Hand-Drawn Circuit Recognition

Ravi Palakodety and Vijay Shah
6.111 Final Project
TA: Jenny Lee
May 2005

Abstract: This project will develop a tool for recognizing components, their values, and their connectivities from a hand-drawn circuit, and will use this knowledge to generate a primitive SPICE netlist. The user will draw his circuit on an 8x8 grid, with each component centered and sized to fill the block. The user's available components include resistors, capacitors, transistors, voltage sources, power supplies, ground terminals, and connectivity nodes. We will display the original hand-drawn circuit, a circuit with hand-drawn components replaced by computer generated pictures of components, and the text of the SPICE netlist onto the screen.

1. Introduction

For our 6.111 final project, we designed a tool that performs recognition of hand-drawn circuits. The user first draws his circuit on a special piece of grid paper. We then scan the picture and resize to 512x512 pixels using commercial software. Vijay designed PERL scripts to convert this bitmap to a csv file which is used to create a ROM on our FPGA. Our code then recognizes the various components and connections and generates a SPICE netlist that can be used to simulate the behavior of the circuit. Finally, we display on an LCD monitor the netlist and an idealized version of the circuit, with hand-drawn components replaced with computer generated ones. The netlist is also sent in text form through the RS-232 port. This project will smoothly take a circuit designer on the path from conception of the circuit to a usable SPICE netlist, without a tedious typing step.

The circuit recognition problem will prove extremely useful to analog circuit designers. Currently, it is fairly time consuming to move from the original "napkin-sketch" circuit to the SPICE netlist required to simulate the circuit. Also, it is fairly easy to make a mistake in typing the netlist, which then requires extra time in debugging the netlist. Tools such as CircuitMaker exist to automate the generation of the netlist. However, the engineer still needs to create his circuit in CircuitMaker, using a drag-and-drop interface with a separate wiring stage. This tool promises substantial time savings by simply requiring the engineer to scan his sketch using a standard scanner. Finally, this project will interest the AI community, many of whom are exploring projects such as "smart" paper. This project offers the possibilities of making your paper think like you, at least in the field of image recognition.

Our project can be broken into three major components: image recognition, netlist generation, and video output. The project was structured in a major/minor FSM structure, with Ravi handling the image recognition and Vijay handling the netlist generation and video output.

2. Design Description

2.1. Recognition Stage

The image recognition stage is implemented using a major/minor FSM design. The major FSM starts the three minor FSMs (memory handling, component recognition, and text recognition) and stores the its results in a shared RAM. The choose_component minor FSM is itself structured as a major/minor FSM, with specialized minor FSMs determining certain types of components.

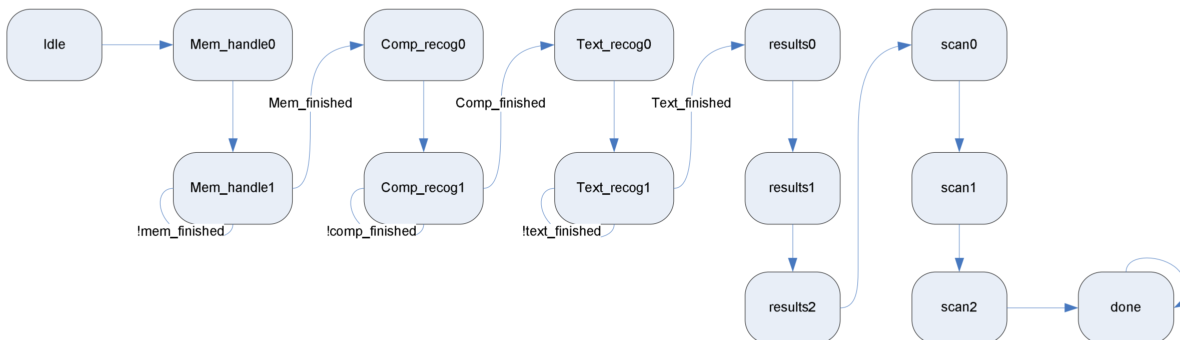
2.1.a. Major FSM (raviFSM.v)

The image recognition major FSM performs five major functions. First for each of the 64 grid blocks, the major FSM extracts the rows and columns from the image ROM to a row and column RAM. These RAMs are then used in component recognition and text recognition stages. Finally, the results of these recognition phases are written to a shared results RAM. After this sequence completes 64 times, the major FSM cycles through the 64 address of the

results RAM, allowing for simple debugging. The following table describes the states of this FSM.

State	Description
Idle	On reset of system, move to mem_handle0 and start recognition process at grid block 0.
Mem_handle0	Start memory handling minor fsm (mem_fsm), move to mem_handle1
Mem_handle1	Wait until memory handling is done, move to comp_recog0 on assertion of finished
Comp_recog0	Start component recognition minor fsm (choose_fsm), move to comp_recog1
Comp_recog1	Wait until component recognition is done, move to text_recog0 on assertion of finished
Text_recog0	Start text recognition minor fsm (text_fsm), move to text_recog1
Text_recog1	Wait until text recognition is done, move to results0 on assertion of finished
Results0	Two-stage writing of results RAM
Results1	Two-stage writing of results RAM
Results2	If finished all 64 grid blocks, move to scan1, otherwise, move to idle and begin process again
Scan1	Three-stage debugging states, cycle through results RAM
Scan2	Three-stage debugging states
Scan3	Three-stage debugging states
Done	Assert overall finished signal, indicate that recognition is complete. Remain in done state for rest of operation.

