio applications.

Video digital signal processing circuits cannot take advantage of sequential processing like their audio counterparts, since the clock rate often equals the sample rate. In this case, the user needs to be able to program the architecture of the system. Since this often means redesigning the chip from scratch, some may criticize that it is easier and faster to design and manually assemble the chip. However, if a sufficient hardware description is given, basic circuit modifications can be made as fast as one might change the program for an audio DSP chip. In the case of manual layout, such a change may take several days instead of a matter of minutes, and is more likely to be error prone.

This paper will begin by discussing the Video Histogrammer application in section 2, describing the desired features of the circuit. In section 3, the specialized architecture will be described, discussing how the design was chosen from a variety of alternatives. In section 4, the relationship between the heirarchy of the CAD tools and the heirarchy of the histogrammer is discussed. A variety of resulting circuits are then shown in section 5.

2. What is a Video Histogrammer?

2.1. The Histogram.

Many image processing techniques use statistical information about an image to improve the quality or contrast of an image. One of the more commonly used statistics is the histogram, which is an estimation of the pixel intensity distribution over a selected portion of an image.

Consider a discrete-time system with finite length signal $S(n)$, where $0 < n \leq N$, and the value of $S(n)$ is one of I discrete values, $S(n) \in \{0, \ldots, I-1\}$. Then, the histogram $H(i)$ of the signal $S(n)$ can be calculated as follows:

$$H(i) = \sum_{n=0}^{N-1} \begin{cases} 1 : S(n) = i \\ 0 : otherwise \end{cases} \qquad 0 \leq i < I \qquad (2.1)$$

The histogram of a signal can be thought of as an estimate of the probability density function $p(i)$ of the signal,

$$p(i) = Prob\{S(n) = i\} = \frac{1}{N} H(i) \qquad \begin{matrix} 0 \leq i < I, \\ 0 \leq n < N \end{matrix} \qquad (2.2)$$

The calculation of the histogram can be thought of as selectively incrementing one of I counters, $H(0)$ through $H(I-1)$. All of the counters are reset to zero before the frame of video data starts. When a pixel of intensity i is received, the counter $H(i)$ is incremented. When the frame has been completely scanned, the values stored in the counters represent the histogram of the signal.

A minimal implementation of of a histogram processor should be able to calculate and download the histogram of an image, once per frame, or every other frame if the image is interlaced. Also, circuitry should be included to support Histogram Equalization for image contrast enhancement, discussed in the following section.

2.2. Histogram Equalization

Histogram equalization or histogram flattening is a method for improving image contrast by changing the pixel intensities of an image to result in a roughly equal distribution of all intensity levels [1].

Ideally, this means that the probability density function, $p_{EQ}(i)$, of the equalized signal should be constant.

To generate the equalized video signal $S_{EQ}(n)$ starting from the image $S(n)$, the histogram is first evaluated. Then, the probability density function $p(i)$ is generated by scaling the histogram coefficients as described in the previous section. From $p(i)$, an approximation of the probability distribution function (PDF) $P(i)$ can be found:

$$P(i) = \sum_{j=0}^{i} p(j) \qquad 0 \le i < I \qquad (2.3)$$

Now, the function $P(i)$ is applied to each value of $S(n)$, to generate a new image, $S_{EQ}(n) = P(S(n))$. This signal, $S_{EQ}(n)$, has the desired property that its histogram is flattened, resulting in an equalized distribution of pixel intensities.

In a system with discrete intensities, the histogram of $S_{EQ}(n)$ will rarely be perfectly flat, for reasons to be mentioned later. To show how the equalization comes about, consider a continuous system, where the image $S(n)$ is represented by the random process S, with non-discrete pixel intensities. Similarly, let S_{EQ} represent the equalized signal $S_{EQ}(n)$. Let $P(i)$ and $P_{EQ}(i)$ be represented respectively by the continuous PDF's $P_{\mathrm{S}}(s)$ and $P_{\mathrm{S}_{EQ}}(s)$, which are defined below:

$$P_{\mathrm{S}}(s) = Prob\{\mathrm{S} < s\} \qquad (2.4)$$

$$P_{\mathrm{S}_{EQ}}(s) = Prob\{\mathrm{S}_{EQ} < s\} \qquad 0 \le s < 1 \qquad (2.5)$$

The range of s in the second equation is restricted since $0 \le P_{\mathrm{S}}(s) < 1$ for any real s by the definition of a PDF. As in the discrete case, the PDF $P_{\mathrm{S}}(s)$ is applied to the original signal S to generate the equalized signal, $\mathrm{S}_{EQ} = P_{\mathrm{S}}(\mathrm{S})$. Substituting for S_{EQ} in equation 2.5, the density function of the equalized curve can be expressed in terms of the original signal, S:

$$P_{\mathrm{S}}(s) = Prob\{P_{\mathrm{S}}(\mathrm{S}) < s\} \qquad 0 \le s < 1 \qquad (2.6)$$

To simplify further, assume that $P_{\mathrm{S}}(s)$ is a continuous function, with a one-to-one mapping from the real line to the real line, with a well-defined inverse, $P_{\mathrm{S}}^{-1}(.)$:

$$= Prob\{\mathrm{S} < P_{\mathrm{S}}^{-1}(s)\} \qquad 0 \le s < 1 \qquad (2.7)$$

Substituting the definition of $P_S(t)$ from equation 2.4, we get:

$$P_{S_{eq}}(s) = P_S(P_S^{-1}(s)) \qquad (2.8a)$$
$$= s \qquad 0 \le s < 1 \qquad (2.8b)$$

The probability density of the equalized signal, in equation 2.9, is constant, since $P_{S_{eq}}$ is linear. This is analogous to a perfectly flat histogram.

$$p_{S_{eq}}(s) = 1 \qquad 0 \le s < 1 \qquad (2.9)$$

In the discrete quantization case, deviations from the ideally flattened histogram are caused by discontinuous nature of the estimated PDF. In the derivation above, $P_S(.)$ was assumed to be a one-to-one function, to justify the relationship $P_S(P_S^{-1}(s)) = s$. In the discrete case, this equality will only be approximate. Two artifacts result from this non-unique mapping. When two or more adjacent intensity quantization levels are mapped to one level, some information is lost. In the opposite case, two adjacent intensities can be mapped to well-separated intensity levels. This results in contouring in the image. Contouring is visually distracting, and introduces edges into an image which can confuse attempts to find and trace the desired edges in an image. Nevertheless, histogram equalization is useful for visually enhancing the contrast of an image.

2.3. Other Applications of the Histogram.

The histogram of an image, by itself, has several applications in image processing. In an environment where lighting conditions may vary, the histogram provides lighting information for a scene. This information can be used to adjust external lighting, or it can be used to adjust the dynamic range of the camera. This can enhance the quality of other recognition schemes.

If the histogram can be taken over arbitrary regions of an image, the results can aid the process of image segmentation [2]. A segment in an image can represent a single object, or part of an object. If the histogram of a region has a well-defined isolated peak, the pixels with intensities around that peak can identify one or more segments. By repeatedly subdividing regions of an image and thresholding, the segments of an image can be identified with more detail.

A histogram processor should be able to perform all of the above tasks, without extensive external support. One objective of this project is to perform the histogram equalization function with two chips, with one chip generating the mapping function, and another performing the remapping of the data. A future possibility is to put both circuits on one chip. The histogram processor should also be designed so that signals other than video data can be processed.

3. Architecture

As the idea for the video histogramming processor took shape, several architectures were reviewed. Power consumption, die area, NMOS or CMOS technology, flexibility and reliability were among the many factors considered before the final design was chosen.

3.1. The Video Signal

Before developing the histogram processor, the nature of the video signal must be defined. The video digitizing test station includes a color camera operating with standard NTSC video timing. In one case, the digitized image consists of 512 columns by 480 rows, requiring a sample rate of 10 MHz. In this case, since the video data is interlaced, the processor must accumulate the histogram during each of the interlaced images or frames, without clearing during the vertical blank period. Alternatively, the interlace can be disabled, and a new histogram can be calculated during each frame. In this case, the number of rows is reduced to 240, while the sample rate remains 10 MHz. The test system can also digitize 256 samples per row, with a reduced sample rate of 5 MHz, in either the interlaced or non-interlaced mode. The histogram processor has access to 8 bit digitized video samples, and must synchronize to the provided horizontal and vertical blank signals.

3.2. Desired Features

Two separate tasks must be performed by the histogram processor. The primary task is the evaluation of the histogram as the video data is received. As mentioned before when the histogram was defined, this can be thought of as selectively incrementing one of several counters, according to the value of each video sample.

The second task is to process the histogram data. This postprocessing may only involve reading the histogram data into another processor. The internal registers of the histogram processor can be refreshed at this time, or they may be reset, to prepare for a subsequent histogram calculation.

For histogram equalization, the data must be accumulated and normalized, to calculate the probability distribution function P(i) discussed earlier. The result is then loaded into a look-up table. If the video signal is applied to this LUT, the resulting signal is the histogram equalized version of the original signal.

Since the LUT was developed separately, the histogram processor must be able to program the existing LUT with the calculated values.

3.3. Implementation of the Histogram Processor

There are several different architectures which can accumulate a histogram. For purposes of comparison, consider an interlaced video image with a total of 512*512 or 2^{18} pixels. Assume that eight bits of digitized video data are available, with 2^8 or 256 possible quantization levels. Also, assume that the video sample rate is 10MHz, allowing 100 ns for the processing of each pixel.

One of the most intuitive structures for calculating the histogram is an array of counters, one counter for each quantized intensity level in the digital video signal, as suggested in figure 3.1. For the image described above, each counter would require 18 bits, since the screen may have a constant intensity level in the worst case. 256 of these counters would be required, one for each level. An advantage of using such an array of counters is that the layout could be straightforward, with simple timing requirements.

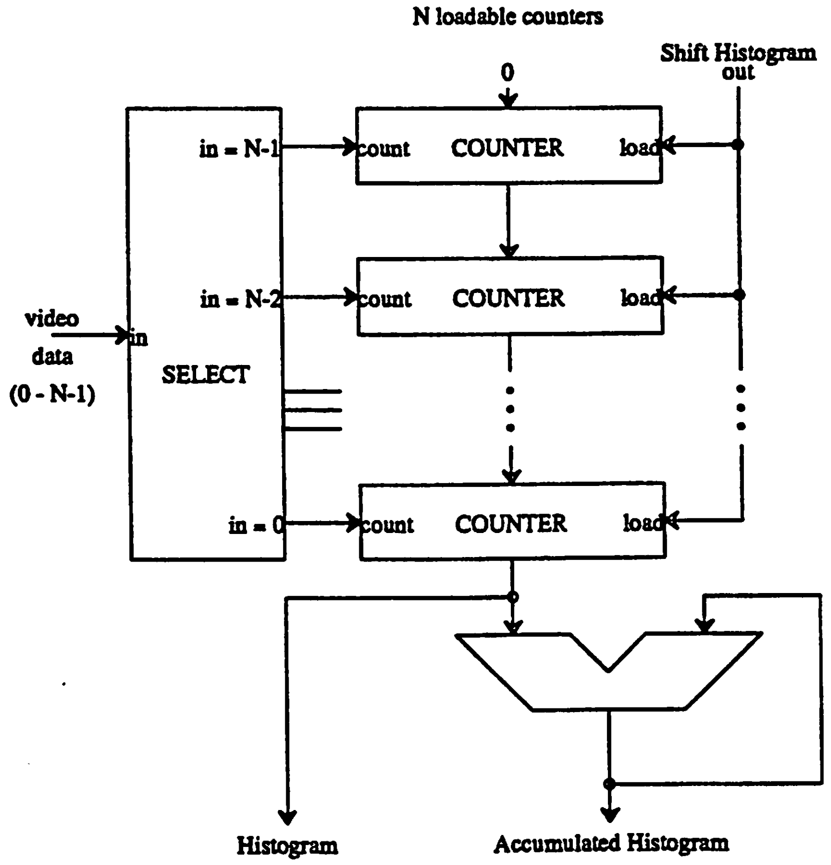


Figure 3.1: Histogramming with Several Counters

If a CMOS 3μ process is used, roughly 128 18 bit counters could be put on a 9000μx9000μ chip, including support logic and control. The CMOS technology is often preferred since it ideally has no static power consumption. At 10 MHz, however, the dynamic power consumption becomes more significant. Consider as an example a CMOS half adder cell with storage, using a 10 transistor exclusive-nor gate, an eight transistor delay cell, and six transistors for a carry logic AND gate. The total capacitance of the transistor gates is:

$$C_{total} = 256 counters \times 18 bits \times 24 gates \times 32\mu^2/gate \times 0.5 fF/\mu^2 > 1700 pF$$

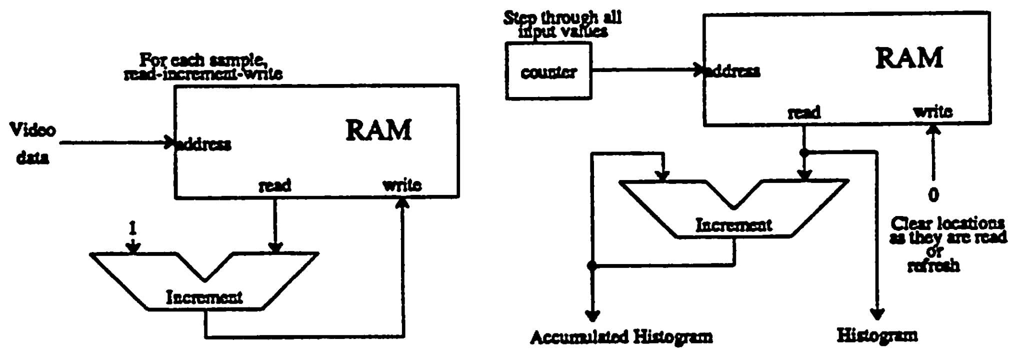
Assume in the worst case that one fourth of these transistor gates change at a time, which can occur when data is shifted out of the counter array. If a 5V power supply is used, then the dynamic power consumption with a 10 MHz clock rate will be:

$$\begin{aligned} P_{dynamic} &= \frac{C_{total}}{4} V^2 f \\ &\approx 400pF \times 25V^2 \times 10^7 Hz \\ &\approx 100mW \end{aligned}$$

If the counter array was implemented in a 3μ NMOS process, the circuit would use slightly less area, but would require substantial power. If each counter cell included 4 load devices, then the counter array would consist of $128x18x4 = 9216$ loads. For a typical $2x4$ load, which may draw $50\mu A$, the average power consumption might be $50\mu Ax5000 = 250mA$. Furthermore, two of these chips would be required for 8 bit video data. Also, over 2000 latches must be clocked. The requirement of driving a large number of gates with a reliable, non-overlapping two phase 10 MHz clock presents a problem for 3μ NMOS technology, since a fast enough buffer cannot be made efficiently. Other video rate signal processing chips [3], can effectively use an off-chip bipolar clock driver, if the number of clocked gates is not exceedingly large. If off chip clock drivers are used, the major problem is skewing, which may result in non-overlapping clocks. Hence the number of clocked devices is restricted by the complexity of layout. RC delays must be minimized in the clock traces to reduce skewing problems. Given these restrictions, the array of counters is likely to fail, due to the excessive number of clocked circuits.

Comparing 3 μ processes, NMOS would draw over 1W compared to the 200mW CMOS power consumption. The CMOS version, however, would require a larger chip area. Ultimately, other considerations, such as the availability and turn-around time of each process will become more important when choosing the technology.

A second approach to the histogram calculation is to use a RAM with an incrementer, as in figure 3.2. For each sample, the digitized video data is used as an address to a RAM bank. The accumulation at that address is incremented and written back into the bank for each video sample.


For each sample, read-increment-write
Video data
address
RAM
read
write
1
Increment
Step through all input values
counter
address
RAM
read
write
Increment
0
Clear locations as they are read or refresh
Accumulated Histogram
Histogram

Acquiring the Histogram Histogram Output

Figure 3.2: Histogram Calculation with RAM and an Incrementer.

The timing for this scheme is critical, since the circuit must run at 10 MHz. During each 100 ns sample period, the selected RAM location must be read, then incremented, and finally the result must be stored back into the RAM. For the above video signal, the processor has a total of 100 ns to read the RAM, increment the value with an 18 bit half adder, and store the data again before the next address is received. To do this with 3μ NMOS is difficult, since the half adder alone will take about 80 ns to ripple the carry signal through all 18 bits in the worst case. A look-ahead carry could be used, although the gain in speed would not be enough, since the RAM access time has not yet been considered. The counter could be split in halves or thirds, by pipelining the carry, but the RAM addressing logic would then become more complicated.

The read-increment-write approach requires fast RAMS. One way to speed up the RAM access time is to use smaller banks. In the above example, a 256 x 18 bit RAM is needed. If this is broken into four 64 x 18 bit RAMS, each one with its own half-adder, then each bank can operate faster than the original full size version. If this approach were chosen, the amount of control and glue logic would increase dramatically.

Notice that the preceding suggestions for speed improvements are compromises between the counter array and RAM-incrementer approaches. The counter array is merely a collection of 1 x 18 bit RAM and half adder pairs. The amount of RAM storage is constant, while the number of incrementers varies. Another alternative architecture involves designing two histogrammers which can operate at half the required speed, say with a 200 ns sample period. Then the two processors can alternately accept video data samples. When the frame of data has completed, the two histograms can be added, and the result is the same as with one histogrammer. This requires roughly twice as much memory, since each processor must now have a 256 x 17 bit RAM. Again, chip size limitations rule out this method, since a single chip implementation is desired.

The choice of technology is still not clear cut, since a power versus area compromise must be made. As a result, the NMOS process was chosen for this project largely for historical reasons. Some of the CAD tools, such as the datapath compiler which will be discussed later, only support NMOS libraries to date. Also, several image processing circuits have already been made successfully using the 3μ and 4μ NMOS processes. This track record, and access to cell libraries with proven cells, resulted in choosing the NMOS technology for fabricating the histogram processor.

3.4. The Chosen Architecture: A Pipelined RAM-Incrementer

In most digital signal processing systems, higher speeds can be obtained from inherently slow circuits by pipelining the system. This method also can be used in the histogrammer. Consider a RAM which can simultaneously write and read different locations within the 100 ns sample period. Also, assume that an adder is available which can settle in less than 100 ns. Then, these two blocks can be connected together, with a delay in between each one. Then, while the current value is being read, the preceding address is incremented, and the address before that one is being updated.

If the video data never contains consecutive equal samples, then this scheme will work well. When two equal samples are separated by one different sample, the RAM can read out the incremented value as it is written, and the results will remain consistent. If, however, two equal samples immediately follow one another, then the corresponding RAM location will not be properly updated with an incremented value before it is re-read for the second sample. As a result, the count will be off by one for each consecutive pair of values. If the video signal is constant, the histogrammer will have counted only one half of the pixels.

To adjust for this miscount, the processor can effectively predict when a duplication occurs, and correct for it. Then, if a duplication occurs, the second sample can be incremented by two, accounting for the unincremented value in the corresponding RAM location. Since there are two pipeline delays, the value in the RAM cannot be off by more than one, hence this correction by prediction is sufficient.

