VIII. *Video Programming*

Jalal Kawash

Outline

1. 2D arrays
2. Memory-mapped I/O
3. Mailboxes
4. Frame buffer architecture
5. Drawing basic shapes

Section 1

2D Arrays

Section 1 Objectives

At the end of this section you will

1. Map 2D arrays to 1D arrays
2. Work with row-major and column-major organizations

2D Arrays

- Two-dimensional arrays must be mapped onto 1D memory
  - Can use *row-major ordering*
    - Store each element of row 0, then row 1, etc.
    - Used in most high-level languages, including C, C++
  - Or *column-major ordering*
    - Store each element of column 0, then column 1, etc.
    - Used by FORTRAN

Example

- Logical arrangement:

10	20	30
40	50	60

- Mapping in row-major order:

10	20	30	40	50	60

- Mapping in column-major order:

10	40	20	50	30	60

Indexing 2D Arrays

- The indices for a multidimensional array must be converted into an *offset*
  - Added to array starting address

- For a 2D array *list*[*1..n*][*1..m*], row-major order, *list*[*i*][*j*] is:

  $$((m * i) + j) \cdot E_{\text{size}}$$

  - E_{size} is the element (cell) size in bytes

Indexing 2D Arrays

- Example C Code declaration of

```
char array[2][3]
array[1][2] maps to an offset of
(3 * 1 + 2) * 1 = 5
```

$$((m * i) + j) \cdot E_{size}$$

0,0	0,1	0,2
1,0	1,1	1,2

Logical view

0	1	2
3	4	5

Physical view

Indexing 2D Arrays

- Example Java Code declaration of

```
char[2][3] array
array[1][2] maps to an offset of
(3 * 1 + 2) * 2 = 10
```

$$((m * i) + j) \cdot E_{size}$$

0,0	0,1	0,2
1,0	1,1	1,2

Logical view

0	2	4
6	8	10

Physical view

Effective Addresses

- A 1D array cell is accessed by:
- *base* + *offset*
- *base* is the starting address of the array
- *offset* = (cell number) * (cell size in bytes)

Examples

- `myArray: .word 10, 20, 30, 40, 50`
- Cell containing 10 has address
  - myArray + 0*4
- Cell containing 20 has address
  - myArray + 1*4
- Cell containing 30 has address
  - myArray + 2*4

Examples

- `myArray:  .byte  10, 20, 30, 40, 50`
- Cell containing 10 has address
  - myArray + 0*1
- Cell containing 20 has address
  - myArray + 1*1
- Cell containing 30 has address
  - myArray + 2*1

Section 2

Memory-Mapped I/O

Section 2 Objectives

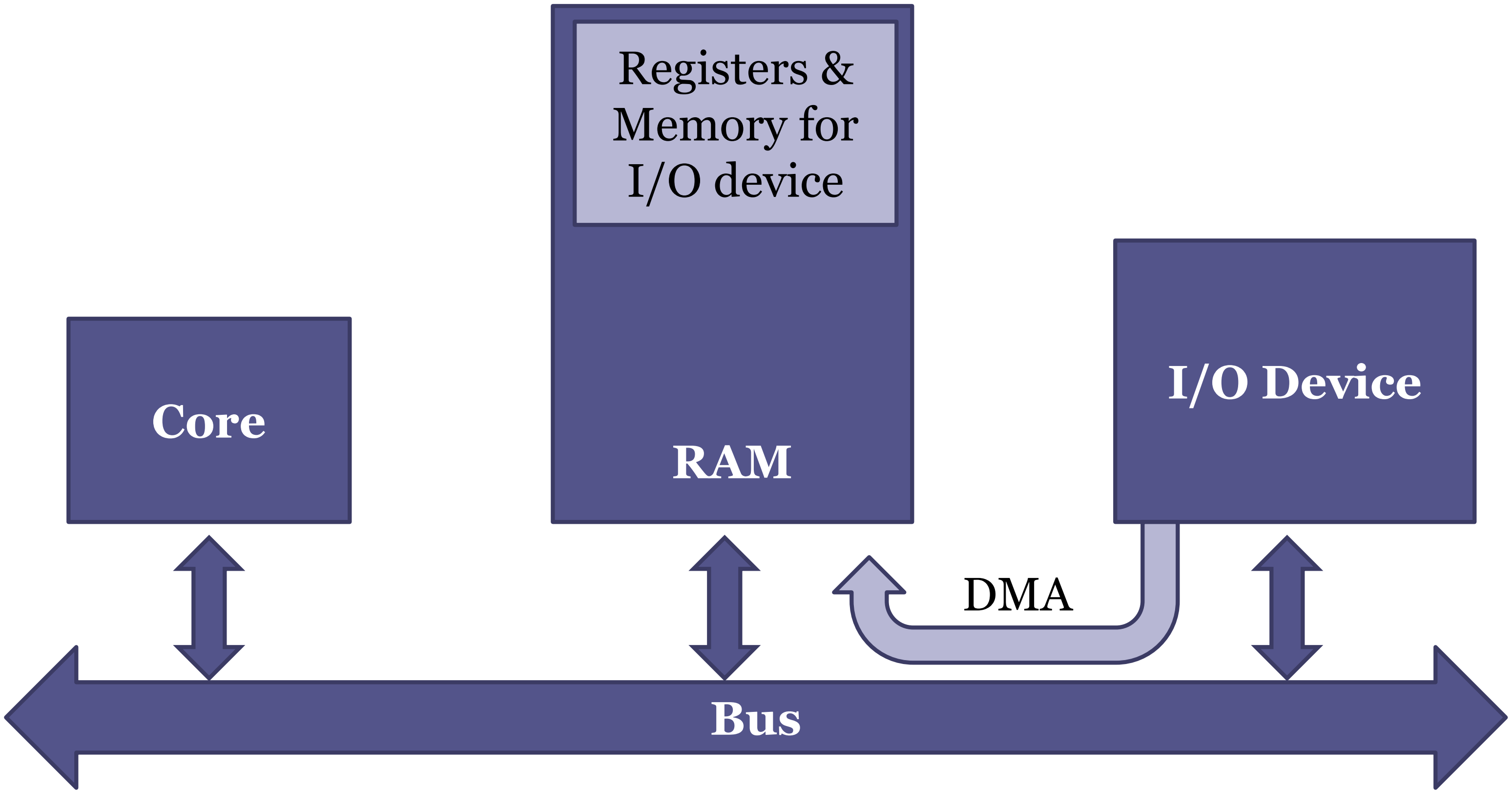
At the end of this section you will

1. Understand memory-mapped I/O
2. Know how frame-buffers are organized

Memory-Mapped I/O

- A method for performing I/O
- Registers and memory of the device are mapped to address values
- A Core can communicate with the device by writing and reading memory
  - Using loads and stores
- To write a register in the I/O device, the core writes to an address in memory
- To read a register, the core reads from memory

