

Graphics and Framebuffers

Baremetal on the Pi

Raspberry Pi A+

ARM processor and memory

Peripherals: GPIO, timers, UART (gpio, uart), keyboard

Assembly and machine language (as)

C and pointers (gcc)

Functions and the stack (gdb)

Serial communication and strings (uart, printf)

Linking and the memory map (ld, memmap, objcopy)

Loading using the bootloader (rpi-install.py, bootloader)

Starting (start.s, cstart.c)

Memory managements

Tools (git, bash, make, brew)

The Force Awakens in You

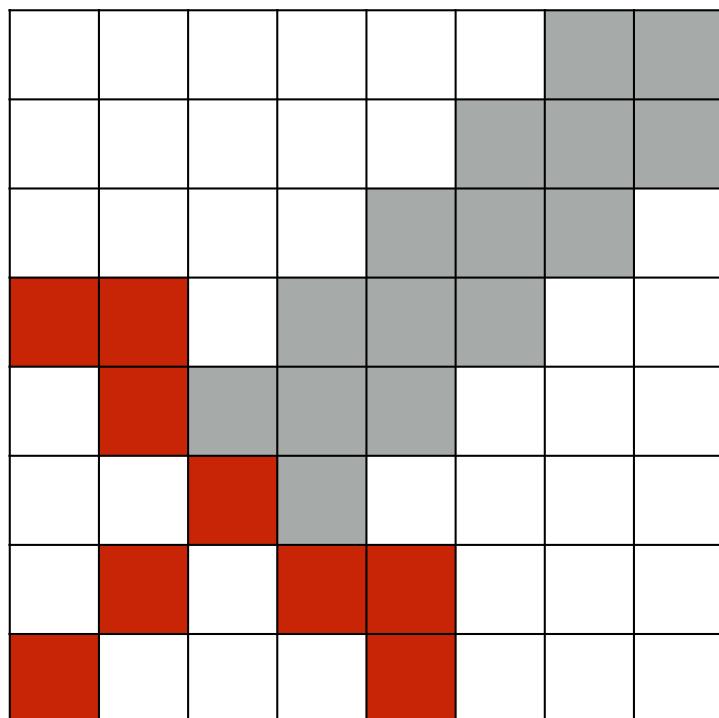




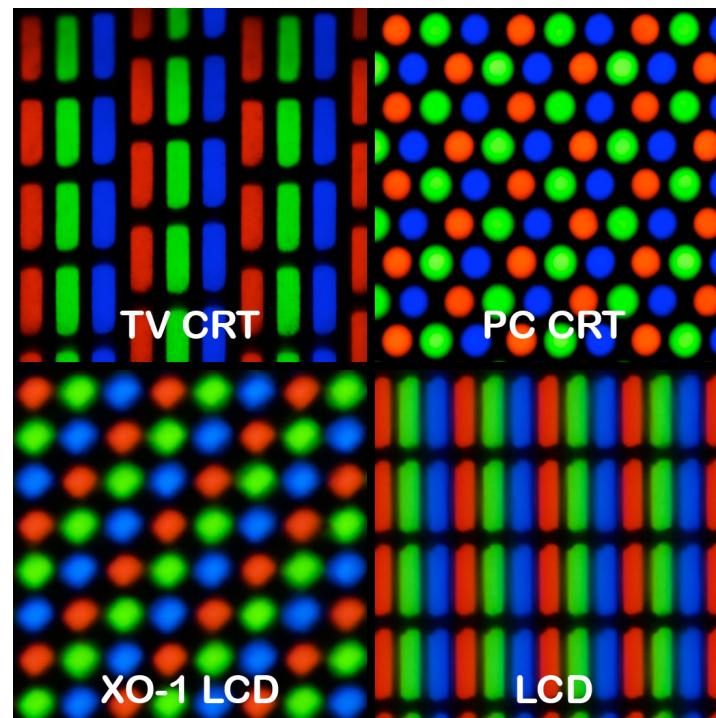
gpio
timer
uart
printf
malloc
keyboard
shell
fb
gl
console