The following state transition diagram summarizes the operation of the major FSM.



2.1.b. Memory Handling Minor FSM (text_fsm.v)

The memory handling minor FSM fills two 64x64 RAMs with the rows and columns of the current grid block. The scanned image was originally a 512x512 bitmap, which was stored in a 64x4096 ROM. Thus, the first address of the ROM contains the first 64 pixels of the first line, while the second address contains the next 64 pixels of the first line – not the next row of the first grid block. Thus, some addressing logic is required to read the rows of

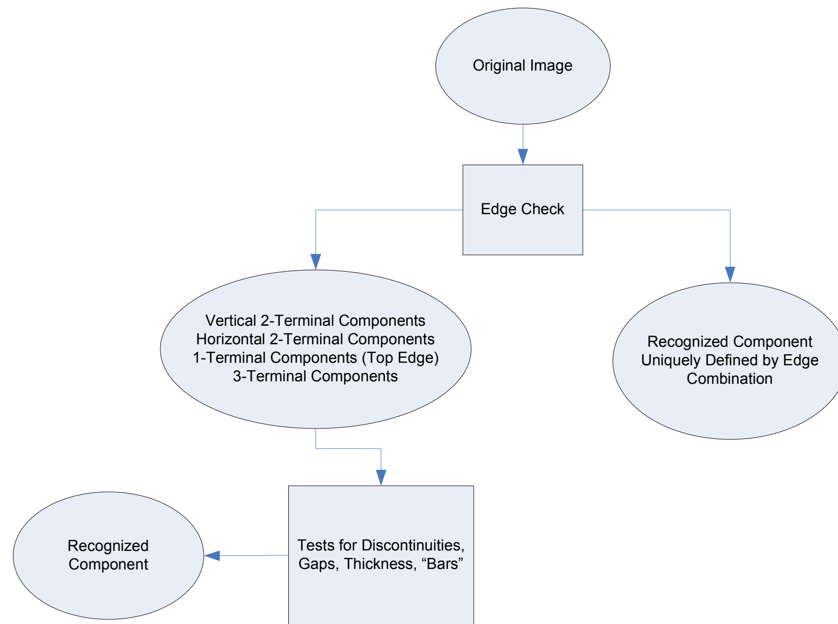
the current grid block from the ROM, and then write these to the row RAM. The FSM then assembles the columns of the grid block from the filled row RAM. Here, we made the tradeoff between area and speed; since we were limited by the space on the FPGA but had no major time constraints, we chose to slowly assemble the columns using a single 64-bit shift register and multiple passes through the row RAM. When both RAMs are full, this FSM asserts its finished signal. The following table describes the states of this FSM.

State	Description
Idle	Stay in idle state until assertion of start signal
Delay0	Delay to ensure correct output of rom
Write_row_ram	Write one row of grid block to ram, increment row_ram_add and move to delay0
Delay1	Delay to ensure correct output of row_ram
Load_reg	Cycle through row_ram extracting a single bit that corresponds to current column. Store bit in shift register
Write_col_ram	Write filled shift register to column ram, return to delay1 if col_ram not full
Done	Assert finished, and return to idle

The state transition diagram for this FSM is located in the appendix.

2.1.c. Component Chooser Minor FSM (choose_fsm.v)

The component chooser minor FSM performs the image recognition necessary to determine the type of component in the current grid block. This minor FSM is itself structured in a major/minor FSM design. Here, the major FSM (choose_fsm.v) determines which edges are terminals of the component. Based on this information, the major FSM starts the appropriate minor FSM which handles that edge combination. For example, if the major FSM determines that the component has terminals on the left and right edges of a grid block, then the component is a horizontal two-terminal element, and the major FSM starts the h2term minor FSM. Specialized minor FSMs exist for horizontal two-terminal elements, vertical two-terminal elements, one-terminal elements intersecting the top edge, and three terminal elements with edge combinations of {left, top, bottom} or {right, top, bottom}. As more components are added, minor FSMs can be modified or added. The following table describes the states of this FSM. State transition diagrams for these minor FSMs are located in the appendix. We use a decision tree methodology for component recognition, with the tree shown below.