Memory-Mapped I/O



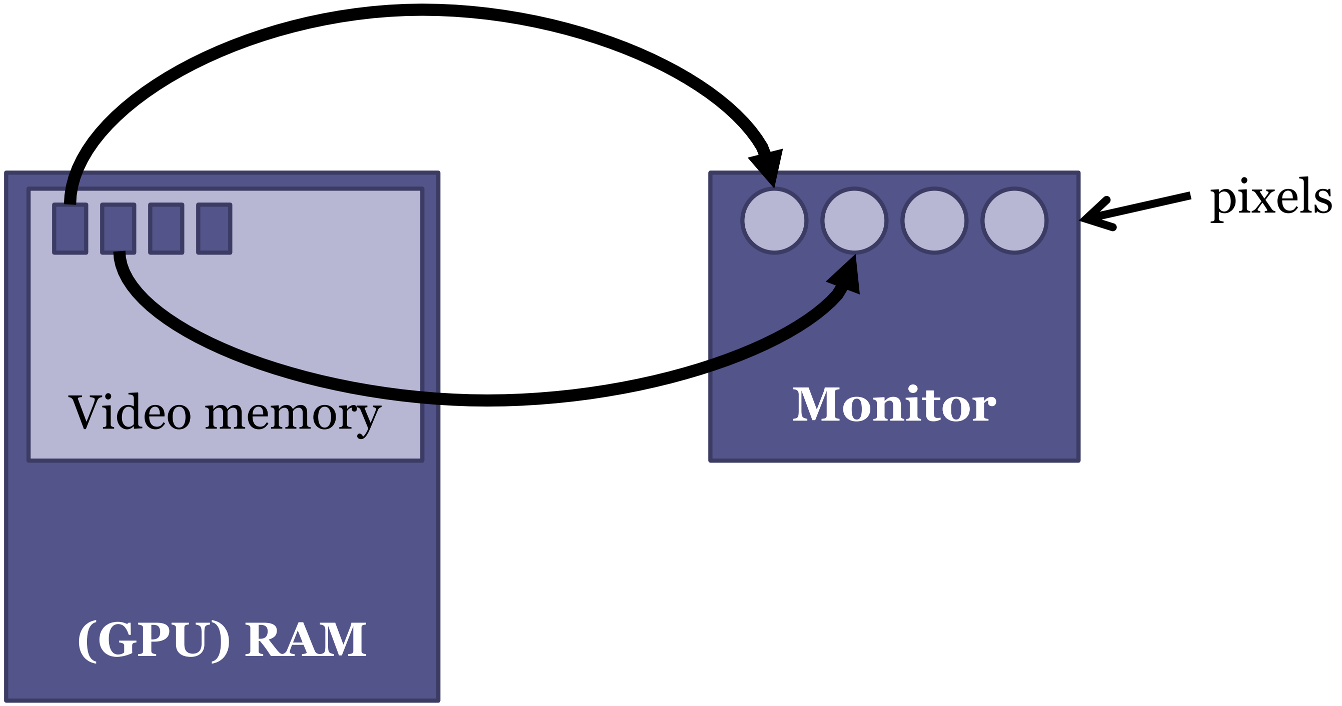
Memory-Mapped I/O

- Easier to program
  - I/O does not need special instructions
- Takes memory
  - Not of a great concern given how cheap RAM is
- Not always implemented using DMA
  - GPIO registers are not really in memory!

Frame-Buffers

- A frame-buffer is a memory mapped arrangement for monitors
- Each pixel is mapped to a memory address
- To write a pixel, the core simply sets the corresponding value in main memory
- Can be implemented in a separate Video RAM or in a reserved section of RAM
- Maps 2D monitor to 1D memory (row-major)

Frame-Buffer



Memory-mapped I/O

- Use ordinary memory instructions to read from or write to the device

- Example

```
MOV  r0, 0x0A0      // address of pixel
STR  r1, [r0]       // set pixel
```

Section 3

Mailboxes

Based on: https://github.com/raspberrypi/firmware/wiki

Section 3 Objectives

At the end of this section you will

1. Understand the mailbox architecture
2. Use its registers to read and write
3. Use the mailbox to initialize a frame buffer

Mailboxes

- Allow communication between the ARM core and the Video Core (VC) in RPi
- Mailbox 0 defines 10 channels
  - Power management, frame buffer, touch screen etc …
  - Frame buffer is channel 1
- Mailbox 1 also exists
  - Not clear what channels it has

Mailbox 0 Registers

Register Name	Offset	Purpose
Peek	0x10	Read but do not delete info
Read	0x00	LS 4 bits = channel number; remaining 28 is data read from mailbox; removes data
Write	0x20	Same as read but for writing
Status	0x18	Tells if mailbox full or empty
Sender	0x14	
Config	0x1C	

Base address for registers is: 0x3F00B880

Status Register

- Bit 31 (MSB) is set when mailbox is full
  - Cannot write to mailbox
  - Use mask 0x80000000 to check it
- Bit 30 is set when mailbox is empty
  - Cannot read from mailbox
  - Use mask 0x40000000

Read & *Write* Registers

Bit 4 to 31 = data bits	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**

Channel number

Reading a Mailbox (for channel *n*)

1. Wait until bit 30 in *status* is clear (=0)
2. data = *read*
3. If data[0..3] != *n*, goto step 1
   (LS 4 bits are the channel number)
4. Return data[4..31]

Writing a Mailbox (for channel *n*)

1. data[4..31] = value to write
2. data[0..3] = *n*
3. Wait until bit 31 in *status* is clear (=0)
4. *write* = data