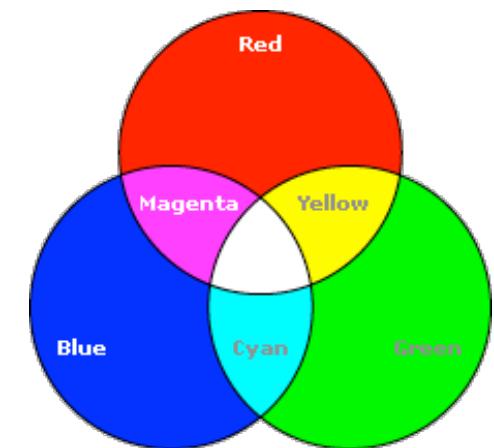
Pixels



Displays



Light



HDMI

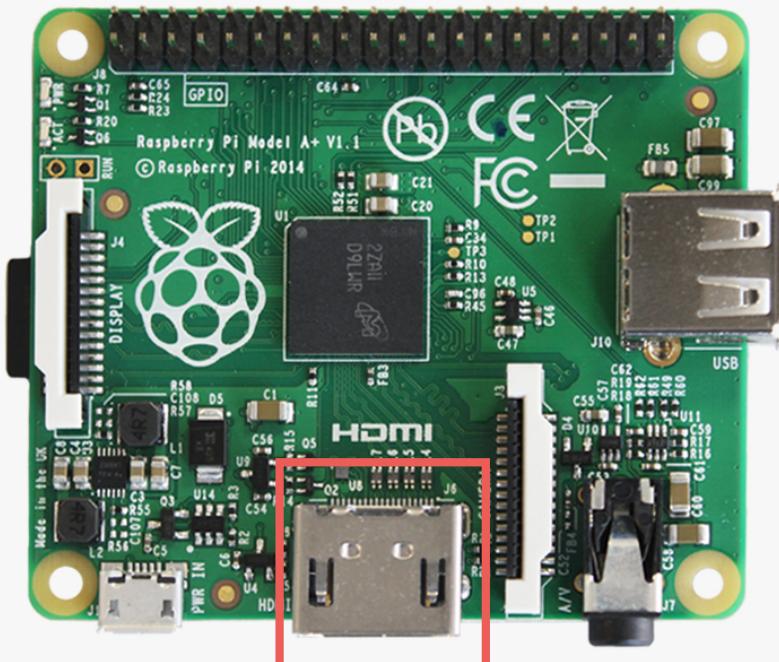


Figure from
High-Definition Multimedia Interface
Specification Version 1.3a

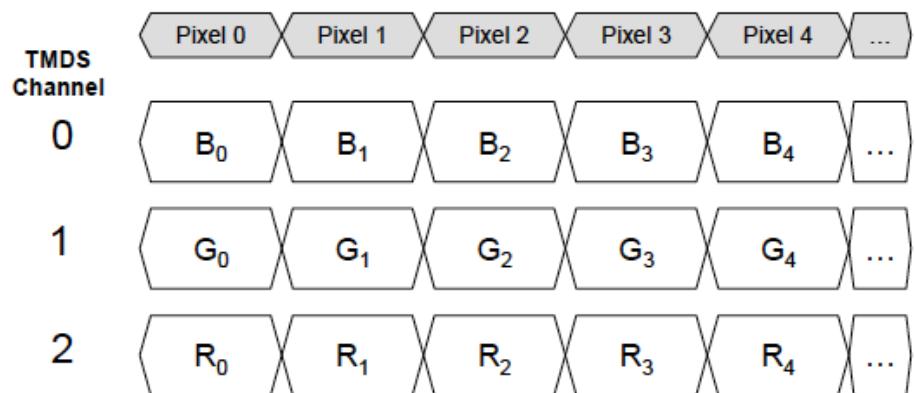
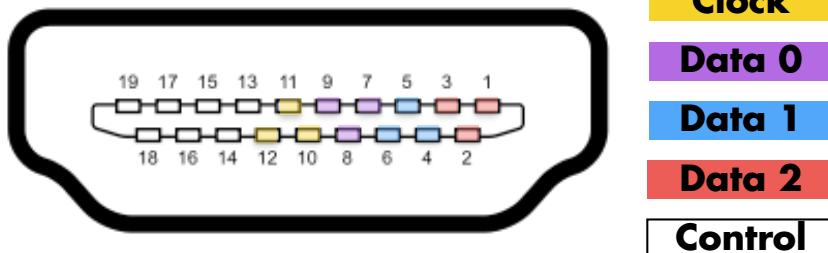


Figure 6-1 Default pixel encoding: RGB 4:4:4, 8 bits/component





The framebuffer stores an image

An image is a 2D array of pixels



RGBA pixel (depth=32 bits)

Red = 8 bits
Green = 8 bits
Blue = 8 bits
Alpha = 8 bits

Framebuffer Resolution

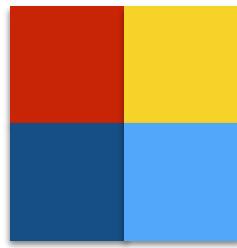
read fb physical size

read fb virtual size

read fb pixel depth

code/video

virtual height = 2



Virtual width = 2

**2x2 image
is interpolated to a
1600x1200 image**



fb_config_t Structure

10 parameters

Field	CPU	GPU	Description
width	write	read	Width of physical screen
height	write	read	Height of physical screen
virtual_width	write	read	Width of framebuffer
virtual_height	write	read	Height of framebuffer
pitch	read	write	Bytes/row of framebuffer
depth	write	read	Bits/pixel of framebuffer
x_offset	write	read	X offset of screen in framebuffer
y_offset	write	read	Y offset of screen in framebuffer
pointer	read	write	Pointer to framebuffer
size	read	write	Size of framebuffer in bytes

