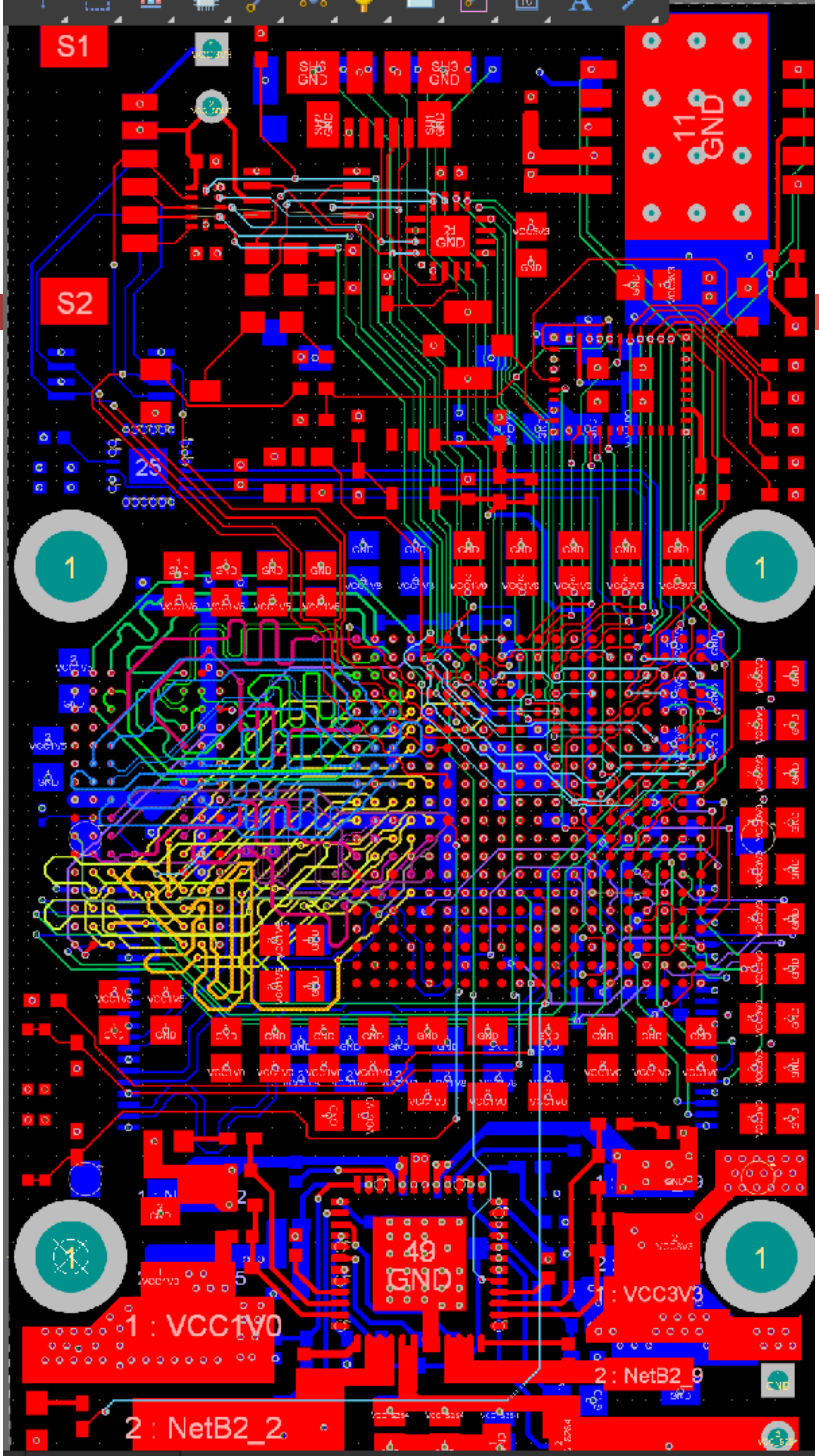
Kestrel Project: FPGA Design

So Why This Lecture?

- We have a slot for a "real world" design example...
  - So lets go with a design that I am actually developing...
- This is also a forcing function
  - Before I start coding in earnest I want to know what I'll be building: ***Design first***, then code.
- How to test this stuff for the exam...
  - Idea: Look at the skills I've used from this class.