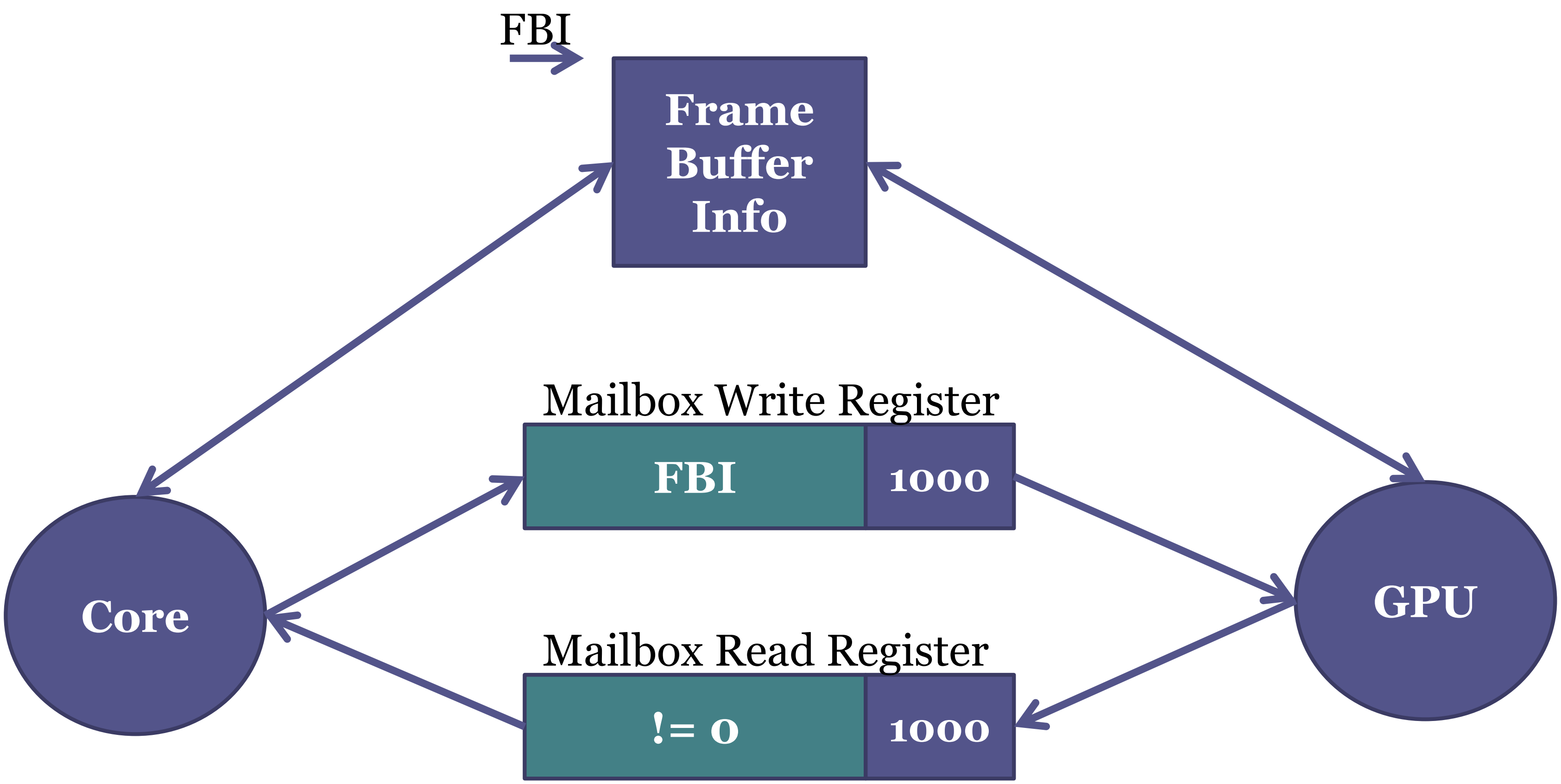
Mailbox 0 Channels

Channel	Description
0	Power Management
1	**Frame Buffer**
2	Virtual UART
3	VCHIQ Interface
4	LEDs Interface
5	Buttons Interface
6	Touch Screen Interface
7	N/A
8	Property Tags (to GPU)
9	Property Tags (from GPU)

Frame Buffer Interface

- Send (write) a message via mailbox 0, channel 8 to initialize frame buffer by GPU
- The MS 28 bits of message contain the address of a structure (frameBufferInfo) that contains information about the frame buffer
  - LS 4 bits contain 8, the channel #
- GPU responds with a non-zero message, setting appropriate values in frameBufferInfo

Initializing the Frame Buffer



The frameBufferInfo Structure

- Created in VRAM
  - Can be part of RAM, but in a dedicated part
- Must be 16-byte aligned
  - Only the MS 28 bits of the address can be passed through the mailbox

- Contains tags
  - A tag contains a value buffer

Tag Structure

- Word 1: Tag identifier
  - Unique identifier for tag
- Word 2: Value buffer size in bytes
- Word 3: MSB is request/response bit, remaining bits are value length in bytes
  - 0 = request; 1 = response
- Remaining words (buffer)

Example Tag

```
.int 0x00048003 // Tag ID
//Set Physical Display width and height
.int 8          //size of buffer
.int 8          //length of value
.int 1024       //horizontal resolution
.int 768        //vertical resolution
```

