Hardware Accelerated AHRS
CSEE-4840 Spring 2026

Final Report

Anubhav Vandkar (av3336) | Jaxson Natalini Robinson (jnr2154) |
Zongyang Li (zl3621) | Darshan Ramakrishnaiah (dr3412)

Contents:

Abstract 2

Block Diagrams 2

a. **System Block Diagram**
b. **I2C Driver**
c. **Kalman FIlter**
d. **VGA Software**

Implementation 4

Resource Budgets 12

1. Abstract

This project presents the design and implementation of an Attitude and Heading Reference System (AHRS) on the Terasic DE1-SoC platform. An MPU-6050 six-axis inertial measurement unit is interfaced to the Hard Processor System (HPS) over I2C, where a userspace driver reads raw accelerometer and gyroscope data at 1 kHz. The HPS performs static bias calibration, computes roll and pitch angles using three-axis arctangent formulas, and encodes all sensor data in a Q3.9 fixed-point format for efficient hardware transfer. The processed data is passed to a custom Kalman filter accelerator implemented in the FPGA fabric of the DE1-SoC via the Avalon-MM lightweight bridge, which fuses the accelerometer angle estimates with gyroscope rate measurements to produce stable, drift-corrected attitude estimates. The filtered roll and pitch outputs are read back by the HPS and rendered in real time on a VGA display as a primary flight display indicator, with a movable horizon line and attitude circle reflecting the physical orientation of the board. The system demonstrates effective hardware-software co-design, combining the flexibility of Linux userspace drivers with the computational efficiency of FPGA-accelerated sensor fusion.

A Kalman filter is an optimal estimation algorithm that predicts the state of a system by combining any noisy sensor measurements with a mathematical model of the system. It works in a continuous cycle of predicting the next state and then correcting that prediction using real-time data to minimize uncertainty. Essentially, it filters out "noise" to provide a much more accurate estimate of our sensor values of roll, pitch and yaw.