The overall block diagram of the pipelined histogram processor is shown in figure 3.3. The delay blocks are roughly time-aligned from left to right, to illustrate the signal flow. The left half of the circuit includes the control logic and address datapath, which together control the RAM banks and the incrementer on the right. Each block is discussed in more detail in the following sections.

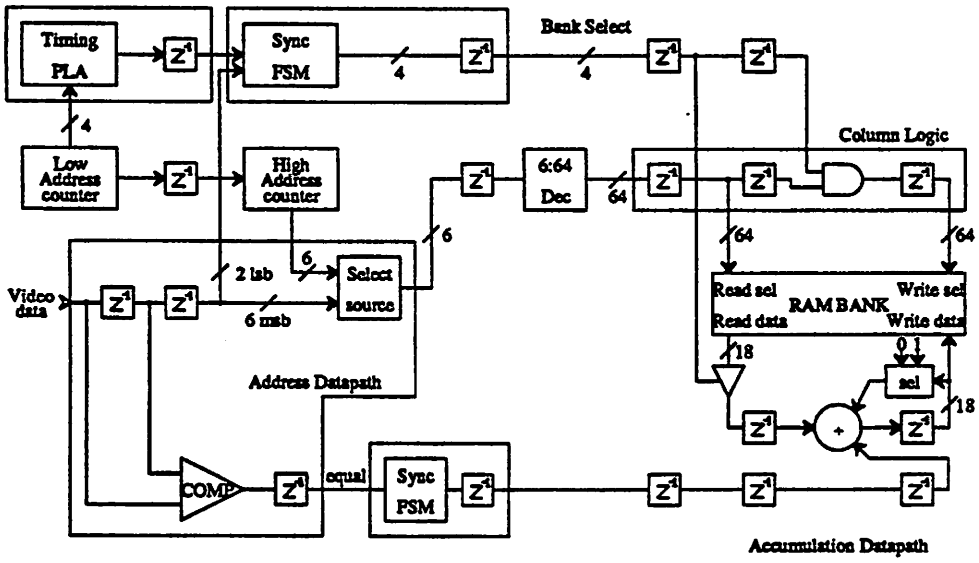


Figure 3.3: Pipeline Structure of the Histogram Processor.

3.4.1. RAM Design

Using the pipelining scheme, the processor must be able to read from one RAM location while writing to another, independently. This led to the choice of a 3 transistor RAM cell, with separate read and write lines, shown in figure 3.4.

To implement the simultaneous read and write, the RAM row drivers include several delays. The write strobe is generated two clock cycles after the corresponding read occurred, to account for the two

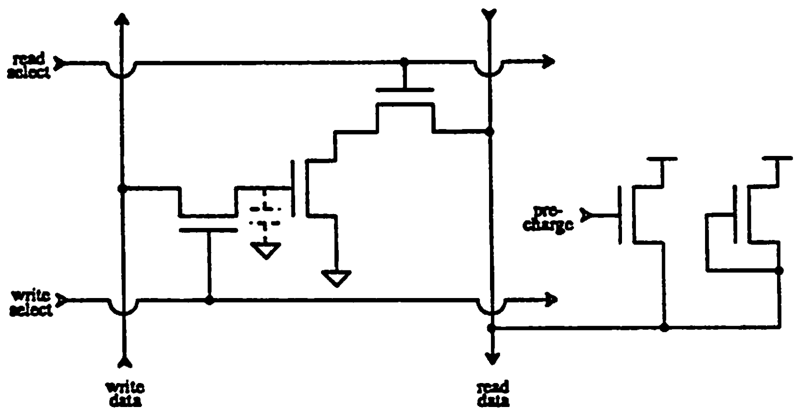


Figure 3.4: Three Transistor RAM Cell.

delays through the incrementer. The timing for a RAM row decoder is shown in figure 3.5, in the case when one address is accessed twice, with a one cycle delay between accesses.

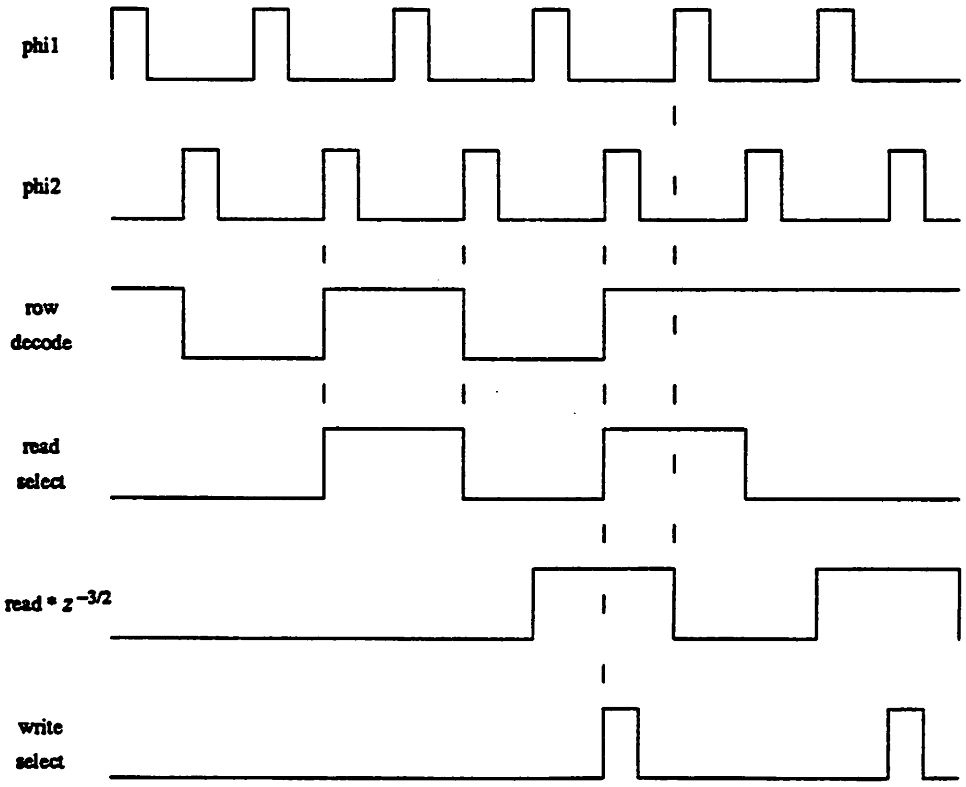


Figure 3.5: RAM Read and Write Select Timing.

Notice that the value written after the first access is available for the second read. If two consecutive accesses occur, both will read the same value, causing an error which can be corrected by the prediction scheme suggested above.

To allow simultaneous reading and writing with a 100 ns cycle, the row decode logic includes several pipeline delays. Figure 3.6 shows the transistor level schematic of the row driver. Each row driver includes a full pipeline register immediately following the decode logic, to insure a stable row select. This signal is buffered to generate the row read strobe. The unbuffered read signal is then delayed one and a half more times, and is gated with the clock phase 2, to generate the write strobe. At any one time, only one read select and one write strobe can occur on all of the RAM rows.

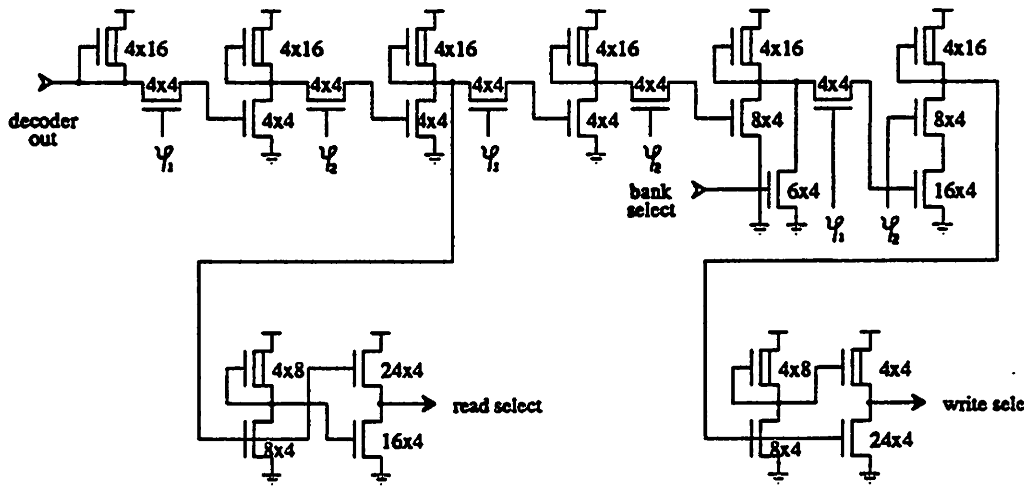


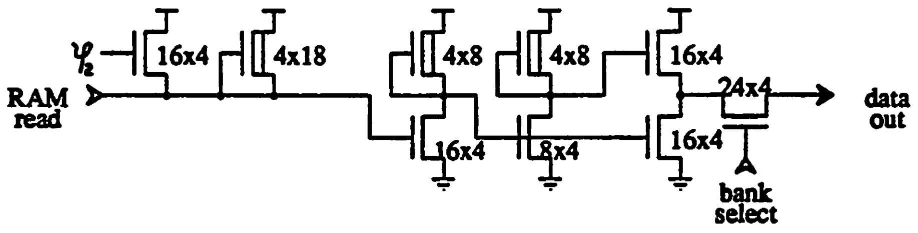
Figure 3.6: Row Driver Circuitry.

The schematic of the column driver is given in figure 3.7. The write drivers for the columns of the RAM are superbuffers, preceded by a nor gate to selectively clear a RAM location. The read driver simply uses an inverter to sense the column data, followed by a superbuffer to drive the data read bus. A pass gate is then used to select which bank is to be read.

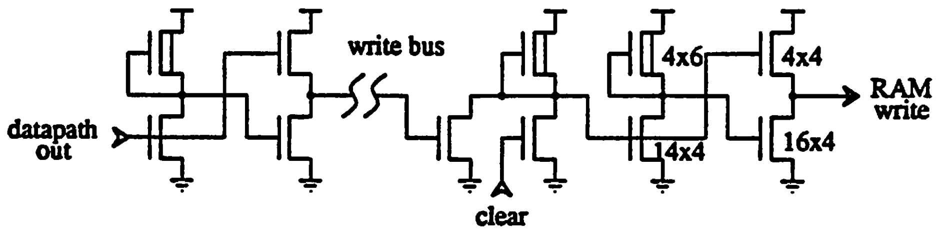
3.4.2. Incrementer Datapath Design

The simplest incrementer for histogram processing is an adder which can selectively add 1 or 2 to the value read from RAM, and store the result. To pipeline the circuit, a delay must be placed before and after the incrementer. Also, depending on the layout, buffers may be used on the input and output, since the data bus lines can get long for the larger circuits. This incrementer is sufficient for collecting the histogram, yet it is not useful as is for the post processing stage.

Post processing, which can mean reading out the histogram or integrating it, was originally



Column Read Sense-amp



Datapath Output Driver Column Write Driver

Figure 3.7: Column Driver Circuitry.

envisioned to require a separate datapath. The prediction scheme, however, requires a full adder so that increments by one or two are possible. Hence, the histogramming datapath already has most of the elements required for the post processing operation. To read out the histogram data, zero can be added to the data read from RAM. Integration of the histogram data requires feedback from the delayed output of the adder back into one of the inputs. One adder input can come from the RAM as before, and the other input can be selected from an input constant or from feedback data. To zero the adder input, a separate zero forcing circuit can be used.

The post processing operation can be much slower than the histogram calculation, allowing adequate time to load the data into a look up table, or temporary memory. Since the same datapath is used for both fast and slow operations, intermediate results for slow operations must be held until they can be used. The alternative of changing the clock speed is not desirable, since the reliability of the clocks is already a critical factor. To store the intermediate results, an additional selector is added after the adder, to maintain the current output value in the output delay register. Also, a zero may be forced into this register, to reset the output register before accumulating a histogram.

3.4.3. Post Processor Timing

When a histogram has been calculated, LUT address and data information is to be generated, with proper control signals to program an existing LUT. For the 8 bit LUT, four registers are loaded with consecutive LUT entries, and a fifth register must be loaded with the 6 most significant bits of the address where these four values will be stored in the table. A 2.4 µs cycle is required to load one register, with an 800 ns valid data strobe. To load the entire LUT, 64 groups of 5 register load cycles must be generated, entirely during the vertical blank period.

The basic 800 ns interval is generated with a programmable 3-bit counter. If the system clock is 10 MHz, the counter will divide the clock by 8; whereas if the clock is 5 MHz, it is divided by four. This is done by designing the counter to load a preset value of zero or four when the MSB generates a carry. An external signal is required, to select between the two clock rates.

One counter is used to generate five 2.4 µs clock cycles. A total of 15 of the basic 800 ns strobe periods are required to load four LUT values. A modulo 3 and modulo 5 counter could be used, but this requires two specialized counter modules. For the sake of simplicity, a 4 bit counter is used, and the first 800 ns interval is ignored. The outputs are then decoded with a PLA, to generate the register addresses and the valid data strobe. The PLA module, called the 'timingpla', is described in figure 3.8.

One more counter is required to generate the 6 most significant bits of the LUT address. A six bit binary counter is sufficient. Notice that a total of 13 bits of counter cells are required for the three counters. The 13 bits of ripple carry may not be fast enough at the 10 MHz clock rate, since there are two gate delays introduced into the carry chain by each counter cell. To introduce a margin of safety, a delay is inserted between the four bit and the six bit counter, pipelining the carry. To counteract this delay, the PLA decoded output from the previous stages is also delayed. This pipelining also serves to guarantee a reliable output from the PLA.

3.4.4. Address Datapath

When the histogram is being calculated, the RAM bank receives its address from the input stream of video data. During the post-processing stage, however, the RAM address is generated by the counters

Timing derivation PLA												
Input Plane							Output Plane					
S8	S4	S2	S1	P4	P2	P1	LOADLUT	ADDR1	ADDR0	OUTSRC	HOLD	RESET
0	0	0	0	X	X	X	1	0	0	1	.	.
0	0	0	1	X	X	X	1	0	0	1	.	.
0	0	1	0	X	X	X	0	0	0	1	.	.
0	0	1	1	X	X	X	1	0	0	1	.	.
0	1	0	0	X	X	X	1	0	1	1	.	.
0	1	0	1	X	X	X	0	0	1	1	.	.
0	1	1	0	X	X	X	1	0	1	1	.	.
0	1	1	1	X	X	X	1	1	0	1	.	.
1	0	0	0	X	X	X	0	1	0	1	.	.
1	0	0	1	X	X	X	1	1	0	1	.	.
1	0	1	0	X	X	X	1	1	1	1	.	.
1	0	1	1	X	X	X	0	1	1	1	.	.
1	1	0	0	X	X	X	1	1	1	1	.	.
1	1	0	1	X	X	X	1	1	1	0	.	.
1	1	1	0	X	X	X	0	1	1	0	.	.
1	1	1	1	X	X	X	1	1	1	0	.	.
0	0	0	1	1	0	0	.	.	.	.	0	1
0	1	0	0	1	0	0	.	.	.	.	0	1
0	1	1	1	1	0	0	.	.	.	.	0	1
1	0	1	0	1	0	0	.	.	.	.	0	1
0	0	0	0	1	1	1	.	.	.	.	1	0
Other Combinations							.	.	.	.	1	1

Figure 3.8: Timing derivation PLA.

described above, to step through the memory sequentially. A simple multiplexer is needed to select the address source, which can be generated as a bit sliced datapath.

Recall that the pipelined incrementer datapath requires a predictor for successively equal address values. The previous sample is made available by introducing a pipeline delay in the address datapath, before the select logic. A comparator, consisting of a collection of exclusive-NOR gates driving an OR gate, can be included in the datapath to determine when duplication occurs. The comparison output is also followed by a pipeline delay, before it is forwarded to the control logic.

3.4.5. Synchronization and Control Finite State Machine

Control logic must be provided to synchronize the two datapaths, 3 counters and at least two RAM banks with the external video system. Several control signals will be made available to the circuit. The Vertical Blank (VBLANK) signal is used to determine whether the circuit is calculating the histogram, or postprocessing the result. The Horizontal Blank (HBLANK) input indicates when valid data is on the video input bus. The Field signal, usually generated for interlaced images to indicate whether the current data is in the even or odd field, is used to select whether the RAM is cleared or refreshed during a vertical blank period. Also during post-processing, the histogram can be read out, or incremented, according to the READ signal.

For each line, the first valid sample of video data is available when HBLANK goes false. At this time, the incrementer must be prevented from incrementing by two, since an invalid duplication might occur. The duplication signal, EQUAL, from the address datapath can be gated with the delayed HBLANK signal, HBLANK1, to avoid errors. Also, since the carry input to the incrementer's adder is used to select between an increment by one or two, this signal must be zeroed during the post-processing stage, by gating EQUAL with the VBLANK signal. The resulting signal, EQUAL1, must then be delayed by the number of pipeline stages in the RAM read and increment circuitry. Hence, the adder will increment by two only when the current value read from RAM is from the same address as the previous value. Figure 3.9 shows the part of the 'fsmsync' PLA which synchronizes with the input data.

Histogram ALU Controller									
Input Plane						Output Plane			
READ	~HOLD	VBLANK	HBLANK	H1	EQUAL	H1'	INC1OR2	~HOLDM1	READM1
X	X	0	1	1	1	1	1	.	.
X	X	0	1	0	X	1	0	.	.
X	X	0	0	X	X	0	0	.	.
X	X	1	X	X	X	0	0	.	.
X	X	0	1	1	0	1	0	.	.
X	0	X	X	X	X	.	.	1	.
X	1	0	X	X	X	.	.	1	.
X	1	1	X	X	X	.	.	0	.
X	X	0	X	X	X	.	.	.	1
0	X	1	X	X	X	.	.	.	0
1	X	1	X	X	X	.	.	.	1

Figure 3.10

The RAM bank select logic which is included with the synchronization logic is in figure 3.11. When the histogram is calculated, the least significant bits of the video input data are used to select the bank. During the vertical blank, the address is available from the PLA which decodes the counter chain. Thus, the control logic can both select the address source and decode the address. Since the RAM bank was designed such that a write always follows a read, the bank select must be gated with the HBLANK1 signal to allow RAM access when video data is read. Also, when reading the data during post processing, each location can be read only once if the memory is being cleared. The counter PLA provides a signal, HOLD*, to indicate when data should be stored in the incrementer, or read from RAM. When HOLD* is true, the addressed RAM location should be read. By additionally gating the RAM bank select with the HOLD* signal, the select logic is complete.