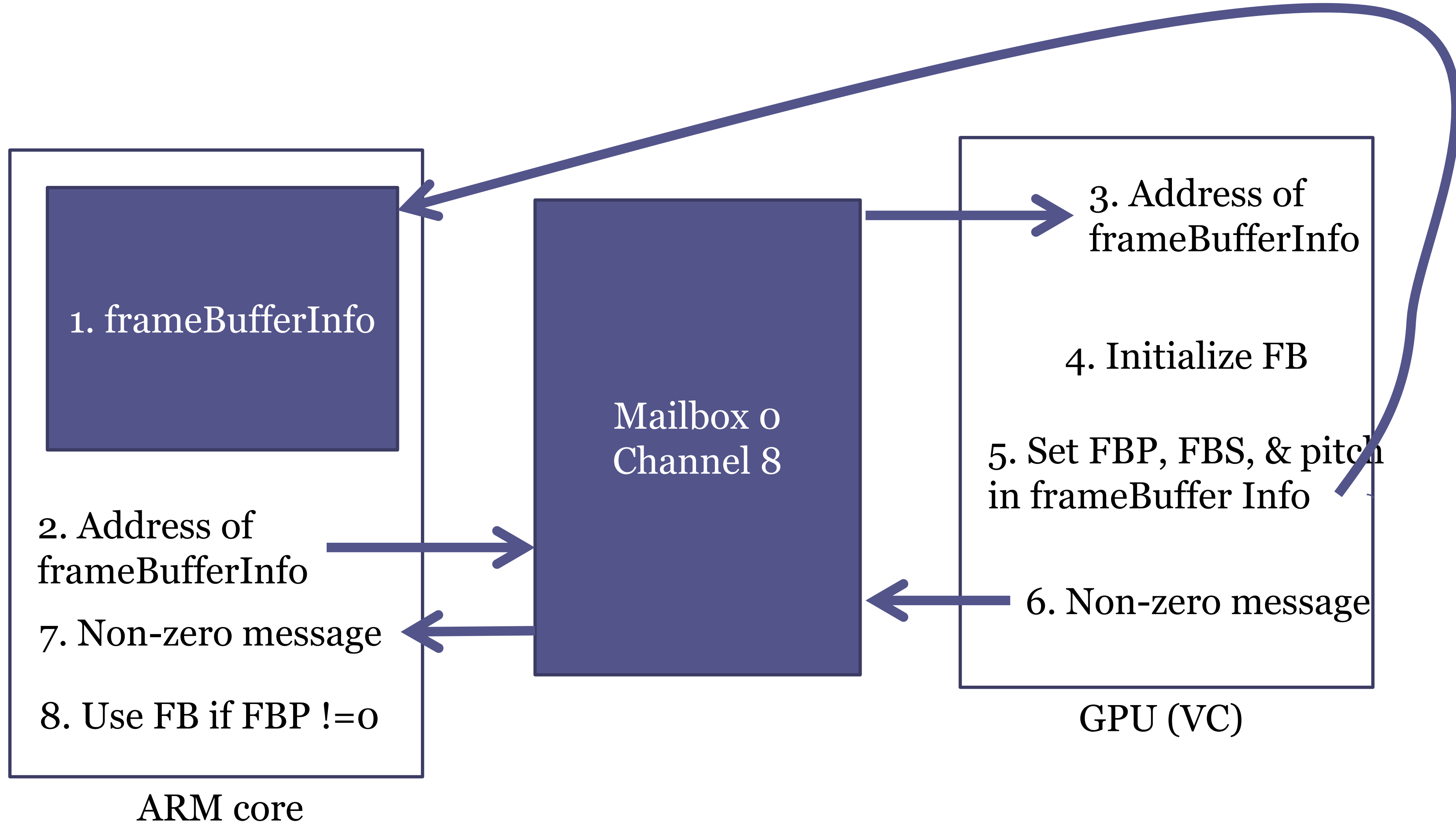
Tag

Buffer value

MSB = 0 => request

```
.align 4
frameBufferInfo:
        .int    22 * 4          //Buffer size in bytes
        .int    0               //Indicates a request to GPU
        .int    0x00048003      //Set Physical Display width and height
        .int    8               //size of buffer
        .int    8               //length of value
        .int    1024            //horizontal resolution
        .int    768             //vertical resolution

        .int    0x00048004      //Set Virtual Display width and height
        .int    8               //size of buffer
        .int    8               //length of value
        .int    1024            //same as physical display width and height
        .int    768

        .int    0x00048005      //Set bits per pixel
        .int    4               //size of value buffer
        .int    4               //length of value
        .int    16              //bits per pixel value

        .int    0x00040001      //Allocate framebuffer
        .int    8               //size of value buffer
        .int    8               //length of value
FrameBuffer:
        .int    0               //value will be set to framebuffer pointer
        .int    0               //value will be set to framebuffer size

        .int    0               //end tag, indicates the end of the buffer
```

Frame Buffer Interface



Frame Buffer Interface (again)

1. Initialize frameBufferInfo array
2. msg[0..3] = 8; msg[4..31] = frameBufferInfo
   1. msg[30] = 1 (forces GPU to not cache FBI!)
3. Write msg to mailbox(0)
4. Read msg from mailbox(0)
5. If msg == 0, return (FB cannot be initialized)
6. Else
   1. FBP = [frameBuffer]
   2. FBS = [frameBuffer]

Virtual Frame Buffer

- This is only used for debugging the frame buffer subsystem
- It is a frame buffer that is created in RAM, rather than VRAM
- Writing to the virtual FB will not be visible
  - Useful for systems that do not have a graphics display (embedded systems)

Section 4

Frame Buffer Architecture

Section 4 Objectives

At the end of this section you will

1. Work with the video frame buffer
2. Understand color resolutions
3. Calculate the effective address of a pixel

Video frame buffer

Frame Buffer Architecture

- Frame buffer:
- An array in memory (GPU RAM), where each element represents a pixel on the display
- The entire array represents one complete frame (screen)
- The 2D frame is mapped to the 1D buffer
    - e.g. VGA is 640 x 480
        - Maps to a 1D array with 307,200 elements

Example Resolutions

Standard	Size	Aspect Ratio
VGA	640 x 480	4:3
SVGA	800 x 600	4:3
XGA	1024 x 768	4:3
SXGA	1280 x 1024	5:4
UXGA	1600 x 1200	4:3

Frame Buffer Architecture

- Each element in the frame buffer represents the pixel's color
  - Size in bits per pixel (bpp) Gives 2^{bbp} colors

- Row major organization

Common Graphics Formats

- Color depths:
- 1-bit
  - monochrome
- 4-bit
  - 16 fixed colors
- 8-bit indexed
  - Choice of 256 colors from a palette
- 16-bit *highcolor*
  - 65,536 colors
  - 5 bits for R, B; 6 bits for G
- 24-bit *truecolor*
  - 16,777,216 colors
  - 8 bits each for R, G, and B
- 32-bit RGBA
  - Like truecolor