2. Block Diagrams

a. System Block diagram

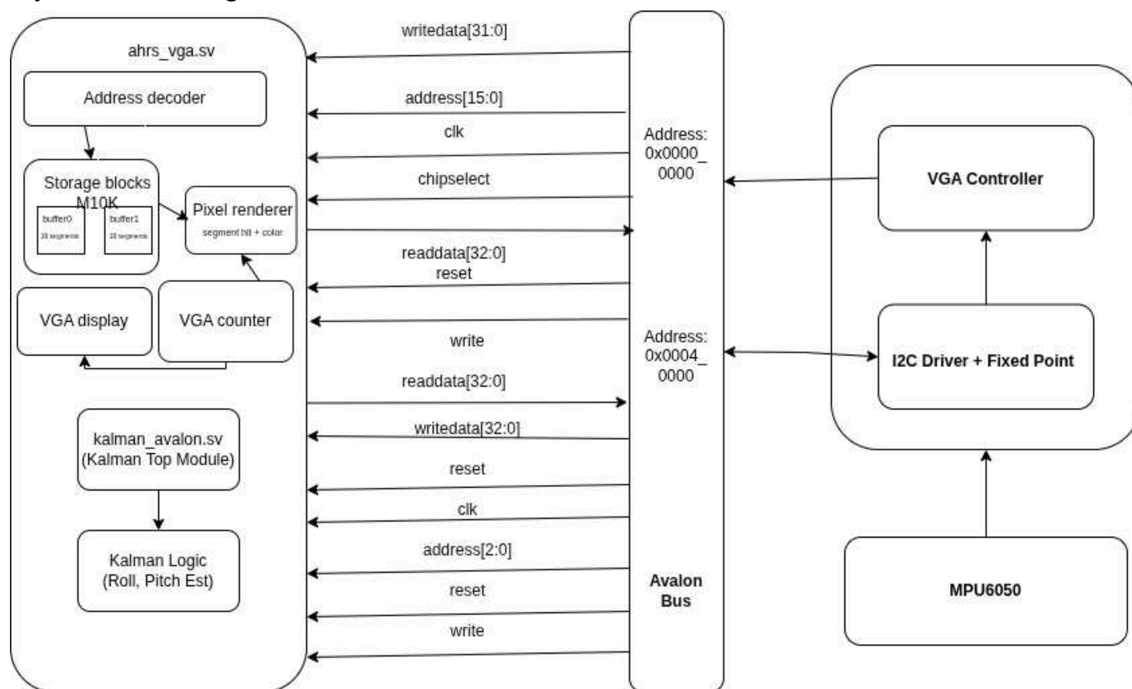

Fig1: System level block diagram

b. Sensor and I2C Driver block diagram

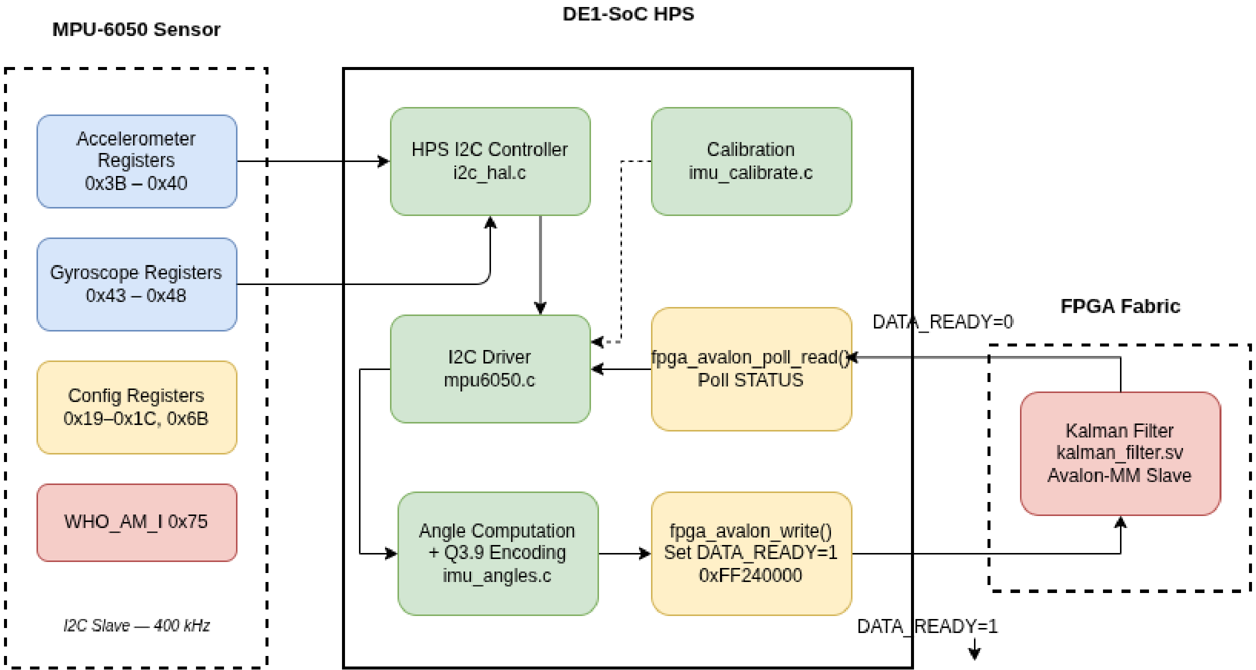

Fig2: Sensor interface

c. Kalman Filter block diagram

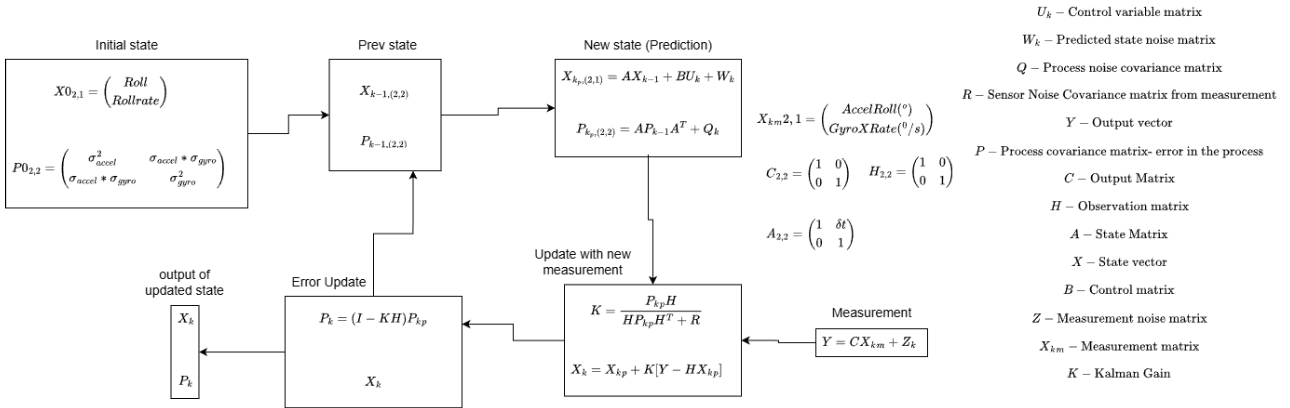

Fig3: Kalman filter estimation algorithm

d. VGA Software Block Diagram

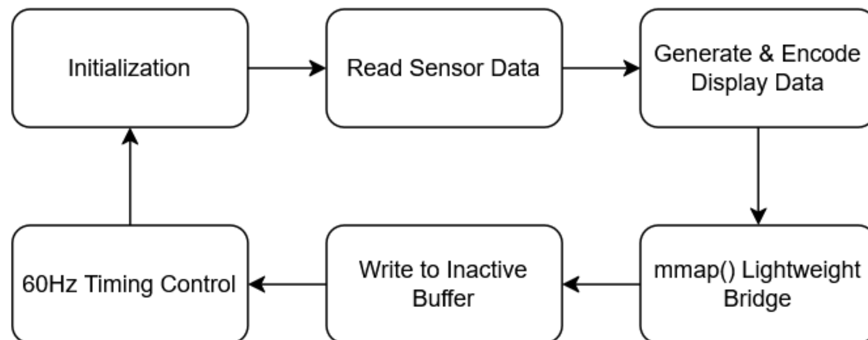

Fig4: VGA Software

3. Implementation

a. I2C Driver

The MPU-6050 is interfaced over I2C using the Linux i2c-dev kernel module, accessed through /dev/i2c-1. No custom kernel driver is written, just the standard userspace ioctl interface is sufficient since the pipeline runs at 1 kHz with no hard real-time interrupt requirements. The hardware abstraction is split across two files: i2c_hal.c handles only the raw bus mechanics (open, write, burst read), and mpu6050.c contains all sensor-specific logic.
Register writes are a simple two-byte transaction. Burst reads use the I2C_RDWR ioctl with two chained message structs, which issues a repeated START between the write and read phases without a STOP in between. This is required by the MPU-6050 to correctly latch its internal register pointer before the read begins.
On startup, mpu6050_init() reads the WHO_AM_I register (0x75), which always returns 0x68 on an MPU-6050. This confirms the device is present and the I2C bus is functional before any configuration is written. The sensor is then woken from sleep and configured: sample rate 1 kHz (SMPLRT_DIV = 0x00), digital low-pass filter at 188 Hz bandwidth (CONFIG = 0x01), gyro full-scale range +-250 deg/s (131 LSB/deg/s), and accel full-scale range +-2g (16384 LSB/g).
Each call to mpu6050_read_frame() polls the register and checks the DATA_RDY_INT bit before reading. When data is ready, a 14-byte burst read starting at ACCEL_XOUT_H (0x3B) captures all six axes in one I2C transaction. The temperature bytes (indices 6-7) are discarded. Each high/low byte pair is explicitly cast to int16_t

```C/C++
frame->ax = (int16_t)((buf[0] << 8) | buf[1]);
```

```
frame->gx = (int16_t)((buf[8] << 8) | buf[9])
```