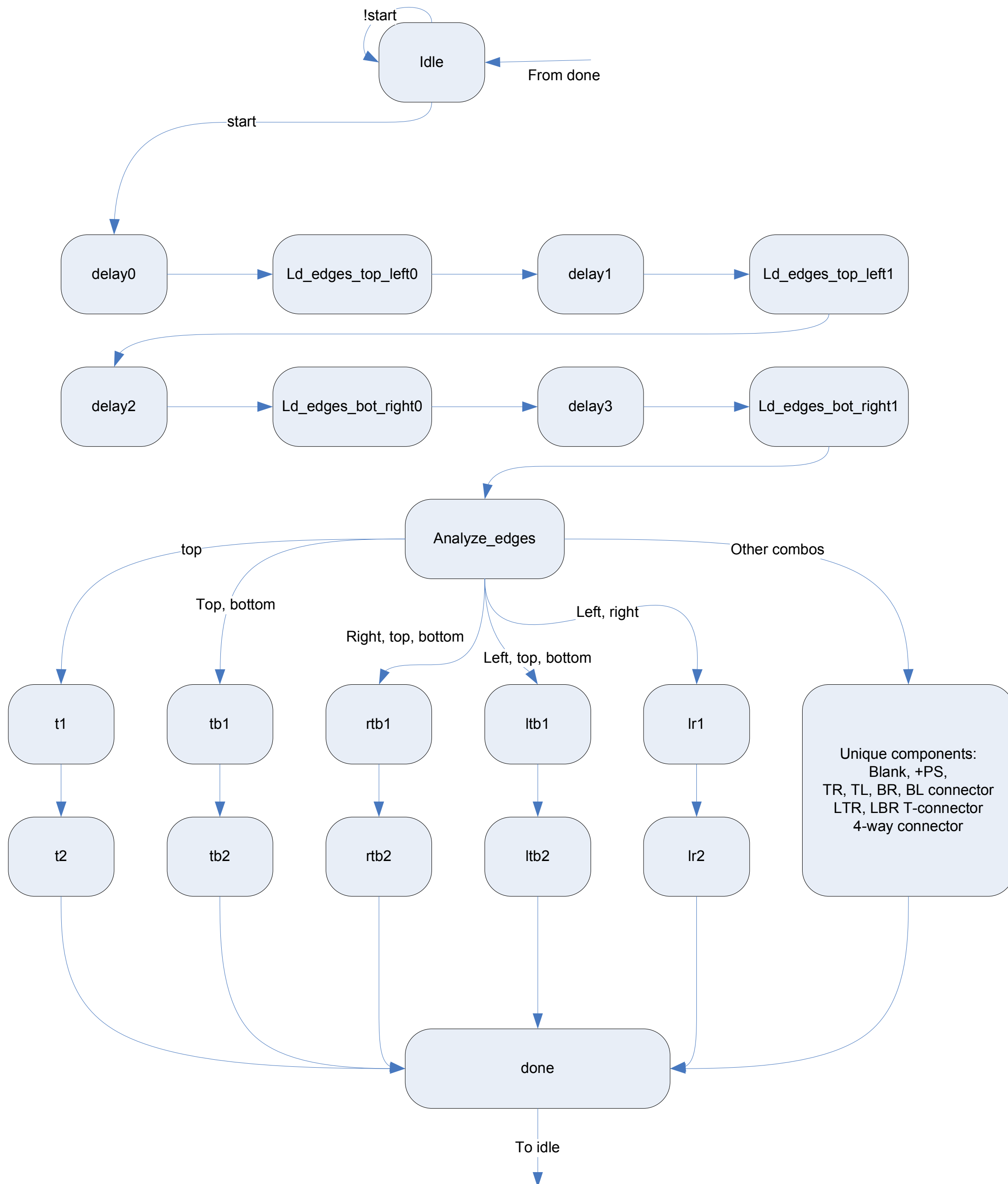
We use a series of different tests for these components. For example, a horizontal or vertical two-terminal component can have a discontinuity (cap), a gap (src), be thick (res), or be thin (wire). A three-terminal component can either have a bar at the base of the transistor or no bar at the base, signifying a t-connector. Finally, a one terminal component can be a ground node, which has a discontinuity, or a power supply, which does not.

The following table describes the states of this FSM.

State	Description
Idle	Stay in idle state until assertion of start signal
Delay0	Delay to ensure correct output of row_ram, col_ram
ld_edges_top_left0	Check row 3, col 3 for intersection
Delay1	Delay to ensure correct output of row_ram, col_ram
Ld_edges_top_left1	Check row 4, col 4 for intersection
Delay2	Delay to ensure correct output of row_ram, col_ram
Ld_edges_bot_right0	Check row 58, col 58 for intersection
Delay3	Delay to ensure correct output of row_ram, col_ram
Ld_edges_bot_right1	Check row 59, col 59 for intersection
Analyze_edges	Based on edge combination, move to specialized states
Unique states: N1, b1, r1, rb1, rt1, L1, Lb1, Lt1, Lrtb1, Lrt1, Lrb1	Edge combinations that uniquely determine component type. N1 is blank grid block. Other state names indicate the edges that are crossed. Each state has a unique type code that will be written into the results RAM
T1, t2	Start the t1term FSM to choose between negative power supply and ground node
Tb1, tb2	Start the v2term FSM to choose between vertical resistor, capacitor,

	source, or wire
Rtb1, rtb2	Start the rtb3 FSM to choose between NPN transistor and T-connector
Lr1, Lr2	Start the h2term FSM to choose between horizontal resistor, capacitor, source, or wire
Ltb1, Ltb2	Start the ltb3 FSM to choose between NPN transistor and T-connector
Done	Assert finished signal and return to idle state

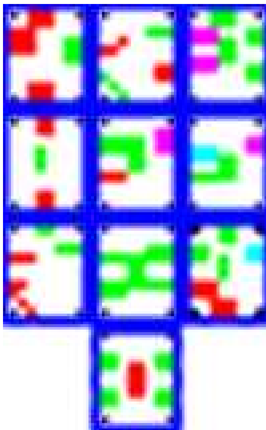
The following state transition summarizes the operation of this FSM.