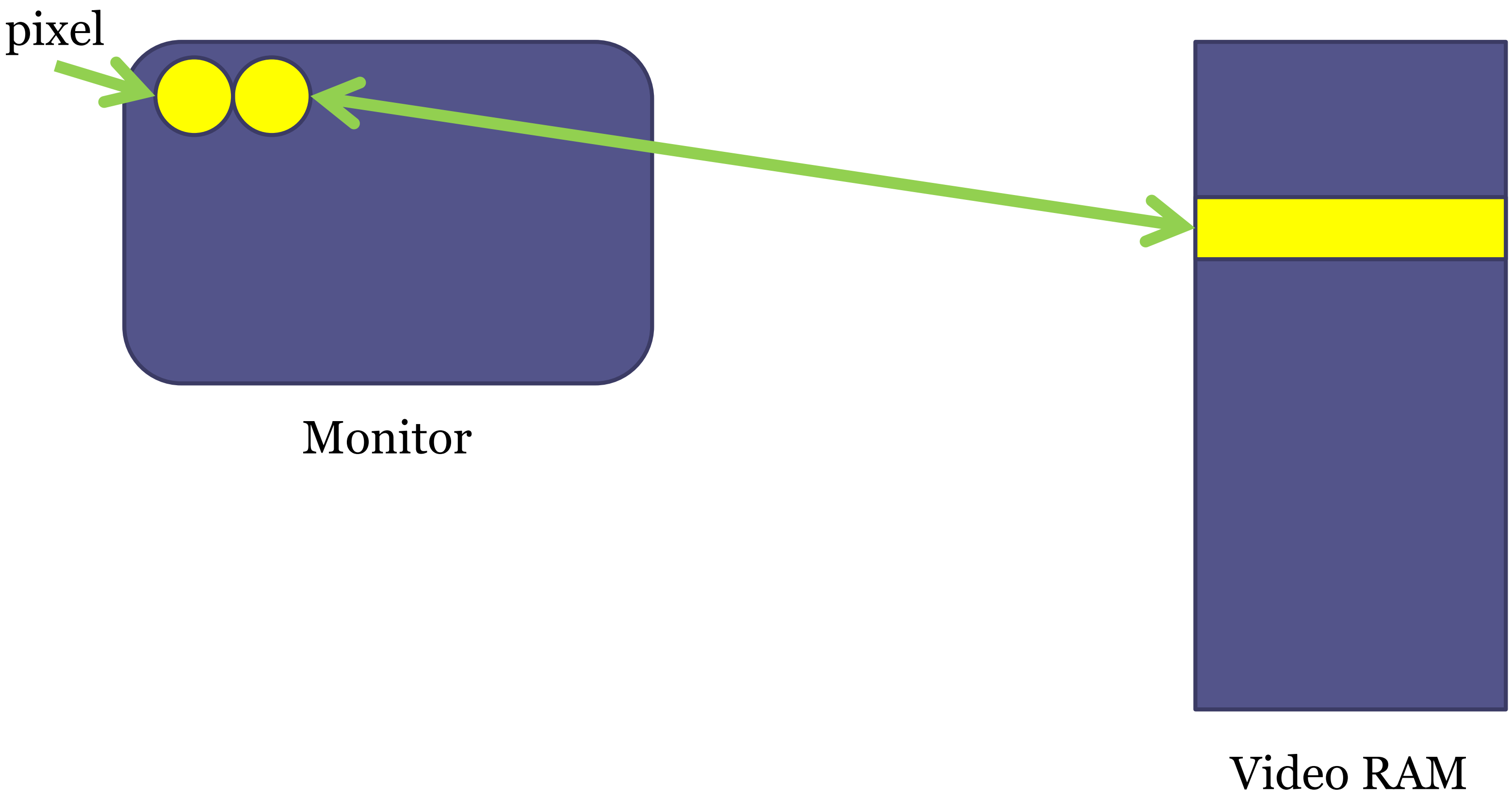
Higher Resolution

- Higher resolution and/or color depth requires a larger frame buffer
  - Practical now since RAM is cheap

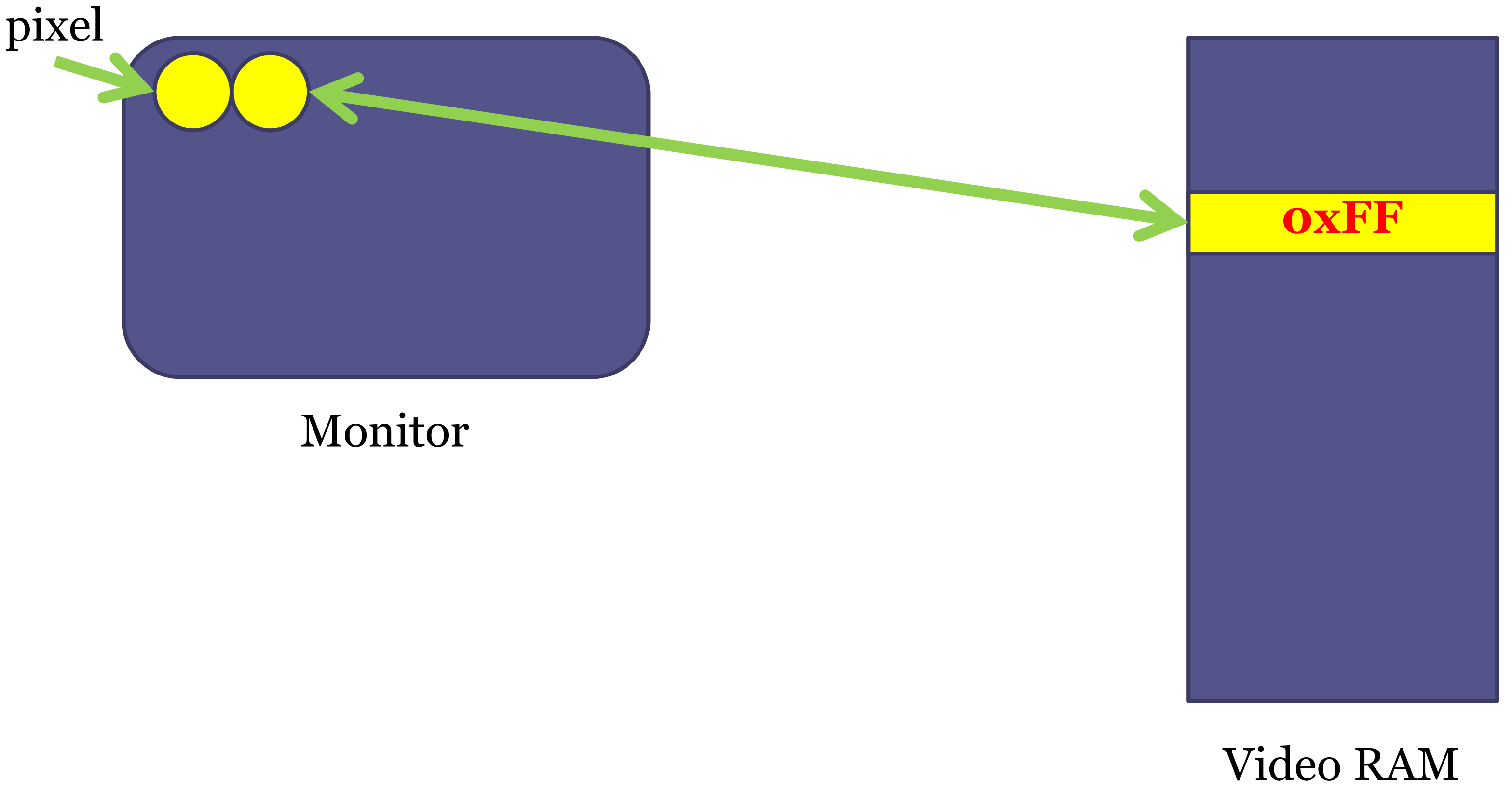
Drawing a Pixel

- To draw a single pixel, set its value in the frame buffer
  - Must map from 2D logical space to 1D physical RAM
    - Must know width and height for particular resolution
  - Use formula: element offset = (y * width) + x
    - x, y are pixel's coordinates
    - Origin is upper left-hand corner
    - x range: 0 to width-1
    - y range: 0 to height-1
  - physical offset = element offset * element size in bytes

