FPGA-based Pedestrian Detection*

Jon Brookshire jbrooksh@csail.mit.edu
Jorgen Steffensen jorgst@stud.ntnu.no
Jianxiong Xiao jxiao@csail.mit.edu

Abstract

For this project, we implemented a computer vision algorithm to detect pedestrians in an image. The algorithm requires many operations to be carried out and is very computationally expensive. To detect pedestrians quickly, we explore the potential parallelism in the algorithm and design an efficient and economical FPGA implementation for the algorithm. Thanks to the convenience of Bluespec, we are able to describe the complicated computation in a simple and intuitive way and also have more time for design optimization. Through careful co-design of both software and hardware, we are able to obtain high throughput performance that is only limited by the memory bandwidth, while keeping chip area as small as possible. Our objective for this project has been successfully met, and we obtain a useful implementation as part of a potential real-time pedestrian detector.

1 Introduction

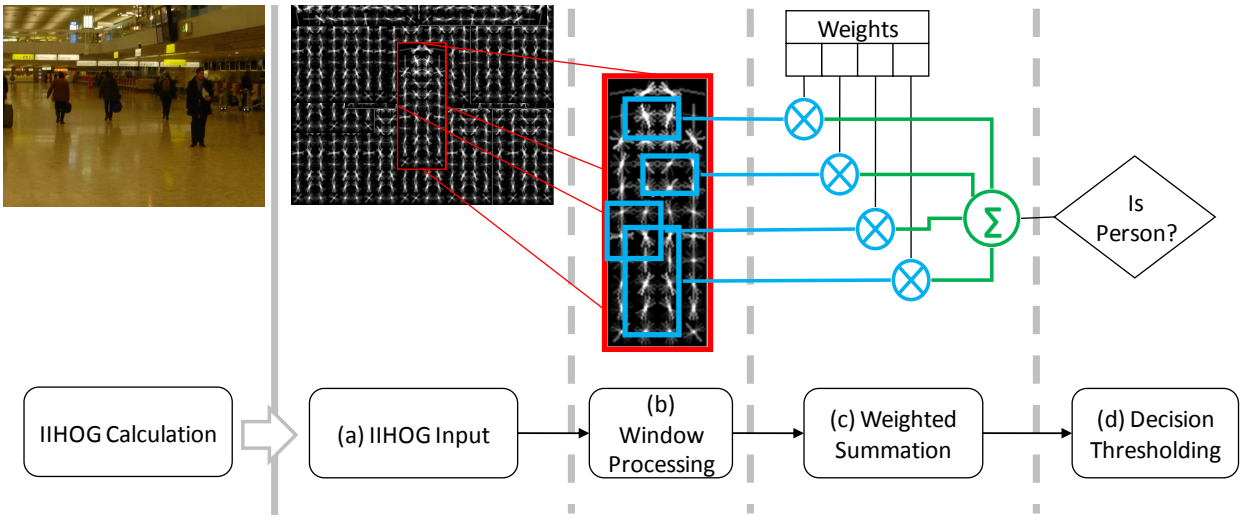

Figure 1: High level algorithm flow chart

The algorithm is outlined in Figure 1. An input image enters on the left and an initial data matrix is constructed. This data matrix is an Integral Image of Histogram of Oriented Gradients

*This work is impossible without the great guidance given by our mentors: Prof. Arvind, Asif Khan, Richard Stephen Uhler.

(IIHOG), and, if the image is 240x320 pixels, then the IIHOG is 240x320x9 32-bit values. The IIHOG is constructed by calculating the directional gradient of every pixel and then accumulating these values in the form of a 9-bin histogram. For this project, we did not implement the IIHOG on the FPGA because (1) the directional gradient requires an arctangent, and (2) since the pixels are accumulated over the entire image, there is little obvious opportunity for parallelism. Instead, the IIHOG is calculated on the on-FPGA PowerPC, and the PowerPC controls the window scanning in Step (a). Our FPGA implementation performs Step (b) and acts as an accelerator for the majority of the computation. The results from the Step (b) computations are returned to the PowerPC and the entire window is classified. In Step (b), each detection window is evaluated by considering a number of sub-regions, or blocks (blue). Evaluation of each block requires 81 32-bit memory accesses, over 100 additions, a 36 element L2 normalization (square, sum, and square root), a 36 element dot product, and a threshold comparison. Each of these blocks can be evaluated independently of the others and, as a result, can be executed in parallel.

In Step (c), the results from the blocks are weighted and summed in the PowerPC. Depending on the threshold in Step (d), more blocks may be evaluated (loop back to Step (b)) or the window is classified.

2 Background

The algorithm works by learning a set of linear Support Vector Machines (SVMs) trained on positive (pedestrian) and negative (non-pedestrian) training images. This off-line learning process generates a set of SVMs, weights, and image regions that can then be used on-line to classify an unknown image as either positive or negative.

2.1 The Features

As with many computer vision applications, part of the challenge is to find a descriptive set of features. If the feature space is rich enough - that is, it provides sufficient information to identify targets - these features can be combined with machine learning algorithms to classify targets and non-targets. The work done in [2] demonstrated success using HOG features for pedestrian detection. A single HOG is a way to encode local gradients. In this process, the gradient is first calculated for each pixel. Next, the training image (see Figure 2) is divided into a number of sub-windows, often referred to as "blocks". The blocks span size from 8 to 64 pixels, have various length-to-width ratios, and densely cover the image (i.e., overlap). Each block is divided into quadrants and the HOG of each quadrant is calculated. A HOG is a histogram with nine evenly spaced bins for orientation into which the gradients vote (nine bins was suggested in [1] as being the most effective for classifying pedestrians). Thus, each quadrant produces nine features for a total of 36 features per block.

The algorithm learns what defines a pedestrian by examining a series of positive and negative training images. Because this process is time consuming and does not need to be repeated, it is performed off-line. Although this off-line process is not part of this project, we give a brief discussion here to motivate the system architecture discussion. During this learning process, blocks are randomly selected (e.g., see Figure 2, right) and evaluated for their ability to distinguish positive and negative images. In general, a single block will not be sufficient to classify positive and negative images successfully. Instead, these "weak" block classifiers can be combined to form stronger classifiers. The AdaBoost algorithm [3] provides a statistically founded means to choose and weight

Figure 2: A typical pedestrian training image (left) and its gradient (right)

a set of weak classifiers. The algorithm repeatedly selects weak classifiers and weights and sums the score from each classifier into an overall score.

The algorithm also uses a rejection cascade. The rejection cascade leverages the observation that the majority of detection windows in any image will not contain pedestrians. Consider, for example, an image with a single pedestrian in it. At all the thousands of positions and scales, only a handful of detection windows will contain the pedestrian. Instead of building one AdaBoost classifier, we build several AdaBoost classifiers. Each classifier, or "stage," in the rejection cascade is trained on increasingly difficult data. If a detection window is classified as negative at any stage, it is overall classified as negative. Although each stage must have a very low false negative rate, there are considerable time savings.

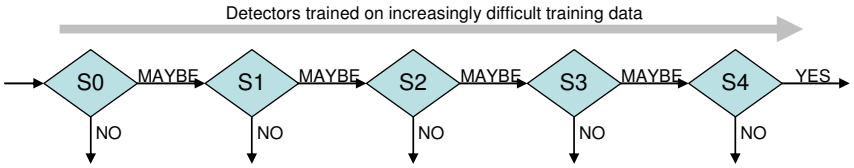

Figure 3: The rejection cascade is a series of stages such that the earlier stages classify "easier" images and the later stages classify "harder" images. Because the earlier stages are trained to handle the more ubiquitous easy windows, more time can be spent processing the harder windows.

In summary, there will be a number of blocks to evaluate for each detection window. These blocks have several learned parameters associated with them, which are fixed for the purposes of this project. The PowerPC is responsible for submitting the appropriate blocks for evaluation.

3 System Architecture

3.1 Algorithm Overview

Algorithms 1 and 2 describe the system execution. In a software-only implementation, both algorithms would be executed by the CPU. In our accelerated implementation, the FPGA is responsible for the block processing in Algorithm 2, and the CPU executes Algorithm 1. In the following

sections, we provide detail describing the implementation of the accelerated design. The function $CalculateHOGFeatureVector()$ creates the 36D feature vector and is detailed in Section 3.6.1.

Algorithm 1 Detect Pedestrians in Image

```
1: Load learned parameters
2: Load image from bitmap file
3: Create IIHOG
4: for each scale factor do
5:    for each (row, column) every 12 pixels do
6:       for each level in the rejection cascade do
7:          score ← 0
8:          for each block in the level do
9:             score ← score + EvaluateBlock() * blockWeight
10:         end for
11:         if score < 0 then
12:            REJECT window (i.e., not a pedestrian)
13:         end if
14:      end for
15:      if not yet rejected then
16:         ACCEPT window (i.e., is a pedestrian)
17:      end if
18:   end for
19: end for
```

Algorithm 2 EvaluateBlock

```
1: v ← CalculateHOGFeatureVector()
2: v̂ ← v/|v|
3: dotProduct ← v̂^T * blockVector
4: if dotProduct > blockBias then
5:    return +1
6: else
7:    return −1
8: end if
```

Figure 4 shows the high-level system architecture. The PowerPC shares 256MB of DDR memory with our Block Accelerator and contains the IIHOG data. We utilize the Xilinx Multi-Port Memory Controller (MPMC) and an interface based on one provided by the TA's. The DDR is also used to contain the instruction, stack, and heap memories. The PowerPC submits job requests to and receives responses from the Block Accelerator via its Processor Local Bus (PLB).