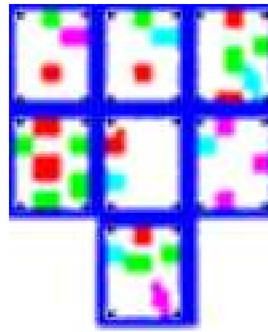
2.1.d. Text Recognition Minor FSM (text_fsm.v)

The text recognition minor FSM performs the text recognition necessary to determine the value (including multiplier) of the component in the current grid block. As mentioned before, each letter or number is written into a 10x8 block in the lower right of the grid block. This FSM contains 10 boolean variables which show whether the handwritten picture could be a certain letter or number. For example, if the pixels in the box meet certain constraints, then the FSM determines that the number might be a one, a two, etc. The sets of pixels (pads) are chosen such that a reasonable representation of the number will yield only one possible choice. These pads are shown in the following graphic. Red shows areas that the letter is not allowed to touch. Green indicates areas that the letter must touch. Purple indicates areas that the letter must not "cross." For example, for a 5, there must not be a continuous vertical path from the top-right to the middle-right. Light blue indicates areas that the letter must cross.

Numbers 1,2,3,
4,5,6
7,8,9
0



Letters F, P, N
U, m, K
M

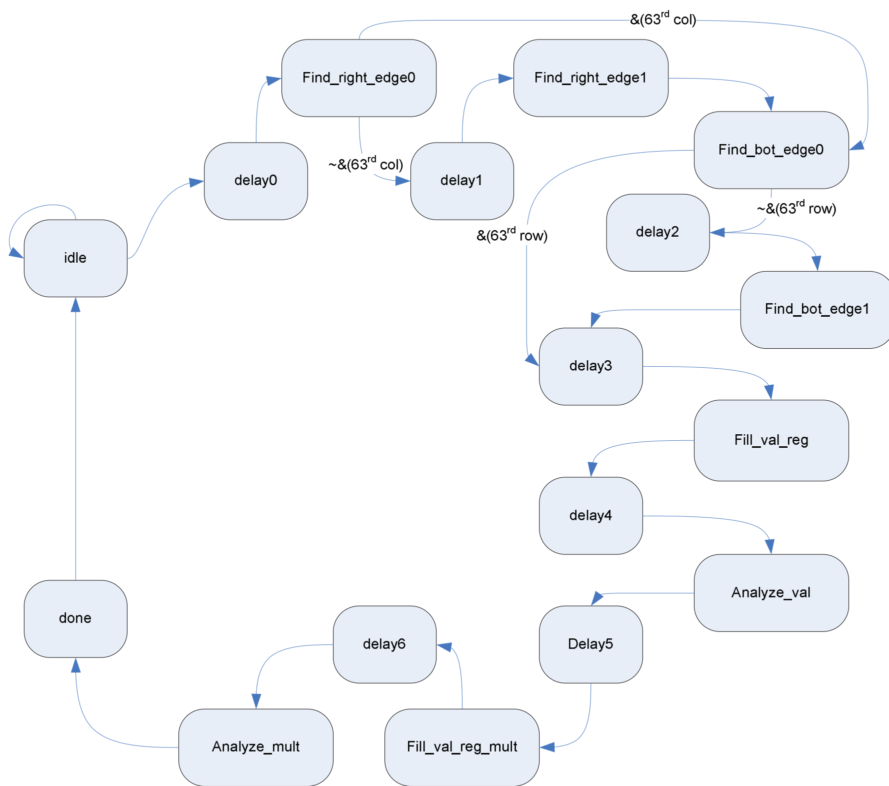


Our text recognition requires that we are able to fill the val_reg with the exact set of 80 pixels corresponding to the handdrawn letter. However, when scanning an image, a small amount of rotation is inevitably added, such that we need to find the location of this 10x8 block. To do this, we added some error correction states which attempt to determine how much the grid block has shifted during scanning. This error correction is able to correct 1-pixel shifts. Also, our pads have a certain amount of leniency such that most of them can handle shifts of one pixel. This table shows the different states of the text recognition FSM.

State	Description
Idle	Stay in idle state until assertion of start signal
Delay0	Delay to ensure correct ram output
Find_right_edge0	Try to determine right alignment
Delay1	Delay to ensure correct ram output

State	Description
Find_right_edge1	Try to determine right aligment
Find_bot_edge0	Try to determine bottom alignment
Delay2	Delay to ensure correct ram output
Find_bot_edge1	Try to determine bottom alignment
Delay3	Delay to ensure correct ram output
Fill_val_reg	Begin filling shift register holding one 10x8 block – number
Delay4	Delay to ensure correct ram output
Analyze_val	Analyze filled val_reg (ie complete 10x8 block) – number
Delay5	Delay to ensure correct ram output
Fill_val_reg_mult	Fill shift register with 10x8 block – multiplier (letter)
Delay6	Delay to ensure correct ram output
Analyze_mult	Analyze filled val_reg – letter
Done	Assert finished signal and return to idle state

The following state transition diagram summarizes the operation of this FSM.