CPU writes/reads to/from GPU

Configure Framebuffer

set physical size

set virtual size

set depth

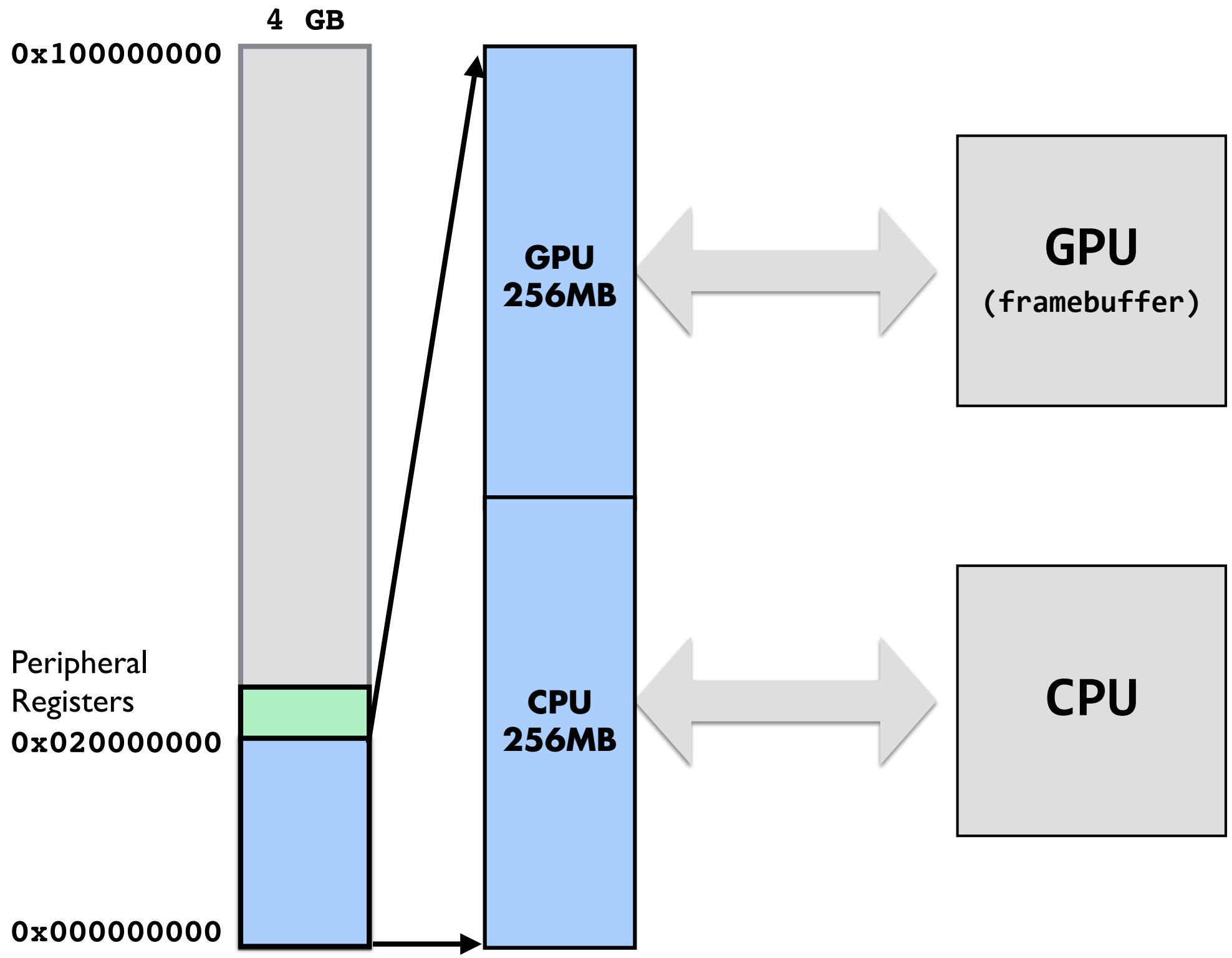
read pitch, size, fb pointer

code/fb

Shared Memory

**frame buffer memory
split between cpu & gpu**

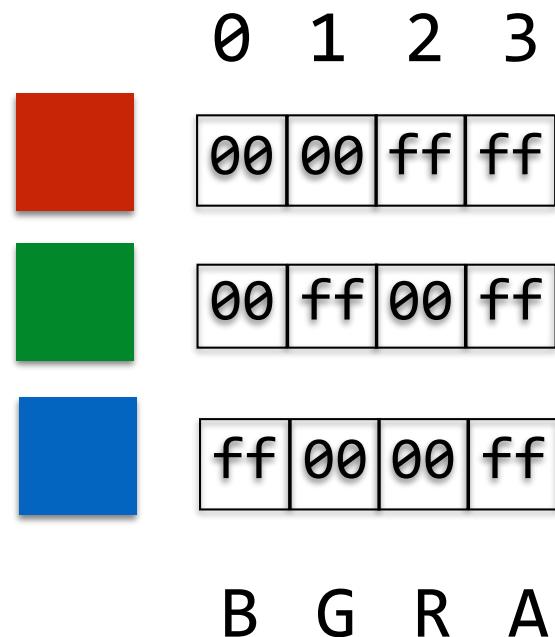
code/video



RGBA (BGRA) Pixel/Color

Pixels have Red, Green, Blue, and Alpha

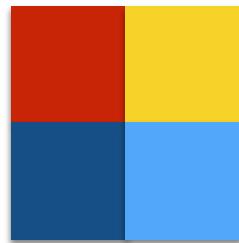
Stored as BGRA in four consecutive bytes



Beware: blue is the first

```
unsigned char fb[2*2*4];  
fb[0] = 0x00; // b  
fb[1] = 0x00; // g  
fb[2] = 0xff; // r  
fb[3] = 0xff; // a
```

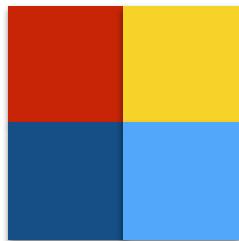
...



<table border="1"><tr><td>00</td><td>00</td><td>ff</td><td>ff</td></tr></table>	00	00	ff	ff	red	<table border="1"><tr><td>00</td><td>ff</td><td>ff</td><td>ff</td></tr></table>	00	ff	ff	ff	yellow	<table border="1"><tr><td>ff</td><td>00</td><td>00</td><td>ff</td></tr></table>	ff	00	00	ff	blue	<table border="1"><tr><td>ff</td><td>ff</td><td>00</td><td>ff</td></tr></table>	ff	ff	00	ff	cyan
00	00	ff	ff																				
00	ff	ff	ff																				
ff	00	00	ff																				
ff	ff	00	ff																				

- (0,0) is at the upper left corner of the display
- The order of pixels is y then x then bgra

```
unsigned char fb[2*2*4];
fb[bgra + 4*(x + 2*y)] = ...
bgra = 0, 1, 2, or 3
```



00	00	ff	ff	00	ff	ff	ff	ff	00	00	ff	ff	ff	00	ff
red				yellow				blue				cyan			

The frame buffer has a width and height

```
#define DEPTH 4
#define WIDTH 2
#define HEIGHT 2

unsigned char fb[WIDTH*HEIGHT*DEPTH];
fb[bgra + DEPTH*(x + WIDTH*y)] = ...
// bgra = 0, 1, 2, or 3
```

The width and height can be changed

```
#define DEPTH 4
#define WIDTH 640
#define HEIGHT 480

unsigned char fb[WIDTH*HEIGHT*DEPTH];
fb[bgra + DEPTH*(x + WIDTH*y)] = ...
// bgra = 0, 1, 2, or 3
```

Drawing

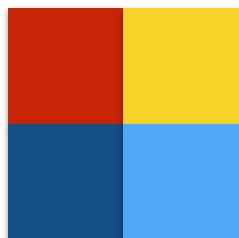
code/clear

Can interpret as an array of unsigned ints

```
#define WIDTH 2  
#define HEIGHT 2
```

```
unsigned int fb[WIDTH*HEIGHT];
```

```
fb[0][0] = 0xffff0000; // x=0, y=0  
fb[1][0] = 0xffffff00; // x=1, y=0  
fb[0][1] = 0xff0000ff; // x=0, y=1  
fb[1][1] = 0xff00ffff; // x=1, y=1
```



00	00	ff	ff	00	ff	ff	ff	ff	00	00	ff	ff	ff	00	ff
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

red

yellow

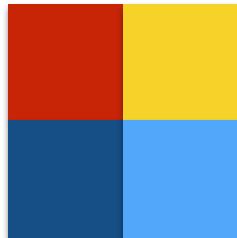
blue

cyan

Casting to a 2D Array

```
unsigned int (*fb)[WIDTH] =  
    (unsigned int (*)[WIDTH])fb.framebuffer;
```

```
// fb[y][x] = color  
fb[0][0] = 0xffff0000;  
fb[0][1] = 0xffffffff00;  
fb[1][0] = 0xff0000ff;  
fb[1][1] = 0xff00ffff;
```



00	00	ff	ff	00	ff	ff	ff	ff	00	00	ff	ff	ff	00	ff
red				yellow				blue				cyan			

What is unsigned fb[WIDTH], (*fb)[WIDTH]? cdecl.org

Demo

code/grid

Single- and Double- Buffering

Single Buffer

Drawing directly into the framebuffer lets you see the graphics as it is drawn.

