Real-Life Design Trade-Offs

(choices, optimizations, fine tuning)

EDAN85: Lecture 3

Real-life restrictions

- Limited type and amount of resources
- Changing specifications
- Limited knowledge

Limited Resources

- **Time**
- **Tools**:
  Build (compilers, synthesis, technology), Test, Debug, Maintain
- Target **hardware** support:
  Available IPs/chips, Fixed architecture
- Target **software** support:
  OS, libraries, drivers, protocols,…

Challenges

1. Design using available components
   - Select the most suitable architecture
   - Adapt components Hw/Sw to the given/fixed parts
   - Allow some flexibility, configurability (shifting specs.)
2. Optimize
   - Do not use more (area, memory, …) than you need
   - Add Hw to speed up/simplify Sw (e.g. DMA ctrl)
3. Test & Debug

Hw/architecture design guidelines

A. Start with a simple, working design

B. Expand gradually by adding tested IPs

C. Design custom IPs only when necessary

D. Communicating data is usually the bottleneck, not computing: choose fast/many memories, buses

E. Improve/Optimize a working prototype

Memory Band-width: VGA frame buffer example

640 x 480
rrrbbbggg
60Hz

640 x 480 x 9 x 60 =
165.888 kb/s
… = 5.184kW/s =
5.2MW/s

…on a Microblaze system

100 MHz bus clock

Approx. 1 word each 19 cycles (read access!)

Memory Band-width: VGA frame buffer example (II)

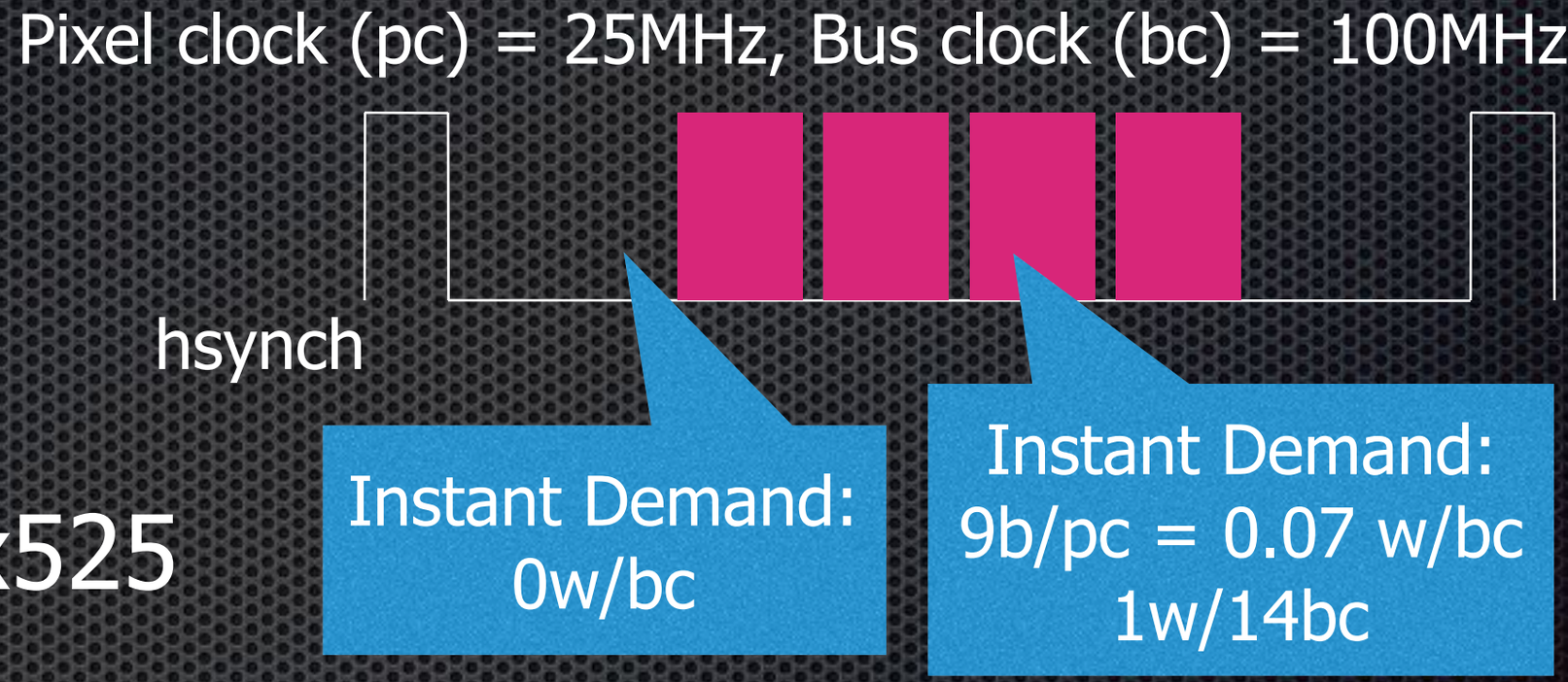
- LMB_BRAM:
  - Single read access: 1clock cycle/word → **no problem**
    Avg. bus utilization due to ctrl = 5% (takes 1 of 20cc)
- PLB_BRAM: (plb_bram_if_ctrl.pdf)
  - Single read: 6cc/w → **OK?**
    Avg. bus utilization due to VGA ctrl =30%
  - Burst reads: ~10cc/4x2w = 1.25cc/w → **no problem**
    Avg. bus U = 7%