2.2. Analysis and Video Output

After image and text recognition is complete, the system can analyze the circuit to discover and label its nodes. An *analysis* module explores the circuit using a Depth-First Search with enqueued list. A *stack* supports the analysis by providing a First-In Last-Out

(FILO) structure to serve as working memory for the search. After the search completes, the system displays the results of recognition and analysis using a standard VGA monitor.

The video output of the system is an 800x600, 1-bit color display. The entire screen contents are held in a dual-port RAM module on the FPGA. The video subsystem is divided into two classes of modules: those that read from the video RAM, and those that write to the video RAM. Because the video RAM uses a dual-port architecture, the read & write sections operate independently of each other.

The read section runs continuously during system operation, driving the display with the video RAM contents and the appropriate control signals. The *sync generator* is responsible for keeping track of the current pixel location and asserting the sync signals during the blanking periods. The *display manager* takes the current pixel location and sends the corresponding video RAM value to the monitor.

The write section consists of three modules to handle the three different video output modes. The *raw circuit display* module reads from the ROM containing the user's scanned circuit image and writes it directly to video RAM, centering the 512x512 image on the screen. The *ideal circuit display* module takes the recognized circuit data stored in the results RAM, activates the analysis module to assign nodes, and then redraws the circuit with predefined sprites stored in ROM, annotating each component with its value and node information. The *spice display* module also uses the analysis module to assign nodes, then combines that data with the information in the results RAM to generate a primitive SPICE netlist describing the circuit. A *Major FSM* delegates memory control and display module activation according to the state of the input switches set by the user. A block diagram for the analysis modules and video subsystem is given on page 74 of the appendix.

2.2.a. Analysis

The analysis module has the task of finding all nodes in the circuit and assigning each node a unique label. The one constraint on node labels is that ground nodes must have the "0" tag; all other node identifiers can be arbitrary character strings. The analysis module uses a simple counter as a source of labels. The counter starts at 1 (because 0 is reserved for ground nodes), and increments whenever a new node is discovered.

Because the circuit is contained within a grid structure, with exactly one component or wire junction per grid location, the set of possible node locations is easily defined as every interior row and column boundary. The *node value* RAM has 112 slots, one for every possible node in an 8x8 grid. When the analysis module begins processing, it initializes every slot to 113, which is a special reserved value for unassigned nodes.

After the node value RAM is initialized, the analysis module begins the task of finding and labeling nodes. A Depth-First search is guaranteed to completely explore the circuit, as any valid circuit must be a connected graph. The steps for the Depth-First search algorithm are:

1. (Initialization) Perform a linear scan of the grid, stopping when an occupied grid block is found. Push this grid location onto the stack.
2. Pop the most recently added grid location off of the stack, and follow the heuristics for node assignment for all row and column boundaries with wire edges.
3. Push all adjacent, connected grid locations that are not already in the enqueued list onto the stack. Loop back to step 2. Stop when the stack is empty.

The enqueued list ensures that the search won't enter an infinite loop. A 64-bit register models the enqueued list. Each bit represents a grid location (bit n represents the grid location in row n div 8, column n mod 8), with a value of 1 meaning the location has been queued and 0 meaning that it has not been queued. The stack module described in the next section provides the FILO memory abstraction.

For node assignment, the basic strategy is to divide occupied grid locations into two categories: wire junctions and components. Wire junctions must have the same node value at all wire edges. If there are conflicting existing assignments, the system gives precedence to the lowest value, reassigns all wire edges to this value, and rescans the table to propagate that reassignment. The lowest value is chosen so that ground nodes are never reassigned. If there are no conflicts, then the system just assigns the same value to all edges, generating a new value if all edges were previously unassigned. A components usually, but not always, has different node values at its wire edges. The system respects any existing assignments, and generates new values for unassigned edges. Ground nodes are the only exception to this rule, as they automatically take the "0" value for the wire edge, and if there was an existing assignment, that change is propagated throughout the node value RAM. A complete flowchart for the node assignment process is given on page 77 of the appendix. Once node assignment terminates, the analysis module asserts its finished signal.

2.2.b. Stack

A stack is a memory element that follows a First-In Last-Out (FILO) policy. The stack supports two operations: push and pop. A push writes a new value to memory. A pop removes the most recently added value from memory and returns that value. The stack in this system exists to support the Depth-First search algorithm used by the analysis module.

The stack uses a 7x64 RAM on the FPGA as the bottom layer for its abstraction. On initialization, the stack module writes a special Start-Of-Stack symbol to address 0 of the RAM and waits for commands. Modules using the stack use a 2-bit command input to send instructions, and a bi-directional 7-bit bus to send and receive data. The stack module only drives the bus while performing a pop operation. Connecting modules should tristate the bus by default, only driving the bus during push operations.