Real World Design Exercise: Kestrel Functionality

- Objective: ***low(ish) cost & compact vision-based*** autonomy for small UAVs
  - Zynq XCZ7010 or 7020
  - Single 1/2 GB, 16b, 800 MHz data-rate DDR3 DRAM
  - Two Raspberry Pi-0 interfaces to OmniVision OV5647 cameras:
    - Two data lane MIPI CSI-2 serial data transfer,
    - 2592x1944x15 frames per second (SXGA), 1920x1080x30 frames per second (1080p)
    - Wide variety of lens options including wide angle and "no IR cut" (aka, 'see into the near IR on the red channel')
  - SD card, WiFi, Bluetooth
  - GPS + two accelerometer/gyros (one with compass)
  - 4 serial wires to control arbitrary RC equipment
- Board status: Out for fabrication...



Major Constraints

- Low Cost:
  - ~$200-300 bill of materials cost in respectable volume
- Very compact & light:
  - 30.5mm x 30.5mm mounting form
  - 35mm width
  - Small size necessarily means light weight
- Drift behind as much existing work as possible:
  - Xilinx FPGA: Rich IP core library
  - Pynq: Full Linux stack w tight integration options to programmable logic
  - Racing drones: Low cost & incredibly agile micro UAVs including 45m endurance fixed wings and 100 mph quad copters
    - Plus highly optimized PID-based control systems and code for low level flight control

Things that ***don't*** concern

- Power Consumption & Power Efficiency
  - Amdahl's law corollary: You don't need to optimize something when something else is already huge.
    Drone motors can be drawing 10+A at 6-24V...
    And quadcopters have 4 of them!
    - Even a high endurance, low cost fixed wing micro UAV is cruising at 5A@8V->40W
- "Deep Learning" AI-based vision processing
  - I'm highly skeptical of the entire field
    - I have a huge bias towards systems that are at least theoretically understandable
    - A neural network is really an (expensive) memory

Design synthesizes many ideas From This Class and Others...

- Awareness of memory latency & bandwidth
  - Want good caching for software, optimized behavior for DRAM
- Design hardware to support software, and software to support hardware
  - Have a vision of both in my head as I go through the design process
    - I want to map out as much of the design as possible in advance:
      Part of the point of these slides is to make the outline concrete
  - Was constrained on platform choice right from the start: This is as much compute & flexibility as I could fit within size/cost constraints.
- Know what fabrics are good for what:
  - FPGA is good at massively parallel operations, optimized for throughput
  - Linux software stack is highly flexible, huge software stack, but poor hard-real-time options
  - Separate processors are still flexible, really good for hard-real-time (no OS): 32 kHz update loops for low level flight controls

The Host Processors: "Mission Planning"

- Running a full Linux stack
  - For ease of development and maximum flexibility, network communication, lots of storage including SSD, etc etc etc...
  - Xilinx provides a good one, complete with python API
    - But in the end I may want to take the underlying C API and create a golang interface
- Has access to a high level view of the current state of all sensors
  - Provided by the programmable logic directly into memory
- Has high level API to direct actuators
  - Servo channel sending
- But Linux ***sucks*** for "hard" real-time activities
  - EG, every 60ms do X
  - Programmable logic will handle ***all logic*** that must be synchronized to an external clock

FPGA Design Components: Multiple Objectives

- Generic glue functions
  - Interfacing to multiple I/O functions
    - The hobby RC community uses somewhat silly pulse-length-encoding signals & some slightly bonkers serial protocols
    - Alternative for a processor would be to "bit bang" a general-purpose I/O line
- Realtime Video Preprocessing
  - Would otherwise use an obscene amount of DRAM bandwidth
- Realtime coprocessing
  - Ensures high response time despite the host processors running a full Linux processor stack by using coprocessors

FPGA Clocking

- Can derive 4 FPGA clocks from the 50 MHz input clock
  - Which is 20x in the internal PLL to 1 GHz
  - Processor core runs at 500 MHz in 6:2:1 mode
- Use some common derived clocks
  - 200 MHz:
    - Fast computation clock for highly pipelined stuff
  - 100 MHz:
    - Per-pixel computation clock for camera input: Exceeds pixel clock of the cameras (barely)
  - 25 MHz:
    - Slow logic clock & pixel clock for grayscale computations