Normally we just show the final drawing. Don't see the drawing process.

Seeing the drawing form can be good for debugging!

Virtual Width

Virtual Height



Drawing can cause flickering

code/singlebuffer

Double Buffer

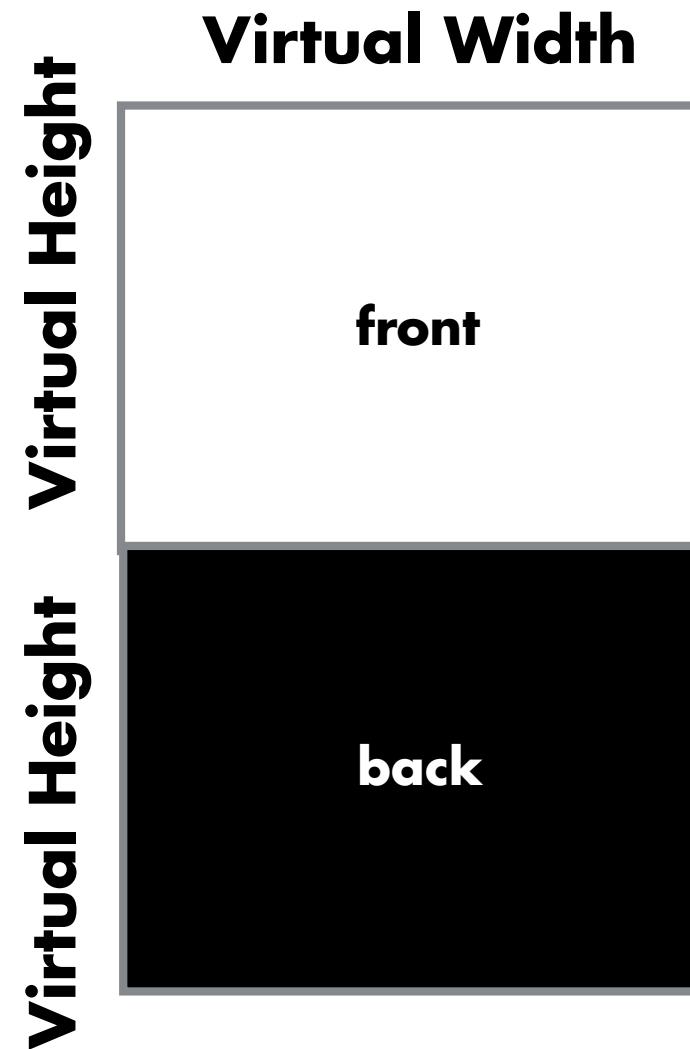
Solution: Two framebuffers

- **Front buffer**
- **Back buffer**

Display the "front"-buffer

Draw into the "back"-buffer

Can't see drawing process



Swap Buffer

**When you are done
drawing.**

**Swap front and back
buffers.**



code / doublebuffer

Display Top Buffer

```
x_offset = 0;  
y_offset = 0;
```

2 * Virtual Height

Virtual Width



Display Bottom Buffer

```
x_offset = 0;  
y_offset = vheight;
```



or



Which arrangement is better?