Memory Band-width: VGA frame buffer example (III)

- PLB_DDR (plb_ddr.pdf)
  - Single read: 14cc/2w → **OK?**, avg. U = 35%
  - Burst read: 16cc/2x2w ~4cc/w → **OK**, avg. U. = 25%
- PLB_EMC (SRAM,Flash): 8-bit access
  - Single read: 10cc/B (40cc/w)→ **Problem**.
    Bus cannot cope: avg. U = 200%
  - Burst read, …etc.

Variable Band-width: VGA frame buffer example (IV)

Total pixels 800x525

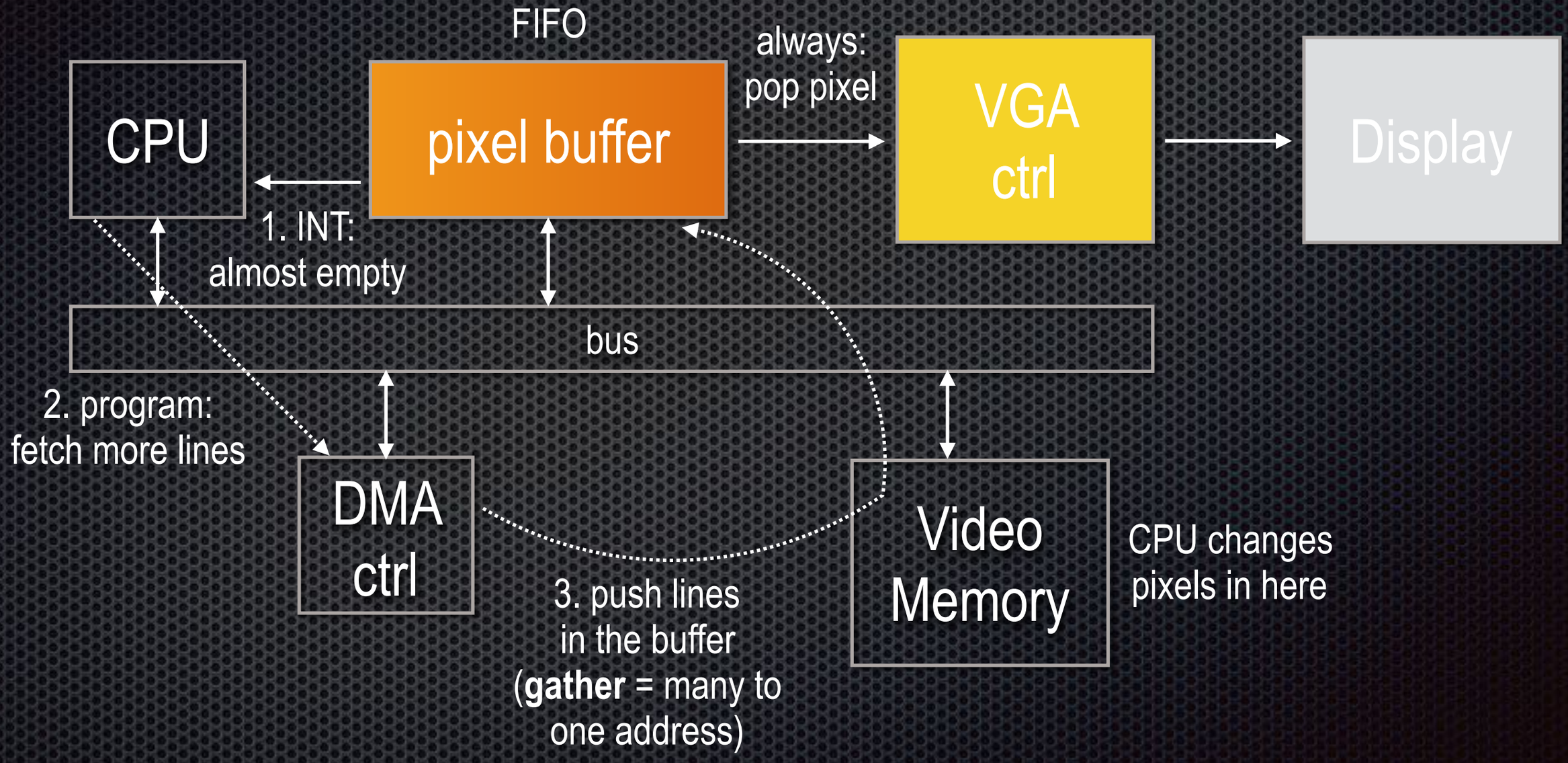


- Band-width demand changes at run-time:
  - High band-width may be too high for the chosen bus
  - Smoother bus utilization may be required
- Solution: **BUFFERING (and prefetch)** !

Variable Band-width: VGA frame buffer example (V)