Drawing a Pixel



Drawing a Pixel

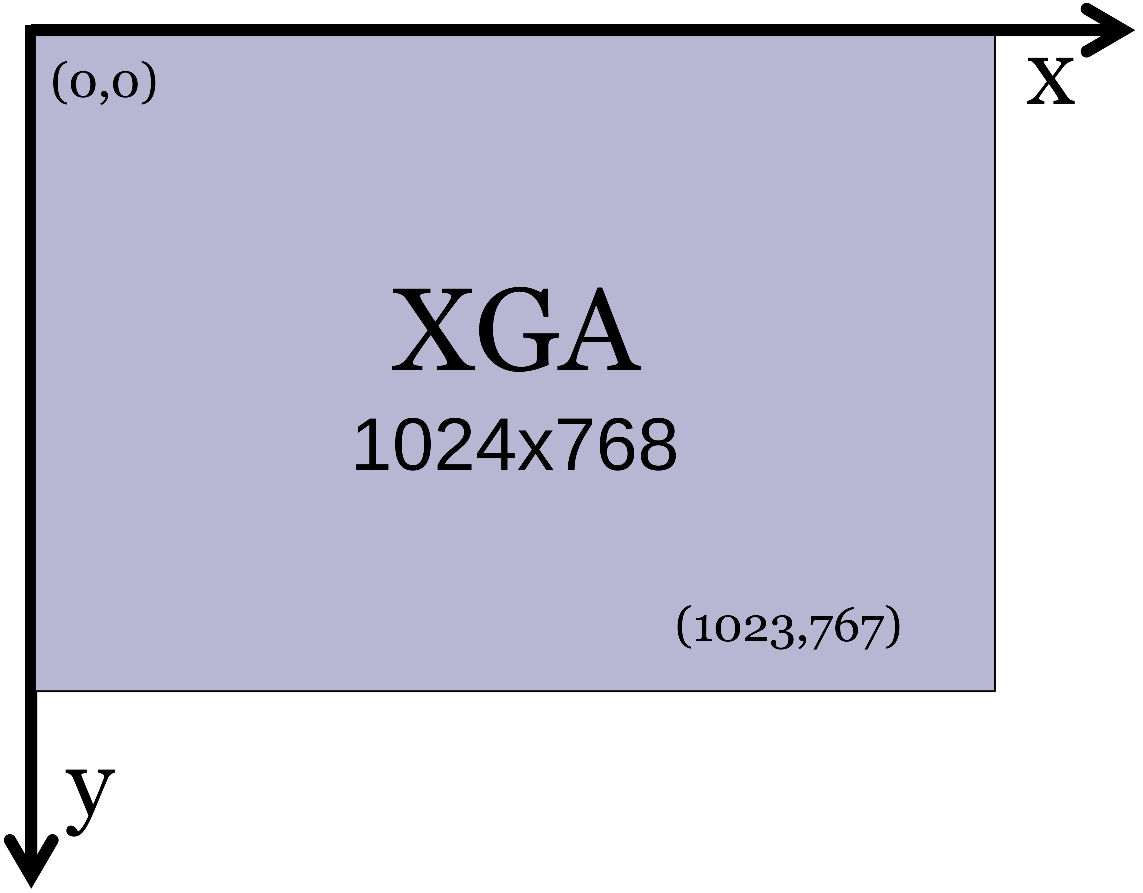


Color Codes

- Refer to: http://www.nthelp.com/colorcodes.htm

Hex Code	Color	Hex Code	Color	Hex Code	Color
0xFFFFFF		0xCCFFFF		0x99FFFF	
0xFFFFCC		0xCCFFCC		0x99FFCC	
0xFFFF99		0xCCFF99		0x99FF99	
0xFFFF66		0xCCFF66		0x99FF66	
0xFFFF33		0xCCFF33		0x99FF33	
0xFFFF00		0xCCFF00		0x99FF00	
0xFFCCFF		0xCCCCFF		0x99CCFF	
0xFFCCCC		0xCCCCCC		0x99CCCC	
0xFFCC99		0xCCCC99		0x99CC99	
0xFFCC66		0xCCCC66		0x99CC66	
0xFFCC33		0xCCCC33		0x99CC33	
0xFFCC00		0xCCCC00		0x99CC00	

Frame Buffer Coordinates



Examples

- E.g. XGA (1 byte per pixel, 8-bit indexed)
- element offset = (y * width) + x
  - Pixel (0,0)
    - [(0 * 1024) + 0] * 1 = 0
  - Pixel (1023, 767) (lower right-hand corner)
    - [(767 * 1024) + 1023] * 1 = 786431

Section 5

Drawing Basic Shapes

Section 5 Objectives

At the end of this section you will

1. Draw one pixel in the frame buffer
2. Learn how to draw basic lines

Setting a Pixel

```
/* Draw Pixel
 *  r0 - x
 *  r1 - y
 */
.globl DrawPixel
DrawPixel:
    px       .req    r0
    py       .req    r1
    addr     .req    r2

    ldr      addr,   =farmeBufferInfo
    ldr      addr,   [addr]

    height   .req    r3 // read FB height from  FBI
    ldr      height, [addr, #20]
    sub      height, #1
    cmp      py,     height
    movhi    pc,     lr
    .unreq   height
```

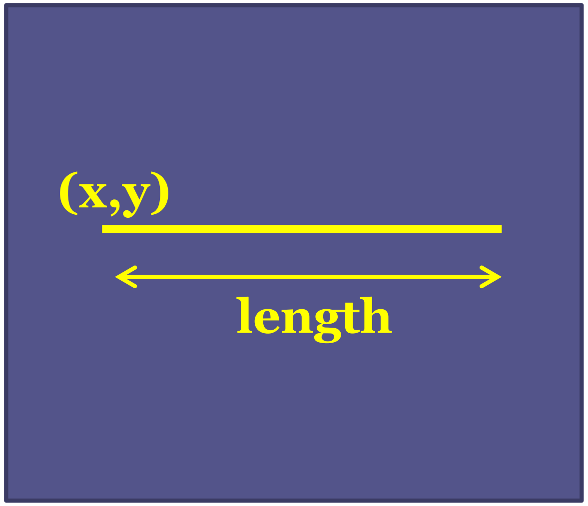
```
width   .req    r3 // read FB width
ldr     width,  [addr, #16]
sub     width,  #1
cmp     px,     width
movhi   pc,     lr

ldr     addr,   =FrameBufferPointer // start of framebuffer
ldr     addr,   [addr]

add     width,  #1

mla     px,     py, width, px    // px = (py * width) + px
.unreq  width
.unreq  py

add     addr,   px, lsl #1 // offset of pixel; 2bytes/pixel
.unreq  px

fore    .req    r3
ldr     fore,   =foreColour // color can be a big constant
ldrh    fore,   [fore]
```

```
strh    fore,    [addr] // set the pixel color
.unreq  fore
.unreq  addr

mov     pc,      lr
```

