ECSE-487
COMPUTER ARCHITECTURE LABORATORY
Assignment #1

Andrei Purcarus Craciun
260631911

January 30, 2017

1 Introduction

This report describes the design, simulation and synthesis of an image processing module that can parse PGM ASCII images byte by byte, store them in an internal memory block, perform pixel by pixel operations on them, and save the resulting images to files byte by byte in the same format. It also describes an application of the module to edge detection in background-difference images.

2 Design Description

2.1 SRAM

First, a basic SRAM module was designed to store the pixels of the images being processed. Due to the size constraints of FPGA devices in terms of memory cells, a maximal image size of 255 by 255 8-bit pixels was decided upon for the simulation. In practical systems, the memory cells could be external DRAM cells, with SRAM caches for performance. This would allow for larger images to be stored.

A pin-out diagram of the component is shown in Figure 1. The component was made generic in terms of address width and data width in order to simplify reuse. The component is synchronized to the rising edge of the clock. To read data from the memory block, the address of the data is set, and the read_en signal is asserted. On the next rising edge, the data will be available on the data_out line. Similarly, to write data, the address to write to and the data_in are set, and the write_en signal is asserted. On the next rising edge, the data will be stored at the specified address. The processor system described in this report uses ADDR_WIDTH = 16 and DATA_WIDTH = 8 in order to satisfy the maximum image size described above.

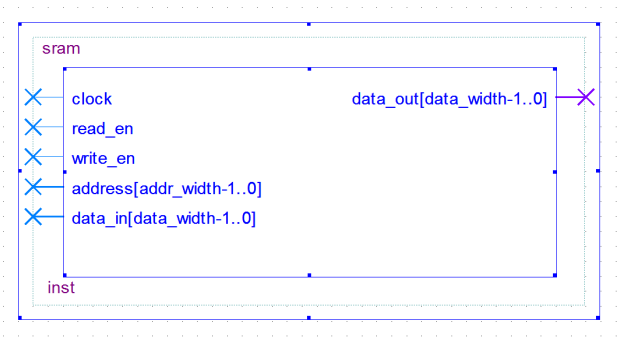

Figure 1: An SRAM module designed to store images.

2.2 Image IO

Next, the image IO modules were designed. In order to make these designs synthesizable, the actual file manipulation is left to the simulation test-benches. The load module assumes that a continuous byte input stream is available as input, and writes its data to an SRAM block as defined in the previous section. The save module assumes that its input is available in an SRAM block and that its output will be written to file when it asserts a signal.

A pin-out of the image load module is shown in Figure 2. The design uses a datapath/controller system to parse a PGM image file. The data is fed continuously into data_in on each rising clock edge, and the load_en flag is asserted for as long as there is data available in the file. The module checks that the file satisfies the PGM image format as it loads data, and outputs an error code if it detects a violation. The possible error codes are described in Table 1, and are defined in the image_io_error package. The pixel data is gradually output as the file is parsed, with the address and write_en lines designed to store the data contiguously in an SRAM block. When the done flag is asserted, the img_width, img_height and maxval signals will contain the correct values assuming no errors, and the data will have been written to memory. The module then needs to be reset through the asynchronous reset flag to be able to parse

another file. Internally, the module discards non-numeric data and comments and stores numeric data in a register, incrementing it as more digits are read before writing it to memory when a whitespace character is reached.

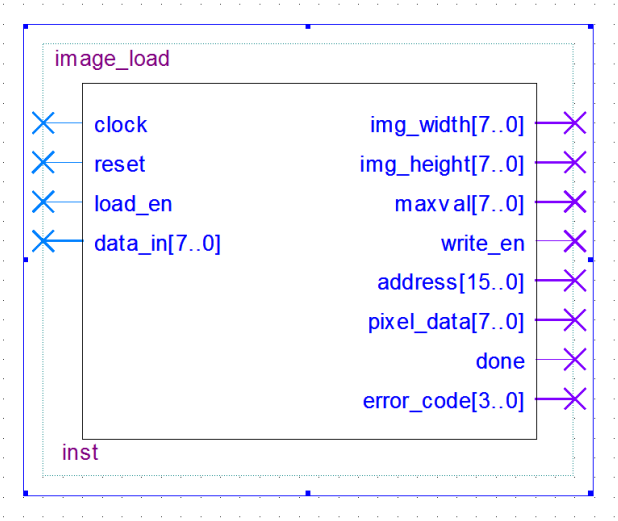

Figure 2: A module that loads PGM images from files into memory.