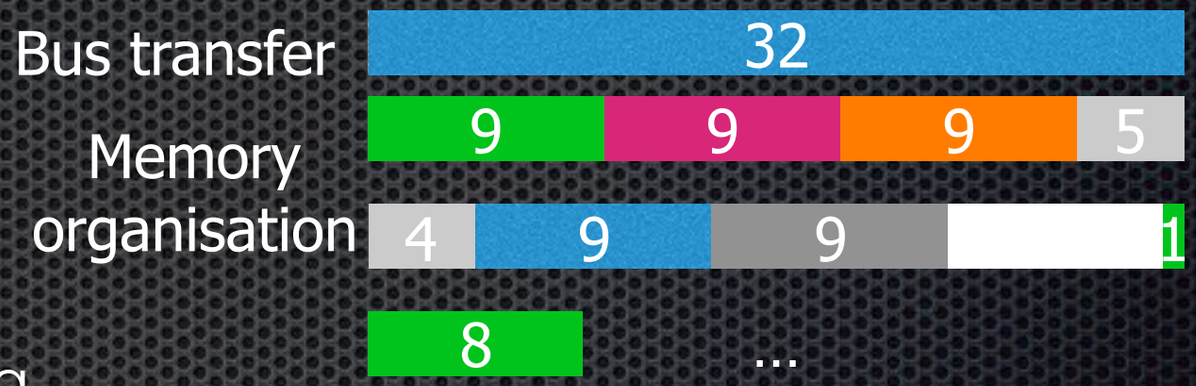
- Challenge: Keep the buffer busy (filled with data)
- Buffer size?
  - Easy way out: full frame - not always possible
  - Trial and error:
    start with a small buffer, increase it if the controller starves
  - Analysis:
    1. Compute the avg. rate (19bc/w ~ 0.052w/bc)
    2. Size = Longest_time_without_using_data x Rate
       (last pixel to first pixel delay)
       vhdl: (525-480+1)x800x0.052 = 1914 words (~2kw)

Intermission:
A VGA buffer architecture



Fine Tuning: VGA frame buffer example (VI)

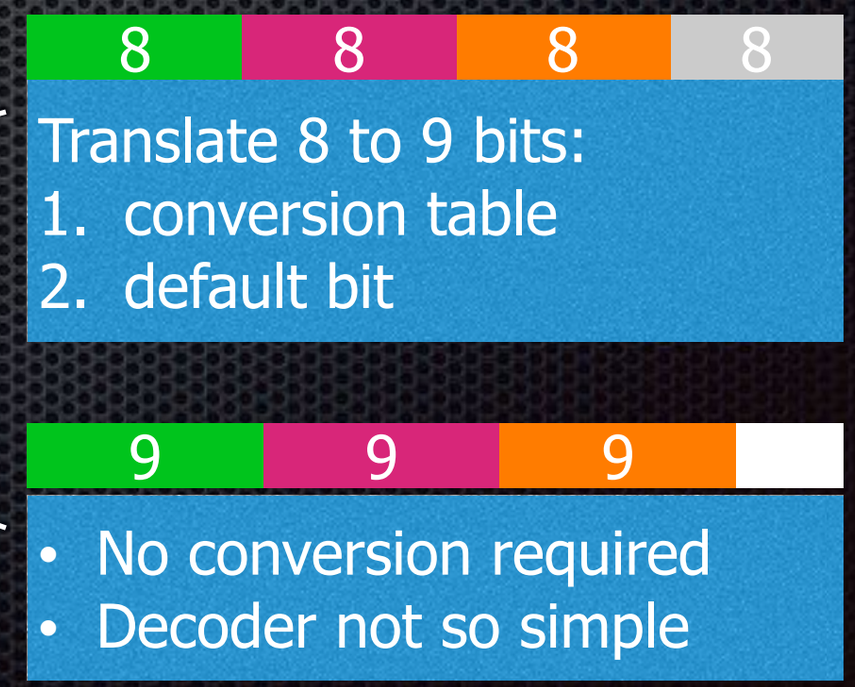
- Initial assumption: all bits in a word carry information!
  - complex decoder and unpacking method