3.2 Software Operations

The PowerPC executes C++ code which is responsible for reading images from the on-board CompactFlash and converting them into the IIHOG data matrix (see Algorithm 1). The PowerPC then orchestrates the detection process by selecting detection windows. A detection window is a 128x64 pixel region of the image. These regions are scanned densely across the image every 12

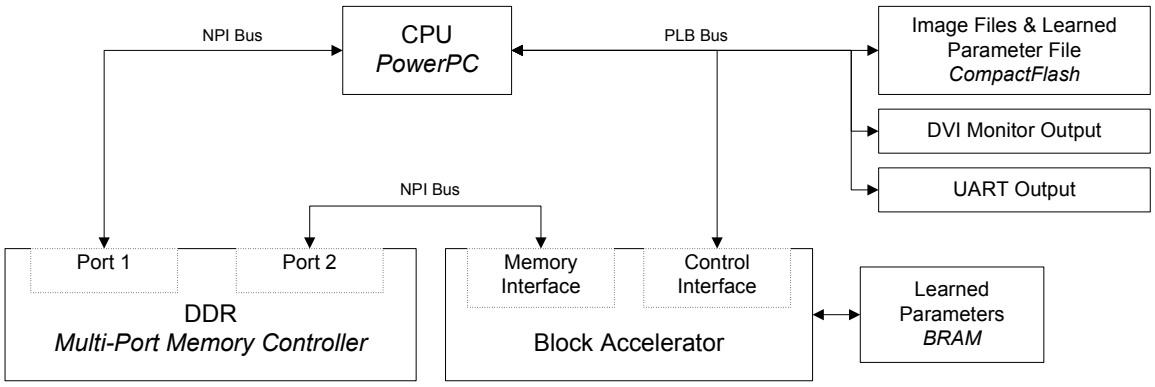

Figure 4: System bus architecture

pixels along the image rows and columns. Additionally, these windows are scanned at various image scales (i.e., the windows are "zoomed in" and "zoomed out" of the image to detect pedestrians at different distances). For each detection window, a number of sub-regions, or "blocks" are considered (see Algorithm 2). It is the evaluation of these blocks that our system accelerates via the Block Accelerator.

Once the Block Accelerator has completed its calculations, it returns either a +1 or -1 response to the PowerPC. The PowerPC performs a weighted summation and a threshold comparison to determine if more calculations are needed.

3.3 Host CPU Interface

Our Block Accelerator interfaces to the PowerPC via four port addresses, accessible via the PLB.

1. Debug port. This port has a FIFO writable from the PowerPC and a FIFO readable from the PowerPC. This port is used to monitor signals of interest during debugging and verify correction operation.

2. Processing port. This port also has an input and output FIFO. The input FIFO is 64 elements long, allowing the PowerPC to queue several block evaluations at a time. Similarly, the output FIFO is 8 elements long, allowing the accelerator to continue to process requests before they have been read by the PowerPC. The input FIFO accepts data from the PowerPC in the form of a single command word, followed by multiple data words. The command can be either `START_EVALUATION` or `WRITE_PARAMS`. The `START_EVALUATION` command is followed by nine parameters which configure the Block Accelerator to evaluate a particular block. The `WRITE_PARAMS` command is followed by address and data words which allow the PowerPC to write to the BRAM which contains the learned parameters (e.g., the $blockBias$, $blockVector$, $blockWeight$ as shown in Algorithms 1 and 2).

3. Base Address port. This register port allows the PowerPC to set the address of the IIHOG in shared memory. Our Block Accelerator and the PowerPC share the entire DDR memory space. Note that the Block Accelerator never writes to the memory, so there are no concurrent write issues.

4. Row size port. This register port allows the PowerPC to specify the image width. The IIHOG is a three dimensional data matrix, stored in a linear memory array. As the Block Accelerator accesses matrix coordinates, it needs the width of the image to calculate the memory addresses.

3.4 Corner And Memory Address Calculation

The IIHOG three-dimensional data matrix stored in the linear memory array is accessed with one-dimensional addresses calculated from the corresponding (row, column) pairs. The (row, column) dimension-reduction consists of a multiplication between the row and the image width followed by addition with the column (e.g. if the width is 320, then the pairs (0,1) and (5,21) are mapped to 1 and 1621 respectively).

$$\begin{pmatrix} (0,0) & (0,1) & \dots & (0,319) \\ (1,0) & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ (239,0) & \cdots & \dots & (239,319) \end{pmatrix} \rightarrow \begin{pmatrix} 0_{(0,0)} & 1_{(0,1)} & \dots & 319_{(0,319)} \\ 320_{(1,0)} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ 76480_{(239,0)} & \cdots & \dots & 76799_{(239,319)} \end{pmatrix} \quad (1)$$

The next dimension-reduction is done in the same fashion; the combined row-column value is multiplied by 9 to get the first 32-bits address that access 8 bytes (two int32's) in the memory. The first address is incremented by 8 four times to create the next four 32-bits addresses. These calculations are implemented as a 2-stage pipeline that is separated with a FIFO between the four increments and the multiplication by 9. For each value from the first stage, the second stage iterates through five states and outputs five addresses. Even though we want to access nine 32-bits values in the memory per pair only five addresses are need since each address access two 32-bits values in the memory. Five addresses implies that ten 32-bits values will be accessed, thus one of them needs to be discarded. Two writeEnable bits are added to tell the memory interface which of the two 32-bits values are wanted. The writeEnable has three options, first, last and both. The second, third and fourth address will always have writeEnable set to both, but the edges depends on each other. If the first address is even then it has writeEnable set to both, and the last address will discard the last of the two values (writeEnable is low). If the first address is odd (writeEnable is high) then the first value will be discarded, and the last address will use both of the values (writeEnable is both). This dependencies is carried out using the fact that if the first address is odd then the last address is odd as well, this is illustrated in table 1 .The first stage is implemented using two pipelined 32-bits multipliers and one adder.

first address (writeEnable)	last address (writeEnable)	action
even (both)	even (low)	use all but the last word
odd (high)	odd (both)	use all but the first word

Table 1: The function for writeEnable-bits in addresses

The input for the memory address calculator is the (row, column) pair we wish to look up in the memory. The pairs are calculated using the parameters (scale, row, column) for the current detection block and the configuration parameters (width, height, row offset, and column offset).

The row and column parameters for the block are relative to the detection window, thus the row and column values relative to the image have to be calculated before it is sent to the address calculator. A corner calculator is implemented to add the block offset and adjust each parameter for scale. The scaled width and height are the actual width and height of the detection block. The new row and column values are the coordinate of the origin of the detection block (point A) relative the image origin. These four values are used to calculate the eight other points (B, C, D, E, F, G, H, I) used for detection, see figure (7). The corner calculator is implemented using a 2-stage pipeline that is separated by four FIFOs holding the new row, column, width and height values after scaling and before the calculation of the eight points. The second stage outputs (sequentially) nine row and column pairs for every set of four input values. The first stage is implemented using one pipelined 32-bit multiplier in addition to one adder, and the second stage is implemented using one adder in addition to one shifter (divide by 2).

The vertical folded pipelined technique used for both of the modules is motivated by the limitation of the DDR access. Since the DDR interface only allows one input at a time, and the address calculator outputs $9 \cdot 5 = 45$ addresses for each set of parameters, there are no need for high speed calculations. Thus, the focus for these modules has been to limit the area. The block diagram for the two modules are given below.