2 * Virtual Height

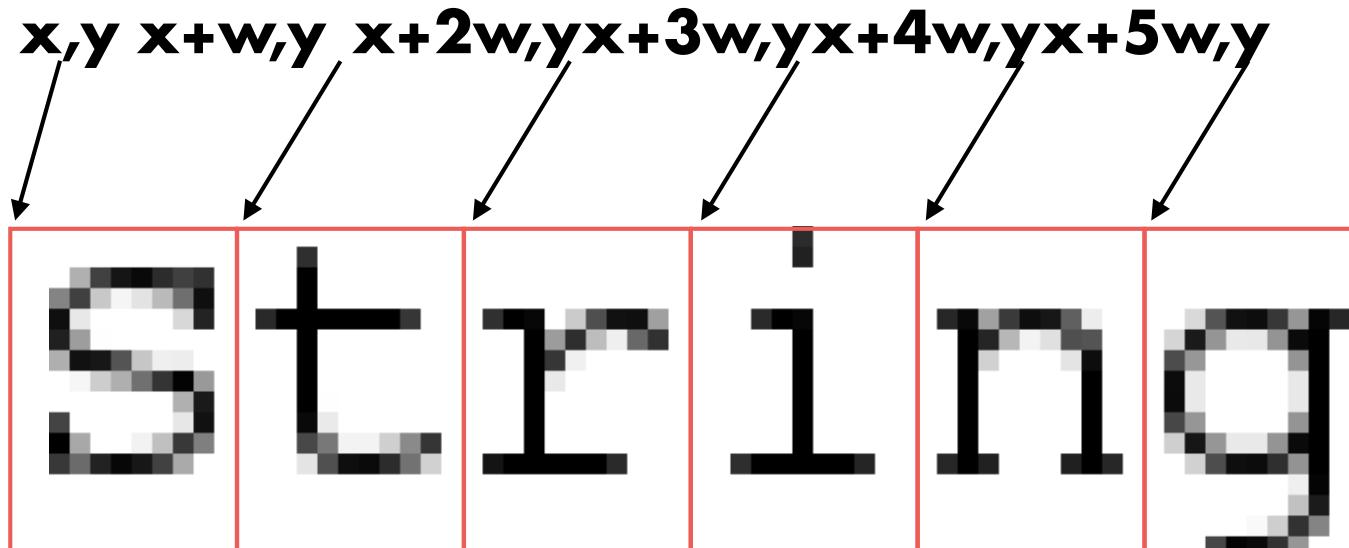
Virtual Width



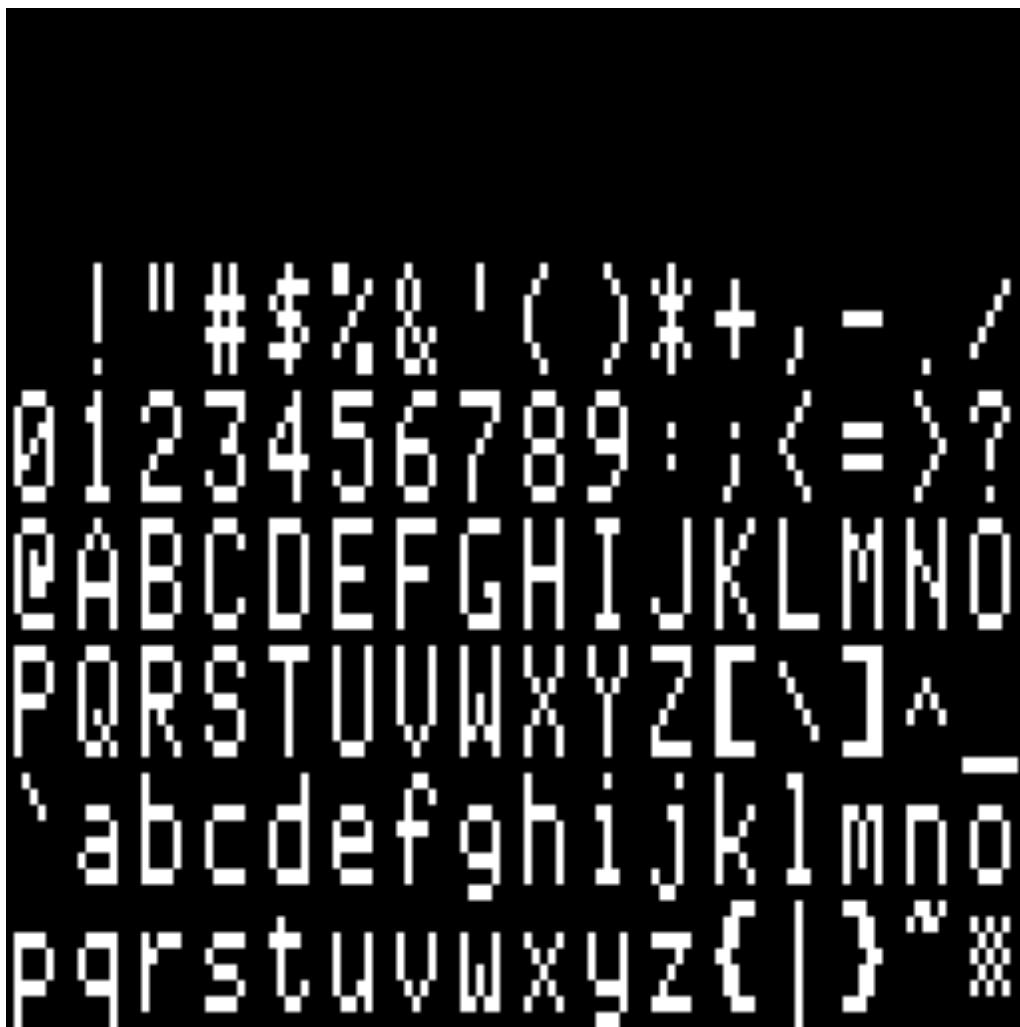
Drawing Text

Fonts: monospaced vs. proportional

Font a set of "glyphs"



Font a set of "glyphs"



Apple II Font (7x8)

Framebuffer Overview

GPU continuously refreshes the display by sending the pixels in the framebuffer out the HDMI port

The size of the image sent to the monitor is called the physical size

The size of the framebuffer image in memory is called the virtual size

The CPU and GPU share the memory, and hence the framebuffer

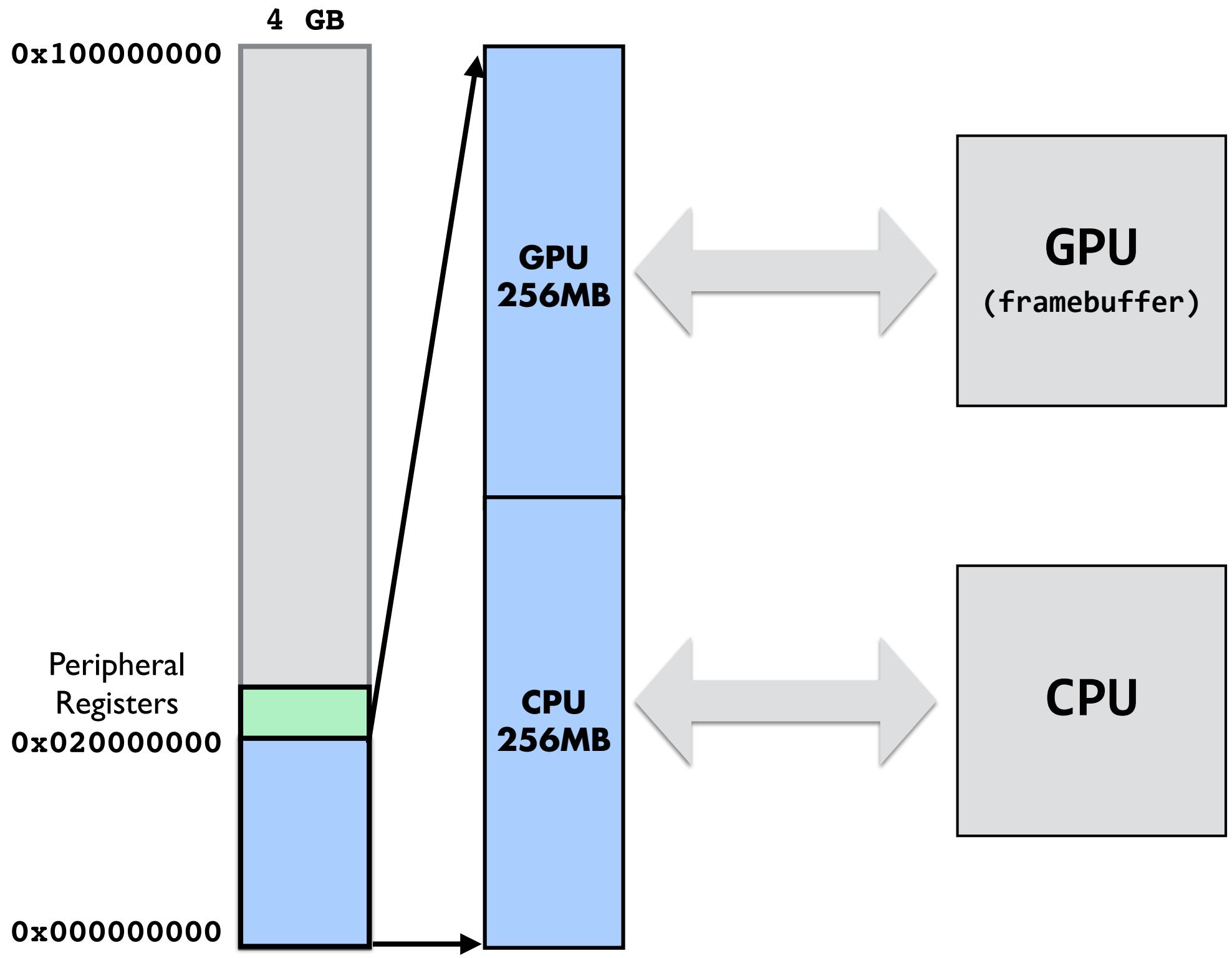
The CPU and GPU exchange messages using a mailbox