Solutions:

A. Reduce bpp: 8 (4p/w)

B. Align & discard bits



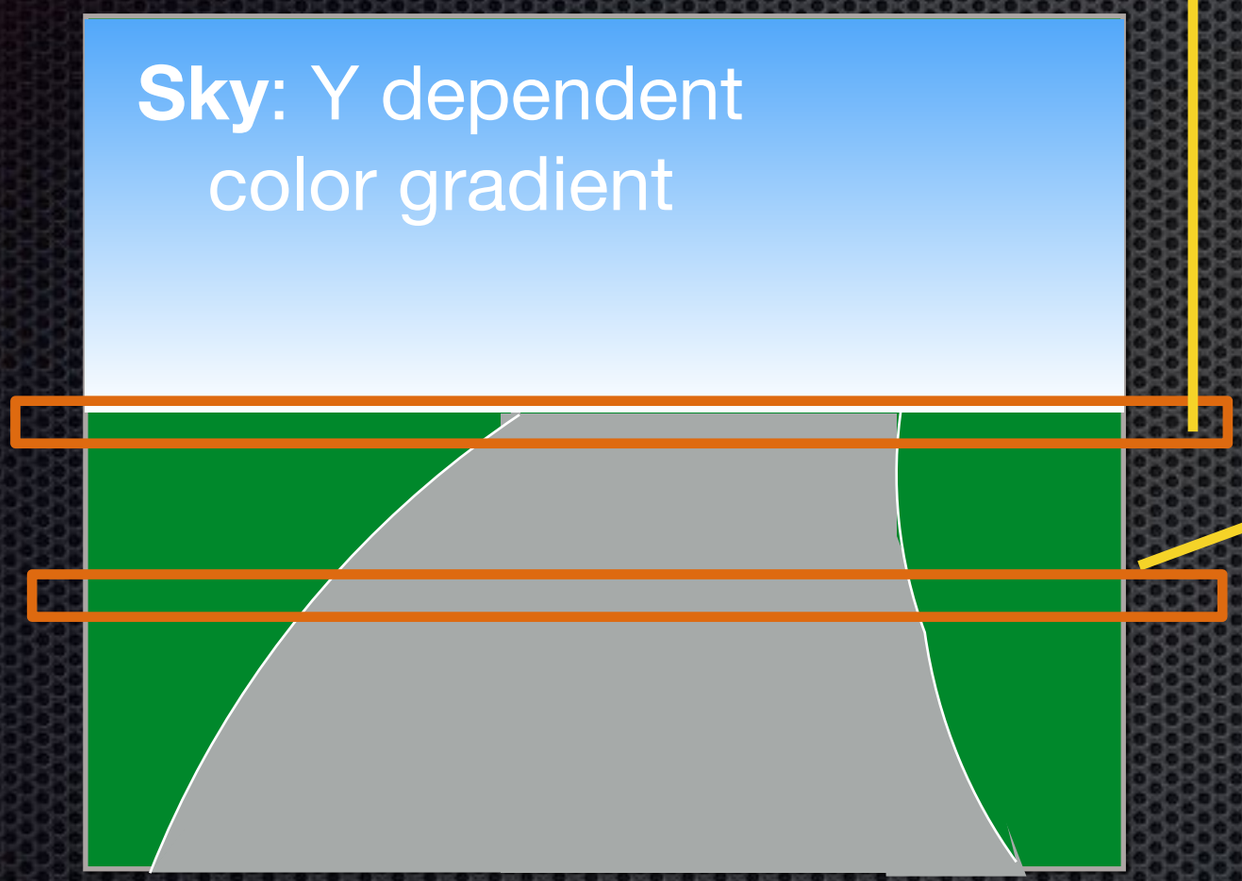
Required band-width and buffer size change!

Other solutions (VGA fb) ...