Straight Lines

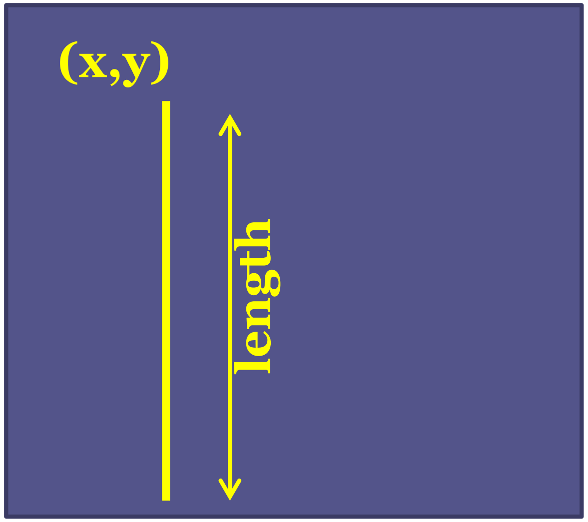
Horizontal Line Pseudo-Code

```
drawHorizontalLine(x, y, length)
        MOV r2, #0 // pixels drawn
drawLoop:
        CMP r2, length
        BGE done
        setPixel(x,y, color)
        ADD r2, #1
        ADD x, #1 // move to the next pixel on x-axis
        B drawLoop
done:
        MOV pc, lr
//does not check if x, y, and length are within the frame
```



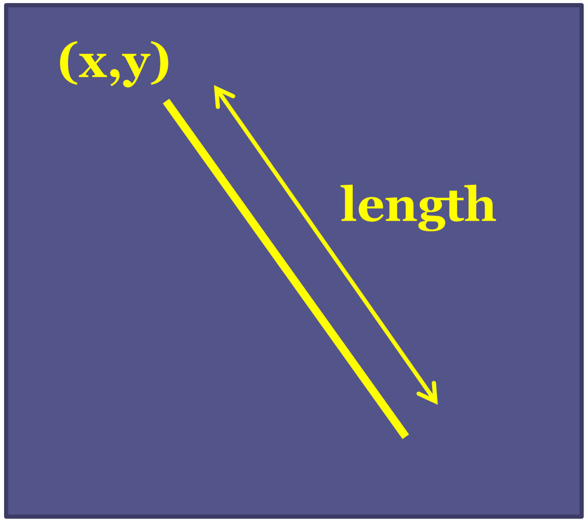
Vertical Line Pseudo-Code

```
drawVerticalLine(x, y, length)
        MOV r2, #0 // pixels drawn
drawLoop:
        CMP r2, length
        BGE done
        setPixel(x,y, color)
        ADD r2, #1
        ADD x, width // move to the next pixel on y-axis
        B drawLoop
done:
        MOV pc, lr
//does not check if x, y, and length are within the frame
```



Integer-Slope Line Pseudo-Code

```
drawIntSlopeLine(x, y, length, slope)
        MOV r2, #0 // pixels drawn
drawLoop:
        CMP r2, length
        BGE done
        setPixel(x,y, color)
        ADD r2, #1
        ADD x, width  +slope
        B drawLoop
done:
        MOV pc, lr
//does not check if x, y, and length are within the frame
```



General Straight Lines

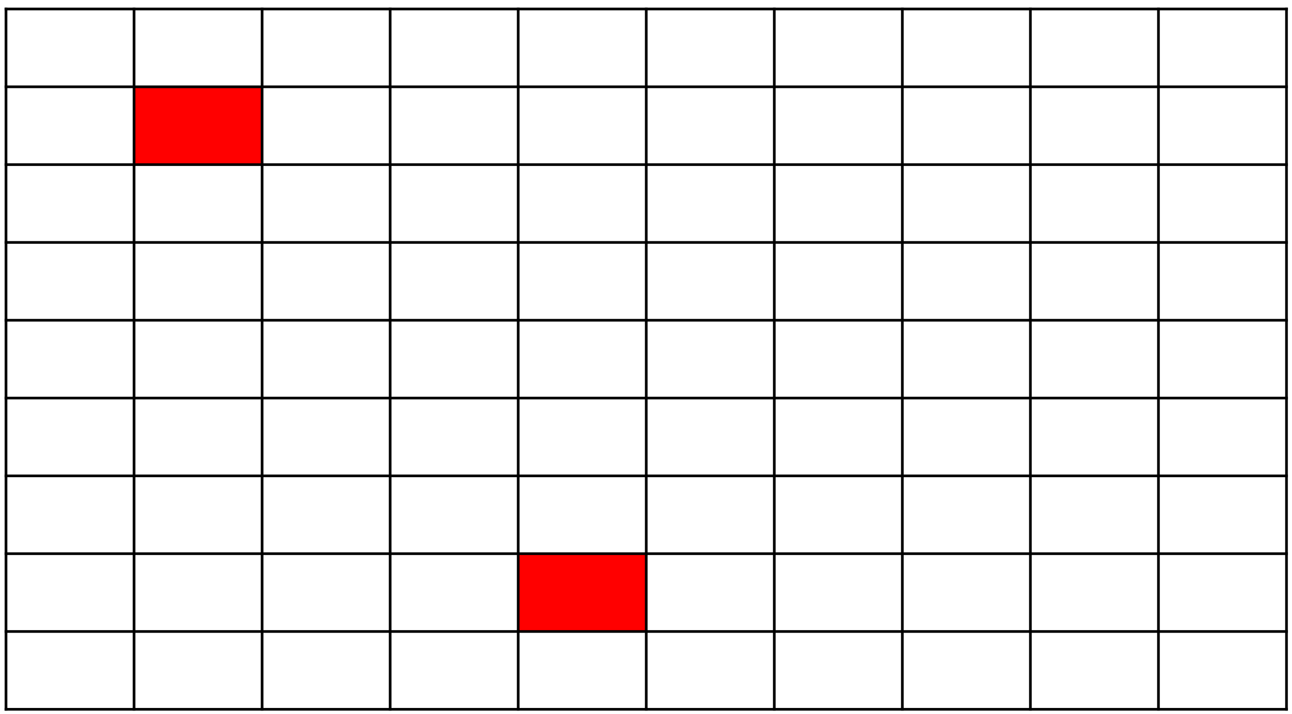
- A line is defined by two points (x1,y1) and (x2,y2)
- Slope = (y2-y1)/(x2-x1) = a
- Equation: (y-y1)/(x-x1) = (y2-y1)/(x2-x1)
- Or: y = a(x – x1) + y1