To push data, a module sets the command input to 2 and places the data on the bus, then waits three clock cycles. The stack module FSM checks the command input on every positive clock edge and moves to the first push state when it sees that command code. The stack RAM address is incremented during that first push state. The stack asserts the write enable signal for the RAM on the next clock cycle, and uses the last of the three clock cycles to hold the data after the write.

A module sets the command input to 3 in order to pop data off of the stack. Because the stack RAM address already points to the most recently added item, the stack module only needs to start driving the bus with the RAM output, and the data is available on the next clock cycle. The data remains available for an additional clock cycle while the stack decrements the RAM address. If the most recent value was the Start-Of-Stack symbol, the stack does not decrement the RAM address. The stack does not need to delete popped values from the RAM, as the address increments only before writing new data, and the address does not decrement past the Start-Of-Stack slot.

2.2.c. Video Read - Sync Generator

VGA output consists of two periods: an active region, when the monitor is receiving user-visible pixel information, and a blanking region, when the monitor is in the middle of moving to the next line or frame. Each blanking region is divided into a front porch, sync pulse, and back porch. During the front and back porch, the color channels are set to output only black pixels. The green channel is set to the sync level during the sync pulse. The sync generator controls the horizontal sync, vertical sync, and blank signals to manage the blanking region portion of VGA output. This module also outputs two counters that define the system's conception of the current pixel location.

For an 800x600 display running at a 72-Hz refresh rate, the sync generator needs to follow the standard VGA timing specifications listed in Table 1.

Format	Pixel Clock (MHz)	Horizontal (in Pixels)				Vertical (in Lines)			
		Active Video	Front Porch	Sync Pulse	Back Porch	Active Video	Front Porch	Sync Pulse	Back Porch
800x600, 72Hz	50.000	800	56	120	64	600	37	6	23

Table 1: 800x600, 72-Hz timing specifications. Reproduced from lab kit website at http://www-mtl.mit.edu/Courses/6.111/labkit/vga.shtml.

Generation of the pixel clock is handled by a simple Digital Clock Manager (DCM). All modules in the video subsystem run at the pixel clock for simplicity. The sync generator uses an FSM to control whether the current display mode is horizontal active video, front porch, sync pulse, or back porch. An additional 2-bit line-mode register controls whether the current vertical region is active video, front porch, sync pulse, or back porch. The vertical line counter is incremented every time the FSM enters the horizontal active video state, and the line-mode register changes when the counter reaches the limit for that mode.

2.2.d. Video Read - Display Manager

The display manager continuously reads from the video RAM, with the address based on the current pixel location data provided by the sync generator. This module also takes the hsync, vsync, and blank signals, generates a new composite sync signal, and sends all sync and pixel data to the monitor and the ADV7125 video DAC.

Each location in the 8x60000 video RAM is an 8-pixel chunk of a row. Therefore, given an 800x600 resolution at 1-bit color, there are 100 locations per row. The display manager sets the read address to ((current column div 8) + (current row * 100)) by continuous assignment. The RAM has a 1-cycle read delay, so the pixel data sent to the DAC is actually the bit corresponding to (current column – 1). All other VGA signals are delayed by 1 clock cycle to compensate for the RAM. The video DAC has a 2-cycle pipeline delay, so the hsync and vsync signals are delayed another 2 clock cycles, as they are sent directly to the monitor.

The composite sync signal is simply the XNOR of hsync and vsync. This sync signal passes through the video DAC, so it is delayed only one clock cycle. The DAC expects pixel data in 24-bit color form, so the display manager simply ties each 8-bit color channel to 0xFF if the current pixel is white, and 0 if the current pixel is black.

2.2.e. Video Write - Raw Circuit Display

Raw circuit display simply sends the contents of the ROM holding the user-supplied hand-drawn circuit bitmap to the video RAM, centered on the 800x600 screen. The word size for the ROM is 64-bits, while the video RAM uses 8-bit words, so the raw circuit display module performs 8 write operations to video RAM for every read from the circuit ROM. The size of the circuit bitmap is always 512x512, so the module only needs to add horizontal and vertical offset constants to the video RAM address in order to center the bitmap.

An FSM controls the flow of operations in the raw circuit display module. When this module is activated, the FSM writes 0xFF to every location in video RAM in order to clear the display. The FSM then starts at the first location of circuit ROM, which corresponds to the upper left-hand corner of the bitmap, and writes 8 8-bit chunks of the ROM data sequentially to the video RAM. Each write takes three cycles: the address and data are set during the first cycle; the RAM write enable signal is asserted during the second cycle; and the address and data are held during the third cycle after the write enable signal is lowered. A simple 3-bit counter keeps track of which 8-bit chunk of ROM data to write to the video RAM.

After all circuit data has been written to video RAM, the module enters a termination state, and remains there until the user chooses to leave raw circuit display mode. The FSM moves to the idle state and awaits the next activation.

2.2.f. Video Write - Ideal Circuit Display

The ideal circuit display module has three different types of data to write to the display: sprites for the components and wire junctions, based on the recognized data in the results RAM; sprites for the characters representing the values for components, also based on the results RAM data; and sprites for the characters representing the node assignments, based on the node value RAM. The module writes all three types of data to the video RAM sequentially for each grid location.