- reduce the visible window (less data)
- reduce the resolution (CGA, blocky)
- dynamic image generation (not fb)
  - custom solution (eg: background, road, cave…)
  - sprites
  - “vector” graphics
- a mix of the above

VGA: custom solutions - a dynamically generated background

Horizon: generates new road “line”



Sky: Y dependent color gradient

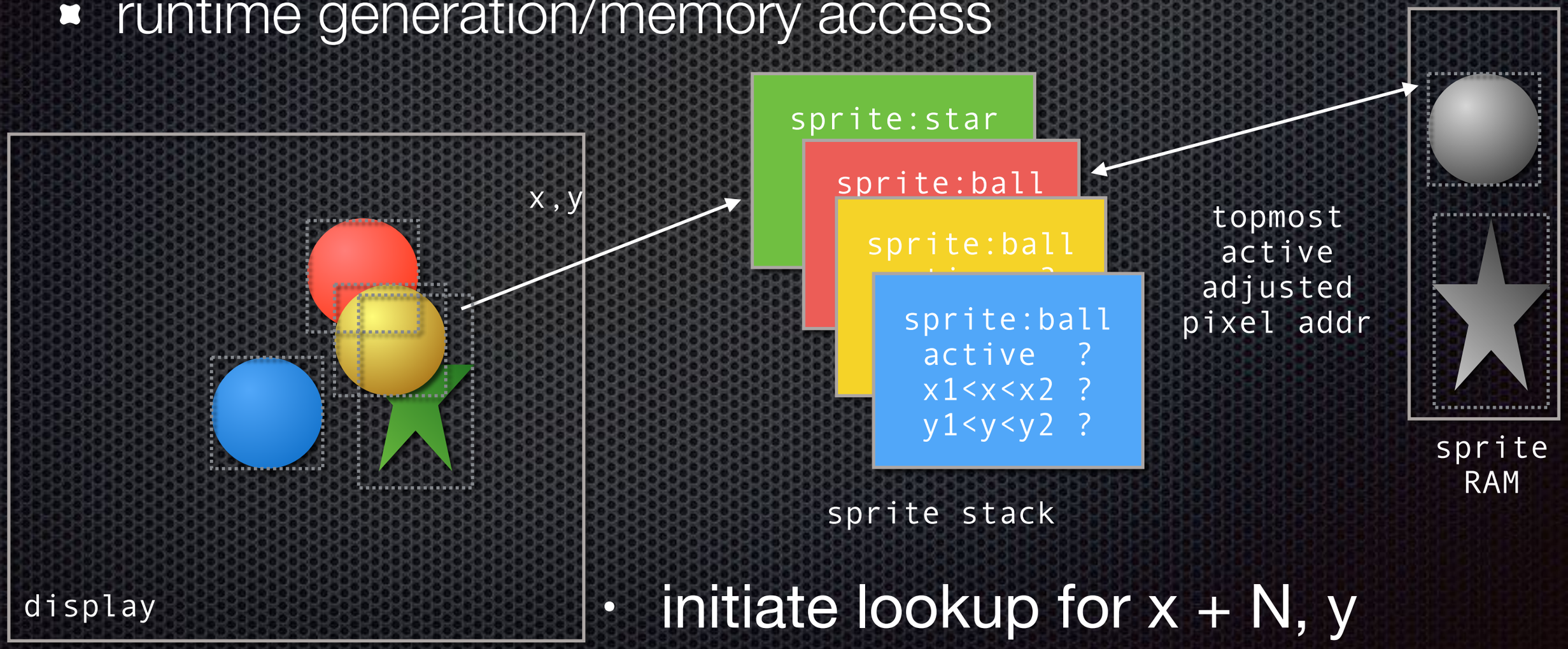
Line: stores start-end “road” shifts “down” and “grows” regularly

Video memory: an array of start-end pairs, shifting regularly (speed)