RAM Bank Select Logic									
Input Plane					Output Plane				
VBLANK	H1	FIELD	ADDR0	ADDR1	¯SEL0	¯SEL1	¯SEL2	¯SEL3	CLEAR
X	1	X	0	0	0	1	1	1	.
X	1	X	0	1	1	0	1	1	.
X	1	X	1	0	1	1	0	1	.
X	1	X	1	1	1	1	1	0	.
X	0	0	0	0	0	1	1	1	.
X	0	0	0	1	1	0	1	1	.
X	0	0	1	0	1	1	0	1	.
X	0	0	1	1	1	1	1	0	.
0	X	1	X	X	.	.	.	.	0
1	X	1	X	X	.	.	.	.	1
X	X	0	X	X	.	.	.	.	0

Figure 3.11

The HOLD* signal also controls the incrementer datapath. When HOLD* is false, the output of the pipeline register following the adder shall be fed back into the register's input, to store the current value. To make sure that this happens only during the vertical blank period, the HOLD* signal is gated with VBLANK. The resulting control signal, HOLDM1*, is delayed to synchronize with the pipeline delays, before it is applied to the incrementer datapath.

Just as the HOLD* signal was gated with VBLANK, so must the FIELD and READ signals. During the vertical blank period, the CLEAR signal is applied to the RAM banks if the input FIELD is true. Likewise, the READM1 signal programs the incrementer datapath to read out the RAM data, or accumulate it, depending on the value of the READ signal. When VBLANK is false, and the image is being scanned, the RAM should not be cleared, and the datapath must not try to accumulate the values. These signals are constant during the vertical blank period, hence no delays need to be added to correct for the pipelining in the system.

The remaining control task is to start and stop the counters which generate the post processor timing and addresses. A four-state finite state machine is used to control this process. Figure 3.12 shows the PLA implementation of this FSM, where S1 and COUNT are the state variables. When the vertical blank first becomes true, the carry input to the counters, COUNT, must be set false, to commence the count. The counting will continue until a carry out from the counter, STOP*, is received. The count cannot commence again until VBLANK toggles false then true again. Along with the counter control, two reset signals are generated by the state machine. The incrementing datapath has an RS flipflop which is set when the adder overflows, to effectively saturate the accumulation. At the beginning of each post-processing cycle, this flipflop must be reset, so that the accumulation can start over. Also, the pipeline register following the adder must be reset to zero, so that accumulations will be initialized properly.

Postprocessing Timing Control								
Input Plane					Output Plane			
S1	COUNT	VBLANK	STOP*	RESET	S1'	COUNT'	RESETC	RSTACC
0	1	0	X	X	0	1	0	1
0	1	1	X	X	0	0	0	1
0	0	1	1	0	1	0	1	0
0	0	1	1	1	0	0	0	1
X	0	1	0	X	1	1	0	1
X	0	0	X	X	1	1	0	1
1	0	1	1	X	1	0	0	1
1	1	1	X	X	1	1	0	1
1	1	0	X	X	0	1	0	1

Figure 3.12

4. CAD Hierarchy

The architecture selected for the implementation of the histogram processor can be broken down to 12 macrocells which must be interconnected. Three sets of pads are needed, for data input, data output and control inputs. The overall timing and control can be generated cleanly with two finite state machine macrocells and two counter macrocells. For the 8 bit video system, two RAM macrocells are needed, each with two banks. One datapath is needed for the incrementing and accumulation logic, and another is used for selecting and monitoring the addresses. A macrocell containing delays of various lengths is also needed, to synchronize the pipeline stages.

Three methods are available for generating macrocells. A module generator, Modgen, can be used to tile several manually designed leaf cells to form a complete macrocell. In this case, no routing is done within the macrocell, requiring that all interconnections must be within the leaf cells. A bit-sliced datapath compiler, Compile, is also available, allowing flexible routing within a slice. Leaf cells can be chosen from a library, and assembled in arbitrary order. The third method is to use the LAGER floorplanning tool, Flint, to assemble a macrocell out of other macrocells. Flint can be used to group macrocells, whereas Modgen and Compile are used to generate the basic macrocells.

To pull the module generator CAD tools together, the 'Connect' program was developed. Connect supports the system level description for a full chip. This description groups macrocells together, and provides a high level net-list style interface to the LAGER system.

4.1. Modgen

Most of the basic circuits of the histogram processor can be generated using the tiling module generator, Modgen. To completely define a macrocell or module, the designer must describe all applicable leaf cells, and a C program must be written to describe how the leaf cells are to be tiled. Modgen reads a parameter description language (PDL) file, specifying such features as the size of a RAM, or the number of bits in a counter. Modgen then calls the C routine for the requested macrocell, which uses the PDL information to control the tiling of the macrocell. This procedure is defined in more detail in the Modgen documentation.

4.1.1. The Delay Macrocell

The delay module generator will generate a selectable number of delays, each with an arbitrary length. To help with the understanding of Modgen, the PDL file for the delay block used in the histogram processor is given below:

```
(module (name delay1) (type delay) (length 3) (width 4)
      (design (array
            DDDD
            xDxx
            DDxx
      ))
)
(term (name in[0]) (net 71) (cable pr|control|fsmsync) )
(term . . .
```

The PDL file begins with a description of the desired macrocell. In this case, four delays are desired, z^{-2}, z^{-3}, z^{-1}, and z^{-1}, in order. The parameter "name" specifies the name of the layout file to be generated. The "type" selects which C routine is called to generate this macrocell. Several macrocells of a given type can be generated, as long as the names are unique. For this example, the maximum delay length is z^{-3}, hence the "length" is set to 3. The "width" specifies the number of delay chains requested. The array "design" consists of symbols to define the macrocell floorplan, where each column represents a delay chain. A z^{-1} delay is generated for each symbol 'D' in the array.

The remainder of the PDL file defines how the terminals on the macrocell boundary connect to other macrocells. The "name" refers to the terminal name defined in the leafcell descriptor file for the requested module type. When a leafcell with a terminal is used more than once, each reference to that terminal is given an index by Modgen, thereby permitting flexible bus widths. The "net" number is an integer used throughout the design to refer to a specific signal. The "cable" gives the name of another macrocell which is attached to this net. The terminal parameters depend on the overall system design, and are generated automatically by the Connect program, which is discussed later.

4.1.2. The Counter Macrocells

Two types of counters can be generated by the "counter" module generator. The first is a presettable counter, which loads a given value when the carry from the most significant bit overflows. This type of counter is chosen by specifying the parameter "(loadable)" in the PDL file, and defining the load value with "value". The value is a string of 0's, 1's or x's, to specify a constant or variable input. The three bit counter in the control logic, called "counter1", is loadable, with the value "x00", which means that it can be selectively loaded with the value "000" or "100", to select modulo 4 or modulo 8 counting. The terminal "load[0]" defines source of the most significant bit.

The four-bit and six-bit counters in the control logic are grouped together, including the delayed carry. These counters are not presettable. The PDL description is given below:

```
(module
        (name counter2)
        (type counter)
        (notloadable)
        (design
                (count 4)
                (delay)
                (count 6)
        )
)
```

If two separate counters and a delay block were used, space would be wasted for redundant power, ground and clock connections, as well as the space required for routing the carry signals.

4.1.3. Finite State Machine Generator

The module type "fsmpr" is used to generate finite state machines or PLA's. The PDL description defines the number of input signals, minterms, and clocked or unclocked outputs. The input and output planes are defined with arrays of 1's, 0's and x's. Also, if the "minimize" parameter is specified, the program *espresso* will be used to optimize the PLA.

Two PLA's are generated for the histogram processor by using "fsmpr". A PLA named "timingpla" is defined to decode the four-bit counter to generate post-processing timing signals. All other synchronization and decoding is done in a second PLA, including the RAM bank select logic, incrementer control, and post-processing synchronization. These could have been included into one PLA, however, its area would be roughly double that of the two put together, since few signals are common to both.

4.1.4. The RAM Bank

The module generator "ramhist" generates a RAM with two banks mirrored around the row select logic. The amount of storage and the size of the data word are parameters. The histogram processor uses two of these modules, "ram1" and "ram2", each with 64 pairs of 18 bit words, to store a total of 256 words.

4.1.5. Pad Generator

A variety of pads can be generated by specifying the module type "padclk". The parameter "(input #)" will specify a number of clocked input pads, and "(output #)" will generate output pads. These pads contain one half of a delay element; the input pads are clocked on phase 2, whereas the output pads are clocked on phase 1. The designer can also specify "(power)", "(ground)", or "(clock)", to generate a power, ground, or a two phase clock pair macrocell, respectively. Only one type can be specified at a time.

The histogram processor has three groups of pads, other than the power, ground and clock pads. The "inpads" accept the video data. The "outpads" program the look up table. The "ctrlpads" provide the synchronization and mode control signals, as well as the output scaling factor for normalization.

4.2. The Bit-Sliced Datapath Compiler

To generate bit-sliced datapaths using a variety of functional blocks, the program "compile" written by Peter Reutz, has been modified to run in the LAGER environment. A text description of the desired datapath block diagram is prepared manually, describing the interconnection of an individual slice. The compile program then follows this description to place and route the slice. The leafcells are placed in the order of their specification, and they are interconnected with a channel router.

Several features of a given datapath must be defined to completely describe it. Signals entering and leaving the datapath are assigned to terminals on the macrocell boundary. The organization is then specified, to direct the placement of adjacent slices. Finally, the placement and interconnection of the individual slices is described, with one description for all types of slices.

The datapath description begins with the placement of external terminals on the macrocell bounding box. By convention, the data flow in a datapath is from left to right, and the least significant bit slice is on the bottom. Five terminal lists are given: CONTROL, RIGHT, LEFT, TOP and BOTTOM. The control signals, usually Vdd, ground and clocks, are listed under the CONTROL heading. Data signals must be in the LEFT or RIGHT lists, and one terminal definition refers to all slices. Signals common to all slices, such as carrys and selects, are on the TOP or BOTTOM of the macrocell. Each terminal has an associated net number and other interconnection information, both of which are generated from the global interconnection description to be discussed later.

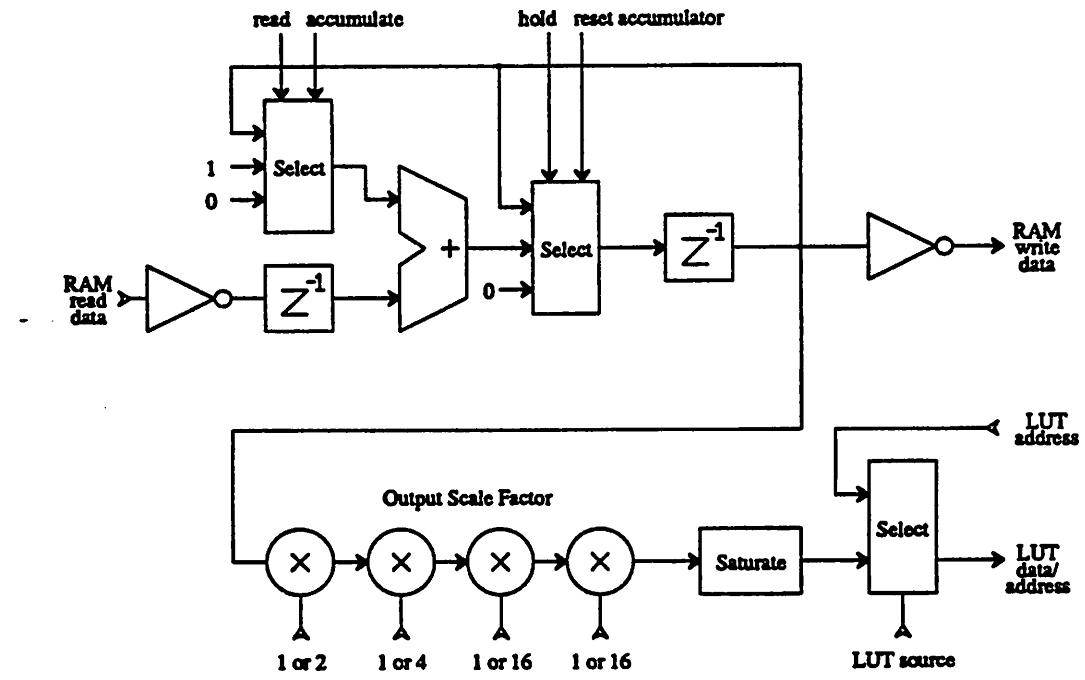
The placement of the slices is described under the heading "ORGANIZATION". The slice names follow, listing the slices as they are stacked from bottom to top. Slices can have arbitrary names, and there are two types of slices. NODATA slices provide the control and power signals, and DATA slices contain the desired functional blocks. In the current library of functional blocks, several naming conventions are used. Power is provided in the CNT slice, and ground and clocks are in the GND slice, both of which are NODATA slices. The slice CELL is used when most or all functional slices are identical, or if the requested slice is undefined. EVEN and ODD slices can be used if adjacent cells in a functional block have different designs, such as in an adder with a fast ripple carry. Also, the MSB and LSB slices can be used if the corresponding slices have special designs.

The final section, "SLICE", defines the interconnection and placement of each slice. A list of quoted net names and unquoted block names delimited by '>' describes the placement of the blocks. Each block has a generic input and output, which are by default connected in sequence. If a net is given instead of a block, the preceding output or following input is attached to that net. Other connections can be made to a block by following the name with a list of terminal-net pairs, surrounded by parentheses. The block terminals are defined in the library, and a net name can have several terminals associated with it. The name of a terminal defined on the boundary of a macrocell is also its net name. This description is used to assemble all DATA slices, and the NODATA slices are expanded to abut with the DATA slices.

4.2.1. The Histogram Incrementer Datapath

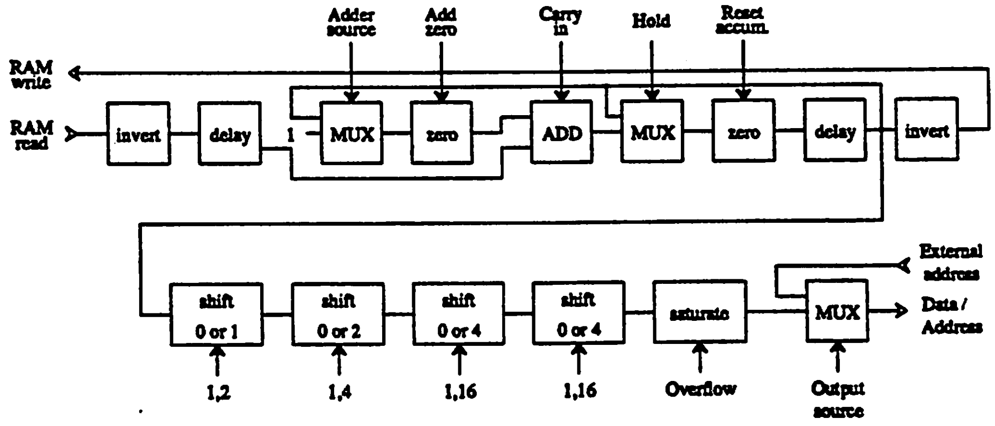
The datapath compiler is well suited to generate the incrementer datapath. If Modgen were used for this application, special cells would have to be designed for nearly every block to route the feedback signals within the leaf cells. With the datapath compiler, features can be easily modified and added to the incrementer.

To implement the incrementer datapath, it must be organized into a one-dimensional collection of functional blocks. For reference, the schematic of the overall datapath is repeated below:



Histogram Main Datapath

The datapath can then be reorganized into the form of a one-dimensional slice which can be duplicated for each bit:



Histogram Datapath Slice Organization

The datapath consists of two sections. The first section includes the increment and accumulate logic with the associated two pipeline delays. This circuitry runs at the 10 MHz clock rate.

The data read from the RAM is delayed, and routed to one input of the adder. To generate the second adder input, a selector was designed to choose between an input value or the constant 1. A zero block, consisting of a pull-down transistor, follows the selector to selectively prevent accumulation of the RAM data. The full adder requires different even and odd slices so that each bit of the adder adds only one gate delay to the ripple carry. With the 18 bit example mentioned earlier, the ripple carry becomes the critical path between the pipeline stages.

When the adder overflows, this condition is stored to force a saturated output. The synchronization FSM could have been used to keep track of the overflow, however, more delays would have been introduced into the system. As a result, an RS flip flop is used in the control slice to store this condition. The flip flop must be reset at the start of both the histogram calculation and the post-processing cycle, to prevent false overflow indications from the preceding cycle.

Following the adder, a multiplexer selects the source of the second pipeline register. The input comes from the adder during the histogram computation and accumulation. To temporarily store an accumulation, the register output can be fed back into the input. At the beginning of the accumulation post processing cycle, this register must be reset to zero, hence another zero block is placed between the multiplexer and the delay. The result is then driven by an inverting super-buffer to write to the RAM.

The second half of the datapath scales and saturates the datapath output, and selects whether this data or the address is sent to the output. This part of the circuit is asynchronous and slower than the first half, and provides meaningful data only during the slower post processing cycle.

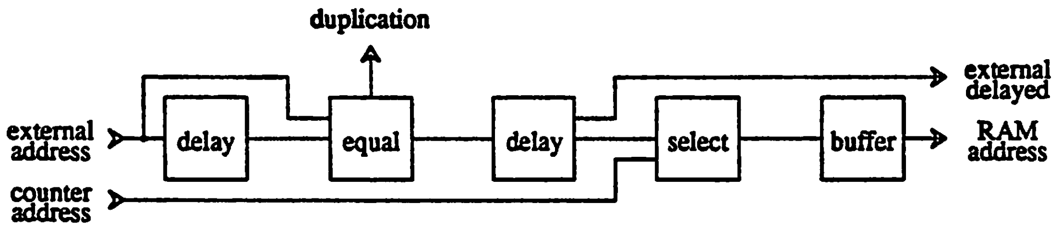
Four shifter stages are used to scale the datapath output. The first stage shifts left zero or 1 bit, the second shifts by zero or 2, and the last two shift left zero or 4 bits. By selecting appropriate shifters, the accumulation result can be shifted left from zero to 11 positions, corresponding to output scaling by powers of two from $2^0 = 1$ through 2^{11}. The control slice of each shifter stage will sense if the shifted value overflows, allowing the output to be clipped later.

The shifters are followed by a saturation register. If the adder or the shifters overflow, the output will be forced to all ones, corresponding to the maximum unsigned value ($2^{18}-1$ in the 18 bit case). Without the saturation, meaningless results would result if the adder or shifters overflowed.

The last stage in the datapath is the selection of the output data. The look up table is loaded with four accumulation values, followed by the most significant six bits of the address for those values. To simplify routing, this selection is included within the incrementer datapath.

4.2.2. The Address Datapath

The datapath compiler is well suited to generate the address datapath. All of the data slices in this macrocell are identical. Although much simpler than the incrementer datapath, the routing within the slice makes it unsuitable as a Modgen macrocell. Below is the slice interconnection:



Address Datapath Organization

After the address is delayed, the current and previous addresses are compared, to recognize successively equal video samples. The result is delayed again to adjust for pipeline delays. A selector can then choose to send this address through a buffer to the RAM banks. During post processing, the address is chosen from the timing counter chain. The delayed address data is also routed to the synchronization logic, where the RAM bank selection is decoded.

4.3. Interconnection and Hierarchy Specification: Connect

Now that all of the macrocells have been defined, the remaining layout task is to assemble the blocks. Connections between macrocells must be completely defined. Once the connections are specified, the macrocells must be positioned in a floorplan to indicate the relative placement, after which the actual connections can be made.