Put It All Together

graphical shell

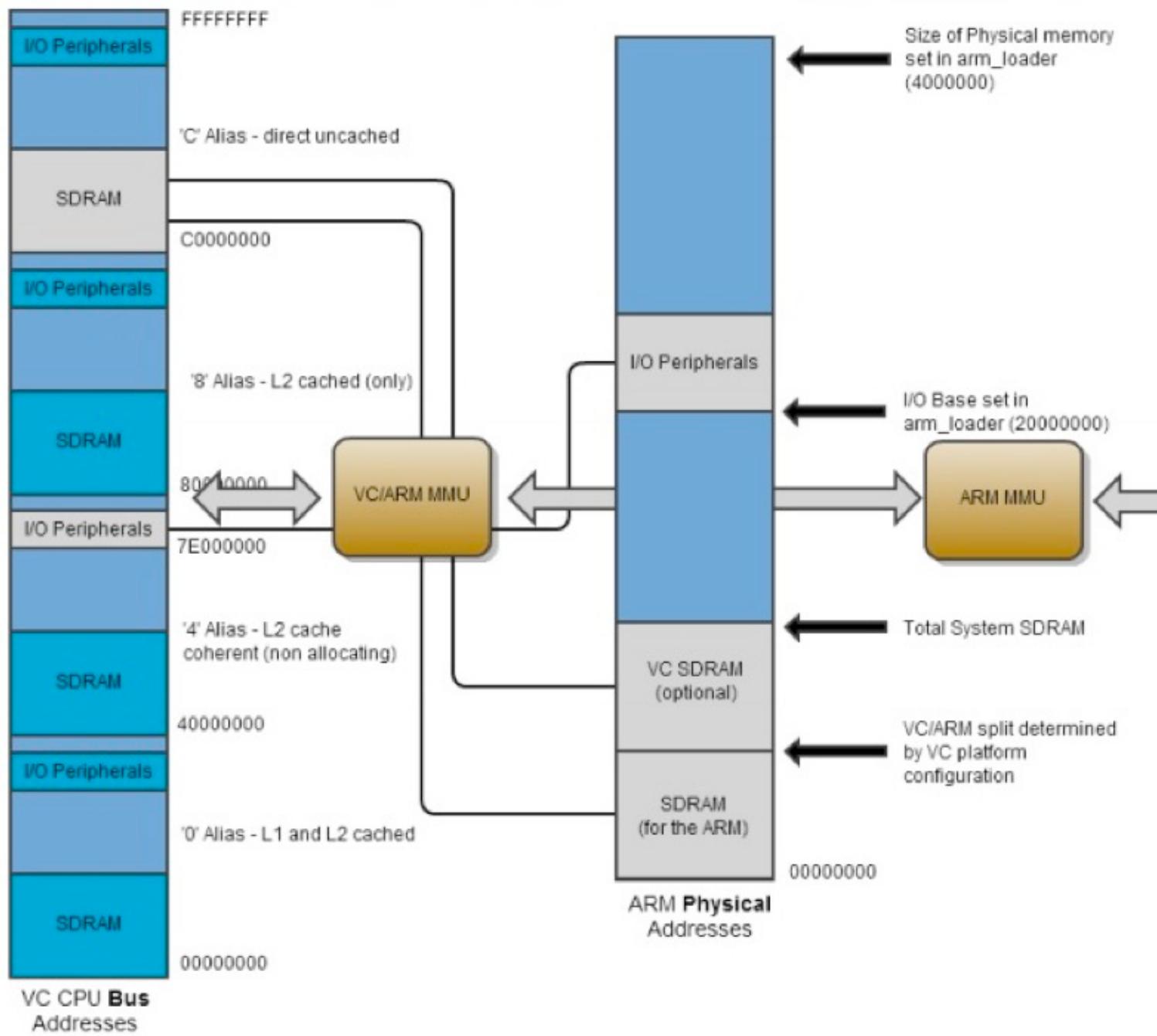


Mailbox





BCM2835 ARM Peripherals



Coordinating CPU+GPU

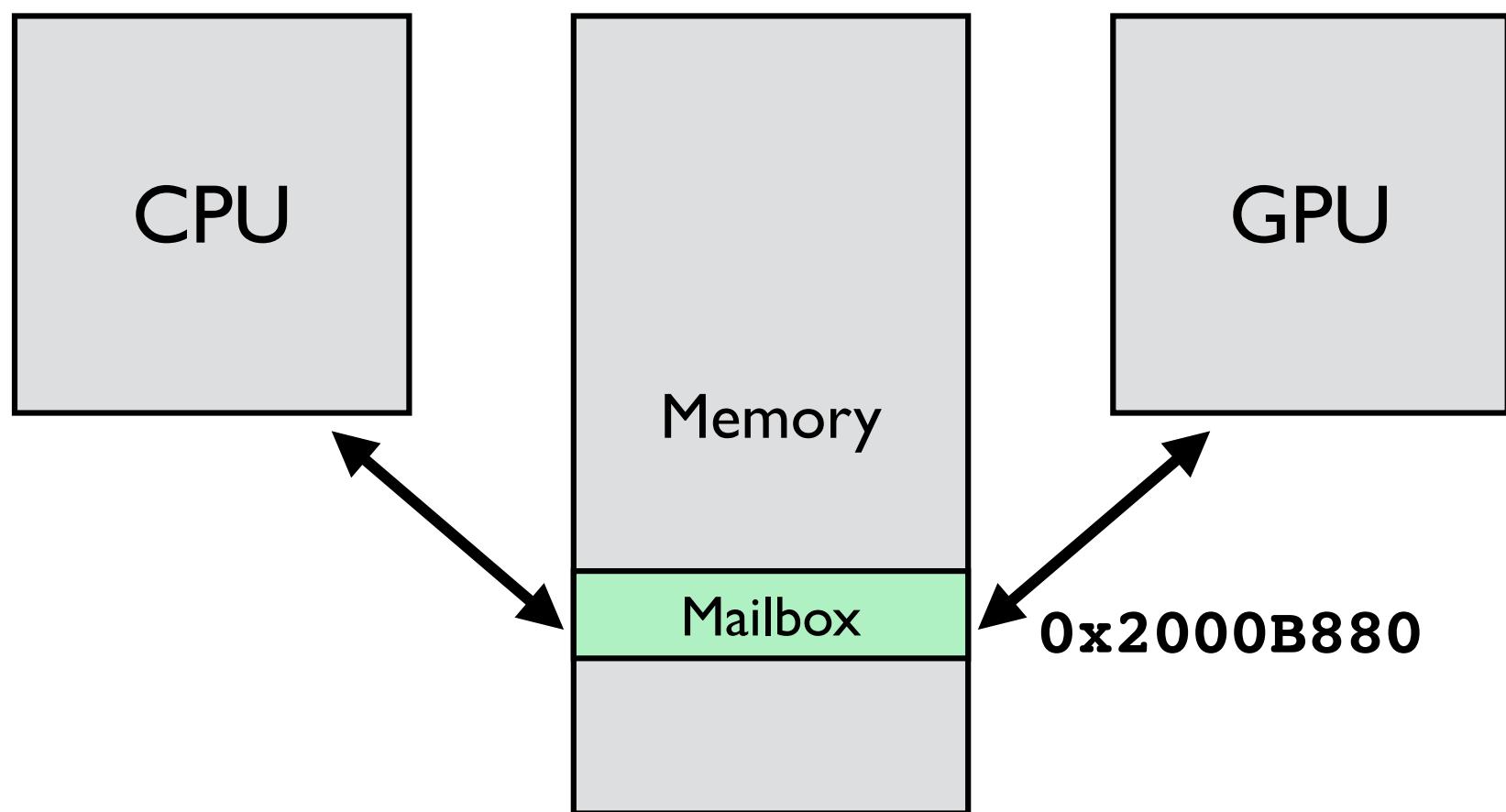
CPU and GPU need to communicate

- **CPU code wants to set/change screen settings**
- **GPU send to the CPU a pointer to the fb_config_t**

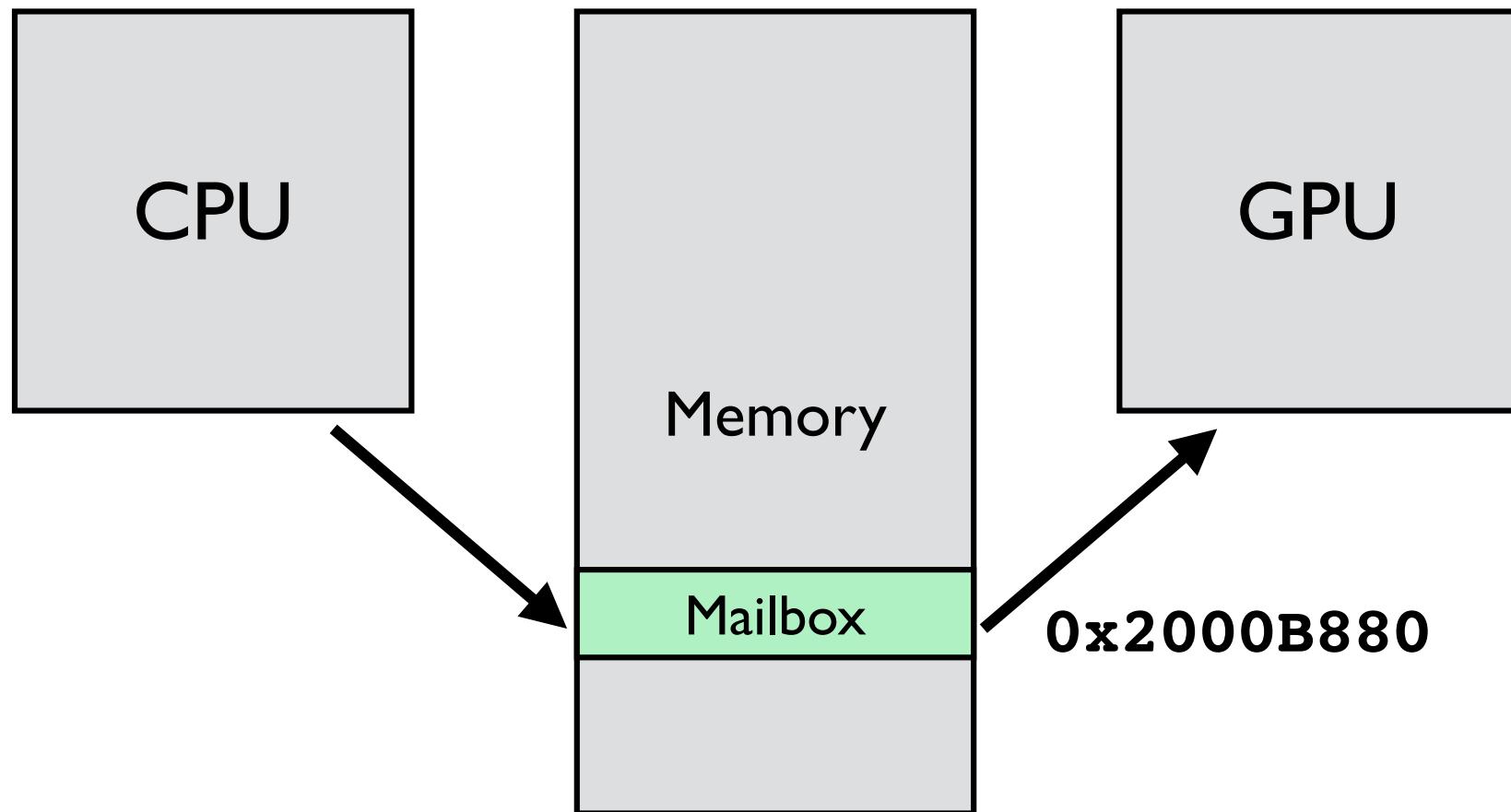
Danger: reading incomplete/partial data

- **They are two processors reading and writing to memory, C compiler has no knowledge that this is happening**
- **Need a simple handshake that depends on a single bit**
 - **"I've set this bit, which means I have sent some data to you. You can write it now."**
 - **"I've cleared the bit, which means I've done writing the data. You can read it now"**