FPGA/Processor Interfacing: AXI interfaces

- A nice bone-simple protocol:
  - Master: Read X bytes starting at this address
  - Master: Write X bytes starting at this address
- 2x AXI high-speed slaves to the memory ports
  - Enables burst-writes from FPGA logic
- 1x AXI high-speed slave that is cache-coherent with the CPU
  - ***Critical***: Allows much lower latency communication between CPUs and FPGA by passing data through the caches rather than DRAM
  - There is a bug: If bursting full cache-lines you have to write the whole line: no masking
- 2x AXI_GP masters
  - Only singleton reads/writes rather than bursts
  - But easy interface for controlling features in the programmable logic from the CPU

FPGA's "Glue" functionality

- Both the 4 output lines and the inputs from the GPS & accelerometers pass through the FPGA
  - Each is fully independent so they all can be controlled independently and in parallel
- They all speak slightly different protocols
  - There are hard macros available on the processor core for I2C etc but there are enough other things hanging off that it isn't always available
  - And if the hard macro is usable, it can be routed through the programmable logic's pins anyway
- So the FPGA's #1 role is to act as glue:
  - Stitching together the problem of interfacing with all the other specialized components
  - Not just getting the data but supporting initial filtering of data

Unique Glue #1: Futaba SBUS

- SBus is a serial protocol used for communicating with servos and controllers
  - Some radio receivers can be in master/slave mode: pass through an SBus signal when they don't have contact with their own transmitter
  - This enables safety override:
    Use a conventional hobby remote control... Turn it on and can take over from the drone's operations
- Enables setting the state on 16 11-bit servo "channels"
  - Traditionally, SBus sources just repeatedly send on regular intervals...
  - But I want to offload the task from the host processor

SBus Funkayyyyness...

- It is pretty much simple RS232 serial at a 100 kHz clock
  - 1 start bit, 8 data bits, 1 parity bit, 2 stop bits, sent with MSB first
- Message is 25 bytes total
  - 1 start byte
  - 22 data bytes
  - 1 flag bytes
  - 1 stop byte
- But the weirdnesses:
  - Output is logic-inverted: Some standard processor UARTs can't handle it
  - Forced to interleave the 16 words across the 22 bytes in a weird byte-order form

SBus Sender: 16 registers...

- There will be 32, 11b registers built using ***two*** simple-dual-port LUT-based SRAM cells
  - Need to be able to mix bits from different registers in the protocol, but still just 16 LUTs using 32x6 SDP mode
    - Control logic to use a single SDP RAM bank would be nearly as much, plus just a right PitA
  - Only using the first 18 locations however
    - 0-15 servos
    - 16 flags
    - 17 commands to the sender logic
    - Just write at 32b and ignore upper bits
- These will be writeable from the processor
  - Part of the AXI-GP slave system
- Processor can then just update a value to change the servo settings
  - No further processor attention needed

SBus Sending

- Simple, counter-based state machine
  - Just run on a 100 kHz clock
    - Don't worry about the processor writing to the reg-file, it will be at the internal logic clock rate instead...
      - Plus things are highly likely not to change much anyway
- Select between the start byte, the right selections from the RAMs, or the stop byte
  - While writing the stop byte also reread the status register
- Standard UART shift-register sending
  - With an optional bit to disable inverting:
    Some downstream devices require you to invert the otherwise inverted SBus signal. 🤷‍♂️

Glue #2: DShot-600

- Standard motor control is "pulse-width modulated"
  - How long is a signal up for...
  - Has issues with drift, etc...
- But the electronics used by these motor controllers are already optimized around decoding pulse-width signals...
  - So a kludge: a 16b serial message (11b data, 1b telemetry request, 4b CRC) encoded with pulse width modulation
- No sending: low voltage continuous
- Sending data: 1.67 µs bit clock
  - 0.625 µs high/1.045 µs low -> 0
  - 1.250 µs high/0.420 µs low -> 1
- 100 MHz -> .01 µs
  - So just count off the 100 MHz clock to determine shift points and sending values
  - Repeatedly send the contents of a register written by the processor

For Both:
Offload repeated bit-banging/sending loops...

- Low level flight control systems expect these signals to be repeatedly resent
  - But I don't want the processor to have to do a thing
- So the glue is two purposes
  - Translating the funkinesses
  - Offloading the need for any real-time operation for sending updates

Glue #3: Gyro polling & integration

- The two gyroscopes are designed to be polled at regular intervals
  - And then you want to do some noise filtering to remove various noise sources and create a coherent reference for probable location & acceleration
- Similar for the GPS
  - Albeit at a slower polling rate
- All have interface pins routed through the programmable logic
- Probably will use a soft core for this (more later)
  - But output to the ***cache coherent*** interface so this data is always available for the processor in a known memory location:
    High level mission planning can always know where it is based on fusion of the 3 sets of location sensors

But the big thing: Video Data...