The result populates imu_raw_frame_t:

```c
typedef struct {
	uint32_t timestamp_us;
	int16_t ax, ay, az;
	int16_t gx, gy, gz;
} imu_raw_frame_t;
```

Before the main loop begins, mpu6050_calibrate() collects 1000 still samples and averages them into imu_bias_t offsets. Accumulators are int32_t to prevent overflow (1000 x 32768 = ~32M). The az bias subtracts the expected gravity component (16384 LSB at +-2g FSR) so that az reads near zero when flat, giving the angle computation a clean reference. mpu6050_apply_bias() subtracts these offsets from every raw frame before any further processing. Calibrated accelerometer readings are converted to roll and pitch in imu_angles.c using atan2f(): Gyroscope readings are converted from raw LSB to radians per second, as required by the FPGA Kalman filter:

```c
float roll_rad = atan2f(ay, az);
float pitch_rad = atan2f(ax, sqrtf(ay*ay + az*az));

float gx = (raw->gx / GYRO_SENSITIVITY) * (M_PI / 180.0f);
float gy = (raw->gy / GYRO_SENSITIVITY) * (M_PI / 180.0f);
```

All four values are then encoded as Q3.9 fixed-point. This format uses 9 fractional bits (scale factor 512), giving a range of +-3.998 and a resolution of ~0.00195. It is chosen because roll and pitch from atan2f are bounded to +-pi (~+-3.14 rad), which fits within Q3.9 with headroom. Gyro values in normal use (60-70 deg/s ~ 1.05-1.22 rad/s) are also well within range. The encoding clips to the 12-bit signed range and masks the upper nibble to zero. The results are packed into imu_angle_frame_t:

```c
static int16_t float_to_q39(float val) {
	float scaled = val * 512.0f;
	if (scaled > 2047.0f)
		scaled = 2047.0f;
```

```
	if (scaled < -2048.0f)
		scaled = -2048.0f;
	return (int16_t)((int16_t)scaled & 0x0FFF);
}

typedef struct {
	int16_t roll;
	int16_t pitch;
	int16_t gx;
	int16_t gy;
	uint8_t data_ready;
} imu_angle_frame_t;
```

The HPS transfers data to the Kalman filter accelerator over the HPS-to-FPGA Lightweight Avalon-MM bridge. The Kalman filter SystemVerilog module is itself an Avalon-MM slave in Platform Designer. The HPS writes directly to the slave's internal registers. The bridge is accessed from userspace by opening /dev/mem and calling mmap() with MAP_SHARED and the page-aligned bridge base (0xFF240000). The Kalman slave sits at a Qsys-assigned offset of 0x40 from the bridge base. All accesses use volatile uint32_t* pointers to prevent the compiler from caching or reordering the writes. The register map is:

```C/C++
REG_ROLL 0x40 /* HPS writes Q3.9 roll */
REG_PITCH 0x44 /* HPS writes Q3.9 pitch */
REG_GX 0x48 /* HPS writes Q3.9 gx (rad/s) */
REG_GY 0x4C /* HPS writes Q3.9 gy (rad/s) */
REG_DATA_READY 0x50 /* HPS sets 1 to trigger Kalman */
REG_DATA_STATUS 0x54 /* Kalman sets 1 when done */
REG_RESULT_ROLL 0x58 /* Kalman-filtered roll */
REG_RESULT_PITCH 0x5C /* Kalman-filtered pitch */
```

