1984: An Object Tracking Surveillance System

by Robert Crowell and Lyric Doshi

Abstract

The *1984*: An Object Tracking Surveillance System will track the motion of any arbitrary target selected using a mouse. When the mouse is clicked, a video camera visually identifies the color of the object the laser was targeting and tracks the movement of the target. A simple particle filter, initialized based on the target's color and location, will be used for image processing. The camera will be mounted to a movable platform mounted to a servo, allowing the system to keep its target within the center of the video screen.

The final system tracked lighter colors fairly well, especially skin color. The VGA display showed could show individual particles or a box around the center of mass of the particles. The tracking experienced issues due to the quality of the image. Some simple filters were used to improve image quality, but high fluctuations in pixel color still adversely affected the system's tracking ability. Improvements to the system center around improving the image data with more sophisticated filters and double buffering the camera data.

Table of Contents

List of Figures

1 Overview - Lyric

The *1984* system is capable of tracking the motion of an arbitrary target selected by the operator of the system. Originally, a laser pointer was going to be used to select a target to track, but the final version of the system uses a mouse to select to target from the VGA display. When the mouse is clicked, the system identifies the color of the object the mouse was targeting and continues to track the movement of this object. A simple particle filter, initialized based on the position of the object selected by the mouse, is used for image processing. The camera is mounted to a movable platform rotated by a servo, allowing the system to keep its target within the center of the video screen. To demonstrate tracking, the system either draws all the particles used for tracking or a box around the center of mass of the particles.

This report explains the theory and modules created to build this system. The system begins with video data from a security camera and decodes the NTSC signal. It interpolates between the YCrCb color data to improve the image color quality and then passes the data to an asynchronous FIFO. Next, the system applies median filtering before storing the image data in a dual port ZBT RAM. From there, the VGA display module constantly reads image data to display. During the blanking period of the camera, the particle filter reads and analyzes the image data when it has a target to track, updating particle positions for display in the process. Section 2 explains the Particle Filter Theory. Section 3 discusses Image Processing Theory, while Section 4 discusses the Design and Implementation. Section 5 explains Testing and Debugging, Section 6 gives the conclusion, and Section 7 presents the Acknowledgements.

2 Particle Filter Theory - Lyric

The Particle Filter uses a set of independent particles to track a given target by color. Each particle has a position, velocity per frame, and assigned color. At initialization, particles are placed on the target object in the image, given random velocities, and assigned the target's color from the image data. At each frame of video, each particle checks to see if it is still on its assigned color. If so, it moves one step using its velocity and awaits the next frame. If the particle is no longer above pixels of its assigned color, it is removed. Finally, new particles are spawned at the center of mass of the currently living particles. They have the same assigned color as the other particles and an initial velocity based on the velocity of a randomly chosen living particle with a little random variation. In short, particles that are successfully tracking the assigned color will survive from frame to frame and spawn new particles that will ideally do the same thing. The random variation in the velocities of newly spawned particles allows the filter to adapt to changes in velocity of the target. A particle may consist of multiple pixels, in which case a threshold is set to determine how many pixels must be of the assigned color for the particle to survive. To check these pixels, the image data is converted to the hue-saturation-value (HSV) color space because HSV makes the system more robust to changes in lighting affecting the target's color.

3 Image Processing Theory - Rob

The *1984* system interfaces with the composite video output of a surveillance camera which encoded NTSC video according to the ITU-R BT.601 standard. Video frames are encoded in the YCrCb color space, which allocates ten bits for the luminance data, Y, and ten bits for each of the two chrominance values, Cr and Cb. The interlaced NTSC video provided by the camera has 720 pixels per line, with 485 lines of active video per frame (accounting for horizontal and vertical blanking, each frame is 858x525). Since the video is interlaced, each frame is actually sent as two consecutive fields, even and odd, each field containing every other row of pixels in the frame. Furthermore, in order to reduce the size of the video, each frame is encoded in a 4:2:2 format in which only half of the chrominance information for

each pixel is provided. While each of the 720 active video samples in a row includes luminance information, the odd pixels include only Cr and the even pixels only Cb. A video decoder must therefore infer each missing chrominance value from the surrounding pixels. The video produced by the camera also includes a significant amount of "salt and pepper noise," which causes the color of a given pixel to fluctuate significantly through time even when the scene is not changing.

Optimized to remove information which is not discernable by the human eye, the compressed and interlaced format of NTSC frames can be problematic for image processing applications. Moving objects appear distorted around their edges due to interlacing; since only half of the rows of video are updated at a time, the "ghost" of quickly moving object will trail behind the object as it moves. This issue is resolved by buffering both fields of a frame before allowing the image processing algorithm access to the new frame. To account for the lossy compression which discards half of the chrominance in each pixel, a simple linear interpolator provides horizontal coherency by averaging the relevant chroma values from the two nearest pixels on the row. While more advanced techniques are available that consider information contained in nearby pixels in adjacent rows, or pixels at the same location in previous and future frames, this simple interpolator is sufficient for *1984*. Salt and pepper noise is reduced by a median filter, which sorts the color values of a pixel and its four nearest neighbors in its row and assigns the current pixel's value to the median of this list. Though it slightly blurs the image, the median filter eliminates much of the noise in a frame and provides the particle filter, which relies heavily on chrominance information, with a more continuous and accurate representation of the scene.

4 Design and Implementation

The discussion of design is split into two parts, video acquisition and particle filtering.

4.1 Video Acquisition - Rob

The video acquisition stage of *1984* is handled by a series of modules which pass pixels through a chain starting from the video decoder chip and ending at the SRAM containing the video buffer. Due to the fact that pixels arrive at a slower rate than the rest of the system, an asynchronous FIFO queue is employed to bridge the two clock domains. Pixel filtering and conversion from YCrCb to RGB is performed before a pixel is stored into the RAM for use by the particle filter and display onto the screen. Figure 1 gives the video acquisition block diagram.

4.1.1 ADV7185 Initialization and the i2c Interface

Two modules provided by Nathan Ickes and available on the 6.111 website, **ADV7185init** and **i2c**, initialize the ADV7185 NTSC decoder chip. The ADV7185 contains a number of internal registers which define any processing that is done on the video input, as well as the video source (composite or s-video) and its properties. For example, the value of one register determines the brightness of a scene, while another determines which of the available filters the video is passed through. The FPGA accesses these registers over an i2c bus, which is a standard single-bit bus with a second clock line. Since *1984* does not need to read data from the registers, the **i2c** module provides a slow clock signal at 250KHz and drives data onto the bus one bit at a time. The **ADV7185init** module defines the desired values for each of the ADV7185's internal configuration registers, and sets them during a system reset by providing data to the **i2c** module 8 bytes at a time.

4.1.2 NTSC Decoder

The **ntsc_decoder** module interprets the raw input stream provided by the camera through the ADV7185 and writing it into an asynchronous FIFO. Samples are provided by the decoder based on an external 27MHz line-linked clock, *llc*, which is determined by the data coming in from the camera; ten new bits of video data are available on each rising edge of *llc*. To eliminate glitches due to the propagation delay between the ADV7185 and the **ntsc_decoder** module on the FPGA, however, the data stream from the decoder is sampled on the falling edge of *llc*.