- Note if x1 = x2, slope is infinite (vertical line)

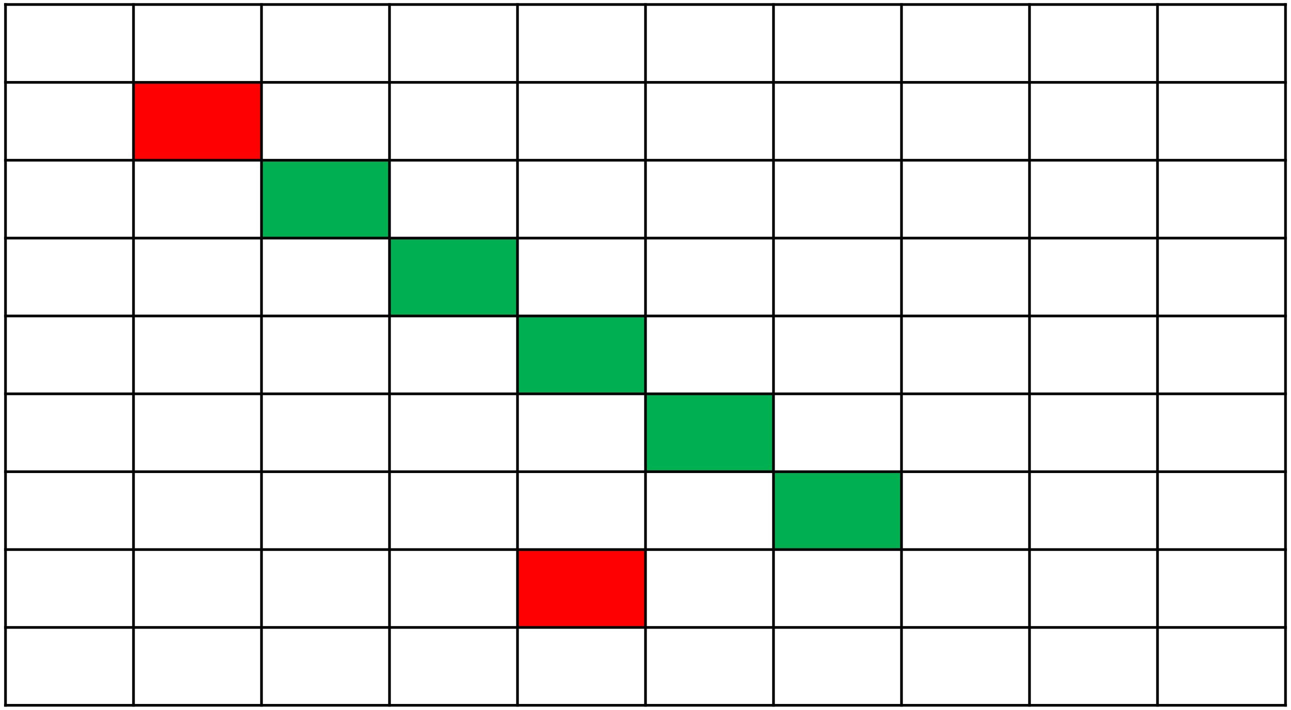
Slopes

- Since we are using integer division only, points must be chosen carefully
- Slope must be an integer

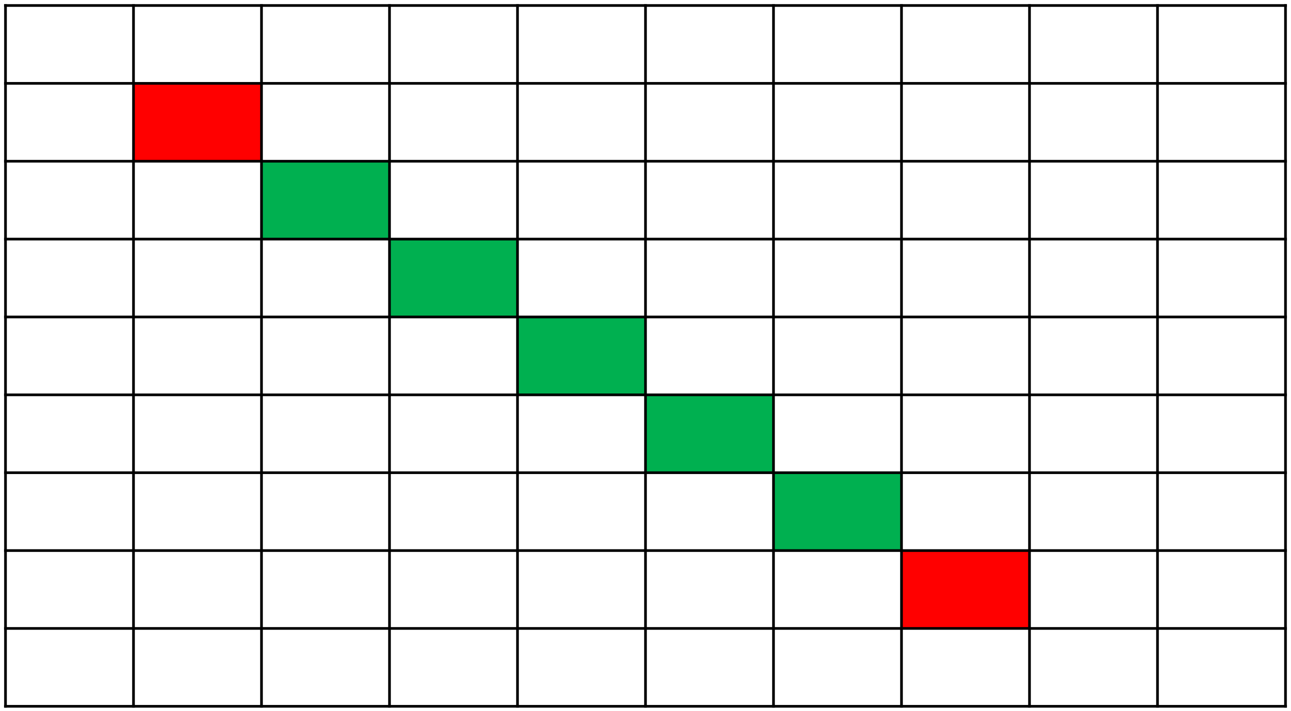
Slopes



Non-integer slope



Integer slope



Slopes

- There are approximation algorithms for straight-lines
  - Beyond the scope of this course (graphics course)
  - Can use them if you use floating-point instructions

Line approximation