The program Connect is used to define the interconnection of the macrocells. The connections to a given macrocell are called terminals, which are located on the cell's boundary. Terminal names generated by Modgen are defined in the individual leaf cells which compose the macrocell, and are fixed for a given type of cell. The datapath compiler allows the designer to specify unique terminal names on the boundary of a macrocell. Several macrocells can share the same terminal names.

To connect several terminals together, they are associated with a common net name. A given terminal or net can be one conductor or a bus of any width. Since terminals and nets can have arbitrary width, they can be parameterized, so that one terminal-net list is sufficient to describe a variety of designs with different sized data and address busses.

If a chip is made up of 10 or more macrocells, the floorplanning effort can become difficult, since all channels and routing must be redefined when the placement of a cell is changed. By grouping macrocells together to form another larger macrocell, hierarchy can be introduced into the floorplanning effort. In this case, a small block can be reorganized without affecting the higher level placement.

The histogram processor can be neatly broken up into groups. The timing and control logic, consisting of two PLAs and two counters, is grouped into the macrocell 'control'. This new macrocell is then included in a larger macrocell containing the RAMs, the datapaths, and the delay block, to form the processor (pr) macrocell. Finally, the full chip consists of the pads surrounding the processor. Each group in the hierarchy can also be debugged separately.

The description of the histogram processor begins by defining parameters. Two basic parameters are needed to specify the processor. The parameter 'width', referring to the number of bits of video data, determines the width of the address datapath and busses, as well as the total number of RAM rows. The parameter 'size' indicates the log base 2 of the number of pixels in an image. This sets the width of the incrementer datapath, and the RAM word size.

After the parameters are defined, the actual hierarchy of the processor is described in the 'design' block. Macrocells can contain cable references if they are generated by Modgen or Compile, or they can list other macrocells for assembly by Flint. The description of a 'narrow' processor is listed below, with cable references and net lists removed:

```
design (narrow) {
    macrocell(pr) {
        macrocell(histdata, datapath) { }
        macrocell(ram1) { }
        macrocell(ram2) { }
        macrocell(addrdata, datapath) { }
        macrocell(delay1) { }
        macrocell(control) {
            macrocell(fsmsync) { }
            macrocell(timingpla) { }
            macrocell(counter1) { }
            macrocell(counter2) { }
        }
    }
    macrocell(inpads) { }
    macrocell(outpads) { }
    macrocell(ctrlpads) { }
    macrocell(power5) { }
    macrocell(ground) { }
    macrocell(clocks) { }
}
```

Now that the hierarchy has been established, the interconnections must be given. All macrocells which do not contain other macrocells will have a list of cables, except for power and clock macrocells. A cable, which refers to a bundle of wires running between two macrocells, is designated by the name of the destination macrocell. The definition of the connections for the "outpads" illustrates this:

```
macrocell(outpads) {
    cable(histdata) {
        in[0,7]=outdata[0,7];
        in[9]=saturated;
    }
    cable(delay1)
        in[8]=loadm1;
}
```

The output pads connect to an eight bit data bus and a status signal from the histogram datapath, "histdata". Also, a control signal comes from the "delay1" macrocell.

The complete description of the system is given in the appendices, listing how all of the signals come together. For more details about the Connect program, refer to the Connect User's Manual in the appendices.

At this point in the design, the electrical connections of the histogram processor have been completely specified, with no explicit information regarding the silicon layout of the processor. Neither the design rules nor the technology have been formally enforced in the Modgen, Compile or Connect input description files, allowing for future changes.

After the Connect program has been run with a given set of parameters, the libraries of leaf cells for the two module generators must be selected. Modgen expects to find a .lib file in the directory from which it is run, or in the user's home directory, with a complete path name to a directory containing the subdirectory 'descript.Modgen'. This directory contains 'descriptor files', which provide all of the information required by Modgen to assemble leaf cells and correctly identify terminals on the macrocell boundary. A different technology can be chosen by naming a functionally identical set of descriptors in the .lib file.

The datapath compiler, Compile, is not quite as flexible, since it was designed for NMOS macrocell generation. The restriction is most evident in that routing can be done in diffusion, between leaf cells. If this layer were ignored, then CMOS datapaths could be assembled given a functionally equivalent library of leaf cells. Minor modifications might be required in the datapath compiler so that it would recognize the different metal and polysilicon layer names. All of the leaf cells are defined in one file, 'celldesc', which is generated automatically by reading the leaf cell layout files with the 'parse' utility. An extensive library of tested NMOS macrocells has grown over the last few years for this datapath compiler, providing much of the incentive for using NMOS rather than CMOS for the histogram processor design.

4.4. Floorplanning and Routing with Flint

Once the macrocells have been generated, the designer must provide a floorplan for the final chip layout. The floorplan is symbolically specified with a floorplan description language (FDL), and a given design can have several FDL layouts. The first purpose of the FDL is to indicate the relative placement of the macrocells. Also, the designer includes the routing strategy in the FDL file.

To support flexibility, the FDL uses minimal information to indicate the relative placement of the macrocells. Each side of a macrocell can be connected to one of several features. If nothing is attached to a side, it is called a 'NULL' side. A single channel can be adjacent to the side of a macrocell, permitting signal routing from that side. Also, a collection of macrocells and channels can be grouped into a rectangular module, introducing hierarchy into the floorplan.

Once the macrocells and channels are in place, the routing strategy must be planned. The routing is controlled by referring to the cables specified in the Connect input description. Since a cable is associated with a pair of macrocells, there is no need to refer to explicit terminal locations and interconnection nets in the floorplan description. For each side of a channel, all cables passing through that side are labeled with a list of macrocell pairs. Power, ground and clocks are routed in a similar manner, although there is no cable in this case. The details of the FDL format are described in the FDL documentation.

The interactive floorplanning tool, Flint, both generates and processes the FDL files. After running the Connect program and generating the macrocells, Flint is used to generate the FDL files for a chip. Once a floorplan has been designed, the same floorplan can often be used for different versions of the same chip. This flexibility means fast turn-around when changing chip size parameters.

The first step for designing a new floorplan in Flint is to place the macrocells relative to one another. Each macrocell can be moved, rotated and mirrored, in attempt to reduce routing complexity. The cables are drawn as lines between macrocells to give a visual estimate of the routing density for a given placement.

Once the macrocells are in place, Flint is switched to channel mode, at which time the routing channels are placed to fill the space between macrocells. Each macrocell side which has terminals must come in contact with a channel, and the channels must abut to provide a complete path for all routing. Flint automatically expands each rectangular channel to the sides of adjacent macrocells.

With the channels in place, the user must then route the cables between macrocells. A cable is interactively selected by pointing to the sides of two macrocells. Then the designer points sequentially to the abutting channels which are to carry that cable. This is repeated for all cables, after which the signal routing is completed. At this stage, the designer can choose to route the signals, and view the result. If desired, the first steps can be repeated to try other floorplans.

When the signal routing is satisfactory, Flint is switched to Power mode, allowing interactive placement of power, ground and clocks. This has no direct effect on the signal routing. With the histogram chip, the circuit was simulated initially without power and clock connections, until the final signal routing strategy was settled upon.

When the full-scale histogram processor was made, the floorplan for the smaller version was reused. To use an existing floorplan within Flint, the Library mode can be selected, and the desired floorplan can be chosen. The new layout is then generated by issuing the Route command, which uses the FDL information along with the database set up by the Connect program. Within seconds, the new layout is ready for testing, and fabrication.

4.4.1. The Floorplan for the Histogram Processor

The floorplan for the histogrammer, which can be used for several parameterized versions, has been chosen to reduce the amount of unused silicon area. To simplify the design effort, the floorplanning was broken into three stages, where each stage required the assembly of a group of macrocells into another macrocell by using Flint. Earlier, the Connect input file described which macrocells were to be grouped together, leaving the actual placement up to Flint.

The first element which is assembled is the 'control' macrocell. This macrocell contains the two counters and two programmable logic arrays (PLA's). The control logic is assembled separately since there is a great deal of feedback and routing within this group. This block is then treated as a black box which generates the timing and control signals for the rest of the system. The symbolic floorplan shown in figure 4.1 places the counters and PLAs around a central routing channel, which contains most of the interconnections.

After the control macrocell is generated, it is included in the main processor macrocell, 'pr', shown in figure 4.2. This macrocell is divided in half by the channel connecting the two RAM banks to the histogram datapath. For the full size histogram processor with an 18 bit datapath, this channel carries 36 bus lines for RAM read and write data, plus the address lines and several control wires. The RAM banks

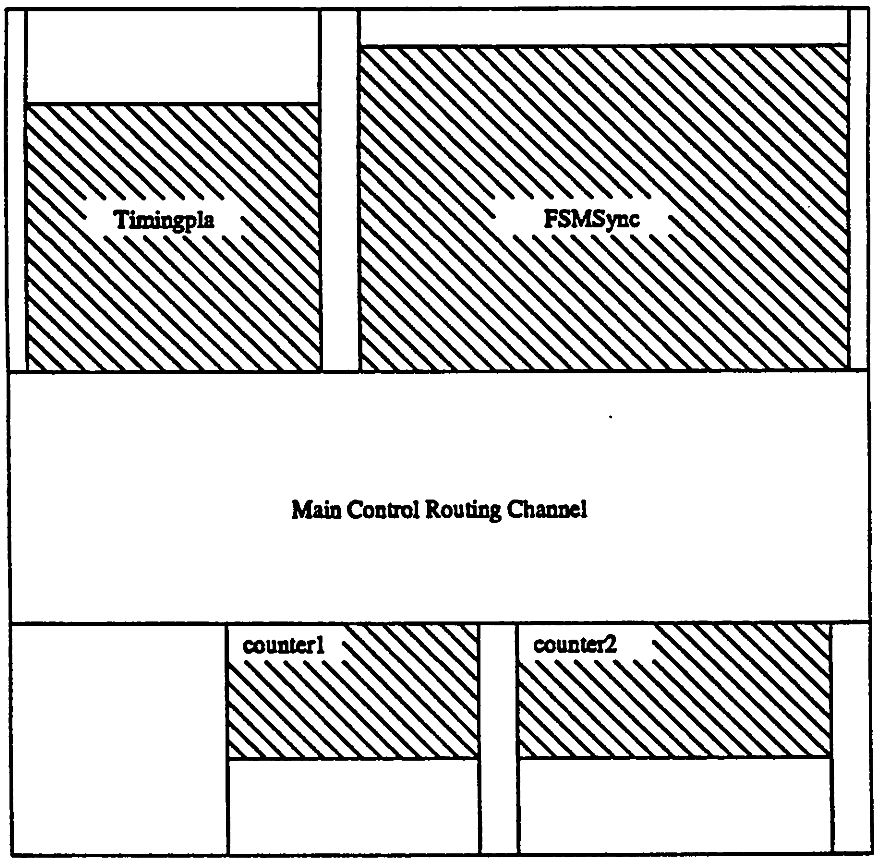


Figure 4.1: The floorplan of the control block, 'control'.

occupy roughly one-half of the chip area. On the other side of the main channel, the histogram incrementer datapath is accompanied by the address datapath, the control macrocell, and extraneous delay circuits.

The highest level macrocell, 'histchip', contains all other macrocells, and has no external terminals, since it contains all of the pads. The main processor macrocell 'pr' is surrounded by the pads, as seen in figure 4.3. The power connections are here, and the pads are routed to 'pr' through channels surrounding the main processor.

The histogram processor is unusual compared to the audio DSP chips generated by the LAGER system, in that more than two levels of calls are made to Flint for floorplanning macrocells. Most DSP chips have a processor which contains macrocells from Modgen, and one or more of these processors are assembled by Flint into a complete chip. The histogram processor has an additional level, in that the control logic is also assembled by Flint.

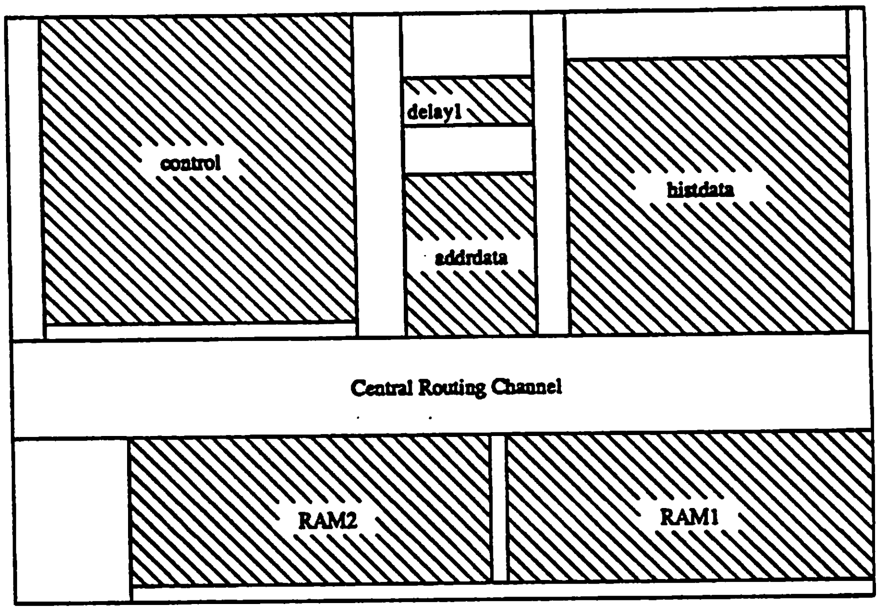


Figure 4.2: The floorplan of the processor main block, 'pr'.

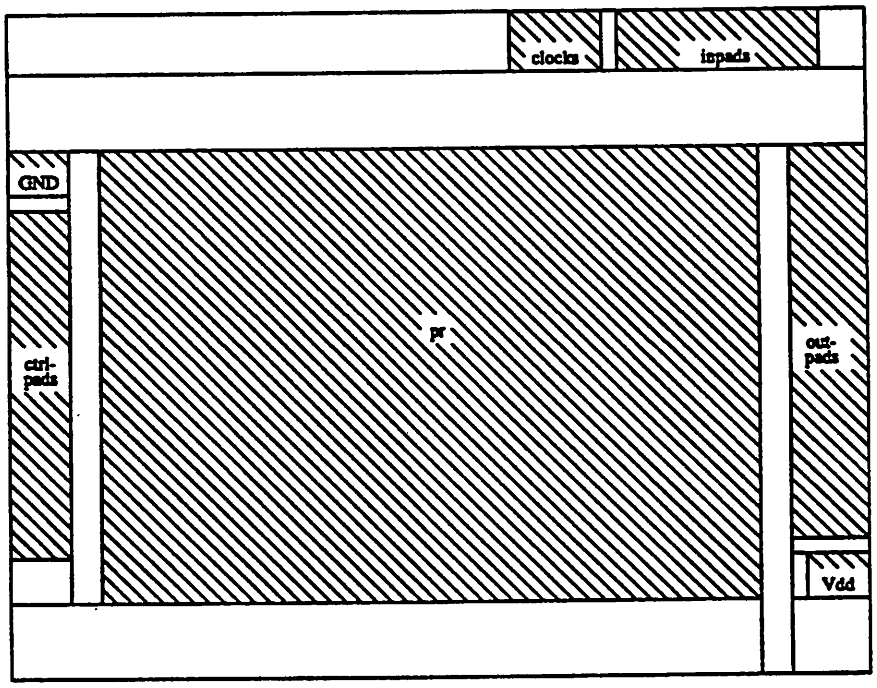


Figure 4.3: The floorplan of the complete histogram chip.

5. Resulting Circuits and Simulations

Two versions of the histogram processor were generated by giving different parameters to the Connect program. To simplify and speed up the testing and debugging procedure, a small histogrammer was generated. The video data bus is only four bits wide, and a ten bit incrementer datapath accumulates up to a 1024 pixel field. Although this is insufficient for practical video signals, it serves well for testing purposes.

Once the smaller processor was debugged, the full size version was generated. The steps for generating the larger chip after the smaller one was designed are straight forward, requiring minimal editing. The Connect input file for the smaller chip, called 'narrow.ch', is edited, and the parameters *size* and *width* are changed to *size* = 8 and *width* = 18. The design name is changed to *design* (*histogram*) to save the old design, and the new file *histogram.ch* is saved. Then the Connect program is run from the directory containing the template files and the *histogram.ch* file:

```
Connect histogram.ch
```

The Connect program has generated the input description files for the module generators. The next step is to run Modgen and Compile on the appropriate files, to generate the hardware description language (HDL) files and the layout files. This is currently done manually using a C Shell script, although this task will eventually be done by Flint.

Once the HDL files are generated, the floorplan files can be copied from the *narrow* directory tree to the newly generated *histogram* tree. Now, all of the files are ready for Flint. The Flint interactive floorplanner is then called for each level in the hierarchy, starting with the innermost macrocells of the directory tree.

Flint is first used to generate the *control* macrocell. From Flint, the control macrocell is selected, and the *control.pr* processor floorplan file is selected by using the library mode. Once the floorplan is loaded, the route command is selected, and the resulting routing is displayed. Then this layout is dumped by selecting the Flint store command, generating the *control* layout file, and the *control.hdl* description of this macrocell.

The same procedure is repeated for the *pr* macrocell, and finally for the *narrow* macrocell, which is in fact the complete chip. The layout files are now distributed throughout the *histogram* directory tree. Rather than listing the paths to each of these directories so that the KIC or Magic layout editors can find the layouts, the next step is to link to all of these files from one directory. The Connect program provides a utility for doing this automatically:

```
Connect -f histogram.ch
```

This 'flattening' option generates the directory *histogram/Layout*, which links to all layout files. Now, the Magic or KIC startup files can be copied from the similar *narrow/Layout* directory, with no changes. Finally, the Caltech Intermediate Format (CIF) file can be generated from the Magic or KIC files, preparing the circuit for simulation and transmission to the fabrication facility.

5.1. Histogram Processor Pin Descriptions

Before discussing simulations, the pinouts of the histogram processor must be defined. The pads can be broken into four groups. The power, ground and clock pad group is self-explanatory. The input pads accept the 5 or 10 MHz video input stream, which is four bits wide for the test chip, and is 8 bits wide for the full scale chip. The output pads consist of 8 *outdata* pads for loading the look-up table, and three bits of LUT register selection address, *addr* 0–*addr* 2, with the *load* signal for strobing the registers.

The remaining group of pads are the control pads, which serve to synchronize the processor with the video circuitry, and to program features of the processor. The vertical and horizontal blank signals, *vblank* is FALSE and *hblank* is TRUE during the visible portions of the video signal. Whenever *vblank* is FALSE, *hblank* is used to enable the histogram accumulation, to prevent histogramming the video data which is presented during the horizontal retrace. When *vblank* is TRUE, the postprocessing cycle commences, reading and refreshing the histogram if the input $\overline{read}$ is FALSE, or integrating the histogram if $\overline{read}$ is TRUE. Also during post processing, the *field* input selects that the histogram is to be cleared as it is read out, if *field* is FALSE. If the video data is interlaced, the memory will be alternately refreshed and accumulated, if histogram equalization is being performed.