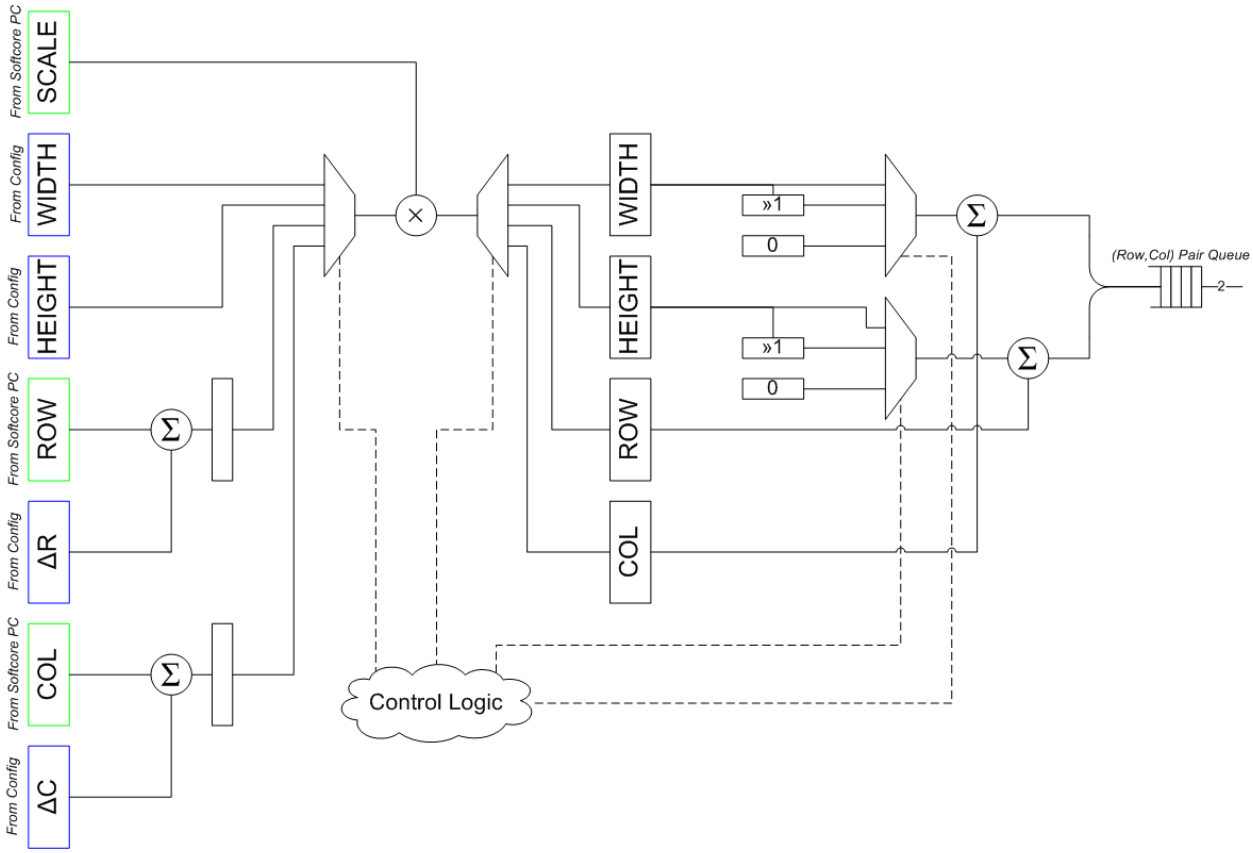

Figure 5: The Corner Calculator calculates the block offset from the image origin, adjusts for scale, and pushes the nine quadrant corners (corners and midpoints) into the "(Row, Col) Pair Queue."

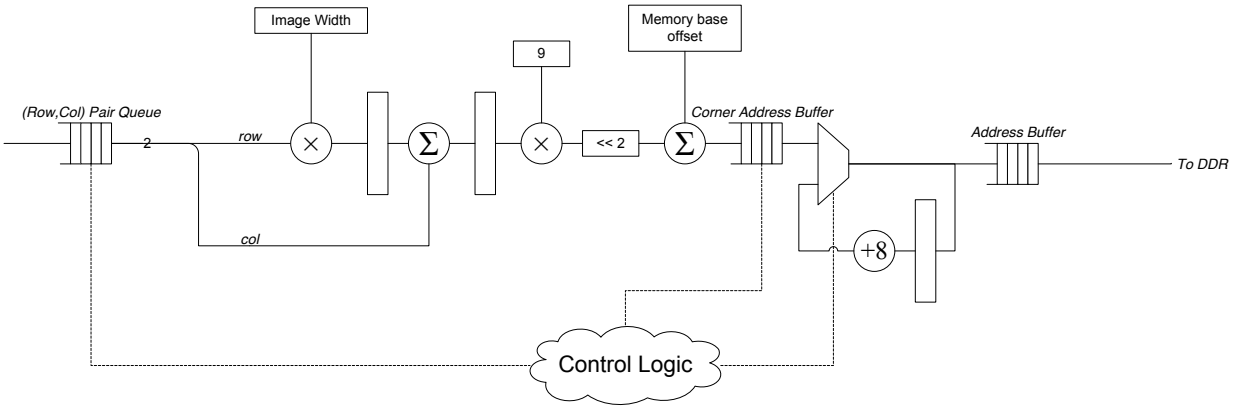

Figure 6: The Address Calculator takes nine (row, col) pairs as input from the "(Row, Col) Pair Queue." It converts each pair into a linear index into the memory array. Each pair corresponds to nine values (int32's) in the memory, so it then pushes the calculated address and the 4 successive addresses.

3.5 DDR Shared Memory

The Xilinx MPMC provides a multi-port memory model for accessing the DDR. As a result, our custom glue logic (written in Verilog and VHDL) interfaces to a "virtual" memory and can ignore the complexities of arbitrating memory requests with the PowerPC. The glue logic makes the DDR appear as three FIFO's in Bluespec. Specifically, it connects a read/write address request FIFO, write data FIFO, and a read data response FIFO. We based our Verilog/VHDL on code provided by the TA's, and corrected a bug that could cause data to be lost (a FIFO ready signal was not properly monitored).

The DDR is accessed via a 32-bit address bus and a (virtual) 64-bit data bus. Since we access values only on a 32-bit boundary, we must occasionally discard the high or low words of a read result. As discussed in Section 3.4, the address calculation routines provide a writeEnable mask which define which words (high, low, or both) should be queued to the feature extractor. These values are simply kept in a FIFO in parallel with the memory read requests. When a response is read, only the selected words are written to the feature extractor FIFO.

3.6 Block Processing

The function of Block Processing is to obtain 81 fixed-point values from the memory, and produce one boolean result. Therefore, it has the following interface:

```
typedef Server#(HOGelem, Bool) BlockProcessor;
```

After we read the integral value of the HOGs from the memory, we have to:

1. Compute the four 9-dimensional feature vectors $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4$ using the integral HOG value, and concatenate the feature vectors into one 36-dimensional vector $\mathbf{v}$.
2. Normalize $\mathbf{v}$ by $\mathbf{v} \leftarrow \mathbf{v}/|\mathbf{v}|$.

3. Compute the dot product $\mathbf{v} \cdot \mathbf{w} + bias$.

4. Report the result as positive or negative.

Therefore, it is equivalent to know if $(\mathbf{v}/|\mathbf{v}|) \cdot \mathbf{w} + bias > 0$. To avoid doing 36 element-wise division, we can transform the algorithm in an equivalent way, i.e., $\mathbf{v} \cdot \mathbf{w} + bias * |\mathbf{v}| > 0$. In this way, we only need one scalar multiplication instead of 36 divisions. Therefore, our algorithm contains three steps:

1. Compute the four 9-dimensional feature vectors $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4$ using the integral HOG value.

2. Compute $\mathbf{v} \cdot \mathbf{w}$, and compute $\mathbf{v} \cdot \mathbf{v}$.

3. Compute the square root of $\mathbf{v} \cdot \mathbf{v}$ to obtain $|\mathbf{v}|$, and test $\mathbf{v} \cdot \mathbf{w} + bias * |\mathbf{v}| > 0$.

Therefore, we divide the computation into the following three modules: FeatureExtractor, DotProduct, and Scorer.

3.6.1 Feature Extractor

The function of Feature Extractor is to obtain 81 fixed-point values from the memory, and produce 36 fixed-point values for the feature vector into the next stages. Therefore, it has the following interface:

typedef Server#(HOGelem, HOGelem) FeatureExtractor;

For each block, let A, B, C, D, E, F, G, H, I denote the values in the 9 corners of the integral HOG, where each of them denotes a 9-dimensional vector of fixed-point values. Note that these "corners" represent a block (region of pixels to be evaluated) within a detection window (a region of pixels being scanned across the image).

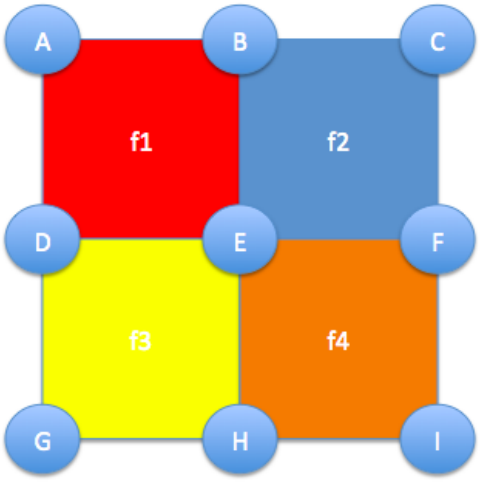

Figure 7: Corner Annotation

The IIHOG allows us to efficiently compute the sum of all values in R1, R2, R3, and R4. These sums become the four feature vectors and are defined as:

$feature1 = E - D - B + A$

$feature2 = H - E - G + D$

$feature3 = F - E - C + B$

$feature4 = I - H - F + E$

To reuse the computation result, we define $Z = E - D$ and $Y = F - E$. Therefore, the new computation requirement is:

$feature1 = Z - B + A$

$feature2 = H - Z - G$

$feature3 = Y - C + B$

$feature4 = I - H - Y$

Because the data from memory is bandwidth limited (and values arrive only 64-bits at a time), these feature values cannot be computed in parallel; therefore, we compute them sequentially. The resulting pipeline is deep and contains many stages, but this is not an issue because we do not have feedback from later stages.

The task of the Feature Extractor, then, is to accept the 32 bit values from memory, compute the 9-element feature vectors, and queue the 36 element concatenated result in the output queue. A naive implementation could store all values from the DDR memory in FPGA RAM, iteratively perform the calculations shown above, and then queue the 36 element output. However, because many of A-I values are used only once, we can conserve FPGA resources by using several of the values as they arrive from the FPGA and immediately forgetting them. Doing so requires that the memory be accessed in a specific order.