- 1080p: 1920x1080 at 30 fps, 10b per pixel
  - Pixel clock rate of 68 MHz
- SXGA: 2592x1944 at 15 fps
  - Pixel clock rate of 80 MHz
- Theoretical DRAM: 2B wide, 800 MHz data clock
  - 1600 GB/s...
  - One camera, reading/writing 16b pixels (for alignment): 125 MB/s
    - Could probably do if I really really wanted to, but it would be painful and probably starve the host of DRAM access, since 2 cameras being written & read is 500 MB/s...
    - And we all know its practically impossible to really use full memory bandwidth unless it is really good at being streamed in the analysis
- So instead image processing ***must*** focus on single frame analysis

Video Pixels

- Alternates between B/G and G/R scanlines
  - Scanlines are horizontal sequences of pixels:
    The camera sequentially sends each pixel in a line, and then moves onto the next scanline
  - 10b of data for each pixel
- 1080p mode is a full pixel subset of the 2592x1944 mode
  - Only going to operate in these modes because these offer the highest resolution while still allowing fast feedback in the computer (15Hz or 30Hz)
  - Probably going to just stick with 1080p: XSGA mode has more "peripheral vision", but doesn't help in the central area of focus

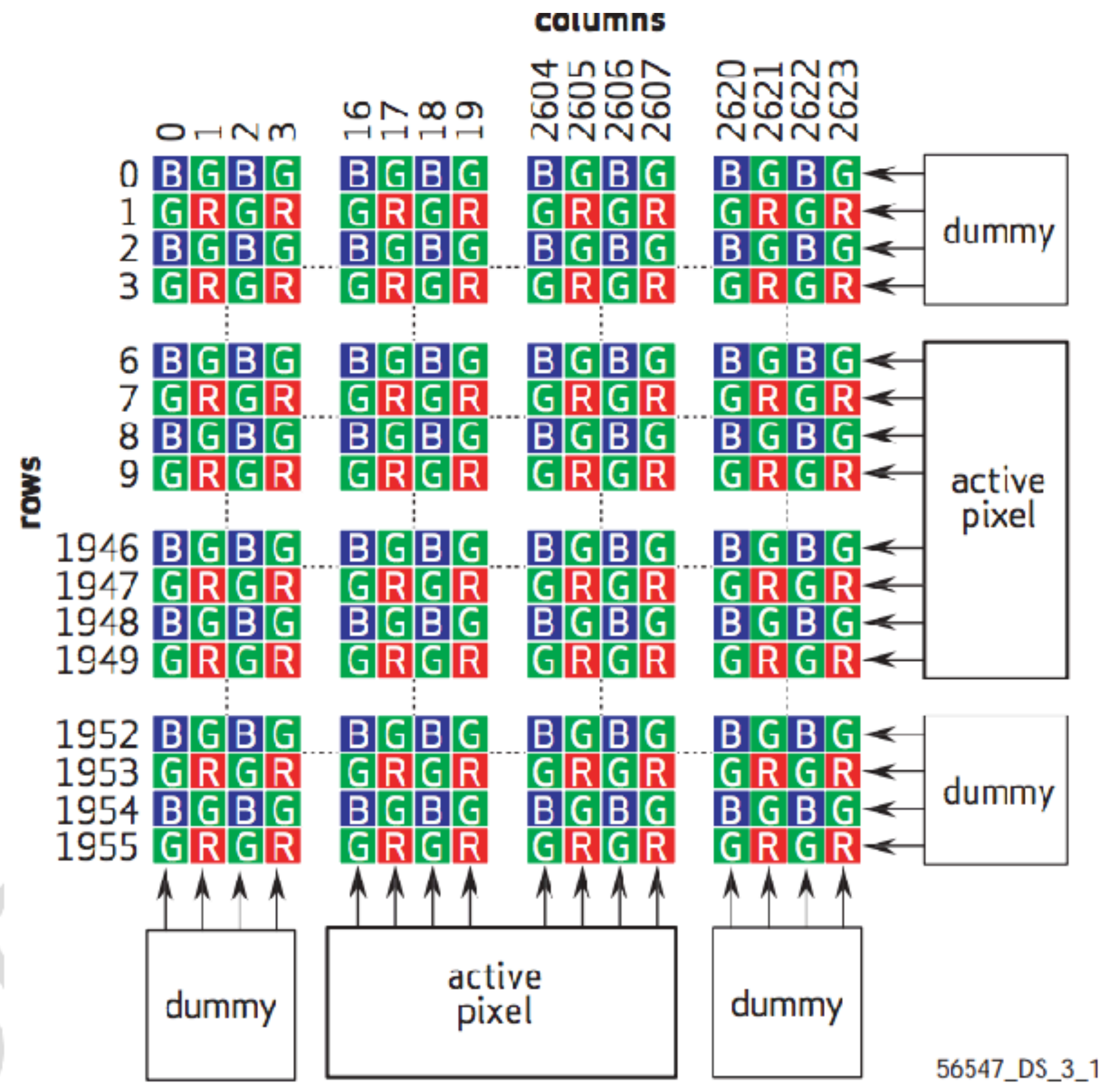


Image Processing Task #1: Image Windowing

- I can't effectively write whole video frame into memory
  - And have a lot of leftover memory bandwidth to do any other cool stuff...
  - Especially if you think programmable logic is annoyed by DRAM latency, processors can't do it nearly as well...
    - So shoving it into the DRAM really doesn't solve the problem anyway!
- But I can establish an "area of focus"
  - AXI-GP request: "I want an X by Y window from this camera with this downscaling to this aligned memory location"
  - As the scan-line comes in, matching pixels are first batched and then written to the ***cache coherent*** interface
    - So writes end up in the caches if the memory is being accessed

Window of Focus model...

- Allows mission planning logic to "scan the sky"
  - Look around for areas of interest and perform more substantial calculations
- Only limitation: You don't know what you aren't looking at...
  - But then again, you'll get the next frame 1/30th of a second later...
- But it allows me to greatly reduce the memory bandwidth:
  - A 256x256 window is 128kB (10b->16b for word alignment), and even if I'm double-buffering I'll never miss in L2 (which is 512kB)!
    - Even on a 500 MHz ARM, the miss penalty to DRAM is huge compared with the hit time for L2
    - And cache associativity is high enough that conflict misses won't kill me either