When a histogram equalization curve is being calculated, the result of the integration must be normalized, or scaled. Four scaling factors can be selected externally to the chip, where each scale factor is applied in sequence. The accumulator output can first be multiplied by 1 or 2, then by 1 or 4, according to the *shift* 1 and *shift*2 signals respectively. Two selectable 4 bit shifters follow, each one multiplying by 1 or 16, according to *shift*4 and *shift*4*a*. By selecting a combination of these, any power of two from 1 to 2^{11} can be used to normalize the output data. Unsigned, saturating shifters are used, to prevent unpredictable overflow errors.

The remaining control pin, *fivemeg*, selects the clock speed. If a 10 MHz clock is used, *fivemeg* should be low. This signal ensures that the postprocessing cycle generates 2.4μs LUT programming cycles for both a 5 and 10 MHz video sample rates.

5.2. Simulations of the Histogram Processor

Now that the histogram processor layout has been completely generated, simulations are run to verify that the circuit is functional. By extracting the circuit directly from the CIF file rather than from schematics, the simulation will also verify that the layout is sound. The CAD utility 'mextra' is used to extract the circuit from the layout. Transistor features and capacitances are represented in the resulting circuit.

The extracted circuit is passed as input to the switch level circuit simulator, 'esim'. Test vectors are then applied to the system, to check out the circuitry. First, the RAM is cleared, then a sample of video data is applied. Finally, the data is read out, to verify the RAM contents.

The RAM is cleared during the post-processing cycle, when *vblank* is high. By forcing the *field* input low, the RAM will be cleared as it is read out. The small test chip, 'narrow', has 4 bits of video input data, and hence contains 16 locations which must be cleared. The processor outputs and clears 4 locations every $800ns \times 16 = 12.8\mu s$. The 16 locations are then cleared over $12.8\mu s \times 4 = 51.2\mu s$. If a 5 MHz video sample rate is used, then at least $51.2\mu s \times 5MHz = 256$ clock periods are required during the vertical blank period to clear the memory. For the simulation, 270 cycles were given to clear the memory. The results of the histogram calculation will verify that the RAM has been cleared.

Now that the RAM has been initialized, the video data can be read in. As long as the *vblank* input is low, the histogram will be calculated on the video input stream. The *hblank* input is used to gate the video data, histogramming input data only when *hblank* is high. For this test, one line of data is read in, consisting of triplets of samples 0, 1, 2, 3, 0, 1, in order, as shown at the top of figure 5.1. A sliding time scale is given for reference, representing 18 clock cycles at the video sample rate. The numbers in this scale represent the video sample values.

The RAM bank selects, *sel* 0 through *sel* 3, are delayed by three clock cycles from the corresponding input. The RAM is actually read a cycle later, with the complementary outputs *ramout* [0–2]. Two cycles later, the incrementer outputs complemented data to be written into the RAM. Notice that the histogram datapath increments by two when the video data and hence the RAM address is repeated.

The histogram data can be observed during simulation on the outputs from the histogrammer, one period after it has been written to the RAM. During actual use, this data will not be reliable, since the output circuitry is intended for much slower operation.

After the video stream has been processed, the result can be read during the next post-processing cycle. For the above example, the video intensities 0 and 1 occurred six times, and the values 2 and 3 occurred 3 times. The result of reading the video data is shown in figure 5.2.

The data can be read out of the RAM only once if the RAM is being cleared at the same time. As a result, valid data is available from the RAM every 12th cycle at a 5 MHz sample rate, or every 24th sample interval at the 10 MHz sample rate. The histogram incrementer datapath stores the current value after reading or accumulating, so that it is stable for the slower output circuitry, including the scaling and saturating logic.

The look-up table expects to receive four consecutive LUT values, followed by the address of these four in the table. The signals *addr* 0, *addr* 1, and *outsrc* indicate which of these values is loaded at one time. If *outsrc* is high, the address signals indicate which LUT register is to be loaded with the data on *outdata* [0–7]. If *outsrc* is low, *outdata* [0–5] provide the most significant six bits of the LUT addresses to be loaded. In either case, the data is valid on the rising edge of the load signal. The full cycle for downloading four values is repeated until all values are loaded into the LUT.

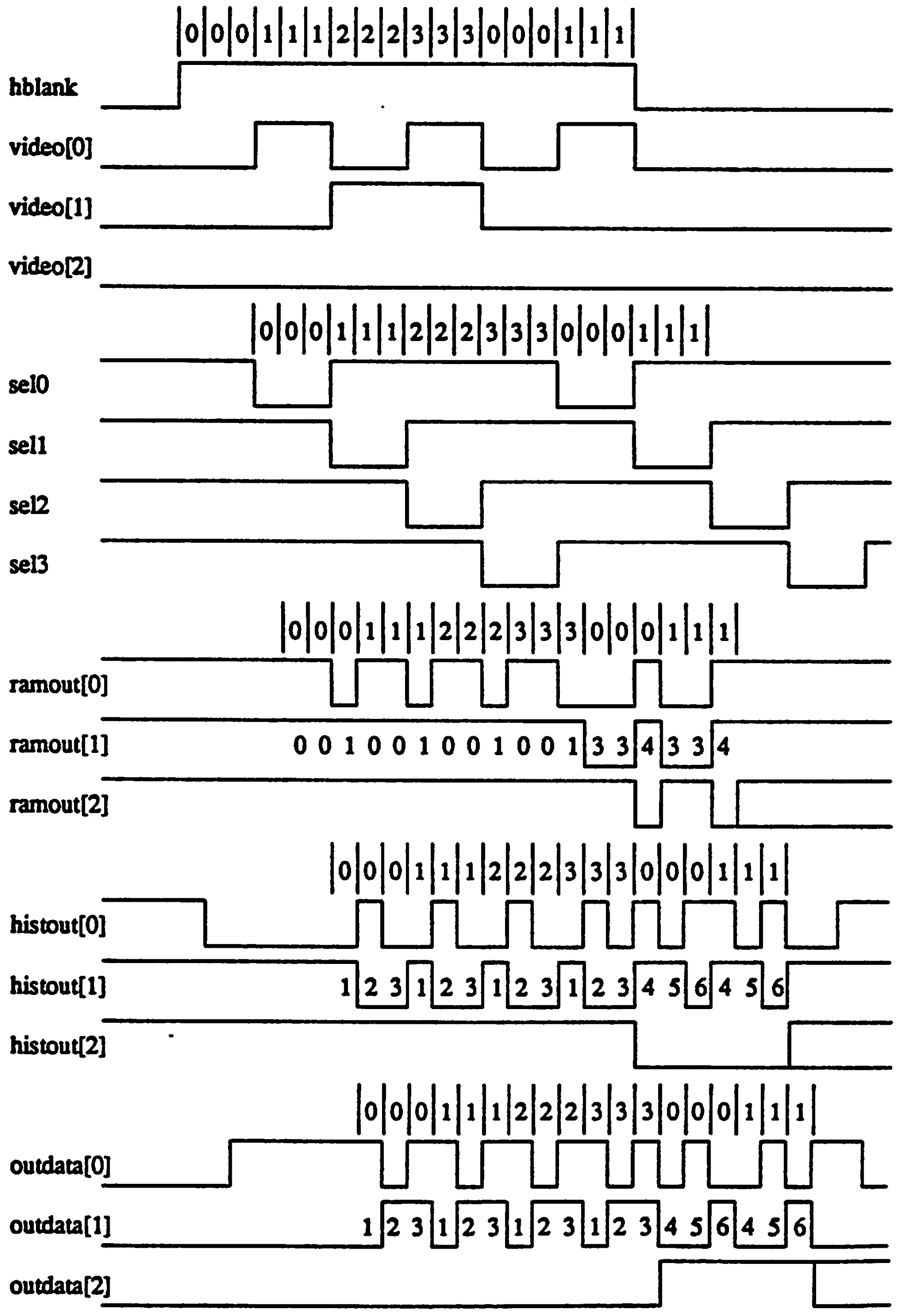


Figure 5.1: Histogram Calculation Simulations

The timing diagrams in figures 5.1 and 5.2 were generated automatically from the simulator results, and indicate the actual functioning of the circuit. Since the circuit simulates as desired, it is ready for fabrication, after design rule checks verify the integrity of the layout.

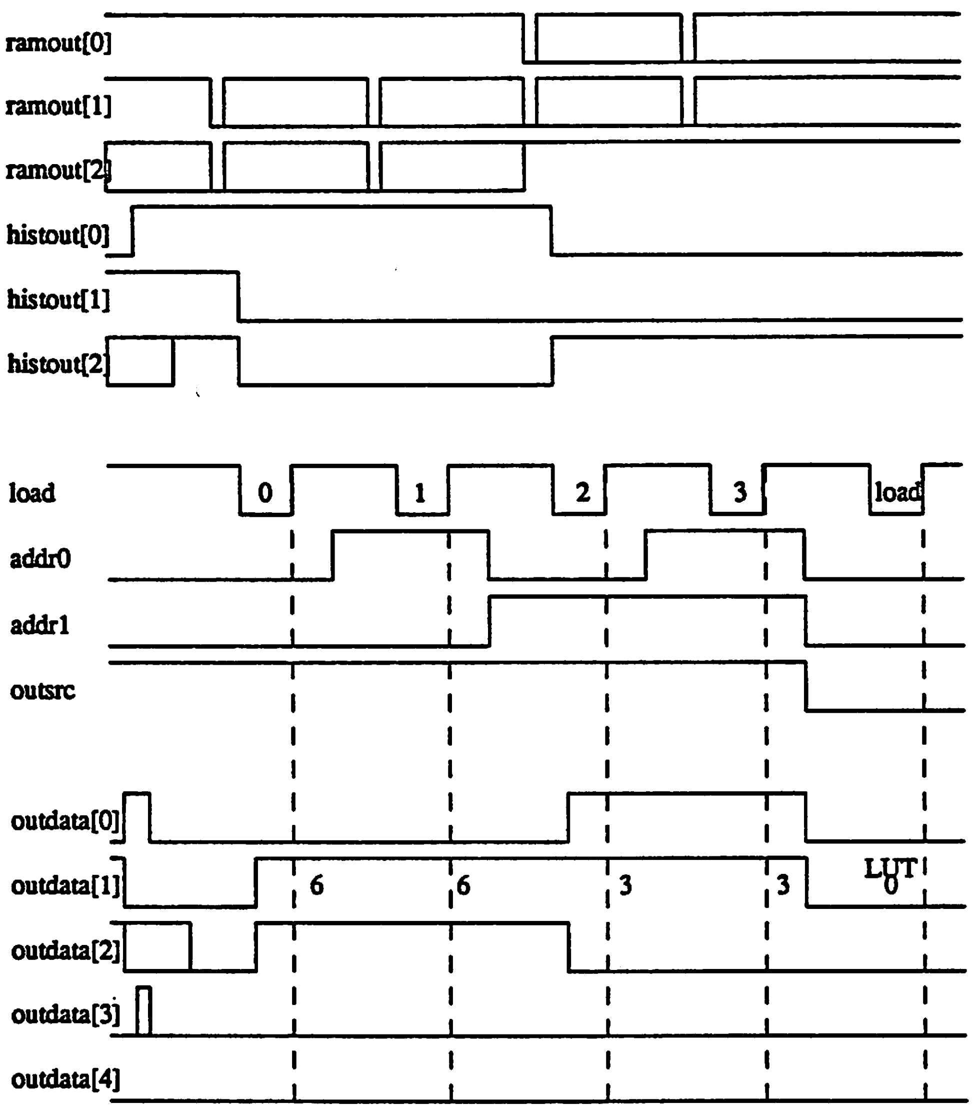


Figure 5.2: Post Processing Simulations

6. Conclusion

Designing the Histogram processor with silicon compiler CAD tools offered many advantages over the alternative of manual layout. The ability to parameterize a design is a big win, compared to redesigning the layout of a similar circuit by hand. Also, fundamental changes can be made to a circuit with a fast turnaround. Debugging can also be simplified by first building a small test circuit for tracking down problems, and then merely specifying new parameters to generate the full size project. In addition, different architectural changes can be evaluated, often with minimal modification of existing descriptions.

The overall design of the histogram processor, including CAD tool development, was often criticized for taking longer than the design time for manual layout of similar circuits. Since new macrocells had to be added to the library to support the circuit, some manual design and layout was required anyway. The design effort for the histogram processor was in fact similar to that for other video chips, in that the same module generators, Compile and Modgen, were used to make many of the macrocells. The difference appeared at the system level, when the blocks were pulled together. The "manually" designed video chips were assembled and routed by hand. For the histogram processor, a symbolic description at the system level was used to generate the macrocells, and to connect them together. Most of the design time attributed to the histogram processor was spent trying to adapt the CAD tools so that they could work together at the system level. Writing the Connect program to unite Modgen, Compile and Flint proved to be a substantial part of the project, since the latter three were originally written independently. Future projects promise to have much more favorable design times.

On the whole, the histogram processor project demonstrated the advantages of using a silicon compiler for designing high speed digital circuits. The flexibility offered by the modified LAGER silicon compiler can be compared to that of a wire-wrap board, by offering a straightforward means for the designer to change a circuit.

REFERENCES

[1] Hall, Ernest L., "Computer Image Processing and Recognition,", p. 166-173, New York: *Academic Press*, 1979

[2] Ohlander, Ron, Keith Price and D. Raj Reddy, "Picture Segmentation Using a Recursive Region Splitting Method," *Digital Image Processing and Analysis*, Vol. 2, p. 226, IEEE, New York, 1985.

[3] Ruetz, Peter A., Robert W. Brodersen, " A Realtime Image Processing Chip Set," *Proceedings of the ISSCC*, Vol XXIX, p. 148; February, 1986.

[4] Ruetz, Peter A., Rajeev Jain, Robert W. Brodersen, " Comparison of Parallel Architectures for Image Processing Circuits," *IEEE International Conference on Circuits and Systems;* San Jose, April, 1986.

[5] Rabaey, Jan M., Steven P. Pope, Robert W. Brodersen, "An Integrated Automated Layout Generation System for DSP Circuits," *IEEE Transactions on Computer-Aided Design*, Vol. CAD-4, NO. 3, p. 285; 1985.

Acknowledgements

This research was sponsored by funds from DARPA, MICRO, Bell Labs, and General Electric.

Many people contributed directly or indirectly to this project. Peter Ruetz deserves a great deal of credit for offering architectural suggestions, numerous circuit layouts, and updating his datapath compiler to fit the needs of the LAGER system.

The automated layout of the processor would have been impossible without the assistance of the LAGER development team. Gordon Jacobs and Rajeev Jain helped with the Modgen support for generating several macrocells. Professor Robert W. Brodersen provided invaluable support for the Flint Floorplanner tool which he and Jan Rabaey have developed over the past two years.

During the development of the Connect program, Chuen-Shen Shung offered feedback on the internal workings of the program. Mats Torkelson helped to debug the Connect program when trying to automate the design of Robert Kavaler's Dynamic Time-Warp circuit. Robert Kavaler also helped to catch bugs in the original code.

Appendix A: Connect User's Manual

Motivation

Over the last few years at Berkeley, several CAD tools were designed, aimed at automated layout. The LAGER system, intended as a digital signal processor design tool, resulted in a variety of useful programs. Modgen, a tiling module generator, can assemble a large variety of macrocells from a simple description. Flint, an interactive floorplanner, allows the designer to try a variety of floorplans. Flint uses a symbolic description of the chip, and can handle the entire place and route task up to the final layout.

Another program, 'compile', is a flexible tool for generating bit sliced datapaths. Each slice of the datapath can contain a list of functional blocks, which will be routed together as specified by the user. Although not originally intended for use with the automatic floorplanner, 'compile' has evolved to fit in with the system.

Simulation programs, such as 'esim', can be used to test a layout after it has been generated. Labels in the layout file can be used to identify inputs and outputs in the circuit. To prevent confusion, the labels should be meaningful, and consistent throughout the layout. The individual programs, however, had their own conventions, occasionally resulting in naming conflicts, or assigning two names to one signal.

The objective of the Connect program is to take full advantage of the individual CAD tools, while pulling them together into a generalized silicon compiler. Any macrocell oriented design should be able to fit into the framework. Also, once a chip has been designed, a similar chip with different data sizes should be realizable with minimal effort.

Terminology

To describe a chip symbolically, a variety of definitions should be clarified. The basic building blocks of a system are the macrocells or modules. In general, macrocells contain other cells or macrocells. To simplify the definition, all macrocells have rectangular boundaries.

The external connections to a macrocell are defined as "terminals", which must be located on the bounding box. The name of a terminal is defined when the macrocell is designed, and does not change between different designs, although the number of terminals might vary. If a terminal occurs more than once in a given macrocell, it will be given an integer index, thereby supporting variable bus widths. Terminals to be connected together are also given a common net number, which is usually assigned by a higher level program. At least one destination macrocell is specified, assigning the net to a "cable", defined below. The macrocell generating programs also specify the exact location of the terminal on the macrocell boundary. The terminals completely define the circuit interconnections while remaining independent of the floor plan.

Routing between macrocells is constrained to rectangular channels. A macrocell can have only one channel touching each side, whereas a channel can have several channels and macrocells adjacent to it. Once the macrocells are placed on the floor plan, the channels are inserted to fill the space in between to provide paths for all terminal interconnections.

Often, there are many possible paths for routing between macrocells, where a path is a series of abutting channels. Rather than specify the path for each interconnection separately, nets are grouped together into "cables". Since a cable is defined by the macrocells at either end, the designer can specify where the routing is to be done without reference to the terminal or net names. This helps reduce the confusion when a designer is interactively floorplanning with Flint.

The Netlist Language for Connect

The file for defining the interconnections and hierarchy of a desired architecture uses a syntax like that of the C programming language, however, it is not a C program. To allow parameterization, integer variables with limited C style expression handling are supported, excluding arrays and pointers. The supported expressions are given in figure A.1, listed in the order of precedence.

These operations can be applied to integer constants or variables denoted by the "symbol" syntax. A symbol is an unquoted character string of alphanumeric symbols, or a quoted string of any symbols, where backslash ‘′ can escape the following symbol. Parentheses can be used to change the order of precedence.

The description of a chip begins with an optional header of parameter definitions. Values can be assigned to variables at this time, and other parameters can be calculated. The syntax of this definition section is shown in figure A.2. Once the parameters have been defined, the netlist can be specified. The direc-

```
?   :
||
&&
|
^
&
==  !=
<  >  <=  >=
>>   <<
+   -
*   /   %
unary minus   !   ~
```

Figure A.1: Precedence of Operators.

```
parameter:
    <symbol> = <expr> ;
|   <symbol> ;        Verify that the symbol is defined.
|   { parameter_list }

parameter_list:
    parameter
|   parameter  parameter_list
```

Figure A.2: Parameter Definitions.

tory containing the Flint database is named in the argument to 'design'. The hierarchy of a design is described by the syntax in figure A.3.

```
design:
    'design' ( <Flint hierarchy> ) statement

statement_list:
    statement
|   statement statement_list

statement:
    { statement_list }
|   <symbol> = <expr> ;
|   <symbol> ;
|   'if' ( <expr> ) statement
|   'if' ( <expr> ) statement 'else' statement
|   macrocell ( <Modgen-macrocell-name> ) cable
|   macrocell ( <Datapath-macrocell-name> , 'datapath' ) cable
|   macrocell ( <Flint-macrocell-name> ) statement
```

Figure A.3: Design Hierarchy Syntax.