Input	Round	sum0		sum1		Next pipeline stage
E	0					
D	1	Z=E-D	[D bed]			
B	2	ZmB =Z-B				
A	3	fea1 = ZmB+A	[A bed] [ZmB end]			featureQ.enq(fea1);
H	4	HmZ = H-Z	[Z end]			
F	5	Y = F-E	[F bed] [E end]			
G	6	fea2 = HmZ-G	[G bed] [Hmz end]	YaB = Y+B	[B end]	featureQ.enq(fea2);
C	7	fea3 = YaB-C	[C bed] [YaB end]	HaY = Y+H	[Y end] [H end]	featureQ.enq(fea3);
I	8	fea4 = I - HaY	[I bed] [HaY end]			featureQ.enq(fea4);

Table 2: Computation Sequences in mkFeatureExtractor.

The data from the DDR arrives in the order shown in Table 2 (top to down), one fixed-point number each cycle. (Recall there are 9 fixed-point numbers in each A-I). sum0 and sum1 are two hardware adders, and the equation in the corresponding column is the computation performed. In the sum0 and sum1 columns, there are also marks indicating the end of usage for particular values. For example, [E end] means that the values of E are no longer useful after this computation and can be safely thrown away. "bed" is short for "begin and end" and means that the value arrived from the DDR in the current clock cycle, was used, and was useless after this cycle. As a result, for variables denoted as [bed], we do not need to keep them in any storage at all. In the column "next pipeline stage", we indicate when the feature vector is computed and can be pushed into the output queue, featureQ.

With this careful design and arrangement, we require only two adders and very few registers, while maintaining the same throughput with the memory. However, the combinational complexity of this block required additional pipelining and proved to be the slowest part of our design (see Section 4.3).

3.6.2 DotProduct

The function of DotProduct is to obtain 36 fixed-point values from the Feature Extractor, and produce two dot-product results, $\mathbf{v} \cdot \mathbf{w}$ and $\mathbf{v} \cdot \mathbf{v}$. Therefore, it has the following interface:

```
typedef Server#(HOGelem, Tuple2#(HOGprodHD64,HOGprodHD64)) DotProduct;
```

This module contains two accumulators, acc_{vv} and acc_{vw}, and two multipliers. The initial values are both zeros. Each time a new value v_i comes in, it computes $v_i * v_i$ and $v_i * w_i$, and increases the accumulators, $acc_{vv} \leftarrow acc_{vv} + v_i * v_i$ and $acc_{vw} \leftarrow acc_{vw} + v_i * w_i$. After 36 values arrive, it outputs the values acc_{vv} and acc_{vw}, and clears the registers, $acc_{vv} \leftarrow 0$ and $acc_{vw} \leftarrow 0$.

At each cycle, we expect one input value and we need to perform two multiplications. We could use two pipelined multipliers, but the memory access and Feature Extractor take 81 cycles to produce the 36 values for this module. This module requires only 72 multiplication operations. Therefore, to maintain the throughput, we actually only need one multiplier. In each cycle, as a new value comes in, we push v_i and v_i into the multiplier, and request to read w_i. When w_i is read in the following few cycles, w_i and v_i are pushed into the multiplier. We use a boolean queue to keep track of which values are pushed into the multiplier. When the result from the multiplier emerges, the module dequeues and accumulates the result appropriately.

3.6.3 Scorer

The function of the Scorer, as shown in Figure 8, is to obtain the two dot-product results, $\mathbf{v} \cdot \mathbf{w}$ and $\mathbf{v} \cdot \mathbf{v}$, from the DotProduct module and produce one boolean value $\mathbf{v} \cdot \mathbf{w} + bias * \sqrt{\mathbf{v} \cdot \mathbf{v}} > 0$. Therefore, it has the following interface:

```
typedef Server#(Tuple2#(HOGprodHD64,HOGprodHD64), Bool) Scorer;
```

The operation of this step is:

1. Compute the magnitude of $\mathbf{v}$, i.e., the square root of $\mathbf{v} \cdot \mathbf{v}$.
2. Compute a multiplication of *bias* and the result of previous step.
3. Compute the sum of $\mathbf{v} \cdot \mathbf{w}$ and the result of previous step.
4. Compare the result of the previous step and zero to decide output as true or false.

In terms of throughput, every set of 81 fixed-point values from memory only needs one square root, and one multiplication with *bias*. Therefore, we do not need a pipelined square root or multiplier. We just need to make sure that everything is computed within 81 cycles. Hence, we implement a 32-cycle square root module and a multi-cycle multiplier.

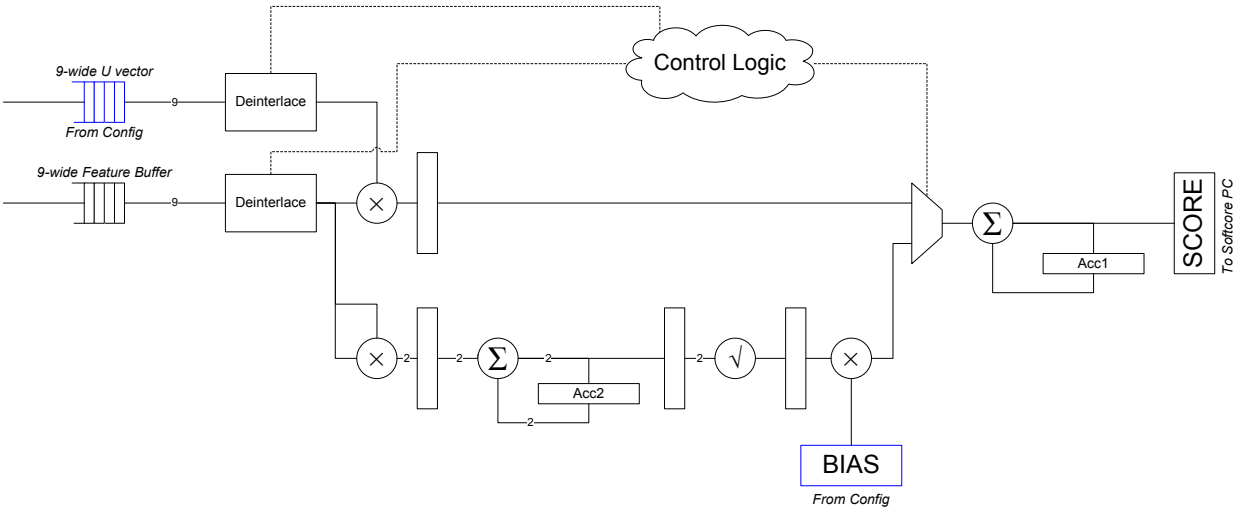

Figure 8: The scorer computes the dot product and L2-normalization of the 36D feature vector.

3.6.4 Square Root Implementation

The function of Square Root is to accept a 64-bit fixed-point value and produce a 32-bit fixed point value. Therefore, it has the following interface:

```
typedef Server#(Bit#(64), Bit#(32)) SquareRooter;
```

There are many different algorithms for hardware square root [5, 6]. We choose a new non-restoring square root algorithm that is very efficient to implement [5, 6][1]. This algorithm has the following features, unlike other square root algorithms: First, it only requires one traditional adder/subtractor in each iteration, i.e., it does not require other hardware components, such as seed generators, multipliers, or even multiplexers. Second, it generates the correct resulting value even in the last bit position.

[1] The algorithm diagram in [6] has incorrect sign in the algorithm pseudo-code.

Algorithm 3 Pseudo-code for 64-bit Square Root

```
Let:
    D be the 64-bit fixed point number
    Q be the 32-bit fixed point number (Result)
    R be the 33-bit fixed point umber (R=D-Q^2)
Algorithm:
Q = 0;
R = 0;
for i = 32 to 0 do
    Rprev = R
    R = ( R<<2 ) || ( D >> (i<<1) & 3)
    Qs2 = Q<<2;
    if (Rprev >=0)
        R = R - (Qs2 | 1)
    else
        R = R + (Qs2 | 3)
    Qs1 = Q << 1
    if (R >=0)
        Q = Qs1 | 1
    else
        Q = Qs1 | 0
```

4 Design Verification and Exploration

4.1 Test Bench

A Bluespec test bench was implemented to verify the full system in BlueSim. It reads in parameter values from a binary file created from the PC-based implementation of the algorithm. The DDR memory is simulated using a hex file, a RegFile, and a few simple rules. Each module in the system uses the Bluespec Get/Put interface, making the module connection particularly clean with the built in Bluespec function mkConnection(). The built in Bluespec state-machine (Stmt) function is used to set-up the test sequence and is outlined in Algorithm 4. We inserted what we termed "sniffers" in between most modules to verify the output of the module and, then, pass the value to the next module. In this way, we could find bugs early in the processing chain.

To simulate the DDR, we created a rule to fire whenever the address buffer has memory read requests. This rule simulates the DDR access behavior as outlined in Algorithm 5.

At the end of the pipeline a "score sniffer" is placed to compare the final result with the "truth" values from the binary file extracted from the original C++ code. This sniffer also store the values

Algorithm 4 Test sequence

```
Load parameter sets
for each parameter set do
    for each parameter in parameter set do
        if block parameter then
            Put parameter in Corner Calculator
        end if
        if parameter is level number then
            Put parameter on Level Queue
        end if
        if parameter is block number then
            Put parameter on Block Queue
        end if
    end for
end for
End of file
```

Algorithm 5 Request memory rule

```
if writeEnable == low then
    Write the low word to the Feature Extractor
end if
if writeEnable == high then
    Write the high word to the Feature Extractor
end if
if writeEnable == both then
    Write the low word to the Feature Extractor
    {wait for next cycle}
    Write the high word to the Feature Extractor
end if
```

in a register, such that at the end of the simulation the total number of True/False values are displayed.