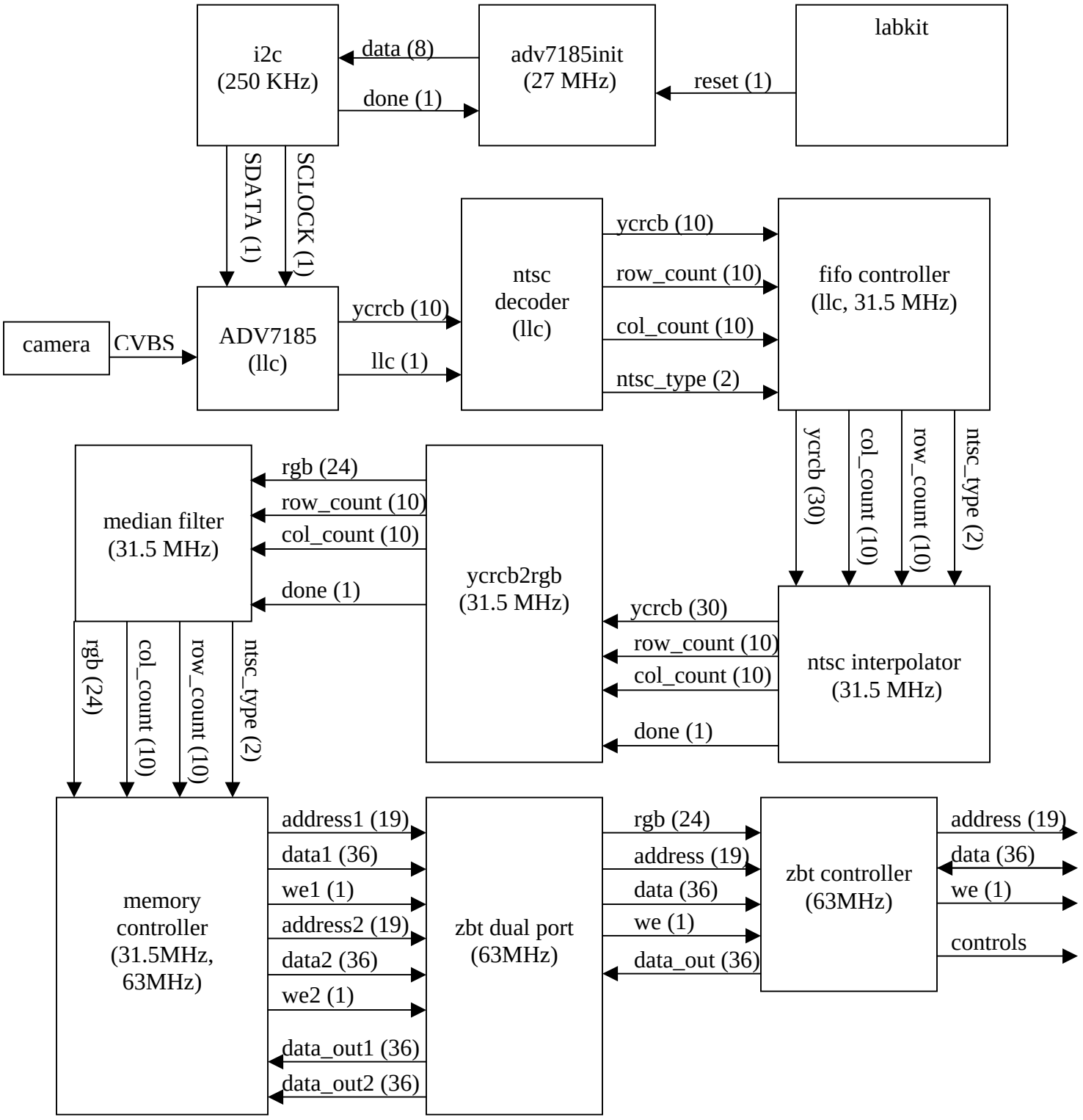

Figure 1: Video Acquisition Block Diagram

Figure 2 shows the FSM for **ntsc_decoder**. Following a system reset, **ntsc_decoder** waits for the upper-left pixel of a frame to be sent by the ADV7185. The decoder watches for the characteristic 3FF_000_000_XY sequence of samples that indicates the beginning or end of a row of active video. The XY data includes a field number (0 or 1 representing odd or even rows) and two boolean flags indicating whether the next set of frames occur during a horizontal or vertical blanking period. Because the current row or column is never transmitted in the video stream, the beginning of the first row of video in a frame is signaled by a transition from vertical blanking of field 0 to active video of field 0. Once this transition has been detected, **ntsc_decoder** begins to send data into the **fifo_controller** module. The decoder must maintain a count of the number of samples it has seen in order to determine whether the data from the ADV7185 is of the NTSC_Y, NTSC_CR, NTSC_CB, or NO_DATA type, as well as the row and column the current sample belongs to. The NO_DATA type accompanies samples which occur during blanking periods, while the NTSC_Y, NTSC_CR, and NTSC_CB types are used to indicate which of the three pixel values the ten bits supplied by the ADV7185 represent.

Pixels are decoded across rows first, each sample representing the next column on the row. Once the horizontal blanking period for a row ends, the row counter is incremented by two and the column counter is reset to zero, indicating that a new row of pixels is about to begin. Once the even field of a frame has finished and its vertical blanking period has concluded, the row counter is set to 1 and its column counter is again reset to 0, indicating that the first row of the odd field is about to be received. Once both the even and odd fields have been received, the entire interlaced frame is finished and the decoder resets itself to receive row 0 from the next frame's even field.

4.1.3 FIFO Controller

Any asynchronous FIFO is used to bridge the two clock domains between *llc* and the global *pixel_clock* used by the rest of the system; the decoder writes data into the FIFO and the linear interpolating filter reads them out. As ten-bit segments of video data are produced by the **ntsc_decoder**, their row, column, type, and 10-bit data are written into a 32-bit wide FIFO (the row and column each occupy ten bits, while two bits are required to indicate which of the four data types the data represents). The **fifo_controller** module provides write access to the video decoder on the negative edge of *llc*, and provides read access to the **ntsc_interpolator** module on positive edges of *pixel_clock*. Because the interpolator is capable of processing a new ten-bit data segment, the read enable of the FIFO is tied directly to the *~fifo_empty* signal. The write enable of the FIFO is tied to high, storing every value made available by the decoder, including the NO_DATA segments.