By using either form of the "if" statement, entire sections of a design can be removed. For example, if

RAM storage in a design is small enough, a designer can choose not to use a second RAM macrocell. The syntax definition of the statement illustrates the three types of macrocells, from the datapath compiler, Modgen and Flint.

The Flint database requires the assignment of net numbers to the terminals of all macrocells. Rather than specify net numbers, the designer can use the cable syntax to assign a net name to a terminal. This net name can be the global signal name on a schematic, for example. The syntax, in figure A.4, goes further to allow busses to be referenced by a single name. This allows parameterization of the bus widths, since integer expressions can be used to index parts of a bus.

```
cable:
'cable' '(' <Destination-macrocell-name> ')'  term_net
|   '{' cable_list '}'
|   '{' '}'

term_net_list:
    term_net
|   term_net  term_net_list

term_net:
    <terminal-name> range '=' <net-name> range ';'
|   '{' term_net_list '}'

range:
    /* no range */
|   '[' <expr> ']'
|   '[' <expr> ',' <expr> ']'
```

Figure A.4: Cable and Netlist Syntax.

The argument to 'cable', the terminal name and the net name are of the type "symbol" discussed earlier.

Notice that there are three forms for a terminal or net range. If no range is specified, this is the same as the range "[0,0]". This is used for most one-bit signals. The range "[value]" is the same as "[value,value]", and is used to attach one-bit terminals to a bus. If two values are given, both the terminal and the net must have double ranges, and the difference between the ranges must be the same. For example, the netlist assignment `term_a[3,6] = net_g[4,7]` is valid, whereas `term_a[3,6]=net_g[3,8]` is ambiguous. The order that nets are assigned to terminals can also be reversed. For example, the assignment

```
        counter_out[0,2]=int_address[5,3];
```

is the same as specifying

```
        counter_out    = int_address[5];
        counter_out[1] = int_address[4];
        counter_out[2] = int_address[3];
```

Since the ranges can contain expressions, the program Connect will report range inconsistencies, to help identify mistakes in the netlist.

At this point, the designer can parameterize the net-list, and conditionally include macrocells. The major task remaining is to pass parameters to the macrocell generators.

Flexible Macrocell Generation

The input files for the datapath compiler and for Modgen specify parameters for macrocell generation, but a different file must be generated for each type of macrocell. Hence a method must be devised to generate the input files without losing the flexibility of each program.

The Connect program uses modified Modgen or datapath compiler input files, called template files, to define all macrocells in a design. A template file is identical to a regular input file to either macrocell generator, except for a variety of substitution commands. After reading and processing the netlist description file, the Compile program filters the template files and generates proper input files for the desired macrocells. Aside from filtering the files, Compile moves the resulting input file to the correct location in the Flint directory tree. This means that all of the template files and the netlist file can reside in the same directory, and several resulting Flint directory trees can start in the same directory.

Modgen Template Commands

To generate a macrocell with Modgen, a processor description language (PDL) file must be generated as input to Modgen. The PDL has a LISP-like format to define module and terminal features. Consider this description of a RAM bank:

```
(module
    (name ram1)
    (type ram_hist)
    (width 10)
    (words 4)
)
(term (name din[0]) (net 11) (cable pr|histdata) )
(term . . .
```

Figure A.5

This PDL file describes a RAM which is to be called "ram1", generated by using the "ram_hist" module generator. This file explicitly specifies a word width of 10, an address of 4 bits. The module description is followed by terminal information, all of which is completely dependent on the netlist data. Now, compare the PDL file above to the corresponding template file:

```
#delimiter &
(module
    (name ram1)
    (type ram_hist)
    (width &width&)
    (words &words&)
)
#pdlterm
```

Figure A.6

Assume that the variables "width" and "words" have been defined in the header of the netlist file (see the previous section). Then, the Connect program will substitute an ascii integer for "&width&" and "&words&". Then, the substitution command "#pdlterm" will be replaced by all of the terminals which are defined in the netlist. The netlist file contains sufficient information so that Connect can fill in the terminal entries completely. Hence, all features of Modgen can be used without requiring that the designer keep track of the confusing terminal information.

Datapath Compiler Template Commands

The datapath compiler has an input specification which is analogous to the PDL file used by Modgen. The first part of the file defines the terminal information, much like the second half of the PDL file. Consider the following excerpt from a datapath description:

```
    . . .
LEFT
    extaddr 58 inpads
    countaddr 62 pr|control|counter2

RIGHT
    ramaddr 21 pr|histdata
    . . .
```

Figure A.7

For each side of the macrocell, the terminals are listed with relevant information. All datapath slices will have terminals on the right and left with the same name and different indices, corresponding to a data bus. The signals on the top and bottom of the macrocell are not duplicated, since they are usually control signals. The terminal name is followed by the net number of the first terminal of that name. Then an expanded cable path is given. All of this information can be generated automatically, and the template file would look like this:

```
    . . .
LEFT
#term    extaddr
#term    countaddr

RIGHT
#term    ramaddr
```

Figure A.8

Again, the Connect program fills in the details. The remaining problem is to parameterize the width of a datapath. The designer defines the width by listing the slices which are to be assembled, rather than by giving a number. To handle this, one more command has been added to conditionally include a template line in the datapath description file. This is illustrated in figure A.9.

The #if command copies the following template line to the destination datapath description file if the expression is true. The expression can only compare variables or constants, and the six arithmetic comparison operators apply ('==', '!=', '<', '>', '>=' and '<='). Figure A.10 shows the result of reading this template file if "size" is set to 6 in the header of the netlist file.

Since the datapath compiler was not originally designed as part of the LAGER system, some net assignment inconsistencies can result. When the datapath compiler assigns nets to terminals, it assumes

```
ORGANIZATION
    CNT NODATA
    CELLDATA
    CELLDATA
    CELLDATA
    CELLDATA
#if size > 4
    CELLDATA
#if size > 5
    CELLDATA
#if size > 6
    CELLDATA
#if size > 7
    CELLDATA
    GND NODATA
```

Figure A.9: Datapath Width Parameterization.

```
ORGANIZATION
    CNT NODATA
    CELLDATA
    CELLDATA
    CELLDATA
    CELLDATA
    CELLDATA
    CELLDATA
    GND NODATA
```

Figure A.10

that the nets for a given terminal name are numbered sequentially. In general, this may not be true, and can cause fatal wiring mistakes. To solve this problem, the Connect program can be used to read the hardware description language (HDL) file which results from running the datapath compiler. The HDL file is the complete macrocell layout description. By reading the terminal name in the HDL file, the Connect program can determine the correct net number, and fix the file if necessary. Aside from fixing the net number, this filter mode will also substitute the net number for the terminal name, in the form "net_##", for debugging purposes. This filter is included in a shell script, "Compile", which runs the datapath compiler 'compile', and fixes up the output. The final result from the datapath compiler is then virtually indistinguishable from the output of Modgen.

Using the Connect Program

After the netlist file and the templates have been prepared, the Connect program is ready to go. The following command should be run in the directory containing the template files and the netlist file netlist.ch:

```
Connect netlist.ch
```

If no errors occur, the subdirectory named by the parameter 'design' should exist in the current directory. The PDL and datapath (*.dp) files should be in the desired places. The next step is to run Modgen and Compile on the appropriate files, to generate the HDL files. At this point, this must be done manually, or in a shell script. This is not done by Connect since a proposal was once made to do this in Flint. Once the module generators have been run, the Flint interactive floorplanner can be run, and the design is complete.

After the chip has been successfully generated by Flint, the actual layout is distributed through several subdirectories in the Flint hierarchy. To flatten out this information, Connect can find the layout files, and place a symbolic link to each of them in the directory <design>/Layout:

```
Connect -f netlist.ch
```

If the layout files are in the KIC format, then the .KIC file only needs to include the datapath and Modgen CELL libraries in its search path. This command is usually needed only once, since the links to the layout files remain whenever Connect or Flint are rerun.

Debugging Aids

The module generators Compile and Modgen place labels in the layout files indicating the location of nets or signals. A circuit extractor, such as 'mextra' will use these labels where possible to identify nodes in the extracted circuit. As a convention, Modgen generates labels of the form "net_##", placing the integer net number in the name. Net numbers are used instead of terminal names, since macrocells can share common terminal names. This can cause one circuit node to have two names, and also two circuit nodes might share the same name. Both cases could confuse a circuit extractor, and could cause unpredictable results when simulating a circuit using an existing set of test vectors. The 'compile' program normally places the terminal names in the layout file, and the Compile script corrects for this by using the Connect program to filter the layouts.

Since net numbers have no obvious relationship to the circuit schematic, the Connect program can generate a list of the net names associated with the net numbers:

```
Connect -n netlist.ch > netlist.alias
```

The resulting list is suitable as input the the 'esim' or 'mosim' simulators:

```
= net_1 ramout[0]
= net_2 ramout[1]
. . .
= net_9 ramout[8]
= net_10 histout[0]
= net_11 histout[1]
. . .
= net_42 saturated
= net_43 readml
```

This assigns the actual net or signal name to a node, allowing the designer to refer to actual signal names when simulating the circuit. Simulator command files with test vectors can be similar for several different versions of a given circuit.

Appendix B
The Connect Program: Internal Representation

Data Structures

The Connect program uses several internal data structures to organize the data from the Connect input database, detailed in the *Connect User's Manual*. Modules, macrocell terminals, cables and nets each have associated data structures. Figure B.1 illustrates the relationship between the structures.

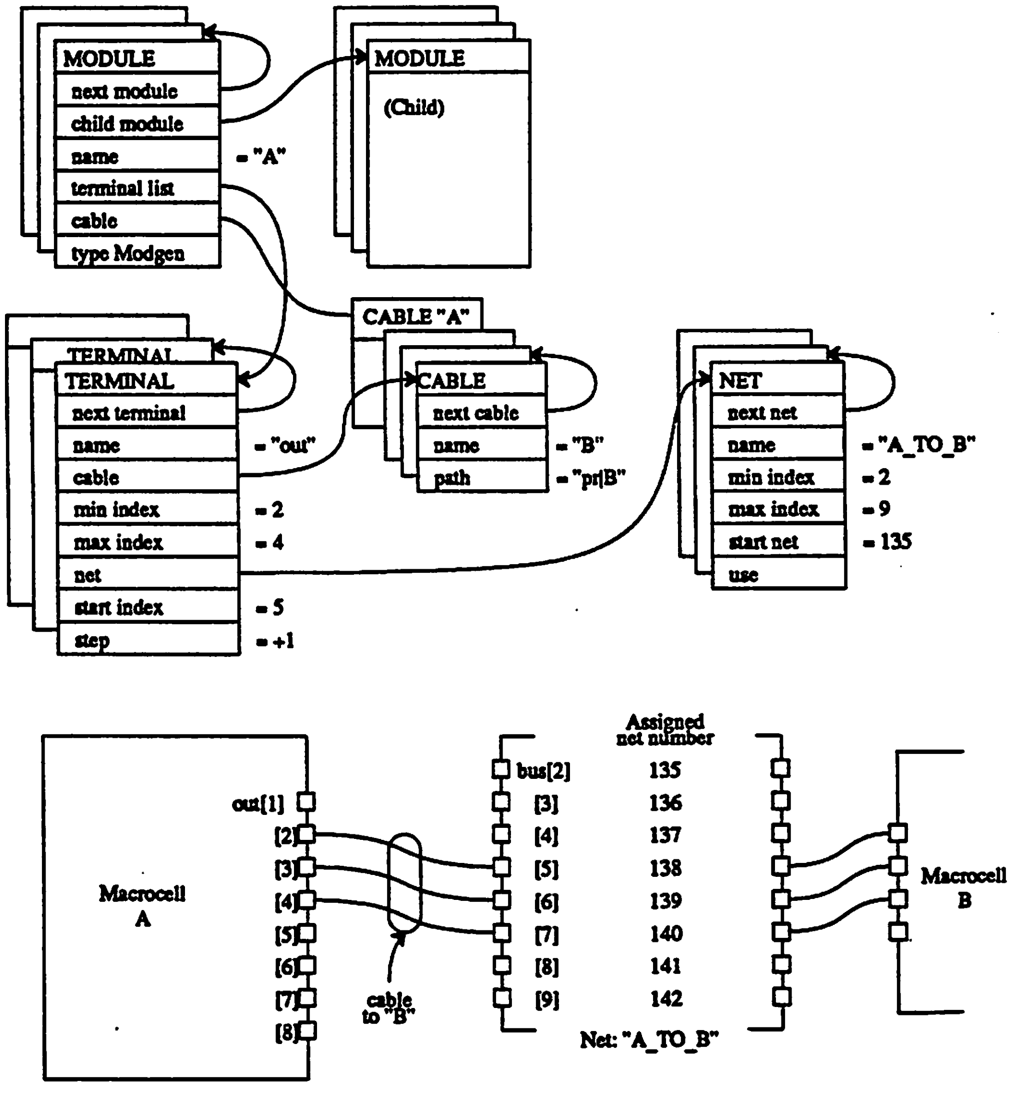


Figure B.1: Internal Data Structures in Connect.

Modules, or macrocells, are stored in a tree data structure, to support the nested hierarchy of macrocells. Modules which share a single floorplan are siblings in the tree. Siblings are organized in a linked list, with each referring to the 'next' one. Similarly, the child pointer points to a list of children. In the histogram processor, the 'control' block has the children 'counter1', 'counter2', 'timingpla' and 'fsmsync', and has several siblings, including the RAM and two datapaths. The module name is also recorded so that cable paths can be built up later. The cable entry points to an entry in a list of cables, which is filled in after the Connect input file has been read. Also, to identify which module generator is to be used, the 'type' of macrocell is stored as a choice between Modgen, Compile or Flint.

A linked list containing terminal information, 'terminal list', is stored for each module which has no children. In general, a terminal on the edge of a macrocell can be a single signal or a bus of several signals. The elements of each bus are indexed so that they can be referenced individually, or in groups. The purpose of each terminal structure in the linked list is to index all or part of such a macrocell terminal, and several such structures may refer to sections of the same terminal. Each structure contains the name of a terminal and a minimum and maximum index range.

To attach two or more terminals together, they are assigned to a common net name, which is stored in the structure 'net'. Each terminal structure can refer to only one net, where a net, like a macrocell terminal, can be a signal or a bus. In general, a terminal can connect to any consecutive elements of a net, in the following form:

```
        term_name[a,b] = net_name[c,d];
```

Since the named net may have a larger range than that indicated in the square brackets, the terminal structure must also store the start of the net range, as 'start index'. In the above example, the start index is set to the smaller of `c` and `d`. If the terminal and net ranges are in the opposite order from one another, the terminal will be connected to the net in reverse order, a condition stored as a 'step' direction of -1 in the terminal data structure.

The net structure is similar in form to the terminal data structure. The name and index range of the net are stored. Since several terminals may refer to the same net, the minimum and maximum range index values must be adjusted to include all ranges given for the named net. Once all of the nets have been assigned, integer net numbers are generated, one for each indexed signal. The lowest net number for each net structure is stored in 'start net'. The last feature in the net structure is the 'use' flag, which indicates whether or not a net is of interest at run time. For example, when debugging, the designer may wish to list only those nets attached to a selected group of macrocells (see the -g option for the Connect program). This is done by marking the desired nets as they are referenced, and then reading the 'use' flag to decide whether or not a net is to be listed.

Cables

The Flint floorplanner uses a mechanism called a 'cable' to refer to macrocell interconnections at a high level. To route the cable of all wires which connect from a side of one macrocell to a side of another, the designer can interactively select the two sides, and then sequentially select the channels through which to route the cable.

In the Connect input file, cables are defined by giving the macrocell names at each end of a net. Flint, however, requires more than the macrocell names; the path to a macrocell is also needed, since each macrocell has a different directory in the Flint design file hierarchy. The path to a macrocell starts at the root of the Flint directory tree for a given design, and a vertical bar, '|', delimits the list of directories which must be traversed to find a desired macrocell.

After the module data structures have been initialized, the tree of modules is traversed. The cable paths are built up at this time, and stored in the cable structure 'path' element. Once the paths are set up, Connect uses these paths to generate input files for the module generators.

For each macrocell which does not contain other macrocells, the Connect program generates Modgen or Compile input files. For each terminal in a macrocell, at least one destination macrocell must be named, to completely specify a cable. For the current versions of Flint, several rules must be followed to avoid erroneous cable designations.

If the path to a destination module is too long, connections between two blocks can become unnecessarily complicated. Consider, for example, the cable between the macrocells `pr|control|counter1` and `pr|control|timingpla`. Although this cable can be entirely within the control floorplan, the paths include the processor macrocell, 'pr'. As a result, Flint may not actually connect them together within the 'control' macrocell, and will route this signal to the boundary of the control macrocell. By eliminating 'pr' from the path of each macrocell, these macrocells will be routed together within the floorplan of 'control'. This problem can be solved in general by removing the path which is common to both macrocells, with the exception of the parent containing both.

If a path is too long, a cable may not be generated. The two macrocells `ctrlpads` and `pr|control|timingpla` illustrate this problem. When routing the full chip, Flint recognizes 'pr' as a black box containing 'control', and knows nothing about the 'timingpla'. Hence, it rejects the cable between these two blocks. To solve this problem, the 'timingpla' is removed from the path, thereby satisfying Flint.

The cable path restrictions, which are adjusted by the Connect program, come about since Flint was originally designed to support two levels of design hierarchy. The procedure for generating a proper path to a destination requires comparing the full source and destination paths. First, the common part of a path name is eliminated, and the result is truncated to two macrocell names if more than two remain in the path.

This cable generation scheme prevents cables from being routed outside of the lowest macrocell containing both the source and destination. If a signal which connects two macrocells together is also to be routed beyond the parent of those macrocells, one of the macrocells within the parent must refer to a macrocell beyond the parent, otherwise no connection will be made.

Normally, each cable will be described twice, once at each end. If more than two terminals are to be interconnected, some cables may be referenced only once, which presents no problem, as long as the cable connects to a sibling macrocell.