Error Name	Error Code	Description
NONE	0000	No error has been encountered.
INVALID_FILETYPE	0001	The P2 file-type specifier was not found.
WIDTH_TOO_LARGE	0010	The width exceeds 255.
HEIGHT_TOO_LARGE	0011	The height exceeds 255.
MAXVAL_TOO_LARGE	0100	The maxval exceeds 255.
PIXEL_TOO_LARGE	0101	A pixel exceeds maxval.
TOO_MANY_PIXELS	0110	The module has read more than width · height pixels.
TOO_FEW_PIXELS	0111	The end of the file has been reached with less than width · height pixels read.
INVALID_TOKEN	1000	The module has encountered an unexpected character.

Table 1: Possible IO errors encountered when parsing a PGM file.

A pin-out of the image save module is shown in Figure 3. The design also uses a datapath/controller system to output a PGM image to a file byte by byte. The img_width, img_height and maxval inputs are used to read in the image metadata, and the read_en and address outputs are used to interface with an SRAM memory block and obtain the pixel_data input. The module is enabled by the save_en flag, and is reset with the asynchronous reset input. On each rising clock edge, the module can output an ASCII character on the data_out line. It indicates that the character should be saved to a file through the write_en flag. Finally, the done flag is used to indicate that the module has finished outputting all the data. Internally, the component stores the input data into a register. It then uses a binary to BCD (Binary-Coded Decimal) conversion circuit and outputs each digit of the data on separate clock cycles after converting it to ASCII.

2.3 Pixel Processor

Next, the basic unit of pixel processing was designed. In addition to the operations specified, the absolute difference operation was added in order to better implement an edge detection algorithm. In addition, it was decided that the addition and subtraction operations should threshold the results at

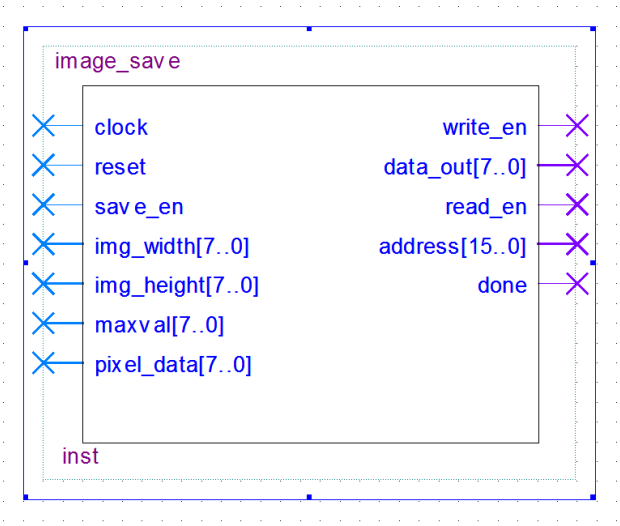

Figure 3: A module that saves PGM images from memory into files.

maxval and 0 respectively, as these behaviors make more sense in the context of pixel manipulation. This implementation also makes all data output by the processor valid. Finally, a third subtraction operation was added, called normalized subtraction. This operation adds maxval to the difference and divides the result by 2, giving a value in [0, maxval]. A description of the supported operations is given in Table 2. The pin-out diagram is shown in Figure 4. The data inputs are placed on the pixel_data and pixel_operand signals, the operation placed on the operation signal and the image maxval placed on the maxval signal. On each rising clock edge, data_out is set to the result if enable is set. The data_ready flag indicates that the data can be stored back in memory, and is included for potential pipelining of more complex operations in the future. The data_valid flag indicates if the data is a valid pixel or if an overflow error has occurred. Currently, both of these flags are not used since the module operates in a single clock cycle and the data is always valid.