Concurrent Access?!?

- Option 1:
  - Window is always writing to one of two buffers
  - Mission planning software just accepts that the buffer contents can change while it works
- Option 2:
  - Mission planning software notifies hardware of "buffer change": ***one*** of the two buffers needs to be switch
    - Hardware switches the buffer it is ***not*** currently writing
  - Software requests acknowledgement from hardware
    - Hardware tells software what buffers it is now using
  - Software now knows which frame won't be modified while it works
- Will use option 2:
  - Concurrency is already a PitA: Being able to say 'no, there is but one owner' makes the process vastly simpler

Image Processing Task #2: Edge Detection

- So many subsequent analyses...
  - Target selection and identification
  - Optical flow to look for obstacles
  - Just knowing an area is "interesting"
- All can benefit from looking at edges rather than objects
  - Plus edges require ***much*** less data to store: 1b per pixel
- And can be done with grayscale
  - Which reduces the pixel clock and count by 75% right from the start
  - So 1080p images become 64kB bitmaps!

Edge Detection Flow...

- Incoming Image, convert 4 pixels to grayscale:
  - Runs at the 100 MHz clock, the rest runs at the synchronized 25 MHz clock
- Gaussian filter
  - For each grayscale pixel, do a 5x5 Gaussian filter to eliminate high frequency noise
- Gradient Edge Detection
  - For each pixel in each of 4 directions, get the gradient. High gradient->edge
- Edge thinning & thresholding
  - For each pixel, if its gradient is the greatest, add in the other gradients. If it is above the threshold, mark the edge.
- Net result requires 4 passes over the entire image...
  - But only need to look at a small window within the image
- This is a hacked-up version of Canny's edge detection algorithm

Image Processing Pipeline: Scanline Buffer...

- Allows evaluating both the current incoming pixel and the same position in the previous scanline
  - Register the pixels to have the previous pixels within the scanlines
- A single 18-bit channel for a 1920b scanline can be buffered in a single 36kb BlockRAM
  - Read Addr = Write Addr - 1920: Gives the corresponding pixel from the previous scan-line
- Other channels of note:
  - 18b, 960 -> 18kb BlockRAM
  - 18b, 2592 -> 2x36kb BlockRAM, 18b, 1296 -> 36kb BlockRAM, 9b, 1296 -> 18kb BlockRAM
- Very simple control logic
  - If byte available to write:
    Write byte, increment both counters, assert byte available to read
- Chain multiple together for larger vertical windows
  - Now can examine an even larger window

Grayscale Conversion...