Appendix C. Listings of the Connect Program

Makefile	The input script for the UNIX 'make' program, for automatic compilation of the Connect program.
Conn.h	Definitions of the internal data structures and global variables for the Connect program.
Common.h	Definitions of variables common to several of the Connect routines.
Connect.c	This is the main C program for Connect, where command line parsing is handled.
BuildFlint.c	Once a database describing a chip has been read in, this code generates the Flint directory tree.
FillTemplate.c	This code filters PDL and datapath input description templates, and generates Modgen .pdl and datapath compiler .dp input files. Variable and terminal information substitutions, as well as conditional line exclusion are implemented here, to support parameterization of a design.
Database.c	This file provides the routines for handling the macrocell, cable, terminal, net and numeric variable data lists.
Flatten.c	These routines traverse a Flint directory tree, and link all layout files to the subdirectory Layout. This simplifies the testing, debugging and fabrication stages in the development of a circuit.
Lists.c	C routines for operating on lists are included here. Functions for appending and sorting lists are provided.
DatapathUndo.c	These filters written in C use the database read from the input description for a design to correct inconsistencies in the KIC layout files, and the hardware description language (HDL) files.
dbaseLex.l	This file is input for the lexical analyzer compiler, LEX, which produces C code for characterizing input data.
dbaseYacc.y	This file describes the syntax of the design input file, and is compiled by YACC to generate a syntactic analyzer in C. The result is used in conjunction with the LEX output to parse the design input file.

Appendix D. Listings of Template Definitions for the Histogram Processor.

addrdata.tem	The address datapath detects repetitions of video samples, and selects the source for the RAM bank address.
clocks.tem	This template describes the pair of unbuffered clock pads.
counter1.tem	This counter generates the postprocessing timing by dividing the sample clock by four or eight.
counter2.tem	This macrocell includes two counters separated by a delay in the carry line. The first counter selects the bank of the RAM for postprocessing, while the second counter indicates the address within each bank. These counters are only used during the post-processing cycle.
ctrlpads.tem	This group of pads provides synchronization and programming signals for the histogram processor.
delay1.tem	This delay block provides four delays for aligning the control signals with the pipelined datapaths.
fsmsync.tem	This PLA macrocell controls the synchronization of the histogram processor with the external circuitry. RAM bank decoding is also handled here.
histdata.tem	This datapath performs the incrementing when the histogram is being calculated. During post-processing, the data can be accumulated, and the result can be scaled by a power of two using barrel shifters.
inpads.tem	The video rate intensity data enters the processor through these pads.
outpads.tem	These pads program a look-up table, providing internally generated data, address and control signals.
power5.tem	This pad provides Vdd to the chip.
ram1.tem	
ram2.tem	These are the RAM banks in which the histogram calculation is stored.
timingpla.tem	The outputs of counter1 are decoded by this PLA, to generate post-processing timing signals.

addrdata.tem

```
/*
Histogram processor address datapath
                Input:      Off-chip RAM address
                            On-chip RAM address from counters
                Output:     Selected RAM address
                            Indication of equal consecutive addresses

                10/23/85
*/

CONTROL
                Vdd 0 Vdd
                GND 0 GND
                ph1 0 phi1
                ph2 0 phi2

LEFT
#term           extaddr
#term           countaddr

RIGHT
#term           ramaddr
#term           extdelayed

BOTTOM
#term           equal

TOP
#term           selsrc

ORGANIZATION
                CNT         NODATA
                CELL        DATA
                CELL        DATA
                CELL        DATA
                CELL        DATA
#if size > 4
                CELL        DATA
#if size > 5
                CELL        DATA
#if size > 6
                CELL        DATA
#if size > 7
                CELL        DATA
#if size > 8
                CELL        DATA
#if size > 9
                CELL        DATA
                GND         NODATA

SLICE
                'extaddr'
```

```
/* Flint pads */
> sideleft (in4=countaddr,out4=countaddrT,out=extaddrT)

/* delay the address */
> delay ('in'='extaddrT')

/* compare consecutive addresses */
> equal ('inb'='extaddrT',B /'equal'='equalundel')

/* Delay the data and the comparison result. */
> delay (out=extdelayedT,B /'in'='equalundel',B /'out'='equal')

/* select input source */
select ('in2'=extdelayedT,'in'='countaddrT',T /'sel'='selsrc')

/* drive the address outputs */
> buffer
> sideright (out=ramaddr,in2=extdelayedT,out2=extdelayed)
```

clocks.tem

```
(module (name clocks) (type padclk) (clock))
#pdlterm
```

counter1.tem

```
(module
            (name counter1)
            (type counter)
            (loadable)
            (value x00)
            (design
                       (count 3)
            )
)
#pdlterm
```

counter2.tem

```
(module
        (name counter2)
        (type counter)
        (notloadable)
        (design
                (count 4)
                (delay)
                (count $counter3width$)
        )
)
#pdlterm
```

ctrlpads.tem

```
(module (name ctrlpads) (type padclk) (input 9))
#pdlterm
```

delay1.tem

```
(module (name delay1) (type delay) (length 3) (width 4)
        (design (array
                DDDD
                xDxx
                DDxx
        ))
)
#pdlterm
```

fsmsync.tem

```
(module
            (name fsmsync)
            (type fsm_pr)
            (in 15)
            (out 13)
            (numclocked 14)
            (minterm 29)
            (minimize)
            (input-plane (array
                        010xxxxxxxxxxxx
                        011xxxxxxxxxxxx
                        0011xxxxxxxxxx0
                        0011xxxxxxxxxx1
                        x010xxxxxxxxxxx
                        x00xxxxxxxxxxxx
                        1011xxxxxxxxxxx
                        111xxxxxxxxxxxx
                        110xxxxxxxxxxxx
                        xx0xx10xxxxxxxx
                        xx0xx0xxxxxxxxx
                        xx1xxxxxxxxxxxx
                        xx0xx110xxxxxxx
                        xx0x1xxxxxxxxxx
                        xxxx0xxxxxxxxxx
                        xxxxxx1xxxxx00x
                        xxxxxx1xxxxx10x
                        xxxxxx1xxxxx01x
                        xxxxxx1xxxxx11x
                        xxxxxx0x000xxxx
                        xxxxxx0x010xxxx
                        xxxxxx0x001xxxx
                        xxxxxx0x011xxxx
                        xx0xxxxxxxxxxxx
                        xx1xxxxxxxx0xxx
                        xx1xxxxxxxx1xxx
                        xx0xxxxxxxxxxxx
                        xx1xxxxx0xxxxxx
                        xx1xxxxx1xxxxxx
            ))
            (output-plane (array                /* negative logic outputs */
                        1000000000010
                        1100000000010
                        0100000000001
                        1100000000010
                        0000000000010
                        0000000000010
                        0100000000010
                        0000000000010
                        1000000000010
                        0001000000000
                        0011000000000
                        0011000000000
```

```
                    0001000000000
                    0000100000000
                    0000100000000
                    0000010000000
                    0000001000000
                    0000000100000
                    0000000010000
                    0000010000000
                    0000001000000
                    0000000100000
                    0000000010000
                    0000000000000
                    0000000000100
                    0000000000000
                    0000000000000
                    0000000000000
                    0000000001000
          ))
)
#pdlterm
```

ground.tem

```
(module (name ground) (type padclk) (ground))
#pdlterm
```

histdata.tem

```
/*

Histogram datapath
                Input: RAM data
                Output: to RAM and output circuitry

                9/23/85
*/
CONTROL
                Vdd 0 Vdd
                GND 0 GND
                ph1 0 phi1
                ph2 0 phi2

RIGHT
#term           Shifted

LEFT
#term           In1
#term           Out
#term           Address

BOTTOM
#term           zero
#term           hold
#term           outsel
#term           shsat
#term           resetc
#term           zeroaccum
TOP
#term           shift1
#term           shift2
#term           shift4
#term           shift4a
#term           cin
#term           accum

ORGANIZATION
                CNT             NODATA
#if width > 16
                ODD             DATA
#if width > 16
                EVEN            DATA
#if width > 14
                ODD             DATA
#if width > 14
                EVEN            DATA
#if width > 12
                ODD             DATA
#if width > 12
                EVEN            DATA
#if width > 10
```

```
            ODD       DATA
#if width > 10
            EVEN      DATA
#if width > 8
            ODD       DATA
#if width > 8
            EVEN      DATA
#if width > 6
            ODD       DATA
#if width > 6
            EVEN      DATA
#if width > 4
            ODD       DATA
#if width > 4
            EVEN      DATA
#if width > 2
            ODD       DATA
#if width > 2
            EVEN      DATA
            ODD       DATA
            LSB       DATA
            GND       NODATA

SLICE
            /* delay the RAM output */
            'In1'
            > sideleft (in2=Out,out2=out,in3=Address,out3=address)
            > minus1                                  /* RAM data is inverted */
            > delay                                   /* delay the RAM data, */
            > 'addin'                                 /* and send it to the adder */

            'selin'
            > select1 (T /'sel'='accum')              /* select the adder input */
                                                      /* zero it if desired */
            > zero (B /'zerobar'='zero')
                                                      /* Add RAM data & selected data */
            > uadder ('inb'='addin',T /'cin'='cin',B /'pof'='addsat')
            > mux2to1 ('inb'='selin',B /'in_to_out'='hold')
            > zero (B /'zerobar'='zeroaccum')
                                                      /* hold output value */
            > delay    ('out'='selin',B /'in'='addsat')

            'selin'
            > driver ('out'='out',B /reset=resetc)

            'selin'
            > shift01 (T /'shift1'='shift1')
            > shift02 (T /'shift2'='shift2')
            > shift04 (T /'shift4'='shift4')
            > shift04 (T /'shift4'='shift4a',B /'satoutb'='shsat')
            > usatregister ('out'='shout')
            mux2to1 ('in'='shout','inb'='address',B /'in_to_out'='outsel')
            > sideright (out=Shifted)
```

inpads.tem

```
(module (name inpads) (type padclk) (input $size$))
#pdlterm
```

outpads.tem

```
(module (name outpads) (type padclk) (output 13))
#pdlterm
```

power5.tem

```
(module (name power5) (type padclk) (power))
#pdlterm
```

ram1.tem

```
#delimiter &
(module (name ram1) (type ram_hist) (width &width&) (words &words&) )
#pdlterm
```

ram2.tem

```
(module (name ram2) (type ram_hist) (width $width$) (words $words$) )
#pdlterm
```

timingpla.tem

```
(module
            (name timingpla)
            (type fsm_pr)
            (numclocked 7)
            (in 8)
            (out 7)
            (minterm 18)
            (minimize)
            (input-plane (array
                        0010xxxx
                        0101xxxx
                        1000xxxx
                        1011xxxx
                        1110xxxx
                        00xxxxxx
                        0111xxxx
                        100xxxxx
                        111xxxxx
                        11x1xxxx
                        010xxxxx
                        01x0xxxx
                        0001100x
                        0100100x
                        0111100x
                        1010100x
                        0000111x
                        xxxxxxx1
            ))
            (output-plane (array
                        1000000
                        1000000
                        1000000
                        1000000
                        1000000
                        0110000
                        0100000
                        0100000
                        0111000
                        0111000
                        0010000
                        0010000
                        0000100
                        0000100
                        0000100
                        0000100
                        0000010
                        0000001
            ))
)
#pdlterm
```

Appendix E. Net Lists for the Histogram Processor.

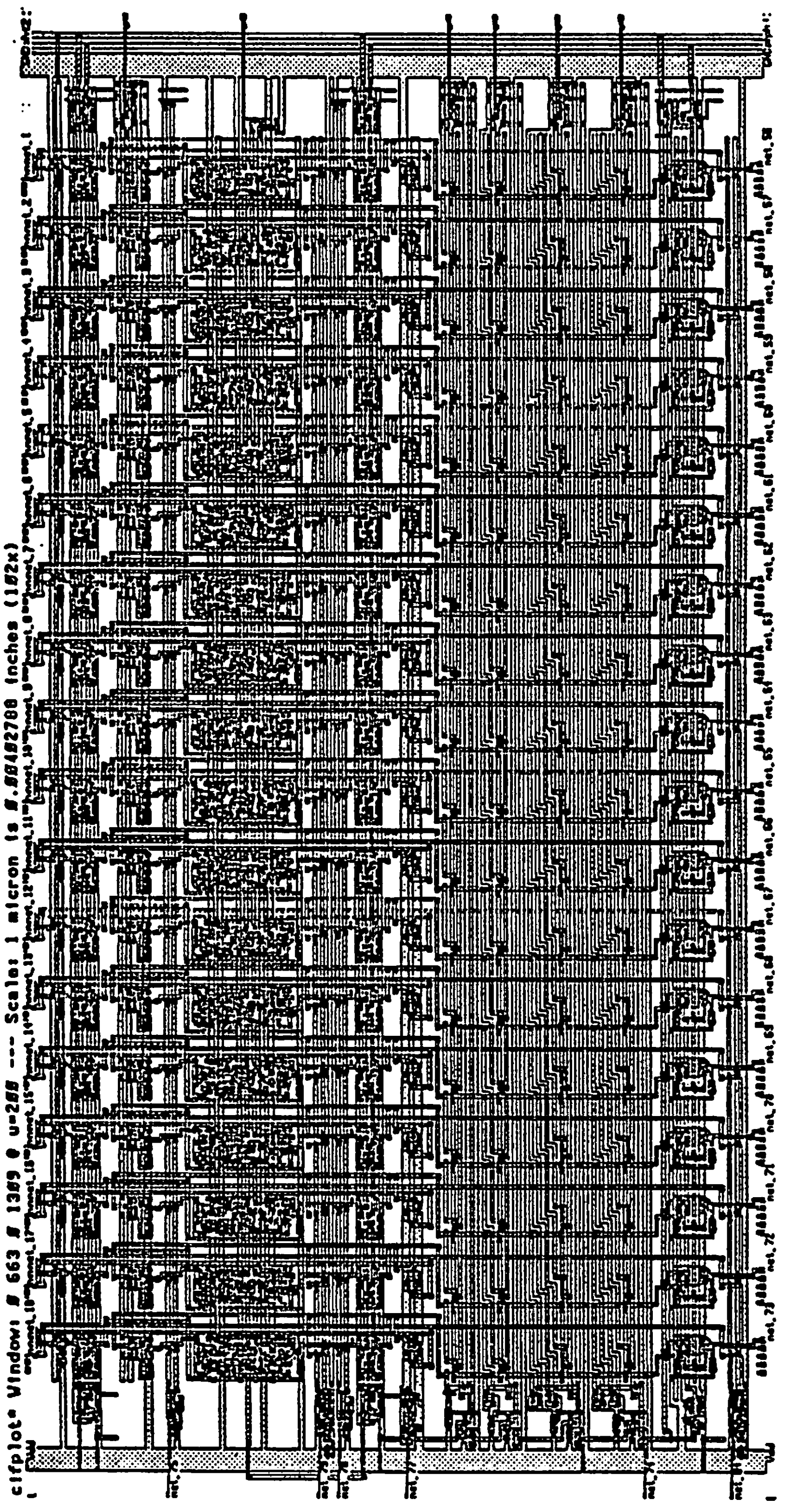
histogram.ch

```
/* Description of the Histogrammer macrocells and blocks */
{
            width = 18;
            size = 8;
            wmax = width-1;
            smax = size -1;
            banks = 4;
            bankbits = 2;
            words = (1 << size) / banks;
            counter3width = size - bankbits;
}

/* The Histogram processor */
design(histogram8) {                                          histogram8
macrocell(pr) {
            macrocell(histdata,datapath) {
                        cable(ram1) {
                                    In1[wmax,0]=ramout[0,wmax];
                                    Out[wmax,0]=histout[0,wmax];
                        }
                        cable(addrdata) {
                                    Address[smax,0]=ramaddr[0,smax];
                                    Address[width-1,size] =
                                                histdataNULL0[0, width-size-1];
                                    accum=vblank;
                        }
                        cable(outpads) {
                                    Shifted[width-1,8]=histdataNULL1[0,width-9];
                                    Shifted[7,0]=outdata[0,7];
                                    shsat=saturated;
                        }
                        cable(fsmsync) {
                                    zero=readm1;
                                    cin=inc1or2m1;
                                    resetc=resetcarry;
                                    zeroaccum=resetaccum;
                        }
                        cable(delay1) {
                                    hold=holdm2;
                        }
                        cable(ctrlpads) {
                                    shift1=shift1;
                                    shift2=shift2;
                                    shift4=shift4;
                                    shift4a=shift4a;
                        }
                        cable(timingpla) {
                                    outsel=outsrc;
                        }
            }
            macrocell(ram1) {
                        cable(histdata) {
                                    din[wmax,0]=histout[0,wmax];
                                    dout[wmax,0]=ramout[0,wmax];
                                    din[width,width+wmax]=histout[0,wmax];
                                    dout[width,width+wmax]=ramout[0,wmax];
                                    address[0,smax-2]=ramaddr[2,smax];
                        }
                        cable(fsmsync) {
                                    sela=sel0;
                        selb=sel1;
                        clear[0]=clear;
                        clear[1]=clear;
                }
        }
        macrocell(ram2) {
                cable(histdata) {
                        din[wmax,0]=histout[0,wmax];
                        dout[wmax,0]=ramout[0,wmax];
                        din[width,width+wmax]=histout[0,wmax];
                        dout[width,width+wmax]=ramout[0,wmax];
                        address[0,smax-2]=ramaddr[2,smax];
                }
                cable(fsmsync) {
                        sela=sel2;
                        selb=sel3;
                        clear[0]=clear;
                        clear[1]=clear;
                }
        }
        macrocell(addrdata, datapath) {
                cable(inpads) {
                        extaddr[0,smax]=extaddr[0,smax];
                }
                cable(counter2) {
                        countaddr[0,1] = addrdataNULL[0,1];
                        countaddr[2,smax]=count3[0,smax-2];
                }
                cable(histdata)
                        ramaddr[0,smax]=ramaddr[0,smax];
                cable(fsmsync) {
                        equal=equal;
                        extdelayed[0,1]=extaddrlsb[0,1];
                        extdelayed[2,smax] = addrdataNULL[2,smax];
                }
                cable(ctrlpads)
                        selsrc=vblank;
        }
        macrocell(delay1) {
                cable(fsmsync) {
                        in[0]=holdm1;
                        in[1]=inc1or2;
                }
                cable(timingpla)
                        in[2]=load;
                cable(ctrlpads)
                        in[3]=hblank;
                cable(histdata) {
                        out[0]=holdm2;
                        out[1]=inc1or2m1;
                }
                cable(outpads)
                        out[2]=loadm1;
                cable(timingpla)
                        out[3]=hblankm1;
        }
        macrocell(control) {
        macrocell(fsmsync) {
                cable(fsmsync) {                fsmin[0]=s1;
                                                 fsmin[1]=count; }
                cable(addrdata)                 fsmin[2]=vblank;
                cable(counter2)                 fsmin[3]=stop;
                cable(ctrlpads)                 fsmin[4]=field;
                cable(delay1)                   fsmin[5]=hblankm1;
                cable(fsmsync)                  fsmin[6]=h1;
                cable(addrdata)                     fsmin[7]=equal;
                cable(timingpla) {                  fsmin[8]=hold;
                                                     fsmin[9]=addr0;
                                                     fsmin[10]=addr1;}
                cable(ctrlpads)                     fsmin[11]=read;
                cable(addrdata)                     fsmin[12,13]=extaddrlsb[0,1];
                cable(timingpla)                    fsmin[14]=reset;
                cable(fsmsync) {                    out[0]=s1;
                                                     out[1]=count;
                                                     out[2]=h1;              }
                cable(histdata)                     out[3]=inc1or2;
                cable(ram1)                         out[4]=clear;
                cable(ram1) {                       out[5]=sel0;
                                                     out[6]=sel1;            }
                cable(ram2) {                       out[7]=sel2;
                                                     out[8]=sel3;            }
                cable(delay1)                       out[9]=holdm1;
                cable(histdata)                     out[10]=readm1;
                cable(histdata)         {           out[11]=resetcarry;
                                                     out[12]=resetaccum; }
}
macrocell(timingpla) {
                cable(counter2)
                                fsmin[0,3]=count2[3,0];
                cable(counter1)
                                fsmin[4,6]=count1[2,0];
                cable(timingpla)
                                fsmin[7]=outsrc;
                cable(delay1)
                                out[0]=load;
                cable(outpads) {
                                out[1]=addr0;
                                out[2]=addr1;
                }
                cable(histdata)
                                out[3]=outsrc;
                cable(fsmsync) {
                                out[4]=hold;
                                out[5]=reset;
                }
                cable(outpads)
                                out[6]=addr2;
}
macrocell(counter1) {
                cable(ctrlpads)
                                load=fivemeg;
                cable(fsmsync)
                                'cin*'=count;
                cable(timingpla)
                                out[0,2]=count1[0,2];
                cable(counter2)
                                cout=cout1;
}
macrocell(counter2) {
                cable(counter1)
                                'cin*'=cout1;
                cable(timingpla)
                                out[0,3]=count2[0,3];
                cable(addrdata)
                                out[4,4+counter3width-1] =
                                                count3[0,counter3width-1];
                cable(fsmsync)
                                cout=stop;
}
}
```