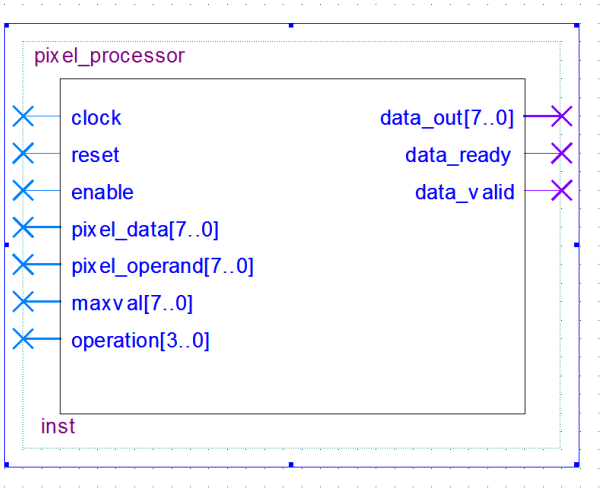

Figure 4: A basic processing unit capable of performing pixel operations.

2.4 Image Processor

Finally, the image processor was designed using the modules defined previously. The pin-out diagram is shown in Figure 5. The processor was designed to be able to perform general purpose operations on

Op Name	Op Code	Description
SET	0000	Sets the output to the value of pixel_operand.
ADD	0001	Adds pixel_data and pixel_operand, capping the result at maxval.
SUB	0010	Subtracts pixel_operand from pixel_data, capping the result at 0.
AND	0011	Performs a bitwise AND on pixel_data and pixel_operand.
OR	0100	Performs a bitwise OR on pixel_data and pixel_operand.
XOR	0101	Performs a bitwise XOR on pixel_data and pixel_operand.
INVERT	0110	Sets the output to maxval - pixel_data.
THRESH	0111	Sets the output to maxval if pixel_data > pixel_operand, and to pixel_data otherwise.
ABSDIFF	1000	Outputs the absolute difference of pixel_data and pixel_operand.
NDIFF	1001	Outputs the normalized difference of pixel_data and pixel_operand.

Table 2: Possible pixel operations performable by the pixel processor.

images stored in 3 SRAM registers. The instruction to execute is specified by the reg_in_0, reg_in_1, reg_out, global_operand, address_increment and operation signals. The types of operations are described in Table 3. The registers can be any of 00, 01 or 10, and the reg_in_1 input can take the special value 11 which indicates that the value of global_operand should be used for all pixels in the register. The address_increment input allows for horizontal left shifting of operation results, with the right-most columns being padded with the last value in each row. The module also includes the data_in_load and read_en_load inputs to provide an interface with the image load module, and the data_out_save and write_en_save outputs to provide an interface with the image save module. Finally, the done and error_code signals specify the state of the processor, with the error codes being the same as those for the image load module.

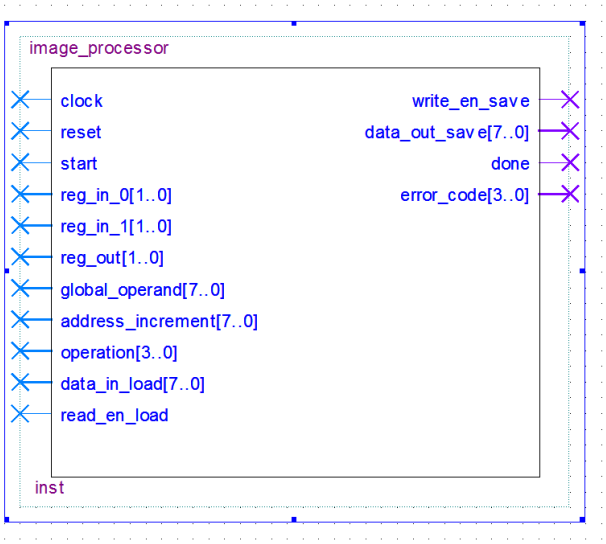

Figure 5: A processing unit capable of performing pixel by pixel operations on images.

2.5 Vertical Edge Detector

The vertical edge detector was implemented as a test-bench that makes use of the image processor described above. The sequence of operations is described in register transfer language in Table 4. First, it loads the main file to be processed into reg0 and the background into reg1, and takes the normalized difference and stores it in reg0. It then shifts it by 1 and stores the result in reg1, and performs an absolute difference between reg0 and reg1 and thresholds the result with a threshold of 7. Then, it saves the result to a separate file. With this sequence of operations, normalization is taken care of by the normalized