The handshake works as follows. The HPS writes all four input registers then sets REG_DATA_READY = 1. The Kalman filter detects this, runs the filter, writes results to REG_RESULT_ROLL and REG_RESULT_PITCH, and sets REG_DATA_STATUS = 1. The HPS polls REG_DATA_STATUS with a 10 us sleep between reads (avoiding busy-waiting) and a 50 ms timeout. Once the flag is set, results are read into kalman_result_t and DATA_READY is cleared to 0. The filtered Q3.9 roll and pitch values are sign-extended from 12 bits before being passed to the display layer.

```C/C++
typedef struct {
	uint32_t kalman_roll;
	uint32_t kalman_pitch;
} kalman_result_t;
```

b. Kalman Filter

The ASM for the controller will be developed during the design phase, Roll and pitch estimation will run in parallel, each estimation will have **11 16-bit matrix multipliers** and **3 16-bit matrix adders,** each matrix multipliers will have 8 16-bit multipliers and 4 16-bit adders each matrix adder will have 4 16 bit adders, so in total **176 16-bit multipliers, 112 16-bit adders**

Fig4: Kalman filter hardware top level block diagram

Fig5: Roll estimation Datapath

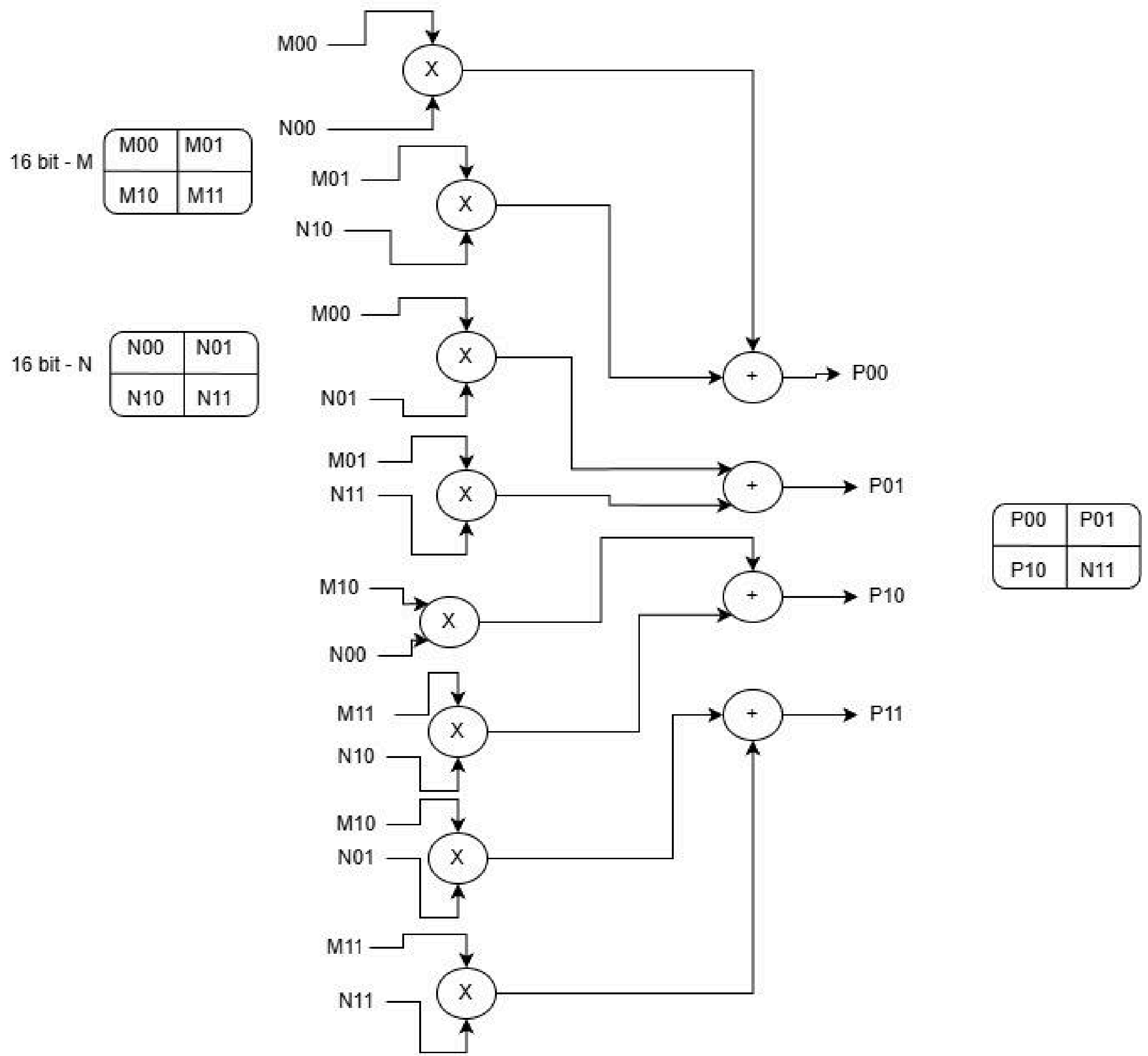

Fig6: Matrix multiplication datapath

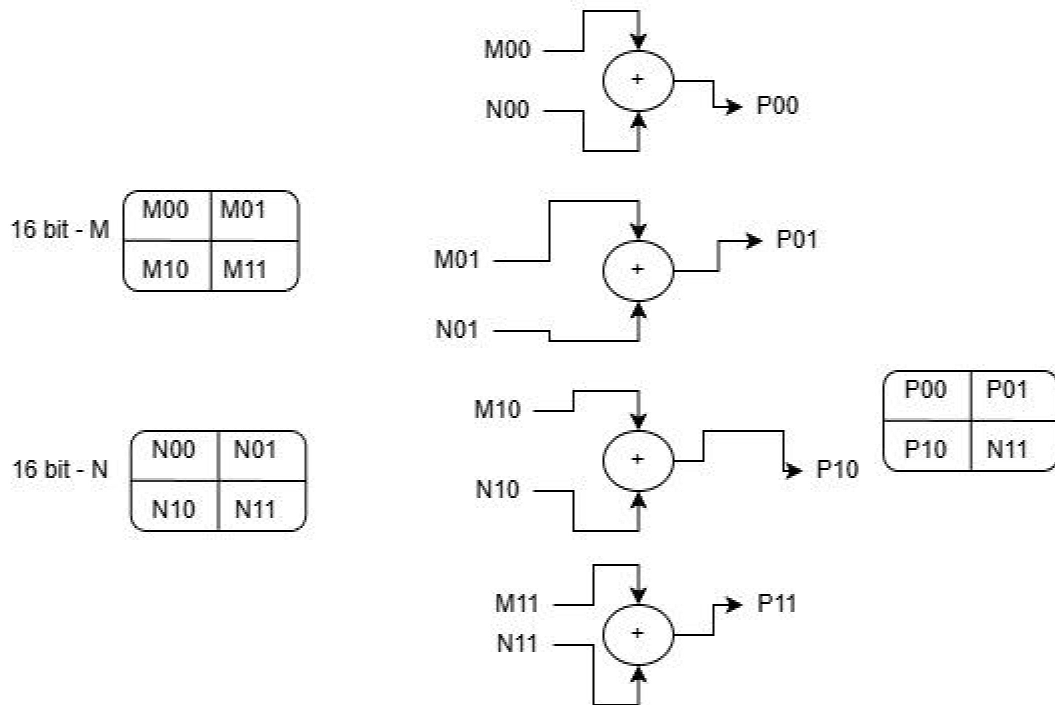

Fig7: Matrix multiplication datapath

c. VGA Display Controller

The VGA display controller is the FPGA block used to show the 640×480 artificial horizon display. It is connected to the Avalon bus, so the HPS can write display data into it. The FPGA then reads the data from on-chip memory and generates the VGA pixels directly. In this design, we did not use an external frame buffer because it would use too much memory.

The main problem we needed to solve was memory usage. A normal VGA frame buffer stores one color value for every pixel. For a 640×480 screen using RGB888, this would require about 921 kB of memory, which is larger than the available on-chip memory. Even using a smaller color format would still take too much space and leave less memory for other parts of the project, such as the Kalman filter.