more: roadside posts, middle marks, …

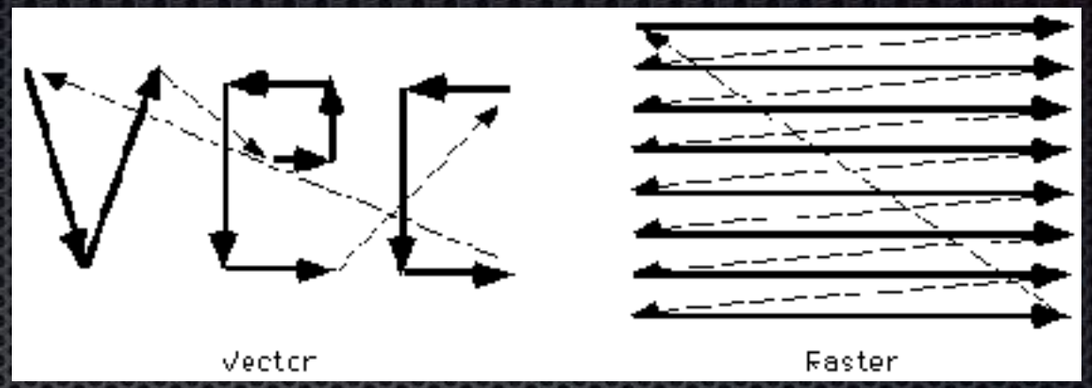
VGA: sprites

- multiple instances of the same image (memory)
- runtime generation/memory access