Pipeline Trace Report

The performance of the design was analyzed after it was verified for correctness. We used a pipeline tracer to get an overview of the modules that where operating (rules firing) at different cycles. Dead cycles are quickly observed in the trace report and it gives an indication of where in the design bottlenecks may be found. Below is a small part of the trace report. This trace report shows only 13 cycles, but we can see that for every cycle the DDR outputs a value. As a result, the post DDR module is able to feed the new addresses every cycle. The trace report in the Appendix A shows all the cycles from the beginning of the simulation until the first score value is outputted.

```
athena% bsv-trace.pl jxiao_trace.cfg err.txt
 CYC:  193  [DDR:1 .  ]  [ 5. 1- 4. 8]  [Cout_MinFW|MoutFF]  [       | |   31|             |        ]
 CYC:  194  [DDR:0 .3 ]  [ 5. 2- 5. 0]  [Cout_MinFW|MoutFF]  [       | |   30|             |        ]
 CYC:  195  [DDR:1 .  ]  [ 5. 3- 5. 1]  [          |MoutFW]  [       | |   29|             |        ]
 CYC:  196  [DDR:0 .3 ]  [ 5. 4- 5. 2]  [          |MoutFF]  [       | |   28|             |        ]
 CYC:  197  [DDR:1 .  ]  [ 5. 5- 5. 3]  [          |MoutFW]  [       | |   27|             |        ]
 CYC:  198  [DDR:0 .3 ]  [ 5. 6- 5. 4]  [          |MoutFF]  [       | |   26|             |        ]
 CYC:  199  [DDR:1 .  ]  [ 5. 7- 5. 5]  [          |MoutFW]  [       | |   25|             |        ]
 CYC:  200  [DDR:0 .3 ]  [ 5. 8- 5. 6]  [          |MoutFF]  [       | |   24|             |        ]
 CYC:  201  [DDR:1 .  ]  [ 6. 0- 5. 7]  [          |MoutFW]  [       | |   23|             |        ]
 CYC:  202  [DDR:0 .3 ]  [ 6. 1- 5. 8]  [          |MoutFF]  [       | |   22|             |        ]
 CYC:  203  [DDR:1 .  ]  [ 6. 2- 6. 0]  [          |MoutFW]  [       | |   21|             |        ]
 CYC:  204  [DDR:0 .3 ]  [ 6. 3- 6. 1]  [Cin__MinFF|MoutFF]  [       | |   20|             |        ]
 CYC:  205  [DDR:1 .  ]  [ 6. 4- 6. 2]  [Cin__MinFF|MoutFW]  [       | |   19|             |        ]
```

4.2 Design Exploration

The initial design used about 1.9 million cycles to complete the test sequence, with numerous dead cycles, as expected. The first design was made up of connected modules that was not pipelined and the full system had to output the score before it was ready for new inputs. After pipelining the modules and adding streaming capabilities to the test bench, the system completed the test sequence after about 1.3 million cycles. At this point of the design process, the tracer was introduced to the simulation as discussed in Section 4.1. By looking at the pipeline trace report, we were able to detect dead cycles and in which modules they might have been introduced. It turned out that the multipliers used did not behave as one would expect from the description. The multipliers used were not pipelined and had to be modified. The modification was done using wrappers around the original multiplier. This increased the speed dramatically, and the system was down to only use 358 838 cycles to complete the test sequence[2].

At this point the pipeline trace report showed that there was one dead cycle between every set of 9 output values from the DDR memory. The origin of the dead cycle was a register in the address calculator storing the column value. This dead cycle was removed by adding an extra FIFO that stores the column value in exchange of the register initially used. As expected this reduced the total number of cycles with about 14% down to 309,258 cycles.

[2]The number of cycles is a valid for the performance comparison since the critical path did not change during optimization.

4.3 Synthesis

Figure 9 shows the resource usage results from our system. The first three columns indicate the resource usage for the Block Accelerator alone at three different stages of our design; the last column shows utilization for the entire, final design. As we optimized our design for speed, the most noticeable increase was in the number of DSP48 blocks used. We designed our multipliers with help from the TA's and based on an Internet design [4]. As described earlier, many of our multiplications have 32-bit inputs and 64-bit outputs. Given the large number of bits, we were initially concerned about space, and used multipliers that took several cycles to complete, but used fewer DSP48s. After our first synthesis, we realized we had plenty of DSP48s, and revised our design to perform more concurrent multiplications.

	Block Accelerator Only						Entire Design	
	Initial Design		First Optimizations		Best Optimizations		Best Optimizations	
Slice Logic Utilization	Used	Utilization	Used	Utilization	Used	Utilization	Used	Utilization
Number of Slice Registers	4684	10.0%	4748	10.0%	4611	10.0%	10220	22.0%
Number used as Flip Flops	4684		4748		4611		10220	
Number of Slice LUTs	6547	14.0%	7646	17.0%	6848	15.0%	11204	25.0%
Number used as logic	6485	14.0%	6766	15.0%	6078	13.0%	10279	22.0%
Number used as Memory	54	1.0%	875	6.0%	764	5.0%	893	6.0%
Number used as Dual Port RAM	48		607		501		499	
Number used as Shift Register	6		268		263		394	
Number used as exclusive route-thru	8		5		6		32	
Number of route-thrus	140	1.0%	48	1.0%	53	1.0%	222	1.0%
Number of occupied Slices	2098	18.0%	2378	21.0%	2184	19.0%	5243	46.0%
Number of occupied SLICEMs	18	1.0%	228	6.0%	199	6.0%	351	10.0%
Number of LUT Flip Flop pairs used	7321		8262		7579		14649	
Number with an unused Flip Flop	2637	36.0%	3514	42.0%	2968	39.0%	4429	30.0%
Number with an unused LUT	774	10.0%	616	7.0%	731	9.0%	3445	23.0%
Number of fully used LUT-FF pairs	3910	53.0%	4132	50.0%	3880	51.0%	6775	46.0%
Number of unique control sets	97		174		163		926	
Number of slice register sites lost to control set restrictions	218	1.0%	320	1.0%	310	1.0%	2122	4.0%
Number of bonded IOBs	341	53.0%	341	53.0%	341	53.0%	236	36.0%
Number of BlockRAM/FIFO	12	8.0%	12	8.0%	12	8.0%	32	21.0%
Number using BlockRAM only	12		12		12		32	
Number of 36k BlockRAM used	12		12		12		32	
Total Memory used (KB)	432	8.0%	432	8.0%	432	8.0%	1152	21.0%
Number of BUFG/BUFGCTRLs	1	3.0%	1	3.0%	1	3.0%	9	28.0%
Number used as BUFGs	1		1		1		9	
Number of DSP48Es	6	4.0%	18	14.0%	17	13.0%	17	13.0%
Average Fanout of Non-Clock Nets	4.4		4		4.1		3.61	
Number of PLL_ADVs							1	16.0%
Number of PPC440s							1	100.0%
Number of IDELAYCTRLs							3	13.0%
Number of BUFIOs							8	10.0%
Number of DCM_ADVs							1	8.0%

Figure 9: Several synthesis results showing the resource utilization at various stages of our optimization

Throughout our design process, the worst case delays did not vary significantly (see Table 3). The largest delay in the Block Accelerator tended to be either signals related to decoding the NPI address or signals in the Feature Extractor. The Feature Extractor has several large combinational blocks. Indeed, this logic caused us to not meet timing several times. However, we were able to pipeline the design and achieve our target of 125 MHz operation.

At a clock speed of 125 MHz, the software-only solution is able to process about 50 detection windows per second. The accelerated solution is able to process over 5100 detection windows per second.

Design	Slowest Net	Description	Delay (ns)
Initial Design	NPI_Addr	NPI address signal from PowerPC to Block Accelerator	7.012ns
First Optimizations	inStageCnt	Signals used to calculate feature vector through a large case statement	7.003ns
Best Optimizations	NPI_Addr	NPI address signal from PowerPC to Block Accelerator	6.632ns
Best Optimizations, Entire Design	delayed_dqs	Signal internal to DDR MPMC	6.463ns

Table 3: Longest path delays at various stages of our optimization

5 Conclusion

A computer vision algorithm to detect pedestrians in an image is successfully accelerated with a FPGA implementation for the majority of the computations. The rest of the algorithm uses the original C++ code and runs on the on-board PowerPC. Some modifications to the algorithm have been made to make it more suited for the FPGA. The system is designed to operate at a 125 MHz clock frequency. At this clock rate, the proposed solution processes 5100 detection windows per second, compared to 50 detection windows per second for a software-only solution at the same clock frequency.

The final detection results tested for three different images are shown in figures 10a, 10b and 11. The blue rectangles are drawn using the detection coordinates outputted from the FPGA.