Op Name	Op Code	Description
SET	0000	reg_out ⇐ reg_in_1
ADD	0001	reg_out ⇐ reg_in_0 + reg_in_1, capped at maxval
SUB	0010	reg_out ⇐ reg_in_0 - reg_in_1, capped at 0
AND	0011	reg_out ⇐ reg_in_0 AND reg_in_1
OR	0100	reg_out ⇐ reg_in_0 OR reg_in_1
XOR	0101	reg_out ⇐ reg_in_0 XOR reg_in_1
INVERT	0110	reg_out ⇐ maxval - reg_in_0
THRESH	0111	reg_out ⇐ reg_in_0 > reg_in_1 ? maxval : reg_in_0
ABSDIFF	1000	reg_out ⇐ abs(reg_in_0 - reg_in_1)
NDIFF	1001	reg_out ⇐ (reg_in_0 - reg_in_1 + maxval) / 2
LOAD	1010	reg_out ⇐ file_in
SAVE	1011	file_out ⇐ reg_in_0

Table 3: Possible image operations performable by the image processor.

difference and absolute difference operations. In addition, the rightmost column keeps its value when the image is shifted left, so the gradient will always be 0 on that column.

reg0	⇐	file_main
reg1	⇐	file_background
reg0	⇐	reg0 NDIFF reg1
reg1	⇐	SET reg0, address_increment = 1 # shift reg0 by 1
reg2	⇐	reg0 ABSDIFF reg1
reg0	⇐	reg2 THRESH 7
file_out	⇐	reg0

Table 4: Operations executed in the edge detection algorithm.

3 Simulation Results

Every subunit of the design was subjected to individual testing before being incorporated into the larger processor unit. Automated tests were used to test reading and writing to an SRAM block, to test every possible operation and special case for the pixel processor, to test file loading with every possible type of error, and to test saving a small sample file. Annotated simulation traces for a selection of these tests can be found in Figures 6, 7, and 8 in Appendix A. These tests can be found in the source code and run using the .tcl scripts. The automated tests all pass.

Then, the processor unit itself was tested by loading the people106.pgm and background.pgm files into memory and performing the NDIFF, XOR, and THRESH 128 operations on them, and saving each result to file. An annotated simulation trace can be found in Figure 9 in Appendix A. The resulting images can be found in Appendix B, in Figures 13, 14 and 15, with the originals available in Figures 11 and 12. It is clear that these operations are performed as expected, as the NDIFF successfully removes the background image, the XOR brightens up the people where the pixels are different to the background and darkens the background, and the THRESH sets the brighter parts of the people image to 255.

Finally, the annotated simulation results for the edge detector test-bench are available in Figure 10 in Appendix A, with the resulting image shown in Figure 16 in Appendix B. The edges of the image have clearly been identified. From these test results, it is clear that the processor unit functions as intended.

4 Synthesis Results

The processor component was synthesized on a Cyclone V 5CSEMA5F31C6 FPGA. This FPGA was chosen due to previous experience with it in ECSE-323. The clock rate was set to 50MHz, the maximum

clock rate on the device. The results are shown in Table 5. The table shows that the processor could be synthesized with the available resources on the device, and that the timing requirements are met.

Family	Cyclone V
Device	5CSEMA5F31C6
Logic Utilization	439 / 32,070 (1 %)
Total Registers	262
Total Block Memory Bits	1,572,864 / 4,065,280 (39 %)
Total DSP Blocks	3 / 87 (3 %)
Fmax (MHz)	66.73
Setup Slack (ns)	5.015
Hold Slack (ns)	0.172

Table 5: Synthesis results on the Cyclone V 5CSEMA5F31C6 FPGA.

The part of the design taking up the most space are the memory blocks. The rest of the processor is quite small, taking up only 262 registers and 1 % of the available logic blocks. In order to make this part of the design smaller, as well as be able to handle larger images, an external DRAM chip can be used, with the internal SRAM blocks serving as a cache. This would reduce the memory block requirement on the FPGA.

The critical path in the circuit was found to be the BCD conversion function in the image save module. This function attempts to produce a 3-digit BCD number in a single clock cycle, which implies a large propagation delay. Synthesis of the (incorrect) design where the function is replaced by a no-op shows that Fmax increases to 119.2 MHz, a significant improvement. To fix this properly, the function could be implemented over several clock cycles, with the image load state machine modified to take this into account. This would potentially increase the latency of the image load circuit, but would decrease it for all the other components. Since saving an image file is a rare operation, this would amount to making the common case faster and thus speeding up the processor as a whole. Finally, the processor could be made much faster by redesigning it to exploit parallelism in the form of pipelining and data-level parallelism. The processor could then operate on multiple pixels simultaneously and deliver results faster.