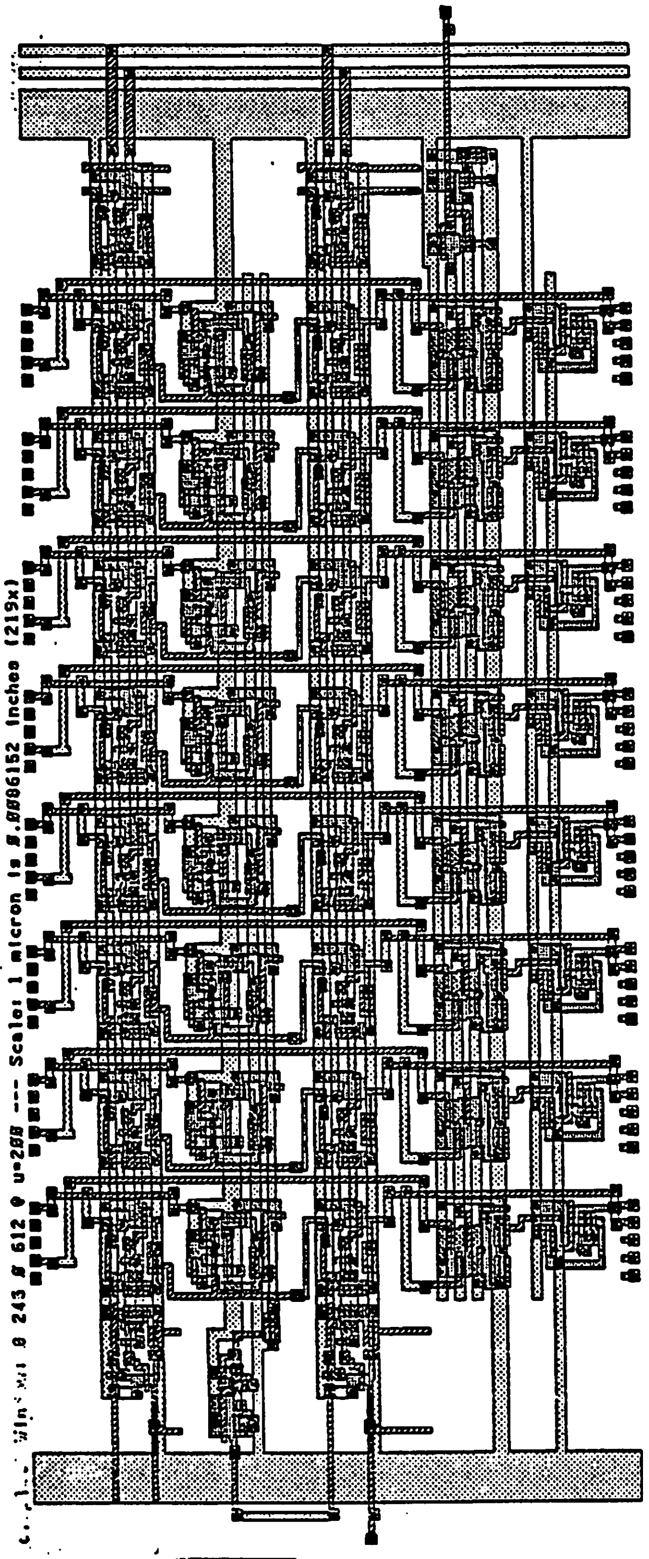
```
}

/* Input pads */
macrocell(inpads)
            cable(addrdata)
                        out[0,smax] = extaddr[0,smax];

/* Output pads */
macrocell(outpads) {
            cable(histdata) {
                        in[0,7]=outdata[0,7];
                        in[12]=saturated;
            }
            cable(delay1)
                        in[8]=loadm1;
            cable(timingpla) {
                        in[9]=addr0;
                        in[10]=addr1;
                        in[11]=addr2;
            }
}
/* Control pads */
            macrocell(ctrlpads) {
                        cable(histdata) {
                                    out[0]=shift1;
                                    out[1]=shift2;
                                    out[2]=shift4;
                                    out[3]=shift4a;
                        }
                        cable(addrdata)
                                    out[4]=vblank;
                        cable(delay1)
                                    out[5]=hblank;
                        cable(fsmsync) {
                                    out[6]=field;
                                    out[7]=read;
                        }
                        cable(counter1)
                                    out[8]=fivemeg;
            }

/* Power, ground, clocks */
            macrocell(power5) { }
            macrocell(ground) { }
            macrocell(clocks) { }
}
```

Appendix F. Layout Examples of the Histogram Processor

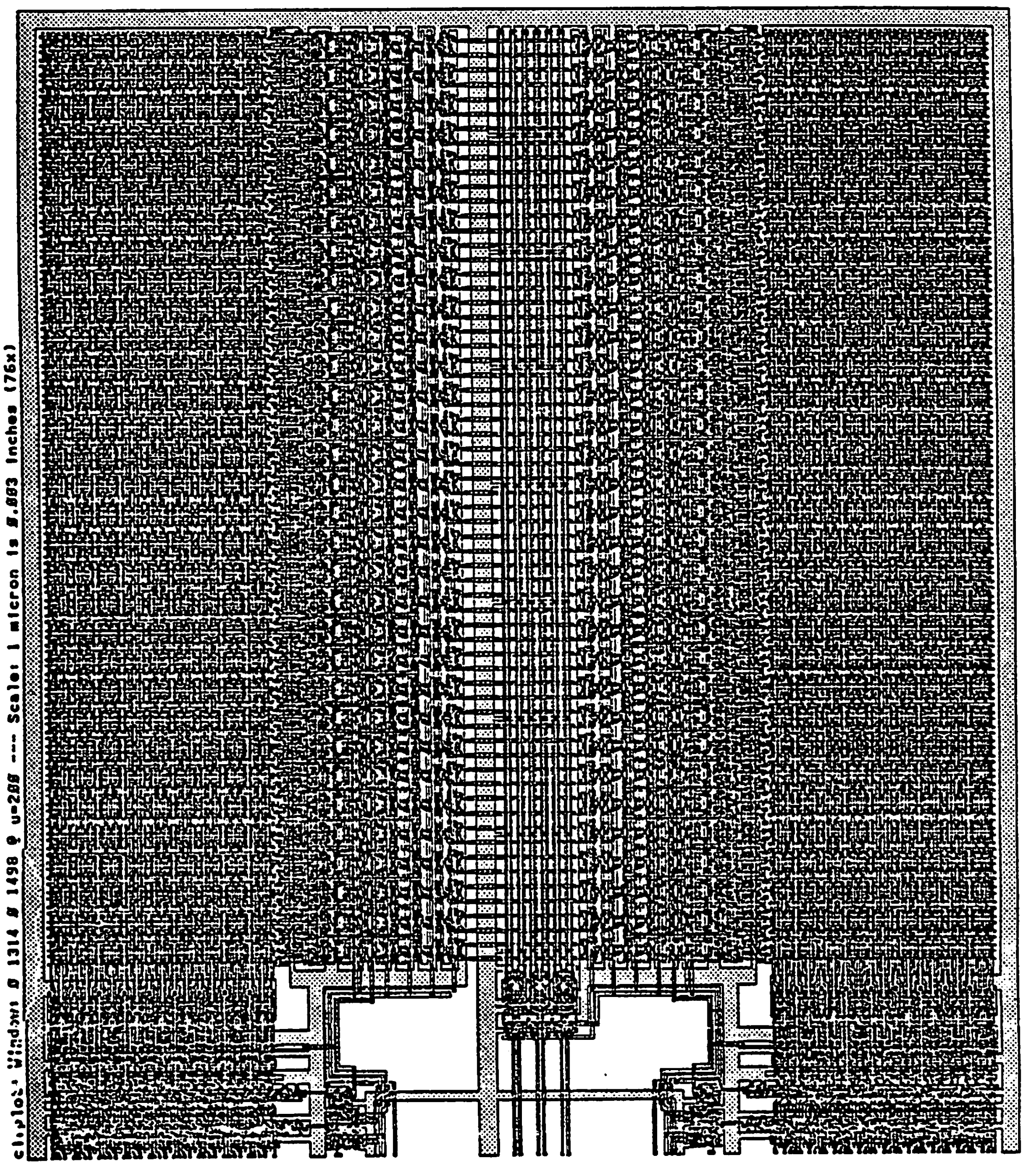
'histdata'	The incrementing and accumulating datapath of the histogram processor. The histogram and post-processing calculations are handled in this macrocell, which includes the output scaling and saturation logic.
'addrdata'	The address source select logic. External or internally generated addresses are directed to the RAM banks and the control logic from this macrocell.
'ram1'	One of two RAM banks. Each RAM bank includes the simultaneous read-write logic. Each macrocell contains two mirrored banks, sharing the row select logic.
'control'	The control macrocell, generated by Flint, consists of two PLA's and two counter macrocells. The timing for the sample rate histogram calculation, and the slower post-processing operations is generated in this section.
'histogram8'	The complete histogram processor, with pads. This version can calculate the histogram of an image with 512 x 512 pixels, with 256 intensity quantization levels (eight bits) per pixel.


cifplot* Window: 0 663 0 1309 0 u=200 --- Scale: 1 micron is 0.00402780 inches (102x)

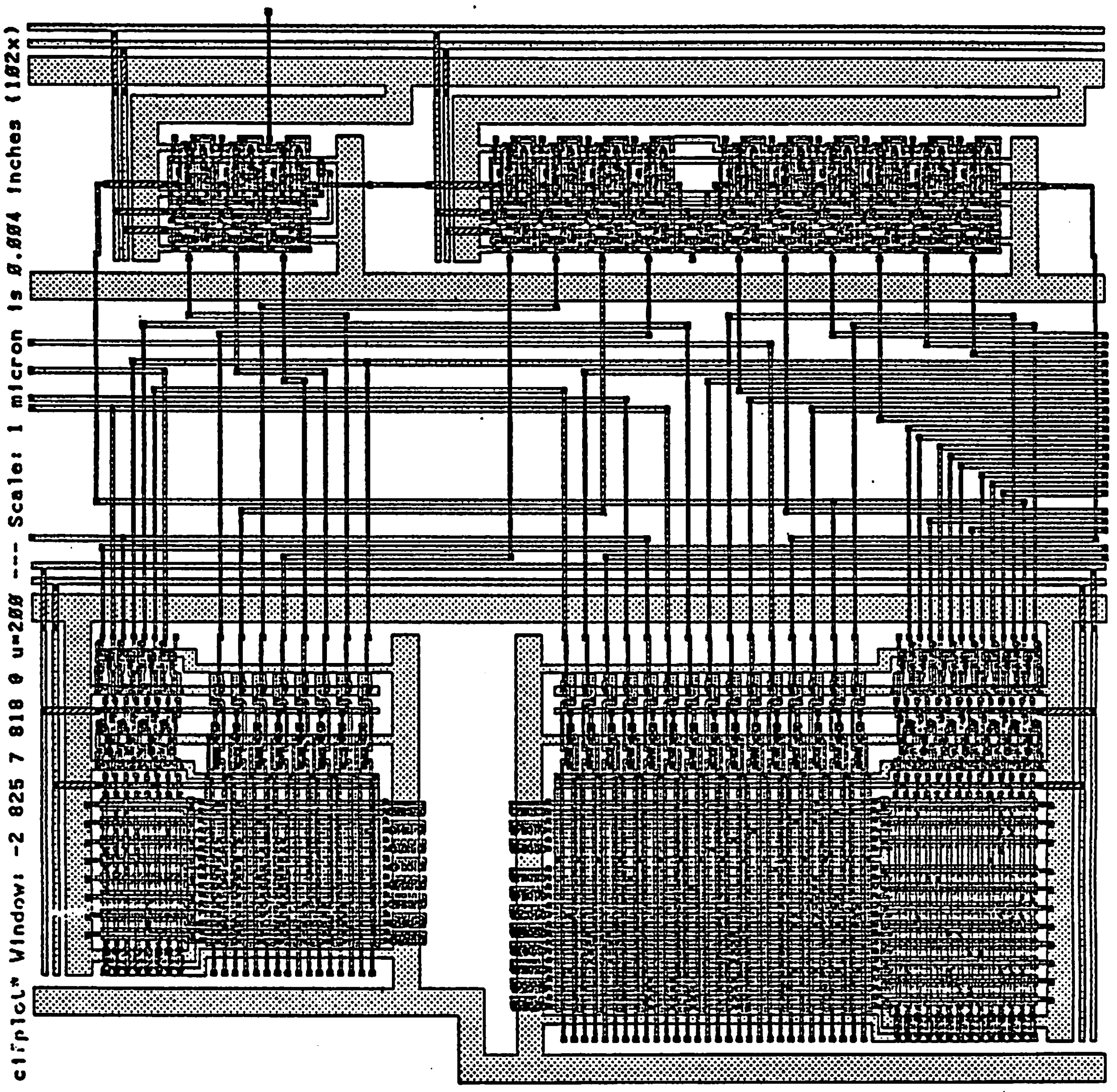
The Histogram Incrementer Datapath, 'histdata'


243 Ø 612 @ u=2ØØ --- Scale: 1 micron is Ø.ØØØ6152 inches (219x)

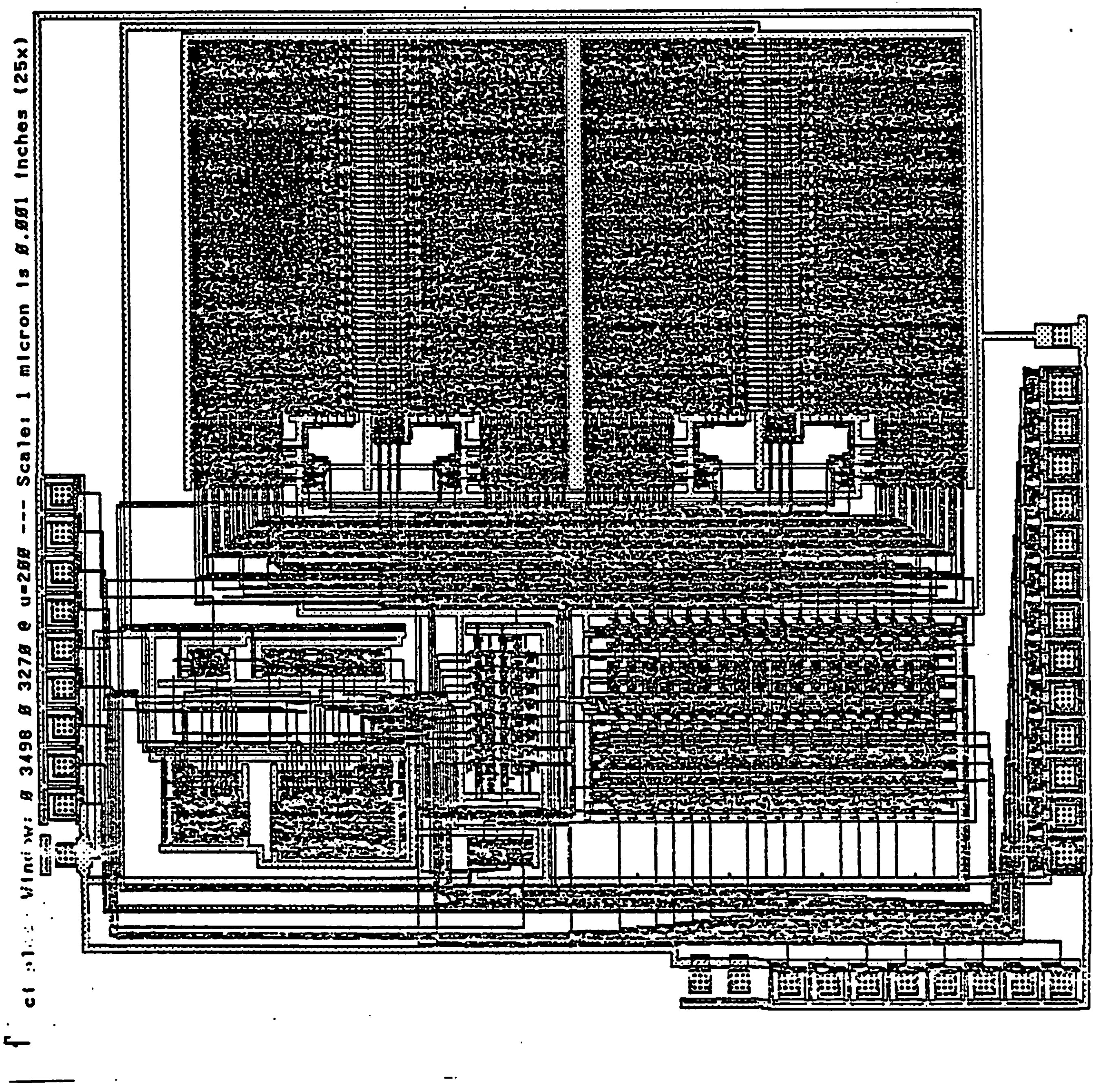
The Address Datapath, 'addrdata'


chipplot Window @ 1314 @ 1498 @ u=200 --- Scale: 1 micron is 0.003 inches (76x)

One of the RAM banks, 'ram_hist'


cifplot* Window: -2 825 7 818 @ u=200 --- Scale: 1 micron is 0.004 inches (102x)

The 'control' block, with two PLA's and two counters.


cif plot Window: 0 3490 0 3270 @ u=200 --- Scale: 1 micron is 0.001 inches (25x)

The Complete Histogram Processor, for a 512 x 512, 8 bit pixel image.