A Pipeline trace report

```
athena% bsv-trace.pl jxiao_trace.cfg err.txt
 CYC:    0  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:    0  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:    1  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:    2  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:    3  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:    4  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:    5  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:    6  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:    7  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:    8  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:    9  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   10  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   11  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   12  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   13  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   14  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   15  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   16  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   17  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   18  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   19  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   20  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   21  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   22  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   23  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   24  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   25  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   26  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   27  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   28  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   29  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   30  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   31  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   32  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   33  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   34  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   35  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   36  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   37  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   38  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   39  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   40  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   41  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   42  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   43  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   44  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   45  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   46  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   47  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   48  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   49  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   50  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   51  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   52  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
 CYC:   53  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
```

```
CYC:   54  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
CYC:   55  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
CYC:   56  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
CYC:   57  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
CYC:   58  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
CYC:   59  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
CYC:   60  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
CYC:   61  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
CYC:   62  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
CYC:   63  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
CYC:   64  [DDR:  .  ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
CYC:   65  [DDR:0 .1 ]  [  .  -  .  ]  [          |      ]  [       | |     |              |        ]
CYC:   66  [DDR:0 .3 ]  [ 0. 0-  .  ]  [          |      ]  [       | |     |              |        ]
CYC:   67  [DDR:1 .  ]  [ 0. 1-  .  ]  [          |      ]  [       | |     |              |        ]
CYC:   68  [DDR:0 .3 ]  [ 0. 2- 0. 0]  [          |      ]  [       | |     |              |        ]
CYC:   69  [DDR:1 .  ]  [ 0. 3- 0. 1]  [          |      ]  [       | |     |              |        ]
CYC:   70  [DDR:0 .3 ]  [ 0. 4- 0. 2]  [          |      ]  [       | |     |              |        ]
CYC:   71  [DDR:1 .  ]  [ 0. 5- 0. 3]  [          |      ]  [       | |     |              |        ]
CYC:   72  [DDR:0 .3 ]  [ 0. 6- 0. 4]  [          |      ]  [       | |     |              |        ]
CYC:   73  [DDR:1 .  ]  [ 0. 7- 0. 5]  [          |      ]  [       | |     |              |        ]
CYC:   74  [DDR:0 .3 ]  [ 0. 8- 0. 6]  [          |      ]  [       | |     |              |        ]
CYC:   75  [DDR:1 .  ]  [ 1. 0- 0. 7]  [          |      ]  [       | |     |              |        ]
CYC:   76  [DDR:0 .3 ]  [ 1. 1- 0. 8]  [          |      ]  [       | |     |              |        ]
CYC:   77  [DDR:1 .  ]  [ 1. 2- 1. 0]  [          |      ]  [       | |     |              |        ]
CYC:   78  [DDR:0 .3 ]  [ 1. 3- 1. 1]  [          |      ]  [       | |     |              |        ]
CYC:   79  [DDR:1 .  ]  [ 1. 4- 1. 2]  [          |      ]  [       | |     |              |        ]
CYC:   80  [DDR:0 .3 ]  [ 1. 5- 1. 3]  [          |      ]  [       | |     |              |        ]
CYC:   81  [DDR:1 .  ]  [ 1. 6- 1. 4]  [          |      ]  [       | |     |              |        ]
CYC:   82  [DDR:0 .2 ]  [ 1. 7- 1. 5]  [          |      ]  [       | |     |              |        ]
CYC:   83  [DDR:0 .3 ]  [ 1. 8- 1. 6]  [          |      ]  [       | |     |              |        ]
CYC:   84  [DDR:1 .  ]  [ 2. 0- 1. 7]  [          |      ]  [       | |     |              |        ]
CYC:   85  [DDR:0 .3 ]  [ 2. 1- 1. 8]  [          |      ]  [       | |     |              |        ]
CYC:   86  [DDR:1 .  ]  [ 2. 2- 2. 0]  [          |      ]  [       | |     |              |        ]
CYC:   87  [DDR:0 .3 ]  [ 2. 3- 2. 1]  [          |      ]  [       | |     |              |        ]
CYC:   88  [DDR:1 .  ]  [ 2. 4- 2. 2]  [          |      ]  [       | |     |              |        ]
CYC:   89  [DDR:0 .3 ]  [ 2. 5- 2. 3]  [          |      ]  [       | |     |              |        ]
CYC:   90  [DDR:1 .  ]  [ 2. 6- 2. 4]  [          |      ]  [       | |     |              |        ]
CYC:   91  [DDR:0 .2 ]  [ 2. 7- 2. 5]  [          |      ]  [       | |     |              |        ]
CYC:   92  [DDR:0 .1 ]  [ 2. 8- 2. 6]  [          |      ]  [       | |     |              |        ]
CYC:   93  [DDR:0 .3 ]  [ 3. 0- 2. 7]  [          |      ]  [       | |     |              |        ]
CYC:   94  [DDR:1 .  ]  [ 3. 1- 2. 8]  [          |      ]  [       | |     |              |        ]
CYC:   95  [DDR:0 .3 ]  [ 3. 2- 3. 0]  [          |      ]  [       | |     |              |        ]
CYC:   96  [DDR:1 .  ]  [ 3. 3- 3. 1]  [Cin__MinFF|      ]  [       | |     |              |        ]
CYC:   97  [DDR:0 .3 ]  [ 3. 4- 3. 2]  [Cin__MinFF|      ]  [       | |     |              |        ]
CYC:   98  [DDR:1 .  ]  [ 3. 5- 3. 3]  [Cin__MinFF|      ]  [       | |     |              |        ]
CYC:   99  [DDR:0 .3 ]  [ 3. 6- 3. 4]  [Cout_MinFW|      ]  [       | |     |              |        ]
CYC:  100  [DDR:1 .  ]  [ 3. 7- 3. 5]  [Cin__MinFF|      ]  [       | |     |              |        ]
CYC:  101  [DDR:0 .1 ]  [ 3. 8- 3. 6]  [Cout_MinFW|      ]  [       | |     |              |        ]
CYC:  102  [DDR:0 .3 ]  [ 4. 0- 3. 7]  [Cin__MinFF|      ]  [       | |     |              |        ]
CYC:  103  [DDR:1 .  ]  [ 4. 1- 3. 8]  [Cout_MinFW|      ]  [       | |     |              |        ]
CYC:  104  [DDR:0 .3 ]  [ 4. 2- 4. 0]  [Cin__MinFF|      ]  [       | |     |              |        ]
CYC:  105  [DDR:1 .  ]  [ 4. 3- 4. 1]  [Cout_MinFW|      ]  [       | |     |              |        ]
CYC:  106  [DDR:0 .3 ]  [ 4. 4- 4. 2]  [Cin__MinFF|      ]  [       | |     |              |        ]
CYC:  107  [DDR:1 .  ]  [ 4. 5- 4. 3]  [Cout_MinFW|      ]  [       | |     |              |        ]
CYC:  108  [DDR:0 .3 ]  [ 4. 6- 4. 4]  [Cin__MinFF|      ]  [       | |     |              |        ]
CYC:  109  [DDR:1 .  ]  [ 4. 7- 4. 5]  [Cout_MinFW|      ]  [       | |     |              |        ]
CYC:  110  [DDR:0 .1 ]  [ 4. 8- 4. 6]  [Cin__MinFF|      ]  [       | |     |              |        ]
CYC:  111  [DDR:0 .3 ]  [ 5. 0- 4. 7]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  112  [DDR:1 .  ]  [ 5. 1- 4. 8]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
```

```
CYC:  113  [DDR:0 .3 ]  [ 5. 2- 5. 0]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  114  [DDR:1 .  ]  [ 5. 3- 5. 1]  [          |MoutFW]  [       | |     |              |        ]
CYC:  115  [DDR:0 .3 ]  [ 5. 4- 5. 2]  [          |MoutFF]  [       | |     |              |        ]
CYC:  116  [DDR:1 .  ]  [ 5. 5- 5. 3]  [          |MoutFW]  [       | |     |              |        ]
CYC:  117  [DDR:0 .3 ]  [ 5. 6- 5. 4]  [          |MoutFF]  [       | |     |              |        ]
CYC:  118  [DDR:1 .  ]  [ 5. 7- 5. 5]  [          |MoutFW]  [       | |     |              |        ]
CYC:  119  [DDR:0 .3 ]  [ 5. 8- 5. 6]  [          |MoutFF]  [       | |     |              |        ]
CYC:  120  [DDR:1 .  ]  [ 6. 0- 5. 7]  [          |MoutFW]  [       | |     |              |        ]
CYC:  121  [DDR:0 .3 ]  [ 6. 1- 5. 8]  [          |MoutFF]  [       | |     |              |        ]
CYC:  122  [DDR:1 .  ]  [ 6. 2- 6. 0]  [          |MoutFW]  [       | |     |              |        ]
CYC:  123  [DDR:0 .3 ]  [ 6. 3- 6. 1]  [Cin__MinFF|MoutFF]  [       | |     |              |        ]
CYC:  124  [DDR:1 .  ]  [ 6. 4- 6. 2]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  125  [DDR:0 .3 ]  [ 6. 5- 6. 3]  [Cin__MinFF|MoutFF]  [       | |     |              |        ]
CYC:  126  [DDR:1 .  ]  [ 6. 6- 6. 4]  [Cout_MinFW|MoutFW]  [       | |     |              |        ]
CYC:  127  [DDR:0 .2 ]  [ 6. 7- 6. 5]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  128  [DDR:0 .3 ]  [ 6. 8- 6. 6]  [Cout_MinFW|MoutFW]  [       | |     |              |        ]