4.1.4 NTSC Interpolator

Because the NTSC video stream includes only half of the chrominance information for each pixel, the missing values are inferred by the **ntsc_interpolator** module which is responsible for producing thirty-bit YCrCb values for every pixel. When a Cr or Cb value is read from the FIFO, the interpolator registers its value for use in inferring the Cr or Cb value for the next sample. Each time an NTSC_Y data type is read from **fifo_controller**, a complete pixel is produced and the *interpolator_done* signal is set high for one cycle. The resulting pixel is a combination of the current Y value, the chrominance data which was received on the previous read from the FIFO, and an inferred chrominance value. To infer missing chrominance information, **ntsc_interpolator** uses a simple linear interpolator which averages the two most recently received chrominance values for pixels on the same row. If the most recently received chrominance value was Cr when an NTSC_Y sample is read from the FIFO, for example, the module returns a pixel whose value is {Y[i], Cr[i], (Cb[i-1]+Cb[i+1]) / 2}. Division by 2 is accomplished by shifting the sum Cb[i-1] + Cb[i+1] to the left once. The module also outputs the correct row and column for each pixel that is produced.

Because the decoder produces ten bits of video data on each cycle of the slower 27MHz clock, the interpolator often has to stall for one cycle if there is no new data in the FIFO. Furthermore, even if data

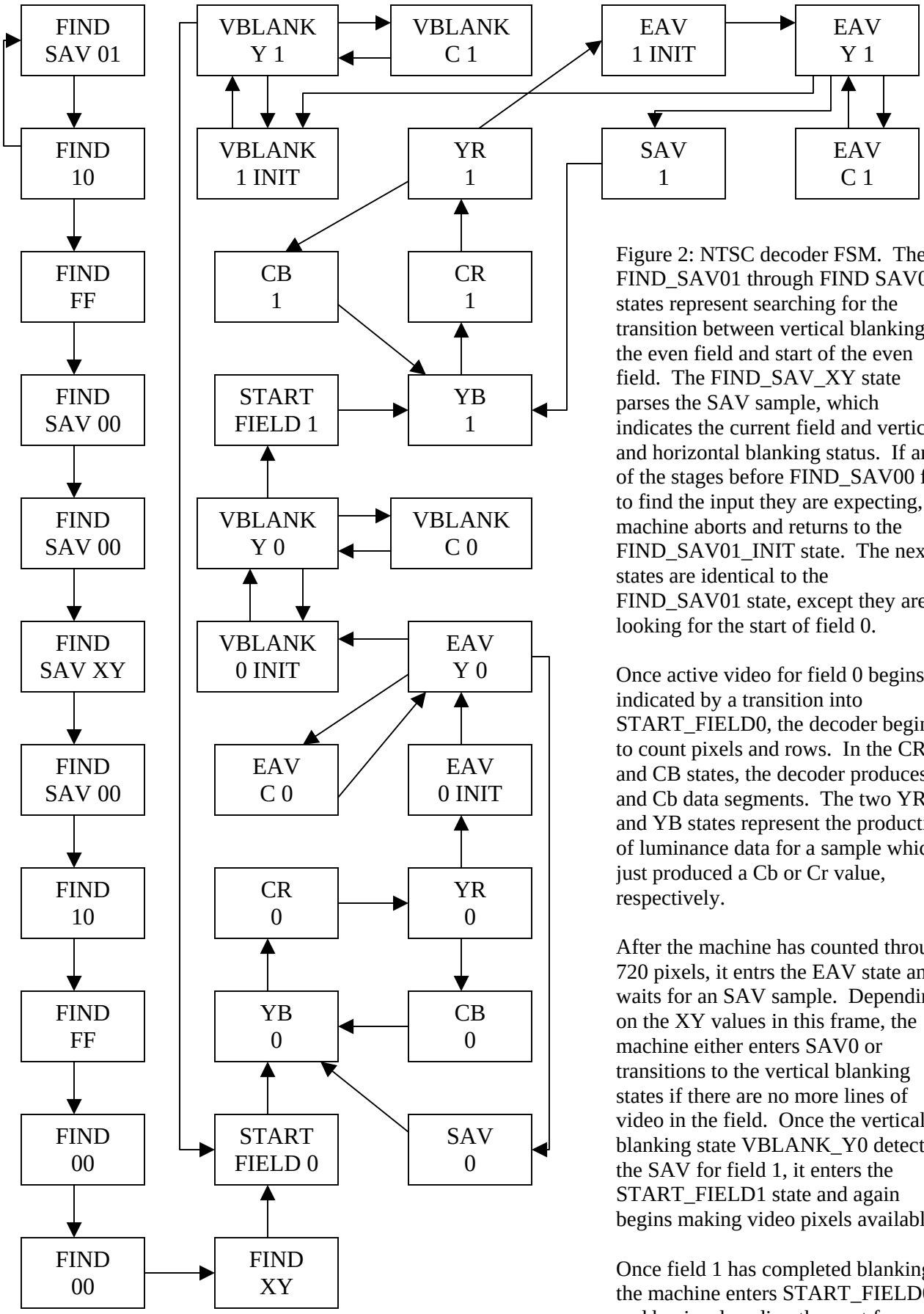

Figure 2: NTSC decoder FSM. The FIND_SAV01 through FIND SAV00 states represent searching for the transition between vertical blanking for the even field and start of the even field. The FIND_SAV_XY state parses the SAV sample, which indicates the current field and vertical and horizontal blanking status. If any of the stages before FIND_SAV00 fail to find the input they are expecting, the machine aborts and returns to the FIND_SAV01_INIT state. The next states are identical to the FIND_SAV01 state, except they are looking for the start of field 0.

Once active video for field 0 begins, as indicated by a transition into START_FIELD0, the decoder begins to count pixels and rows. In the CR and CB states, the decoder produces Cr and Cb data segments. The two YR and YB states represent the production of luminance data for a sample which just produced a Cb or Cr value, respectively.

After the machine has counted through 720 pixels, it entrs the EAV state and waits for an SAV sample. Depending on the XY values in this frame, the machine either enters SAV0 or transitions to the vertical blanking states if there are no more lines of video in the field. Once the vertical blanking state VBLANK_Y0 detects the SAV for field 1, it enters the START_FIELD1 state and again begins making video pixels available.

Once field 1 has completed blanking, the machine enters START_FIELD0 and begins decoding the next frame.

were being provided by the FIFO every cycle, complete pixel values could only be produced every other cycle since two data segments define each video sample.

4.1.5 YCrCb to RGB Converter

The complete thirty-bit YCrCb pixels produced by the interpolator are converted into 24-bit RGB pixels by the **ycrcb2rgb** module provided by Xilinx as part of xapp283. This three-stage pipelined module performs several multiplications and additions on the YCrCb output before producing valid RGB output that is used by the rest of the system. To account for the delays of this module, the row, column, and *interpolator_done* signals are delayed by three cycles.

4.1.6 Median Filter

To clean up salt and pepper noise that prevents the particle filter from accurately processing a frame, the **median_filter** module contains a 120-bit line buffer that stores the five most-recently received pixels from the **ycrcb2rgb** module in the order in which they were received. When a new RGB pixel arrives as indicated by the delayed *interpolator_done* signal, it is placed into the front of the line buffer while the rest of the pixels advance one position through the buffer. The last pixel in the buffer, which has the lowest column value, is pushed out of the buffer and discarded. A series of comparisons are performed on the five pixels currently in the line buffer in order to determine which of the five values is the median; each of the three RGB components of the pixels are considered separately to compute a median pixel value. To compute the median value of R, for example, two numbers are computed for each of the five R values: *number_leq* and *number_geq*. For a given R value, *number_leq* indicates the number of other pixels in the line buffer whose R value is less than or equal to its value, while *number_geq* indicates the number of pixels whose R value is greater than or equal. The median R will have a value of at least 2 for both of these numbers.