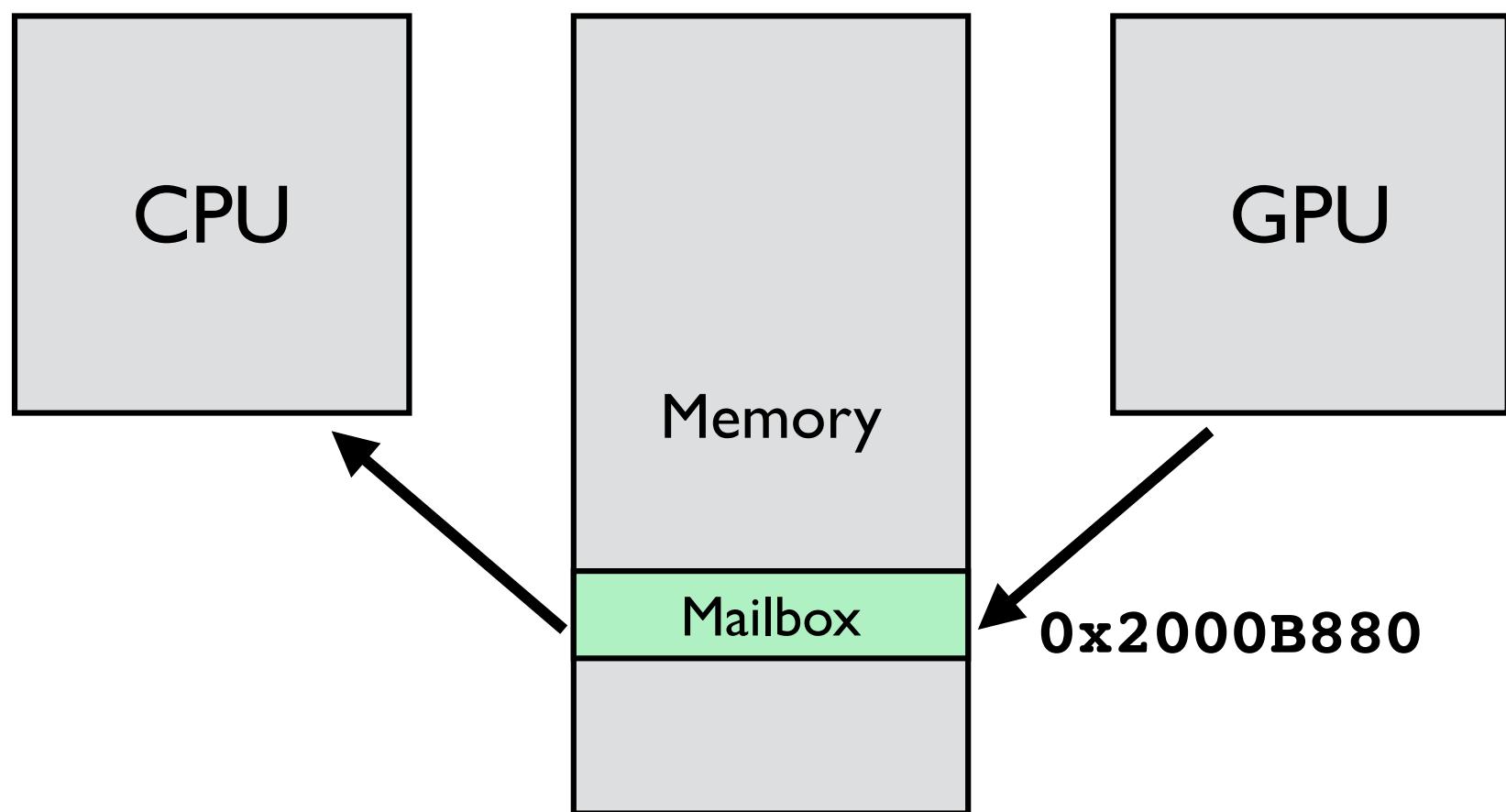
Mailbox



CPU "Mails" Message to GPU



GPU Mails Reply to CPU



Mailbox Format

Register	Offset	R/W	Use
Read	0x00	R	Destructively read value
Peek	0x10	R	Read without removing data
Sender	0x14	R	Sender ID (bottom 2 bits)
Status	0x18	R	Status bits
Configuration	0x1C	RW	Configuration bits
Write	0x20	W	Address to write data (GPU addr)

F | E

undocumented/unused?

F = Full

E = Empty

```
#define MAILBOX_BASE    0x2000B880
#define MAILBOX_FULL     (1<<31)
#define MAILBOX_EMPTY    (1<<30)
#define GPU_NOCACHE 0x40000000

typedef struct {
    unsigned int read;
    unsigned int padding[3]; // note padding to skip 3 words
    unsigned int peek;
    unsigned int sender;
    unsigned int status;
    unsigned int configuration;
    unsigned int write;
} mailbox_t;
static volatile mailbox_t *mailbox = (volatile mailbox_t *)MAILBOX_BASE;

void mailbox_write(unsigned int channel, unsigned int addr) {
    // mailbox has a maximum of 16 channels
    if (channel >= MAILBOX_MAXCHANNEL)
        return;
    // addr must be a multiple of 16
    if (addr & 0xF)
        return;
    // wait until mailbox is not full ...
    while (mailbox->status & MAILBOX_FULL) ;
    // set GPU_NOCACHE bit so that the GPU does not cache the memory
    addr |= GPU_NOCACHE;
    // addr is a multiple of 16, so the low 4 bits are zeros
    // 4-bit channel number is stuffed into those low bits
    mailbox->write = addr + channel;
}
```