CYC:  129  [DDR:1 .  ]  [ 7. 0- 6. 7]  [Cin__MinFF|      ]  [       | |     |              |        ]
CYC:  130  [DDR:0 .3 ]  [ 7. 1- 6. 8]  [Cout_MinFW|      ]  [       | |     |              |        ]
CYC:  131  [DDR:1 .  ]  [ 7. 2- 7. 0]  [Cin__MinFF|      ]  [       | |     |              |        ]
CYC:  132  [DDR:0 .3 ]  [ 7. 3- 7. 1]  [Cout_MinFW|      ]  [       | |     |              |        ]
CYC:  133  [DDR:1 .  ]  [ 7. 4- 7. 2]  [Cin__MinFF|      ]  [       | |     |              |        ]
CYC:  134  [DDR:0 .3 ]  [ 7. 5- 7. 3]  [Cout_MinFW|      ]  [       | |     |              |        ]
CYC:  135  [DDR:1 .  ]  [ 7. 6- 7. 4]  [Cin__MinFF|      ]  [       | |     |              |        ]
CYC:  136  [DDR:0 .2 ]  [ 7. 7- 7. 5]  [Cout_MinFW|      ]  [       | |     |              |        ]
CYC:  137  [DDR:0 .1 ]  [ 7. 8- 7. 6]  [Cin__MinFF|      ]  [       | |     |              |        ]
CYC:  138  [DDR:0 .3 ]  [ 8. 0- 7. 7]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  139  [DDR:1 .  ]  [ 8. 1- 7. 8]  [Cin__MinFF|MoutFF]  [       | |     |              |        ]
CYC:  140  [DDR:0 .3 ]  [ 8. 2- 8. 0]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  141  [DDR:1 .  ]  [ 8. 3- 8. 1]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  142  [DDR:0 .3 ]  [ 8. 4- 8. 2]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  143  [DDR:1 .  ]  [ 8. 5- 8. 3]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  144  [DDR:0 .3 ]  [ 8. 6- 8. 4]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  145  [DDR:1 .  ]  [ 8. 7- 8. 5]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  146  [DDR:0 .3 ]  [ 8. 8- 8. 6]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  147  [DDR:1 .  ]  [ 0. 0- 8. 7]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  148  [DDR:0 .3 ]  [ 0. 1- 8. 8]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  149  [DDR:1 .  ]  [ 0. 2- 0. 0]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  150  [DDR:0 .3 ]  [ 0. 3- 0. 1]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  151  [DDR:1 .  ]  [ 0. 4- 0. 2]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  152  [DDR:0 .3 ]  [ 0. 5- 0. 3]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  153  [DDR:1 .  ]  [ 0. 6- 0. 4]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  154  [DDR:0 .2 ]  [ 0. 7- 0. 5]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  155  [DDR:0 .1 ]  [ 0. 8- 0. 6]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  156  [DDR:0 .3 ]  [ 1. 0- 0. 7]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  157  [DDR:1 .  ]  [ 1. 1- 0. 8]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  158  [DDR:0 .3 ]  [ 1. 2- 1. 0]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  159  [DDR:1 .  ]  [ 1. 3- 1. 1]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  160  [DDR:0 .3 ]  [ 1. 4- 1. 2]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  161  [DDR:1 .  ]  [ 1. 5- 1. 3]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  162  [DDR:0 .3 ]  [ 1. 6- 1. 4]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  163  [DDR:1 .  ]  [ 1. 7- 1. 5]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  164  [DDR:0 .3 ]  [ 1. 8- 1. 6]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  165  [DDR:1 .  ]  [ 2. 0- 1. 7]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  166  [DDR:0 .3 ]  [ 2. 1- 1. 8]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  167  [DDR:1 .  ]  [ 2. 2- 2. 0]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  168  [DDR:0 .3 ]  [ 2. 3- 2. 1]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  169  [DDR:1 .  ]  [ 2. 4- 2. 2]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  170  [DDR:0 .3 ]  [ 2. 5- 2. 3]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  171  [DDR:1 .  ]  [ 2. 6- 2. 4]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
```

```
CYC:  172  [DDR:0 .2 ]  [ 2. 7- 2. 5]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  173  [DDR:0 .1 ]  [ 2. 8- 2. 6]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  174  [DDR:0 .3 ]  [ 3. 0- 2. 7]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  175  [DDR:1 .  ]  [ 3. 1- 2. 8]  [Cout_MinFW|MoutFW]  [       | |     |              |        ]
CYC:  176  [DDR:0 .3 ]  [ 3. 2- 3. 0]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  177  [DDR:1 .  ]  [ 3. 3- 3. 1]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  178  [DDR:0 .3 ]  [ 3. 4- 3. 2]  [Cin__MinFF|MoutFF]  [       | |     |              |        ]
CYC:  179  [DDR:1 .  ]  [ 3. 5- 3. 3]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  180  [DDR:0 .3 ]  [ 3. 6- 3. 4]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  181  [DDR:1 .  ]  [ 3. 7- 3. 5]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  182  [DDR:0 .1 ]  [ 3. 8- 3. 6]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  183  [DDR:0 .3 ]  [ 4. 0- 3. 7]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  184  [DDR:1 .  ]  [ 4. 1- 3. 8]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  185  [DDR:0 .3 ]  [ 4. 2- 4. 0]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  186  [DDR:1 .  ]  [ 4. 3- 4. 1]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  187  [DDR:0 .3 ]  [ 4. 4- 4. 2]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  188  [DDR:1 .  ]  [ 4. 5- 4. 3]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  189  [DDR:0 .3 ]  [ 4. 6- 4. 4]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  190  [DDR:1 .  ]  [ 4. 7- 4. 5]  [Cout_MinFW|MoutFW]  [       | |     |              |        ]
CYC:  191  [DDR:0 .1 ]  [ 4. 8- 4. 6]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  192  [DDR:0 .3 ]  [ 5. 0- 4. 7]  [Cout_MinFW|MoutFF]  [Rin_Cin|P|     |              |        ]
CYC:  193  [DDR:1 .  ]  [ 5. 1- 4. 8]  [Cout_MinFW|MoutFF]  [       | |   31|              |        ]
CYC:  194  [DDR:0 .3 ]  [ 5. 2- 5. 0]  [Cout_MinFW|MoutFF]  [       | |   30|              |        ]
CYC:  195  [DDR:1 .  ]  [ 5. 3- 5. 1]  [          |MoutFW]  [       | |   29|              |        ]
CYC:  196  [DDR:0 .3 ]  [ 5. 4- 5. 2]  [          |MoutFF]  [       | |   28|              |        ]
CYC:  197  [DDR:1 .  ]  [ 5. 5- 5. 3]  [          |MoutFW]  [       | |   27|              |        ]
CYC:  198  [DDR:0 .3 ]  [ 5. 6- 5. 4]  [          |MoutFF]  [       | |   26|              |        ]
CYC:  199  [DDR:1 .  ]  [ 5. 7- 5. 5]  [          |MoutFW]  [       | |   25|              |        ]
CYC:  200  [DDR:0 .3 ]  [ 5. 8- 5. 6]  [          |MoutFF]  [       | |   24|              |        ]
CYC:  201  [DDR:1 .  ]  [ 6. 0- 5. 7]  [          |MoutFW]  [       | |   23|              |        ]
CYC:  202  [DDR:0 .3 ]  [ 6. 1- 5. 8]  [          |MoutFF]  [       | |   22|              |        ]
CYC:  203  [DDR:1 .  ]  [ 6. 2- 6. 0]  [          |MoutFW]  [       | |   21|              |        ]
CYC:  204  [DDR:0 .3 ]  [ 6. 3- 6. 1]  [Cin__MinFF|MoutFF]  [       | |   20|              |        ]
CYC:  205  [DDR:1 .  ]  [ 6. 4- 6. 2]  [Cin__MinFF|MoutFW]  [       | |   19|              |        ]
CYC:  206  [DDR:0 .3 ]  [ 6. 5- 6. 3]  [Cin__MinFF|MoutFF]  [       | |   18|              |        ]
CYC:  207  [DDR:1 .  ]  [ 6. 6- 6. 4]  [Cout_MinFW|MoutFW]  [       | |   17|              |        ]
CYC:  208  [DDR:0 .2 ]  [ 6. 7- 6. 5]  [Cin__MinFF|MoutFW]  [       | |   16|              |        ]
CYC:  209  [DDR:0 .1 ]  [ 6. 8- 6. 6]  [Cout_MinFW|MoutFW]  [       | |   15|              |        ]
CYC:  210  [DDR:0 .3 ]  [ 7. 0- 6. 7]  [Cin__MinFF|      ]  [       | |   14|              |        ]
CYC:  211  [DDR:1 .  ]  [ 7. 1- 6. 8]  [Cout_MinFW|      ]  [       | |   13|              |        ]
CYC:  212  [DDR:0 .3 ]  [ 7. 2- 7. 0]  [Cin__MinFF|      ]  [       | |   12|              |        ]
CYC:  213  [DDR:1 .  ]  [ 7. 3- 7. 1]  [Cout_MinFW|      ]  [       | |   11|              |        ]
CYC:  214  [DDR:0 .3 ]  [ 7. 4- 7. 2]  [Cin__MinFF|      ]  [       | |   10|              |        ]
CYC:  215  [DDR:1 .  ]  [ 7. 5- 7. 3]  [Cout_MinFW|      ]  [       | |    9|              |        ]
CYC:  216  [DDR:0 .3 ]  [ 7. 6- 7. 4]  [Cin__MinFF|      ]  [       | |    8|              |        ]
CYC:  217  [DDR:1 .  ]  [ 7. 7- 7. 5]  [Cout_MinFW|      ]  [       | |    7|              |        ]
CYC:  218  [DDR:0 .3 ]  [ 7. 8- 7. 6]  [Cin__MinFF|      ]  [       | |    6|              |        ]
CYC:  219  [DDR:1 .  ]  [ 8. 0- 7. 7]  [Cout_MinFW|MoutFF]  [       | |    5|              |        ]
CYC:  220  [DDR:0 .3 ]  [ 8. 1- 7. 8]  [Cin__MinFF|MoutFF]  [       | |    4|              |        ]
CYC:  221  [DDR:1 .  ]  [ 8. 2- 8. 0]  [Cout_MinFW|MoutFF]  [       | |    3|              |        ]
CYC:  222  [DDR:0 .3 ]  [ 8. 3- 8. 1]  [Cin__MinFF|MoutFW]  [       | |    2|              |        ]
CYC:  223  [DDR:1 .  ]  [ 8. 4- 8. 2]  [Cout_MinFW|MoutFF]  [       | |    1|              |        ]
CYC:  224  [DDR:0 .3 ]  [ 8. 5- 8. 3]  [Cin__MinFF|MoutFW]  [       | |    0|              |        ]
CYC:  225  [DDR:1 .  ]  [ 8. 6- 8. 4]  [Cout_MinFW|MoutFF]  [       | |     |Rout_Cout_Min|        ]
CYC:  226  [DDR:0 .2 ]  [ 8. 7- 8. 5]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  227  [DDR:0 .1 ]  [ 8. 8- 8. 6]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  228  [DDR:0 .3 ]  [ 0. 0- 8. 7]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
CYC:  229  [DDR:1 .  ]  [ 0. 1- 8. 8]  [Cout_MinFW|MoutFF]  [       | |     |              |        ]
CYC:  230  [DDR:0 .3 ]  [ 0. 2- 0. 0]  [Cin__MinFF|MoutFW]  [       | |     |              |        ]
```

```
CYC:  231  [DDR:1 .  ]  [ 0. 3- 0. 1]  [Cout_MinFW|MoutFF]  [       | |     |             |        ]
CYC:  232  [DDR:0 .3 ]  [ 0. 4- 0. 2]  [Cin__MinFF|MoutFW]  [       | |     |             |        ]
CYC:  233  [DDR:1 .  ]  [ 0. 5- 0. 3]  [Cout_MinFW|MoutFF]  [       | |     |             |        ]
CYC:  234  [DDR:0 .3 ]  [ 0. 6- 0. 4]  [Cin__MinFF|MoutFW]  [       | |     |             |        ]
CYC:  235  [DDR:1 .  ]  [ 0. 7- 0. 5]  [Cout_MinFW|MoutFF]  [       | |     |             |        ]
CYC:  236  [DDR:0 .1 ]  [ 0. 8- 0. 6]  [Cin__MinFF|MoutFW]  [       | |     |             |        ]
CYC:  237  [DDR:0 .3 ]  [ 1. 0- 0. 7]  [Cout_MinFW|MoutFF]  [       | |     |             |        ]
CYC:  238  [DDR:1 .  ]  [ 1. 1- 0. 8]  [Cin__MinFF|MoutFW]  [       | |     |             |        ]
CYC:  239  [DDR:0 .3 ]  [ 1. 2- 1. 0]  [Cout_MinFW|MoutFF]  [       | |     |             |        ]
CYC:  240  [DDR:1 .  ]  [ 1. 3- 1. 1]  [Cin__MinFF|MoutFW]  [       | |     |             |Mout_res]
```