- Incoming pixels:
  - `B/G/B/G/B/G`
  - `G/R/G/R/G/R`
- Scanline buffer for grayscale conversion:
  - Register 1 input & 1 output to get a 2x2 moving window
  - Only compute every other pixel on the odd # scan-lines
- Fixed-point luminosity grayscale conversion:
  - 0.3 * R + 0.295 * G1 + 0.295 * G2 + 0.11 * B
    - Use fixed-point approximations to actually implement
    - Notice green is more important for total intensity: matches human perception
  - But we can go from 10 bits to 16 bits in the process since 10 is already 1b too many for the BlockRAMs
- Also serves to ***radically*** reduce the pixel rate
  - Now it is 1/4 the pixels and 1920x1080 becomes 960x540, 2592x1944 becomes 1296x972
  - Drops the pixel clock by 75%: Everything downstream can now use the 25 MHz clock!
    - Could even ***possibly*** contemplate saving to DRAM: 31 MB/s/camera, 1MB images at 960x540

Handling the edges of the frame...

- Option 1: Don't bother...
  - For 5x5 analyses, each pass adds corruption to 2 pixels worth at the edge: With 3 passes we corrupt a 6 pixel border
- Option 2: Special case the edges...
  - Just repeat the same pixel that is on the edge itself
- Going to go with the latter
  - Since this is now operating on grayscale at 25 MHz, the additional muxes are eh, who cares.

Gaussian Blur

- A standard image filter designed to remove high-frequency noise
  - Since most camera noise appears on just one or two pixels, a little blurring goes a long way
- For each pixel in a 5x5 grid
  - Multiply it by the constant weight
- Sum all the pixels
- Divide by the constant to normalize things
- Available parallelism: Obscene
  - 25 MACs per pixel, and computation is per-pixel independent, too

$$\frac{1}{273}\begin{array}{|c|c|c|c|c|}\hline 1 & 4 & 7 & 4 & 1 \\ \hline 4 & 16 & 26 & 16 & 4 \\ \hline 7 & 26 & 41 & 26 & 7 \\ \hline 4 & 16 & 26 & 16 & 4 \\ \hline 1 & 4 & 7 & 4 & 1 \\ \hline \end{array}$$

The Multiplication & Summation...

- Its just an adder tree
  - x1, x4, x16 -> shift
  - x7 -> shift and subtract (X << 3 - X)
  - x26, x41 -> shifts and 3 adds
  - And then just add everything up in a tree
- So 16 + 2 * 4 + 5 * 3 = 39 adders
  - Increase of bit width for the output is 9 bits wider than the input
  - And tree structure -> 6 adders deep and even better critical path
- Pure feed forward
  - So can pipeline it trivially if needed, but its at the 25 MHz clock anyway so...

The Division in Hardware

- 1/273 is the same as multiply by 0.00366300366...
  - But floating point multiply sucks...
- Use an online IEEE floating-point mapping tool, find out...
  - Can represent (in binary) as 1.111000001111 * 2^{-9}
    - With a trivial error
  - Can also thus represent (in binary) as 1111000001111 * 2^{-21}
- Thus this becomes a multiplication by a 13 bit constant followed by a constant right shift of 21 bits
  - So 8 adders
  - But right-shift by 13 instead to create an 18b range output

Gradient Detection...

- Want to look for the maximum gradient in all 4 main directions
  - Horizontal, vertical, two diagonals
- Gradient calculated by selecting the maximum pixel in the line of 5
  - If on left, subtract min of center and two on the right
  - If on the right, subtract min of center and two on the left
  - If center, subtract min of other 4
- Store 4 separate gradients, one for each direction
  - Truncate to 9 bit values
- Output goes from 18b to 36b

Thinning and Thresholding...