To produce a value for a pixel, the median color values of the pixel and its two nearest neighbors to the left and to the right must be considered. Therefore, this module adds an additional two cycle delay between receiving a pixel and producing its final RGB value in the best case. However, because a pixel can only be produced after another has arrived into the buffer, the delay due to **median_filter** is often four or more clock cycles, thanks to the asynchronous production of pixels by the decoder and the fact that video samples cannot be produced faster than 13.5MHz.

4.1.7 Video Buffering

The most recent frame of video received from the camera is stored in a large off-chip single-port ZBT 512kx36 SRAM, whose filtered RGB pixel values are read by both the particle filter and the VGA display. By clocking the ZBT's *ram_clock* at twice the rate of the rest of the system, the other modules can behave as if the ZBT were actually a dual-port RAM supporting two operations in a single *pixel_clock* cycle. To compensate for the propagation delay between the FPGA and the off-chip ZBT, and ensure that the setup and hold times for registers on the FPGA were not violated, the *ram_clock* sent to the ZBT is 180 degrees out of phase with the internal *ram_clock* used to generate and latch in the data sent to and received from the RAM. Each 24-bit pixel is allocated a single 19-bit address in the RAM, which is calculated by concatenating the lower nine bits of the pixel's row with the ten bits of its column. Due to the fact that the NTSC signal contains only 485 rows of active video, none of the video pixels are lost despite the fact that only nine bits are used for the address since up to 512 rows can be addressed in nine bits.

The ZBT is pipelined, resulting in a two *ram_clock* cycle delay between making a read request and receiving a response. Since the RAM uses the zero bus-turnaround protocol, write data is also provided two cycles after the address is set, allowing one request to be handled each clock cycle without any dead cycles when performing a write-after-read operation. The **zbt_controller** module handles the timing of sending the address, write enable, and data signals to the RAM. When a read or write is requested,

zbt_controller sends the address and write enable signals to the RAM immediately. If the request is a write operation, the data to be written is driven onto the ZBT's data bus two cycles later. If the request is a read operation, however, **zbt_controller** tristates the data bus two cycles later and reads the value being driven onto the bus by the RAM.

A higher level module **zbt_dual_port** multiplexes its input with **zbt_controller**, sending two requests to the controller during each of the system's clock cycles. The **zbt_dual_port** module has two identical sets of input ports, supporting two addresses, two write enables, and two data inputs to be handled during the cycle. The module also provides two data output buses corresponding to the two values returned by the controller. To prevent the RAM from writing to incorrect addresses due to glitches in the address or write enable signals, the address, write enable, and data inputs are registered during the first cycle of *ram_clock*. During the second cycle, the dual port ram sends *address1, we1,* and *data_in1* to **zbt_controller.** The values of *address2, we2,* and *data_in2* are sent to the controller during the third cycle. The value *data_out1* returned by the RAM containing the contents of *address1* is registered by **zbt_dual_port** during the fourth cycle, and *data_out2* is ready during the fifth cycle. Due to timing constraints, the value of *data_out2* cannot be registered, and therefore must be registered by the particle filter before it is used.

The **memory_controller** module interacts with **zbt_dual_port** to provide read access to the VGA module, read access to the particle filter, and write access to the median filter. Because a momentary lapse in displaying video on the screen is immediately apparent and unacceptable, one of the two ports on the dual-port ZBT is dedicated exclusively to read operations requested by the VGA module. Read operations by the particle filter and write operations by the median filter are multiplexed by **memory_controller** onto the second port of the ZBT. When the delayed *interpolator_done* signal is high, indicating that a pixel of video is ready, the memory controller sends the row, column, and 24-bit data of the pixel to the **zbt_dual_port** module and sets *we2* high. At all other times, *we2* is low and the row and column requested by the particle filter are sent to **zbt_dual_port**.

4.2 Particle Filtering - Lyric

The particle filter is managed by the **particle_fsm** module. This module in turn makes use of the **particle_is_alive** module, the **vga_display_rectangle** module, the linear feedback shift register modules (LFSR_23 and LFSR_25), the pseudo-random number generator modules (PRNG_23 and PRNG_25), a single port BRAM called **particle_memory** and a dual port BRAM called **particle_map_dual_port**. The **particle_memory** is 36 bits x 512 rows, while the **particle_map_dual_port** is 1 bit x 2^20 rows. The particle filter consists of 169 particles. The number is a perfect square to make initialization velocities uniformly distributed. The exactly value is arbitrary and can be changed. 169 was found to work, so it was used. To convert colors from RGB to the HSV color space, the **hsv2rgb** module was used. Figure 3 shows the block diagram for the particle filter portion of the system.

4.2.1 Particle FSM

This module consists of the FSM shown in Figure 4 that controls the particle filter. This FSM consists of five phases: Initialization, Particle Checking, Calculating Center of Mass (COM), Spawning, and Activating. Each phase and its associated states and functions are described below. After every field is decoded by the **ntsc_decoder**, the **particle_fsm** receives a *start* signal. It then has over 30,000 cycles to use the **memory_controller** to complete its function, but requires many fewer.

4.2.1.1 Initialization

When the FSM receives an *initialize* signal, it moves from IDLE to INITIALIZE_WAIT_FOR_START and sets the hasValidTarget signal high and registers the target location. The FSM then moves to INITIALIZE_POSITION after the next *start*. In this state, the initial positions of all 169 particles are written to a first 169 even addresses of **particle_memory** which stores all the particle's positions and

velocities in two lines each. These initial positions are all the same, and they are inputs to the module. The position is selected by the user using the mouse from section 4.4. Then the FSM moves to INITIALIZE_VELOCITY. Here all the particles are given velocities starting with (-12,-12) and stepping by two in both directions to every pair of possible velocities up to (12,12). These velocities are written in the first 169 odd addresses of **particle_memory** to correspond to the positions written there in the