References

[1] Dalal N (2006) Finding People in Images and Videos. Dissertation, Institut National Polytechnique de Grenoble

[2] Dalal N, Triggs B (2005) Histograms of Oriented Gradients for Human Detection. In: Proceedings of IEEE Conference Computer Vision and Pattern Recognition, pp 886-893

[3] Schapire R (2001) The boosting approach to machine learning: An overview. In: MSRI Workshop on Nonlinear Estimation and Classification

[4] `http://memocode2010.csail.mit.edu/redmine/repositories/browse/platforms/trunk/modules/bluespec?rev=282`

[5] Yamin Li And and Yamin Li and Wanming Chu (1996) A New Non-Restoring Square Root Algorithm and Its VLSI Implementations. Proc. of 1996 IEEE International Conference on Computer Design: VLSI in Computers and Processors

[6] K. Piromsopa and C. Aporntewan and P. Chogsatitvataa. An FPGA Implementation of a Fixed-Point Square Root Operation

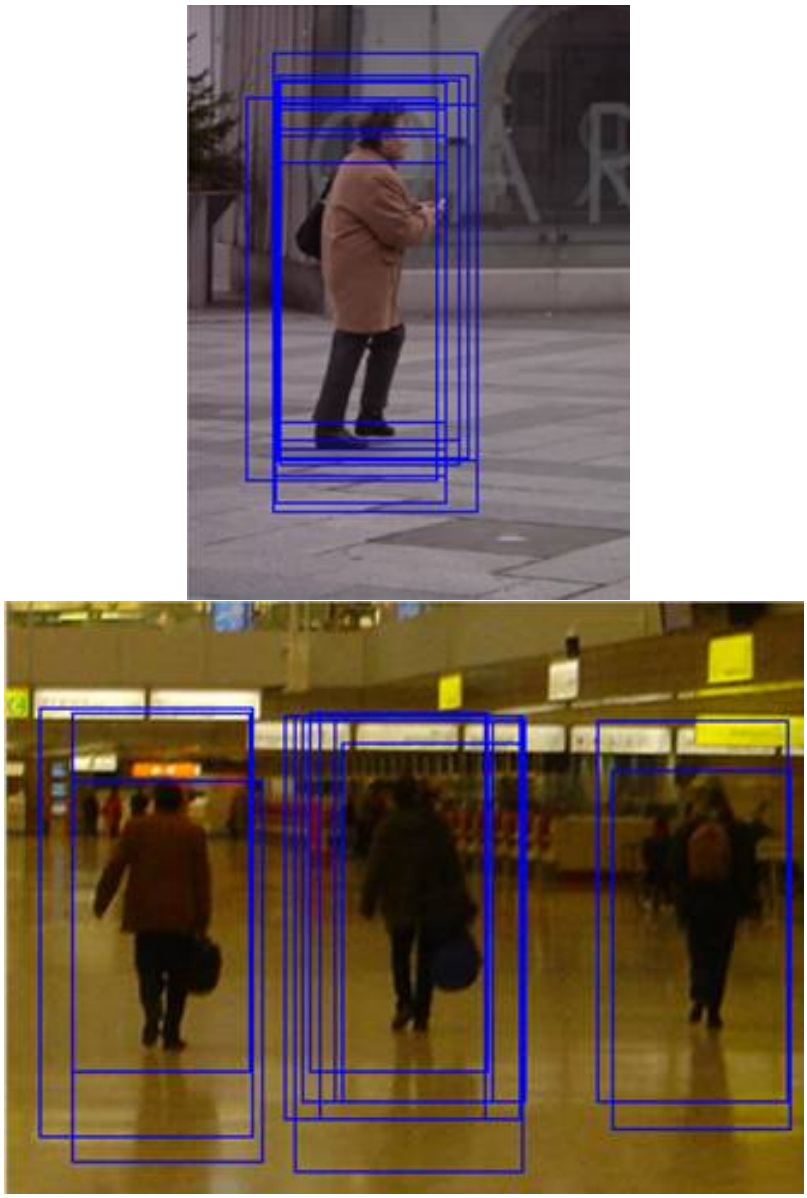

Figure 10: Sample outputs from our detector

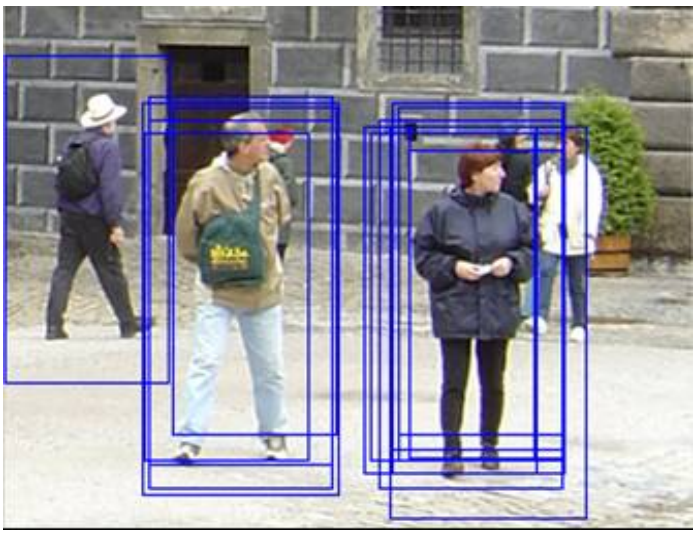

Figure 11: Sample outputs from our detector