- For each cardinal direction
  - If not the maximum gradient, not an edge in this direction
  - If is the maximum gradient, sum up all gradients on this direction
    - If sum(gradients) > threshold: pixel is an edge
- Pixel is an edge if it is an edge on any cardinal direction
  - Reduces image to a single-bit value: 960x540 image -> 64 kB
    - But pad the output a bit, so that a scan-line occupies an 8-word alignment:
      Makes accessing arbitrary lines easier and keeps things staggered just enough to prevent conflict misses
- Write resulting image into main memory
  - 64 kB still big for BlockRAM but no longer an issue with memory bandwidth for shoving into the DRAM
  - Also small enough to buffer a second worth of frames, not just a single frame

Memory Budget for a Camera Channel Edge Detection at 1080p

- 1 36kb BlockRAM for the grayscale conversion
- 4 18kb BlockRAMs for the Gaussian Blur input
- 4 18kb BlockRAMs for the Edge Detection input
- 8 18kb BlockRAMs for the Thinning & Thresholding
- So this becomes 9 36kb BlockRAMs
    - Have either 70 or 120 depending on the part

Optical Flow...

- Given two images taken in different positions
- Find regions that correspond between the two images but represent transformations in scale & position
- For such regions: find direction & magnitude of the necessary transformations
- Then with some trigonometry based on the the direction & magnitude of the scalings and knowledge of the two positions...
  - Can know how far each region is away from the camera
    - Let the host processor do this: its way easier with floating point
- This is a significant component of how humans work for longer distances
  - Stereo vision is really only good at distinguishing things up close

Optical Flow Cheats For This Application...

- Assume we are doing optical-flow on a ***forward mounted*** camera...
  - Limits ***most*** displacement to the horizontal direction, and objects get bigger
  - So although optical flow clearly requires looking at a significant X by Y window between the two images...
    - We should bias our search space for horizontal displacements
- And we can do multiple frames at the same time:
  - Take the current frame as the reference and do optical flow comparison with targets of the last frame, frame-15, frame-29
    - The last frame is very low error for even close-in stuff: very low displacement
    - The -29 frame is high error but good for far away stuff: large displacement
- Can also possibly use this to detect moving objects
  - They will stand out as different on the optical flow: moving "odd" compared to everything else

Optical Flow Line Buffer...

- Can't do a standard line-buffer approach
  - Need to compare a segment of the reference image with a larger hunk of the target image
- Instead need to do a "load and read" approach
  - Load the area under examination, then repeatedly read it out
- Pad a 960 pixel line to 1024 (makes addressing easier)
  - 18kb BlockRam: can hold 16 lines and read out 16b at a time
- Probably want to run at the 100 MHz clock
  - For each 32x4 window of the reference frame, see where in a 64x16 window of the output is the best match of the pattern and determine the scaling & transformation
    - Overlap the 32x4 regions however and only record the center 4x4 value

Comparisons...

- With no edges...
  - Can't compute, ah well
- With just a single clean edge...
  - Can only get displacement, not scaling
  - And select the one closest to the origin that matches the basic shape
- With 2+ edges
  - Can get displacement & scaling
- Yes, this requires a ***lot*** of comparisons!
  - But there are tons of parallelization opportunities

Parallelization Opportunities

- Each comparison is, itself, done in parallel for all bits in the reference image
  - Each pixel in the target compared against its corresponding pixel + 3 surrounding in the reference
    - Output is 2-bit, not 1-bit
      Bits selected for biasing towards the object being bigger in the reference frame
  - All comparisons are summed to create a target confidence
    - Confidence is biased by how far from the center it is & # of edge pixels in reference
  - Also comparing against upsampled & downsampled reference for scaling measurement
- Compare against multiple source windows simultaneously
  - Natural breakdown: Compare with all windows in a vertical slice
  - Then each clock cycle, shift each bit by 1
  - Does require a ***lot*** of registers & comparisons, but we knew that coming in!
- Keep a running total of "best match"
  - And then when completed, we have it...
- And then compare with multiple frames at the same time...
  - If the resources allow it

Final Component: Slave Coprocessors

- The race-drone community has some nice low-level flight control software
  - Designed to run on ~70-150 MHz 3-stage pipeline ARM cores with single-precision FPU
  - Basically performance-equivalent to a Xilinx Microblaze soft-core
- Initial deployment will use external flight controller board...
  - But will eventually want to migrate that functionality onto the FPGA itself:
    Why spend an extra $50/drone on a separate FC when the board already has 2 independent gyros and all the other stuff needed?
- Second slave coprocessor to do input filtering on GPS/gyros
  - Its a continuous, real-time task
    - So don't want it on the host processor
  - But its complex, branchy, coded in C, and uses floating point
    - So can't directly implement on the FPGA fabric directly