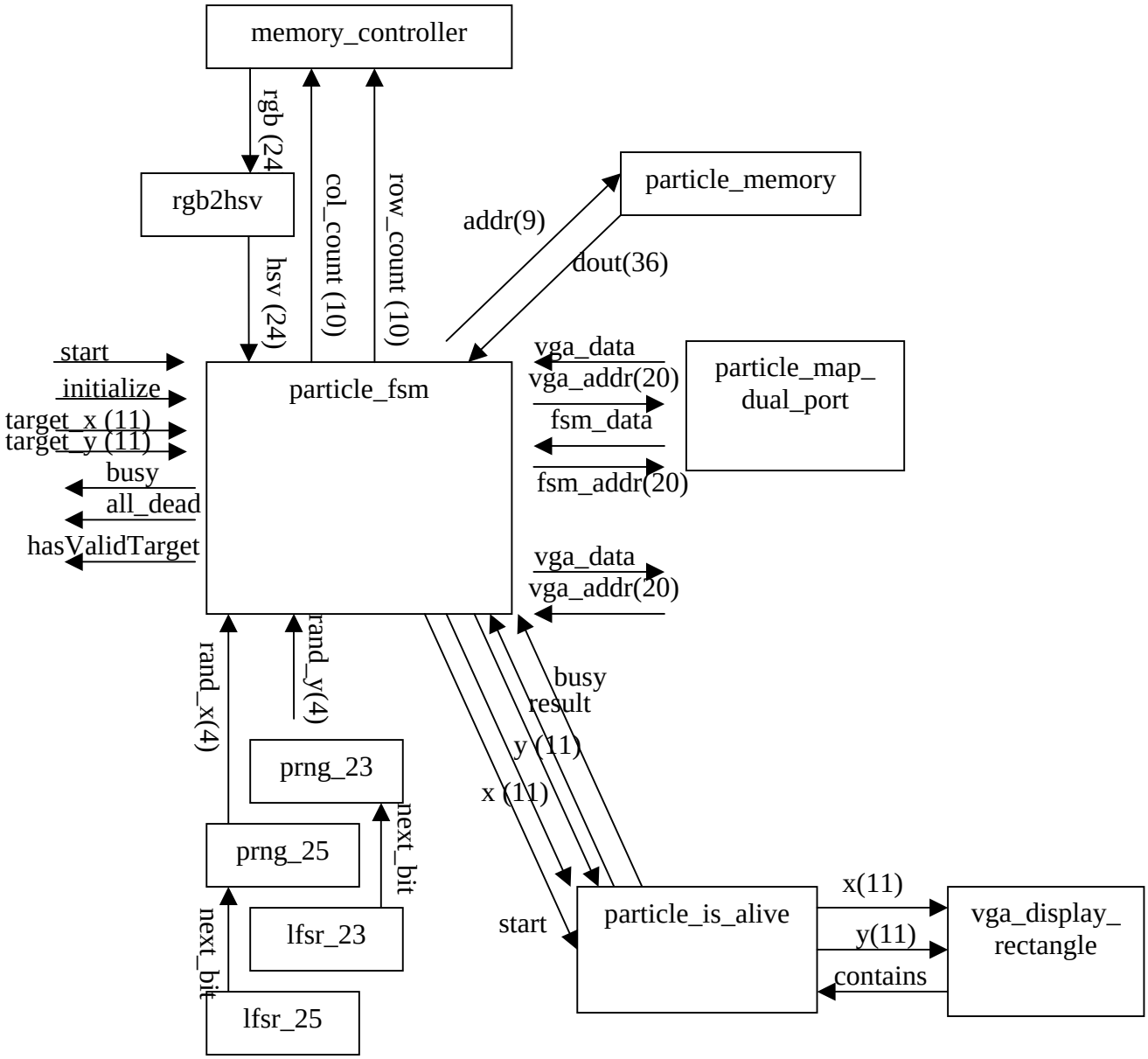

Figure 3: The block diagram for the particle filter. The row (10), column (10), and pixel_color (24) sent to particle_fsm to forward to the memory_controller are not drawn. All modules have a reset and 31.5 mhz clock input.

previous state. Next, the FSM moves through SEND_TARGET_COLOR_ADDRESS, WAIT_FOR_TARGET_COLOR and SAVE_TARGET_COLOR during which it sends the particles' initial position to the **memory_controller** and awaits and then registers the returned HSV 24-bit color of the target. Finally, the FSM goes to CLEAR_RAM in which clears every location in

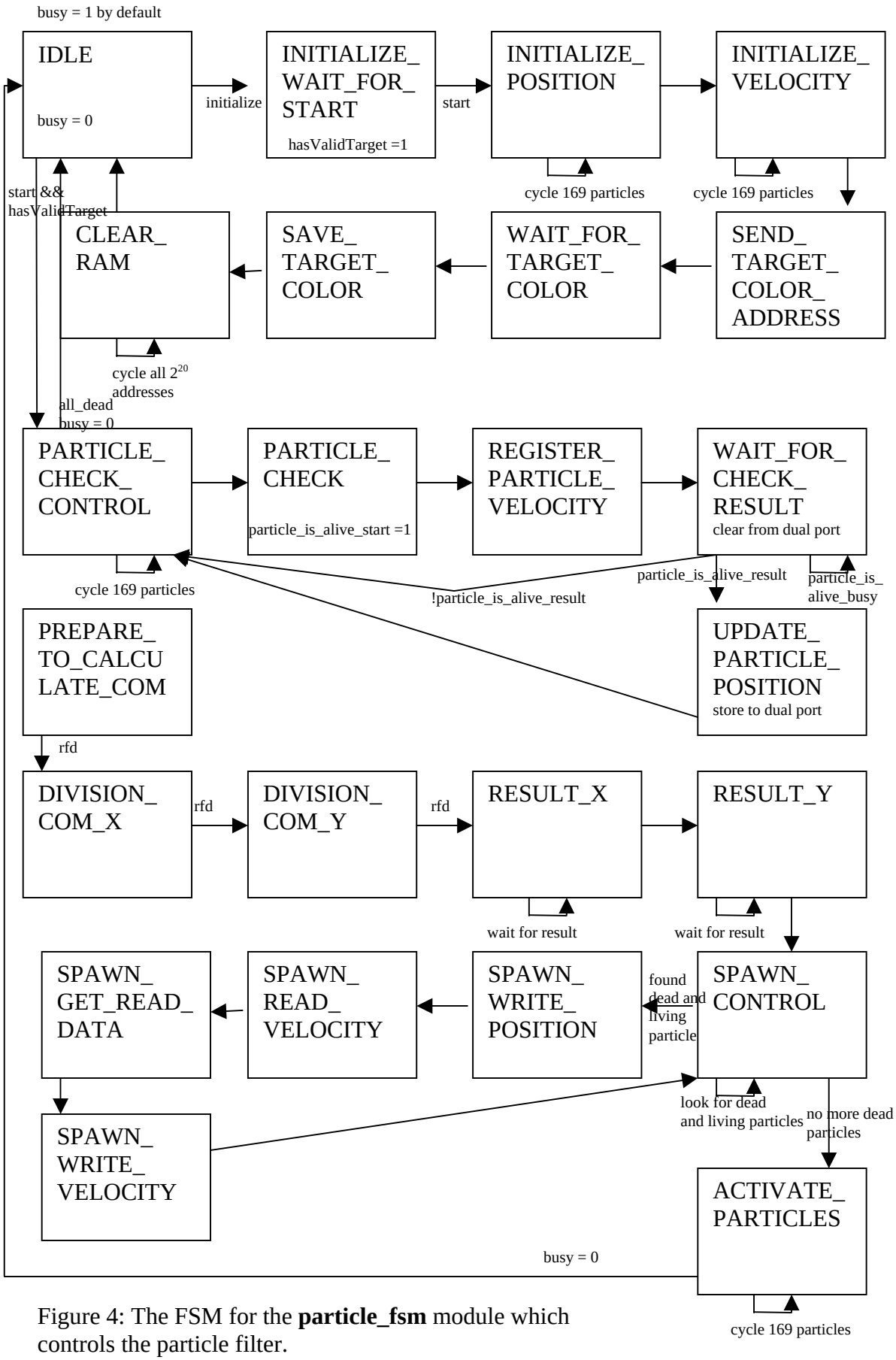

Figure 4: The FSM for the **particle_fsm** module which controls the particle filter.

particle_map_dual_port. The **particle_map_dual_port** stores a bit for every position on the VGA display. The bit is high if the position is held by a particle and low otherwise. The module makes displaying the particles simple and efficient. Upon initialization, the RAM is completely cleared. After that, only changes for moving, dying, and spawning particles are written to the RAM so that they can be done quickly. The RAM is dual port so that the VGA display can continually query the **particle_map_dual_port** module to check for the presence of a particle at a given location. Once the RAM is cleared, the FSM returns to IDLE to await the next *start*.