A Simulation Traces

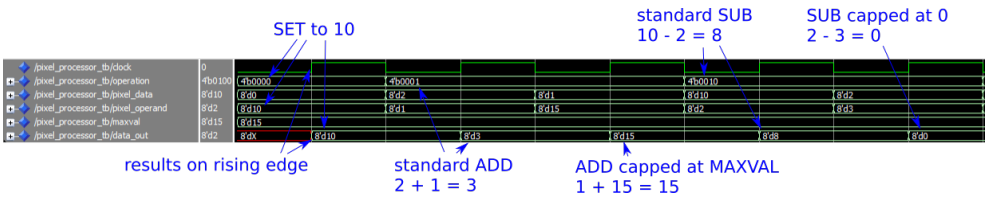

Figure 6: Simulation showing correct operation of the SET, ADD and SUB operations of the pixel processor.

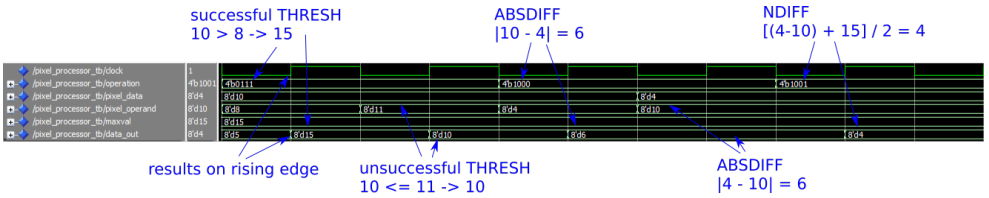

Figure 7: Simulation showing correct operation of the THRESH, ABSDIFF and NDIFF operations of the pixel processor.

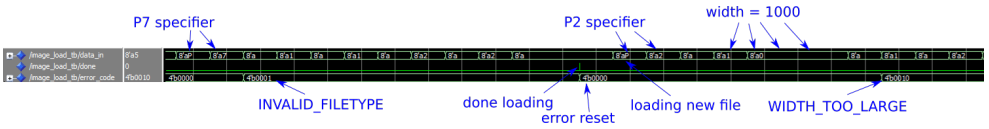

Figure 8: Simulation showing correct error identification when loading an image.

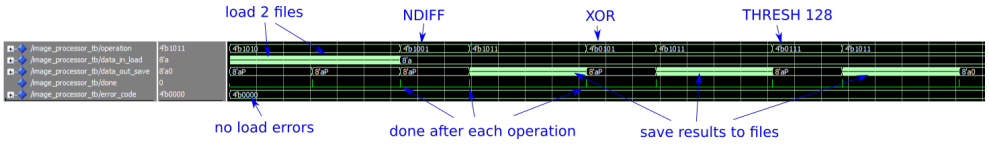

Figure 9: Simulation showing correct operation of the LOAD, NDIFF, XOR, THRESH and SAVE operations of the image processor.

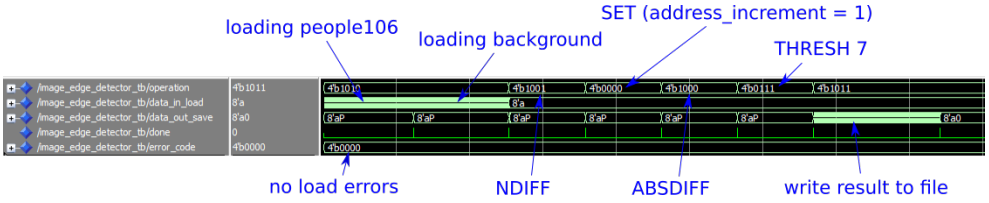

Figure 10: Simulation showing the correct generation of the edge detection image through the LOAD, NDIFF, SET, ABSDIFF, THRESH and SAVE operations of the image processor.