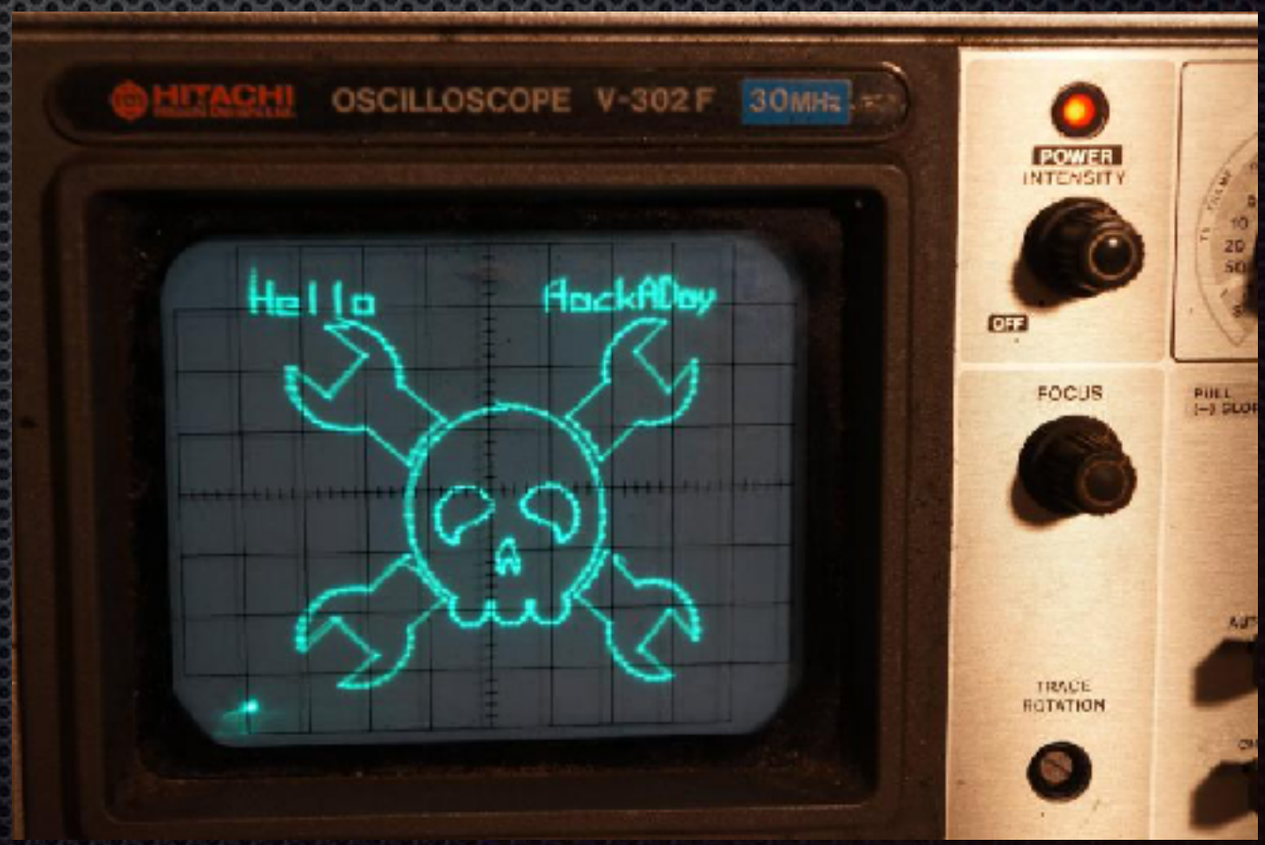


- initiate lookup for x + N, y

VGA: vector graphics

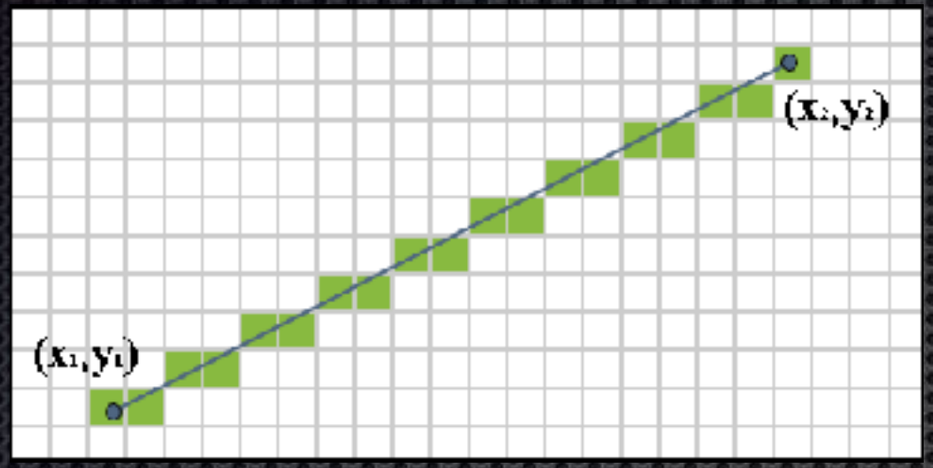


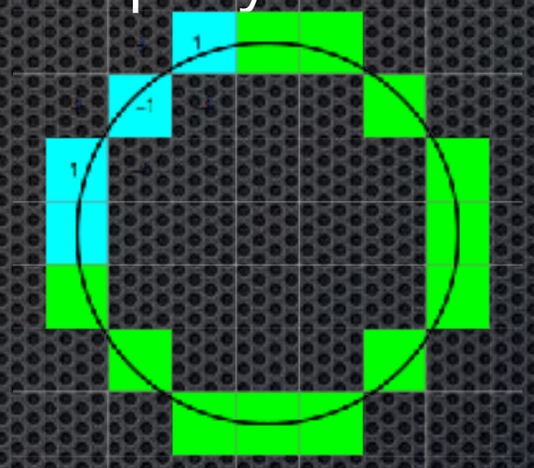
- initially made for vector displays
- based on primitives
  - lines, triangles, polygons
  - circles, curves
- store minimal info
  - start-end points
  - center-radius…



Vector Graphics Challenges

- rasterization is needed on modern displays





See Bresenham: http://www.cs.columbia.edu/~sedwards/classes/2012/4840/lines.pdf, http://members.chello.at/easyfilter/bresenham.html

- a frame buffer is often assumed!
- computationally intensive…
- dynamic generation - interesting problem