The module's FSM moves from the idle state to an initialization state when it receives an activation signal. The FSM clears the video RAM contents during initialization, writing 0xFF to every location, and then sets the results RAM address to 0. For every grid location, the ideal circuit display module reads the data from the results RAM, reads the appropriate component/wire junction sprite from the component ROM and writes it to video RAM, reads the character sprites for the component value from the character ROM and writes them to video RAM, and finally reads data for the four node locations from the node value RAM and writes the corresponding character sprites to video RAM. The sequence terminates when all 64 locations in the results RAM have been processed.

The sprites in the component ROM are 64x64, so the ideal circuit is 512x512, the same as the raw circuit. The circuit is centered on the display by adding constant vertical and horizontal offsets to the video RAM address. The 8x8 character sprites for the component values are written in the lower right-hand corner of each grid block, with the numeric value starting at a relative location of (40,48), and the multiplier value below it at (56, 56). Node annotations are only written for components, in order to avoid unnecessary visual clutter. Each component has at most four possible node locations at its top, left, right, and bottom edges. The relative locations for these edges are (24,-8), (-16, 32), (48, 32), and (24, 56). Note that the negative offsets for the top and left node locations means that they will

overwrite the bottom and right node locations for the top and left adjacent grid blocks. This strategy lets the system simply avoid repetitive node annotations when two components are adjacent.

After all 64 grid locations are processed, the FSM loops in a termination state, until the activation signal for the module is lowered. The FSM then returns to the idle state until the activation signal is raised again. A simplified state transition diagram is given on page 76 of the appendix.

2.2.g. Video Write - Spice display

Spice display writes a basic SPICE netlist to the video RAM. For every component in the circuit, the SPICE description requires a unique label, identifiers for the nodes at its terminals, a type string (only for certain components), and a component value. All information is represented on the display using the character ROM sprites.

The spice display FSM follows the same basic process flow as the ideal circuit display FSM. The FSM reads the results RAM for a grid location and, if it holds a component, it generates a label by concatenating the one-letter component code with a counter value specific to that component. Next, the FSM reads the node value RAM locations corresponding to that component's terminals, and writes the three terminal values, separated by spaces. For two-terminal devices, the third terminal is replaced by whitespace. Voltage sources and transistors are the only two components that fill the type field. The current circuit recognition routines do not support specification of component types, so these values are hardcoded to "DC" for voltage sources and "NPN" for transistors. All other components fill this field with whitespace. The FSM writes the component value and multiplier code found in the results RAM value, and then finally writes an End-of-Line symbol.

All SPICE information is written first to a spice RAM before the appropriate character sprites are written to video RAM. The spice display module takes this approach to support the serial export module, which reads the spice RAM and exports an ASCII text file containing the SPICE data to a PC over a null-modem serial cable. When the spice display module is done writing to the spice RAM, it writes an End-of-File symbol, so that the serial export module knows where to end transmission. After finishing with writing to the spice RAM, the spice display module first clears the video RAM, then starts reading the character codes from the RAM and writing the corresponding character sprites from the character ROM. Because all null fields are filled with whitespace, the lines are neatly formatted with each field lining up vertically, and the width of each line is known to be 21 characters. The output is centered horizontally on the monitor by adding a horizontal offset constant to the video RAM address. As with the other display modules, the spice display module enters a termination state when video output completes, looping there until the module is deactivated. The FSM then returns to the idle state until the next activation. A simplified state transition diagram is given on page 75 of the appendix.

3. Design Decisions

The image and text recognition was implemented using simple and intuitive logic techniques rather than sophisticated signal processing. As mentioned before, a decision tree was used for component recognition, as we looked for noticeable features of different

components. Text recognition also used a decision tree, where the noticeable features were different sets of pixels on the 10x8 letter area. Here, we made a trade-off between scalability and feasibility. For our limited component library, the heuristic based decision tree was extremely accurate for component recognition. We originally tested recognition using a 2-D correlation in Matlab, and found that additional heuristics would be needed for accuracy; after testing our heuristics, we determined that the 2-D correlation was unnecessary. Another trade-off was one of size versus amount of information. Our images were saved as monochromatic bitmaps (ie 1-bit color). This severely limited the accuracy of some of the image recognition algorithms. For example, if we had used a 2-D correlation, a black grid block would correlate well with every component. Also, we used a rather low resolution for our images, again limiting our ability to effectively use image recognition algorithms. Finally, as mentioned before, we faced a space constraint in the capacity of our FPGA, but no time constraints. As a result, we chose to implement a slower algorithm rather than use large registers and memories, as shown in the memory handling FSM.