B Images

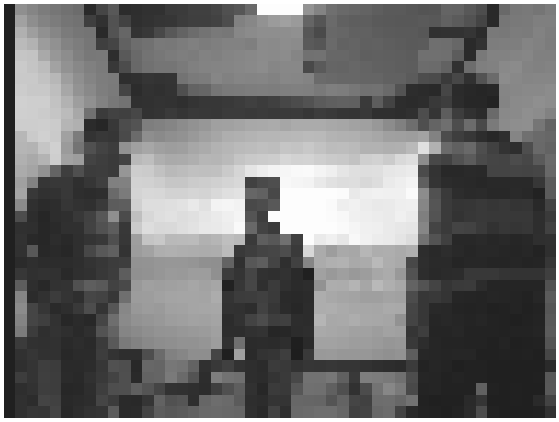

Figure 11: Image of people provided for testing.

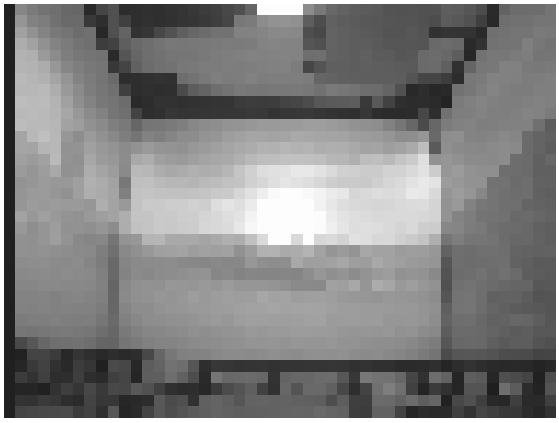

Figure 12: Image of background provided for testing.

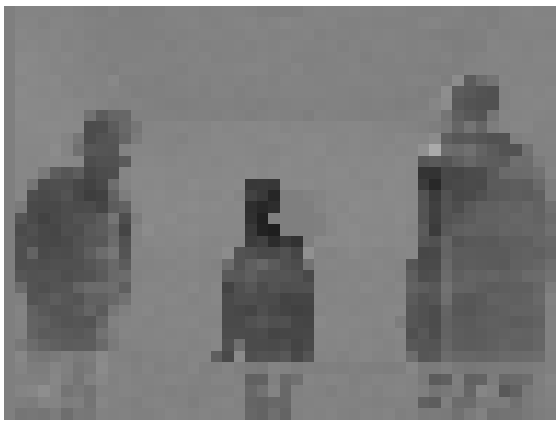

Figure 13: NDIFF of Figures 11 and 12. Note the gray background resulting from normalization and the clear identification of the people in the picture.

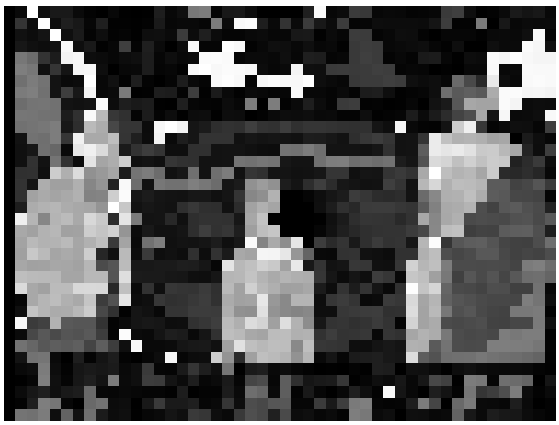

Figure 14: XOR of Figures 11 and 12. Note that in general, the differences are set to white (1) and the similarities are set to black (0).

Figure 15: THRESH 128 of Figure 11. Note that the brighter parts of the image have been set to MAXVAL (255).

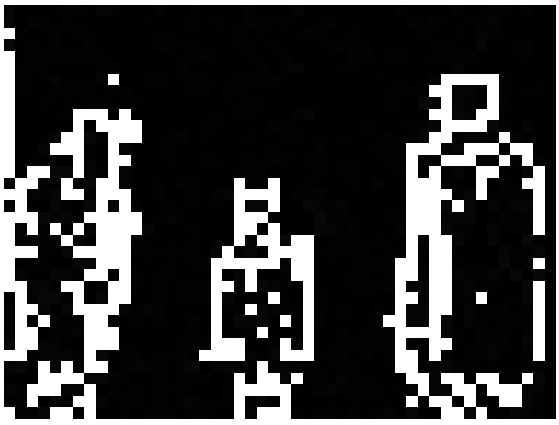

Figure 16: Result of edge detection algorithm performed on Figures 11 and 12. Note the correct identification of the edges of the people in the picture.