Instead of storing every pixel, I store each row as several horizontal line segments. Each row can have up to 16 segments, and each segment is stored as one 32-bit word:

word[11:0] = xstart
word[23:12] = xstop
word[31:24] = color

Here, xstart and xstop define the horizontal range of the segment, and color uses RGB332 format. When the VGA controller draws a pixel, it checks whether the current x position is inside one of the segments. If it is, the controller outputs that segment's color. If xstop <= xstart, that segment is treated as empty, so software can easily disable a segment.

This method is enough for our AHRS display because we mainly need to draw the horizon line, sky/ground areas, and some simple markers. It reduces the required display memory from about 921 kB to around 64 kB.

For the memory structure, we used two complete picture buffers for double buffering. Each buffer is divided into 16 small RAM blocks, one for each segment slot. Each RAM is 512 entries deep and 32 bits wide, and I used the Quartus ramstyle = "M10K" attribute to make sure they are implemented using M10K block RAM.

The reason to split the 16 segments into separate RAMs is that the renderer needs to check all 16 segments for the current row at the same time. If all segments were packed into one large word, the design would be harder and less efficient. By using 16 separate RAMs, the hardware can read all segment slots in parallel and compare them with the current pixel position.

One issue we had during implementation was that Quartus did not infer M10K memory correctly at first. My first version put RAM reads, writes, control registers, and reset logic in the same always_ff block. Because M10K memory does not support asynchronous reset in that way, Quartus implemented the arrays using flip-flops instead. This made the design too large and it failed to fit on the FPGA.

To fix this, I separated the RAM logic from the control register logic. The RAM block uses a purely synchronous always_ff block without reset, while the control registers are handled separately. After this change, Quartus correctly inferred the memory as M10K blocks, and the design fit much better.

The VGA timing is generated by a small counter module. It uses the 50 MHz clock and generates a 25 MHz VGA pixel clock from hcount[0]. The display uses standard 640×480 timing. Since M10K memory has synchronous reads, the segment data comes out one cycle after the row address is given. To handle this delay, I also delay the pixel x position, y position, blanking signal, and selected buffer by one cycle. This makes sure the pixel coordinate and RAM output match correctly.

For each pixel, the renderer checks the 16 segment slots. Segment 0 has the highest priority, then segment 1, and so on. If a segment matches the current x position, its RGB332 color is expanded to RGB888 and sent to the VGA output. If no segment matches, the controller shows a default background pattern, so the screen still displays something even before software writes any data.

The design also uses double buffering to avoid tearing. Software writes the next frame into the buffer that is not currently displayed. Then it writes to the control register to request a buffer swap. The hardware does not switch buffers immediately. Instead, it waits until the start of the next frame, when hcount = 0 and vcount = 0. This makes the buffer swap happen during vertical blanking, so the user does not see half of the old frame and half of the new frame.

In the final version, the resource usage was much better than the first version. The first flip-flop-based version used too many LABs and could not fit. After the M10K fix, the design used only 568 ALMs, 964 registers, and 64 M10K blocks. The block memory usage was 524,288 bits, which is about 13% of the device memory.

Overall, this VGA controller works well for the project. It avoids the large memory cost of a full frame buffer, supports clean updates using double buffering, and can still draw the AHRS display at 60 Hz

without visible tearing. The software test can update the full segment table every frame, which is enough for our display requirement.

d. Software *(AHRS display)*

While the Kalman filter produces the roll and pitch values all by itself, the hardware can't really do anything meaningful with that data on its own. If you were to leave it all up to the vga controller modules, everything would start to fall apart once you attempted to do floating-point division to find the slope of the horizon line. Even if you managed to do that, how would you encode that line in such a way that the vga controller knows what pixels to light up to display it in the center of the screen? Clearly, this is all above the capabilities of the hardware alone. Thus, software is an excellent fit for this implementation.