4.2.1.2 Particle Checking

If the *initialize* signal is low and the *hasValidTarget* signal is still high, the FSM moves from IDLE to PARTICLE_CHECK_CONTROL at the next *start*. This state manages cycling through all living particles to check if they are still alive. Living ones will be moved to their new position, while ones that just died will be removed. For each living particle with status ALIVE, the state first sends the address in the **particle_memory** corresponding to that particle to the **particle_memory** to discover its position and moves into PARTICLE_CHECK. From here, the minor-FSM **particle_is_alive** module is sent a *start* signal along with the position of the particle that was just read out from the **particle_memory**. At the same time, the velocity of the particle is requested from the **particle_memory**. After that velocity is registered in REGISTER_PARTICLE_VELOCITY for use after the particle is determined to be alive or not, the FSM waits in WAIT_FOR_CHECK_RESULT until the **particle_is_alive** *busy* signal goes down. In this state, the particle is also cleared from the **particle_map_dual_port** using its current location. If it survives, its new position will be written to **particle_map_dual_port** during UPDATE_PARTICLE_POSITION. If *result* is low, the particle's status is set as DEAD and the FSM moves back to PARTICLE_CHECK_CONTROL to continue to the next particle. If *result* is high, the particle's (x,y) coordinates are added to a summer over x positions and a summer over y positions respectively. These sums will be used to calculate the center of mass of the living particles later. The new positions of the particles, the (x,y) with the registered velocities added to them, are sent to particle_memory as the FSM leaves WAIT_FOR_CHECK_RESULT if *result* is high. The FSM moves to UPDATE_PARTICLE_POSITION, where the new position of the particle is written to the **particle_map_dual_port**. Finally, the FSM returns to PARTICLE_CHECK_CONTROL to continue to the next particle. If all 169 particles are found to be dead, the *all_dead* signal goes high, *hasValidTarget* is set low, and the FSM returns to IDLE. Otherwise, the FSM now moves to PREPARE_TO_CALCULATE_COM to begin the next phase.

4.2.1.3 Calculating Center of Mass

The center of mass is required for the Spawning phase, so it is calculated between Particle Check and Spawning. The phase starts at PREPARE_TO_CALCULATE_COM, which waits for the division units *rfd* ready signal. It then moves to DIVISION_COM_X and at the next *rfd*, sends the division unit the sum of all x coordinates as the dividend and number of particles currently alive (total particles - number dead) as the divisor. The FSM moves to DIVISION_COM_Y and does the same with the dividend as the sum of all y coordinates. Then the FSM waits in RESULT_X for the division to complete and registers the result. It moves to RESULT_Y and does the same for the y center of mass before moving on to SPAWN_CONTROL.

4.2.1.4 Spawning

In this phase, the FSM spawn new particles to replace those that died since the last field. The state maintains two pointers to the list of particle statuses. One will be used to find living particles, and one will be used to find dead ones. In SPAWN_CONTROL, the FSM increments both pointers until the live pointer points to a living particle and the dead pointer points to a dead one. A new particle will now be spawned to replace the dead one using the velocity information from the living particle pointed to by the live pointer. The FSM transitions to SPAWN_WRITE_POSITION and writes the position of the dead particle to the **particle_memory** as the current center of mass. During this state, the status of this newly

spawned particle is set to SPAWNING. The purpose of this is so that the newly spawned particle is not in turn used to spawn other particles during this iteration. Only currently living particles that have shown they are tracking the target by being alive should be used to generate new ones. The FSM then moves to SPAWN_READ_VELOCITY to read the velocity of the living particle out of the **particle_memory**. In SPAWN_GET_READ_DATA, the velocity data is delayed a single cycle. This is because the velocity data needs to be used to determine and write the new velocity for the spawning particle, but the BRAM does not support a write immediately after a read. Next clock cycle, the FSM moves to SPAWN_WRITE_VELOCITY and writes the new velocity of the spawning particle to the **particle_memory**. This is determined by adding random numbers ranging from -7 to 8 to the x and y components of the velocity of the living particle to get the velocity for the spawning particle. The random number generation is discussed in Sections 4.2.3 and 4.2.4. The FSM finally increments the live and dead pointers by one to make sure the next spawn check does not use the same particles and moves to SPAWN_CONTROL.

The FSM continues this process in SPAWN_CONTROL, looking for the next living and dead particles using the two pointers. When one pointer finds an appropriate particle, it stays with it while the other continues to increment. The living pointer may cycle back to the same living particles if the number of particles to be spawned exceeds the number currently alive. When the dead pointer reaches 169, however, all dead particles have been attended to and replaced with newly spawned ones. Now all that remains is to change the status of the new particles from SPAWNING to ALIVE. This is done in the final phase as the FSM moves to ACTIVATE_PARTICLES.

4.2.1.5 Activating

This is the shortest and simplest phase. The FSM sits in ACTIVATE_PARTICLES and cycles through all 169 particles and sets all these statuses to be ALIVE. Now the particle_fsm has completed its work for the current field of video and can return to IDLE to await the next *start*.

4.2.2 Checking Particles

This module **particle_is_alive** checks to see if a given particle is still alive. The FSM takes the current position of the particle of interest as well as the color the particles is tracking and a start signal as initial inputs. Once the FSM, shown in Figure 5, is given a particle position and a start signal, it spends a cycle in each of nine states, from IDLE to ADD8, sending a row and column identifying a different pixel in the 3 x 3 pixel particle to the ZBT RAM to learn the color of each of the pixels in the current camera image. Then the FSM waits 17 cycles in WAIT for the pixel color data to be read out from the ZBT RAM and converted from RGB to HSV for better analysis. Then as the data comes in over 9 consecutive cycles in the DATA state, the FSM compares each pixel's color against the color the particle is tracking. Small variations in hue and larger ones in saturation and value are permitted. Under the current settings, hue may vary by 10, saturation may vary by 31, and value may vary by 15. These may be tweaked as desired. Because the hue and saturation play much less of a role in affecting the color when value is very high or low, the variation in hue and saturation could be very large and the color should still be considered white or black. Thus, a few extra special thresholds are added to check for these cases. If the value of both the pixel and particle's assigned color is greater than 235 or less than 25, hue and saturation are ignored. If saturation is under 25, only value is considered as normal and hue is completely ignored. If a pixel is on the screen, as checked by the **vga_display_rectangle** module, and its color matches the particle's assigned color, then a counter is incremented. If the counter exceeds a preset threshold (currently 3), then enough of the particle's pixels match the assigned color for the particle to be considered to be still alive. In this case, a high signal for result is returned during the RESULTS state when the busy signal goes low. Otherwise, if the particle is deemed to be dead because too few pixels match the particle's assigned color, then the result remains low during the RESULTS state.