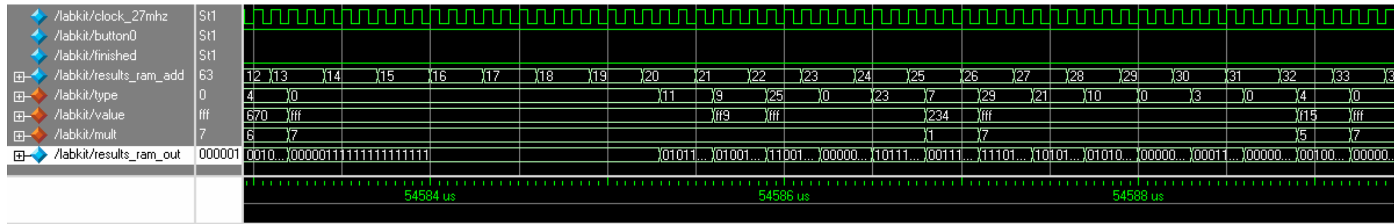
The analysis module uses the Depth-First search algorithm for efficiency in exploring the circuit. A linear scan of the results RAM would also return correct results, but spends a lot of time unnecessarily searching unoccupied grid locations for sparsely populated grids. For an 8x8 grid, the effect on system speed is negligible, but the improvement grows rapidly as grid sizes increase. The enqueued list is modeled as a 64-bit register, instead of using a RAM, because the small amount of information the system needs to store is judged to be not worth the complexity of a RAM interface. On the other hand, the stack is implemented as a stand-alone module instead of a simple 64-slice array because a stack is a useful abstraction to have, even if it is only currently used by one module. Also, although the addition of a stack module increases the total amount of complexity in the system, it reduces the complexity of the analysis module, and makes the Depth-First search state flow more natural and comprehensible.

The 800x600 resolution for the video display was selected because the system needed to display the 512x512 circuit bitmap. 800x600 was the smallest standard VGA resolution that could fit a 512x512 bitmap. An alternative choice would be to scale the bitmap down, but it is desirable to keep the reference bitmap on screen as faithful to the original as possible. 1-bit color was chosen for two reasons: there wasn't much need for more than two colors, as the recognition routines only operate on black and white values, and the FPGA would not be able to support the RAM size if the color depth increased much further. The use of more color would have allowed the system to present its results in a more engaging manner, perhaps by color coding components, but this benefit was judged to be not worth the added complexity of moving the video RAM to the external ZBT SRAMs.

4. Testing

Testing was first carried out using a behavioral simulation in ModelSim. As seen above, Ravi added 3 states at the end of his major FSM that simply cycled through the results RAM so he could observe the accuracy of the recognition. Multiple real circuits were prepared and scanned. In addition, circuits were prepared that simply filled the 8x8 grid, without regard to connections. Finally, a circuit was prepared with all the numbers and letters written multiple times, to test the text recognition. The component recognition was 100% accurate when the user followed the specified rules for drawing. Text recognition was

slightly more difficult, but we still reached 95% accuracy, with the "eight" being the most difficult number to correctly characterize. The following simulation shows a typical test attempt. The various codes for components and multipliers are shown in the appendix.



Because Vijay finished the video portion of the project before Ravi finished the image recognition, we were able to test the image recognition on the video output as well. The only problem with this was the extremely long wait to generate a programming file.

The stack was implemented and fully tested before work on the analysis module began, because subtle errors in the stack functionality could have been hard to detect within the context of the Depth-First search. ModelSim behavioral analysis combined with manual forcing of relevant parameters was used first to verify that high-level state flow and signal assertions proceeded as designed. Next, post-place & route simulation, combined with a simple testbench, ensured that the timing contract for pushes and pops worked within the specific timing specifications of the FPGA.

Once the stack was satisfactorily tested, the analysis module was implemented and subjected to a similar series of tests. The results RAM was pre-populated with a test circuit, and then the waveforms in behavioral analysis visually verified that the first few steps worked as expected. The process takes too long to completely inspect step-by-step, so the “mem display” ModelSim command was used after the analysis module terminated to dump the contents of the node value RAM, and the values were compared against hand-calculated assignments. A few different sample circuits were also tested to guard against corner cases, and then finally the tests were repeated under post-place & route simulation to ensure that there were no timing violations.

The video display modules were much easier to test, as their output was directly verifiable on the monitor. Basic video output was easy to implement, as the sync generator was mostly completed from a previous laboratory assignment. Prepopulating the video RAM with images ensured that the display manager worked correctly. Raw circuit functionality was straightforward and showed that the dual-port video RAM architecture worked. Ideal circuit and spice display were both tested by prepopulating the results and spice RAMs with sample data and visually verifying that the screen contents matched expectations. In the few instances where the reason for failure was not immediately apparent from the screen output, the logic analyzer was used to examine the state variable and RAM address & data values.

5. Conclusion

Our project demonstrates the ability to:

- Perform image and text recognition on a scanned bitmap in the domain of an electrical circuit

- Analyze the circuit to discover and label electrical nodes
- Output the results of recognition and analysis in both graphical and textual form

Our project uses no ICs or connections to external devices (besides the serial cable for exporting data), showing that it is possible to implement a complex and interesting digital system entirely within an FPGA. Our design makes heavy use of RAMs and ROMs for data storage and sharing, and FSMs for coordination of process flow. The logic for recognition and analysis is extensive, and implementation was manageable only because significant time was spent designing flowcharts and state transition diagrams beforehand. The logic analyzer was invaluable in debugging, because many subtle timing issues did not manifest themselves in behavioral simulation, and post-synthesis simulation proved to be just as slow as compiling the project and outputting from the labkit to the logic analyzer.