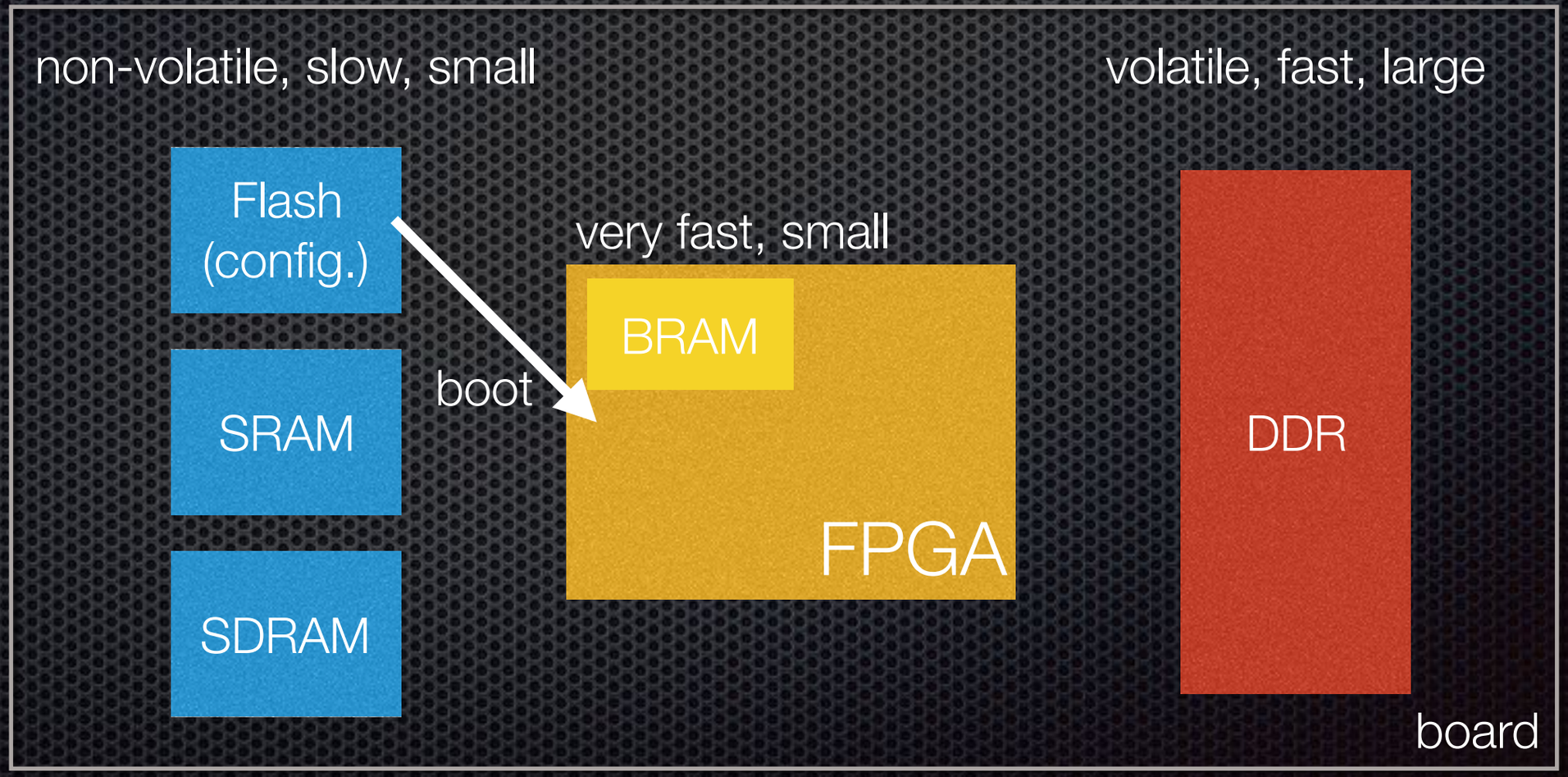
IP Configuration

Trade-off area/power for performance:

- Processor
  A. Cache type/size
  B. Floating point support
  C. Pipeline depth (?)
- Memory sizes
- Interconnect type/width (buses)
- Timing/wait states

Memory Size Issues

- **problem**:
  the program does not fit in the available on-chip BRAM



Memory Size Issues

- Many solutions:
  - A. compile with -Os, remove debug info.
  - B. put the stack and heap in off-chip memories
    - need to use available SDRAM, SRAM/Flash, DDR
  - C. execute from non-volatile off-chip memory
    - boot from BRAM, jump to an executable off-chip
    - use caches to speed up
  - D. decompress exec. from SRAM/Flash to DDR at boot

Memory Size Issues: SRAM executable example

Steps:

1. Link the main application from SRAM_BASEADDR
2. *mb-objcopy -O binary main_app.elf main_app.bin*
3. Write/compile/link a bootloader from 0x0000

```
typedef int (*maintype)(int,char**);
maintype maincode = (maintype)SRAM_BASEADDR;
int main(int argn, char **argv) { return maincode(argn, argv); }
```

4. Add MDM debug periph., set mblaze DEBUG_ENABLE flag
5. Download configuration, connect in xmd: *mbconnect mdm*
6. *xmd> dow -data main_app.bin SRAM_BASEADDR* **once!**
7. Run or Download configuration again

Or… flash both Hw and Sw:

https://reference.digilentinc.com/learn/programmable-logic/tutorials/htsspisf/start

Software fine tuning

To adjust code speed and size:

A. Algorithm selection (e.g. bubble vs. quick sort)

B. Compiler optimization options

C. Linking options

- segment splitting: distribute code, stack, heap,…

D. Driver/library choices

- low level, small footprint, reduced functionality vs. high level, large footprint, loads of functionality

Drivers - Hw/Sw interface: VGA frame buffer example (VII)

A. 8bpp (4p/w)
   - Easy to modify single pixels (Xio_Out8) by writing single bytes

B. 9bpp (3p/w)
   - Single pixels: read, modify & write
   - Exact address/offset computation more complex

C. Packed 9bpp
   - Even harder to compute the offset/address, build masks, access split pixels, etc.

Conclusions

- Trade-offs are very common
  (e.g. band-width vs. simplicity, Hw vs. Sw)
- Hardware, software, and interfaces:
  must be designed together!
- Knowledge about the available components is key