There are many features that can be displayed on an AHRS. Since our sensor can only detect pitch and roll, the scope of features that we can include are mostly limited to those two values. The first major feature is a horizon line that rises with pitch and turns with roll. By converting the roll angle to a clean slope using trigonometry, we can essentially encode the line as follows: for each scanline y, mark the pixels that our line formula would generally intersect *(given some thickness).* Now, we essentially have a list of start and stop positions for each scanline that tell the vga controller where to write each part of the line on the screen. Fortunately, this clever encoding is very scalable. For example, what if you want the space above the horizon to be blue and the space below to be green? You could encode two more segments per scanline for sky and ground. To differentiate them, you could add 8-bit color data to each segment's word. Now, we have our official 32b segment-word structure: [12b start | 12b stop | 8b color]. Since adding a new segment to each scanline only adds 1.92KB of memory overhead per buffer, we can afford to add quite a few more. Ultimately, we found that having 16 segments per scanline was a good tradeoff between low memory usage and output capabilities.

Retrieving raw roll and pitch values is simple enough: the I2C driver calls **ahrs_display_render(uint16_t roll_raw, uint16_t pitch_raw)** and kindly provides the program with up-to-date values constantly. Sending values down to the vga controller, however, is a bit more involved. Since the VGA controller is implemented as an Avalon-MM peripheral, the HPS accesses it through the FPGA lightweight bridge using memory-mapped I/O. The software opens **/dev/mem** and calls **mmap()** to map the VGA controller's physical address space into userspace. Once mapped, the controller appears as a normal array of **volatile uint32_t** words that software can write to directly. After the frame is constructed, the software determines which VGA buffer is currently inactive. It writes the entire segment table into that back buffer row-by-row and segment-by-segment using normal memory writes over the Avalon bridge. Once the transfer is complete, the software writes to the VGA control register to request a buffer swap. The hardware does not immediately switch buffers. Instead, it waits until the next frame boundary during vertical blanking before swapping the displayed buffer. This double-buffered approach prevents visible tearing artifacts and allows the software to safely generate the next frame while the current one is still being displayed.

4. **Resource Budgets**

Kalman:

4939 ALMs used
3884 Registers
64 M10K blocks
87 DSP blocks

VGA:

524k memory bits
964 registers
Memory usage of display
2 buffers * 480 rows * 16 words / row * 32 bits / words = 61.44KB

```
+-------------------------------------------------------------------------------+
; Fitter Summary                                                                ;
+---------------------------------+---------------------------------------------+
; Fitter Status                   ; Successful - Tue May 12 19:59:52 2026       ;
; Quartus Prime Version           ; 21.1.0 Build 842 10/21/2021 SJ Lite Edition ;
; Revision Name                   ; soc_system                                  ;
; Top-level Entity Name           ; soc_system_top                              ;
; Family                          ; Cyclone V                                   ;
; Device                          ; 5CSEMA5F31C6                                ;
; Timing Models                   ; Final                                       ;
; Logic utilization (in ALMs)     ; 4,939 / 32,070 ( 15 % )                     ;
; Total registers                 ; 3884                                        ;
; Total pins                      ; 362 / 457 ( 79 % )                          ;
; Total virtual pins              ; 0                                           ;
; Total block memory bits         ; 524,288 / 4,065,280 ( 13 % )                ;
; Total RAM Blocks                ; 64 / 397 ( 16 % )                           ;
; Total DSP Blocks                ; 87 / 87 ( 100 % )                           ;
; Total HSSI RX PCSs              ; 0                                           ;
; Total HSSI PMA RX Deserializers ; 0                                           ;
; Total HSSI TX PCSs              ; 0                                           ;
; Total HSSI PMA TX Serializers   ; 0                                           ;
; Total PLLs                      ; 0 / 6 ( 0 % )                               ;
; Total DLLs                      ; 1 / 4 ( 25 % )                              ;
+---------------------------------+---------------------------------------------+
```