4.2.3 Linear Feedback Shift Registers

Linear feedback shift registers (**lfsr_23** and **lfsr_25**) are used to generate a pseudo-random sequence of bits. Each cycle, the shift register shifts all the bits in the bus to the right one register so a single bit is pushed out as the output of the module. A new bit is added on the left with a value determined by an xor on the bits contained in some specified subset of the registers. An n-bit shift register using a good set of registers for the xor will produce all 2^n - 1 n-bit strings on the bus (except for all 0's). The registers used

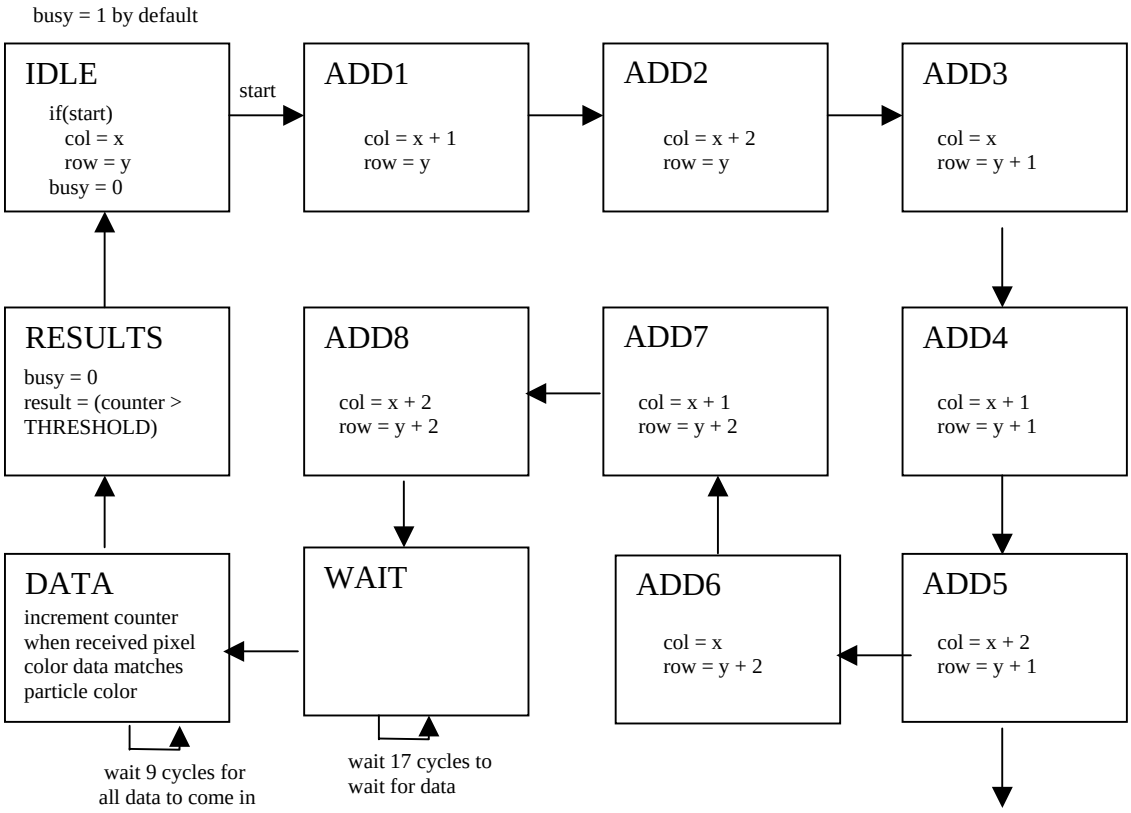

Figure 5: The FSM for the **particle_is_alive** module which checks all the pixels in a particle to see if they are the same color as the particle's assigned color.

for the xor can be determined using a polynomials from a look-up table of good polynomials. The 23-bit shift register uses the 5th and 23rd bits xor-ed together to get the replacement bit while the 25-bit shift register uses the 3rd and 25th bits.

4.2.4 Pseudo-Random Number Generation

These modules serve as pseudo-random number generators to produce the pseudo-random numbers (**prng_23** and **prng_25**) needed to alter velocities when spawning new particles. Each module consists of a simple four-state FSM that pulls four bits at a time from the LFSR. Specifically, the **prng_23** uses **lfsr_23** and the **prng_25** uses **lfsr_25**. Every four cycles, the module has a signed pseudo-random number between -7 and 8 ready. When the module receives the *next* signal, it begins the calculation of the next random number which is ready four cycles later and registered until the following *next* signal is received. Because the LFSRs are of relatively prime sizes, four bits are used at once, and the random numbers from the two modules are always used at the same time, the exact same pair of random numbers will not be repeated for the order of 2^25 * 2^23 * 4 cycles, which is more than enough for this application. Furthermore, the changing motion in the camera image of the targets being tracked also adds external random variation to the system since the random velocities are only generated to help track this

random/unknown motion.

4.2.5 VGA Display Checker

The **vga_display_rectangle** simply checks whether an (*x*,*y*) point is within the VGA display, i.e. if x is between 0 and 639 inclusive and y is between 0 and 479 inclusive. If so, *contains* is high, and low otherwise. The module is used when checking pixels on particles because if part of a particle is off of the VGA display, it should automatically be considered to be a non-matching pixel without needing to use the image data from the RAM. This also means the **pixel_is_alive** module does not depend on the ZBT RAM's behavior in this situation because the return data is ignored.

4.2.6 RGB to HSV Converter - Rob

A standard conversion algorithm is used to convert the 24-bit RGB pixels contained in the ZBT video buffer into 24-bit HSV pixels used by the particle filter. The **hsv2rgb** module computes the maximum and minimum of the three R, G, and B values for a pixel in order to assign each pixel a value V. Several 8-bit multipliers and two 16-bit, eighteen cycle pipelined dividers are used to compute the hue, H, and saturation, S, for each pixel. In total, the **rgb2hsv** module adds twenty cycles of delay into the system.

4.3 Servo - Lyric

There are two modules used to control the servo motor which rotates the camera: **servo_pwm** and **servo_feedback**.

4.3.1 Servo PWM

The angle the servo turns to is controlled by the width of a periodic pulse sent to the servo every 20 ms. The servo was calibrated to find the widths corresponding to 0 and 180 degrees. Every .5 milliseconds, the **servo_pwm** module checks to see if the pulse width needs to be increased or decreased based on input signals. A sustained signal to turn clockwise decreases the pulse width at the rate of 1 degree / .145 seconds. Likewise, a counterclockwise signal causes the pulse width to increase at the same rate. This rate was chosen because it empirically was slow enough for the camera to maintain a clear image. However, it can be easily adjusted. If clockwise and counterclockwise signals or no signals are given to the module, the module maintains the current pulse width. The pulse width is only adjusted if the system has a target it is trying to track.

4.3.2 Servo Feedback

The goal of the *1984* system is to keep the target horizontally centered in the camera image. The **servo_feedback** module takes the current center of mass of the particles as an input. If the center of mass is to the left of a defined center region of the screen (currently 270 pixels to 370 pixels), then the **servo_feedback** module is told to move the camera counterclockwise until the center of mass reaches the center region of the image. Likewise, if the center of mass is to the right of the center region, the **servo_feedback** is given a high clockwise signal. Both signals are low when the center of mass is centered.

4.4 PS2 Mouse

The **ps2_module** was taken in its entirety from the 6.111 Fall 2005 website. It was used to allow the user to select targets much more easily because the laser pointer proved to be impossible to track due to size and color variation. The module takes information from a PS/2 mouse as input and outputs the (*x*,*y*) position of the mouse as well as signals corresponding to mouse clicks. Whenever the user left-clicks, the position of the mouse is sent to the **particle_fsm** module as the target location input.

4.5 VGA - Rob

The VGA interface takes advantage of the **vga** module written previously, which generates the *line_count* and *pixel_count* signals that indicate which pixel is currently being drawn to the screen. The module also generates the horizontal sync, vertical sync, and screen blank signals consistent with the delays expected

by the pipelined ADV7125 used to drive the monitor.
In addition to showing the buffered video frame held in the ZBT SRAM, the VGA module is responsible for displaying a square representing the location of the mouse pointer, a box around the target currently being tracked, and the location of each of the 169 particles. A cyan square representing the mouse cursor location is displayed by the **mouse_location** module, which compares the values of *line_count* and *pixel_count* with the current x and y coordinates of the mouse, *mouse_x* and *mouse_y*. The cursor has a pre-defined size WIDTH. If the vga's current pixel is within WIDTH units along the x or y axis from *(mouse_x, mouse_y)*, the pixel provided by the video frame is discarded and replaced with a cyan pixel instead. A frame surrounding the current target is drawn similarly by the **crosshair** module; the frame has a defined width and thickness, and a video pixel is replaced with a blue one whenever it lies within the bounds of the frame. The frame is centered around the center of mass of the particles.

The **particle_map_dual_port** module contains a single bit for each of the display pixels. If a particle is currently present at a given location (*particle_x, particle_y*), the location in the RAM corresponding to (*particle_x, particle_y)* contains a boolean 1. Otherwise, the location's value is set to 0. When displaying a pixel, the vga interface queries **particle_map_dual_port** with its (*pixel_count, line_count*). If the location contains a 1, the video pixel is replaced with a green pixel and a particle is displayed on the screen instead. Querying this BRAM incurs an additional delay; thus, the vga signals are delayed an additional cycle.

5 Testing and Debugging - Rob and Lyric
All significant modules were first tested using Modelsim Behavior Simulation. This step alone allowed for the removal of many bugs. Some test benches were also constructed to test several modules working together. These tests were particularly useful in checking basic timing and ensuring the right signals went high during the correct states. They also helped make sure the FSMs traversed their states correctly and that calculations were being performed correctly. Further testing continued on the logic analyzer. The logic analyzer was particularly useful in debugging the ZBT and particle filter. Outputting RAM addresses, RAM data, FSM states, done signals and a few other signals to the analyzer uncovered bugs missed in Modelsim and helped identify their source.

In general, an incremental approach toward implementing and integrating modules allowed the system to be assembled and verified in a rational manner, and helped identify which modules were causing problems. For example, a very noticeable flickering in the VGA screen was debugged by stepping through the partially assembled system, adding modules individually until the culprit--the hex display module on the front of the labkit, was identified and removed.

In order to test the video decoder, a large BRAM capable of storing a 256x256 black and white image was implemented before the dual port ZBT RAM was added. Since luminance in the YCrCb color space is the equivalent of a black and white image, only 8 bits per location had to be stored. Since the BRAM experiences a dead cycle during a write-after-read operation, the BRAM module was clocked at 94.5MHz to allow the module to simulate a dual-port RAM. While the temporary BRAM buffer allowed the video decoder's operation to be verified, its large size and fast clock resulted in routing and propagation delay issues which were resolved by switching to the ZBT.

Once the particle filter components were successfully tested on Modelsim, the integrated particle filter was tested using a simple image written to a small BRAM. The image consisted of a green square on a black background that could be moved using switches on the FPGA. Another switch was used to signal that another iteration of the particle filter should be calculated. Using the two switches, the particle filter could be tested to make sure it could track the green square as it started and stopped moving. The test combined with data from the logic analyzer also helped identify and eliminate a few timing and logic bugs in the particle filter. For example, the test made it possible to check whether the right pixels were

being analyzed for each particle and ensure the correct decisions were being made based on the pixel colors. The test also uncovered a bug in the spawn code that cause an infinite loop when all the particles died.

After all the components were completed, another similar tested was performed with the entire screen split into two halves of different colors. The colors were written from the video decoder module instead of the correct video data to ensure the full system was being tested. This test also uncovered a few small timing bugs. Once those were fixed, the test showed a great example of the correctness of the particle filter on the fake video data and showed all the system pieces worked together properly after integration.

6 Conclusion - Rob

The *1984* system successfully tracked objects as they moved through a scene. While its performance on dark colors was poor, the system was surprisingly successful tracking skin and faces. By filtering noise out of the image and rotating the camera to keep its target in view, the system can keep track of mobile objects for around thirty seconds before losing its target. Video quality issues associated with interlaced video could be addressed by adopting a double-buffering scheme in which one RAM bank is reserved for buffering the current frame of video while the other provides the VGA module and particle filter with video pixels from the previously buffered and de-interlaced frame. Furthermore, adding vertical and temporal coherency into the **ntsc_interpolator** module would increase the quality of the resulting image. A more sophisticated particle filter that weights each particle based on how well it is currently tracking its target would improve the system as well. Combined with more advanced image filtering techniques, a cleaner and higher-resolution video source would enable the particle filter to track the motion of smaller, faster moving objects, including the laser pointer which proved to be impossible to follow reliably.

7 Acknowledgements - Rob and Lyric

The authors would like to thank the 6.111 staff for their help throughout the course of the project. Everyone was very willing to help with any issues encountered with the design, Xilinx, or the FPGA. In particular, we'd like to thank Javier Castro for his guidance and continued attention as well as empathy when Xilinx's interface and behavior left everyone frustrated and confused. Additionally, the assistance of Nathan Ickes in explaining the i2c protocol and the finer points of using DCMs and timing constraints proved invaluable. Furthermore, Rob would like to add that he would love to see more infinite 'Find' boxes in the future.