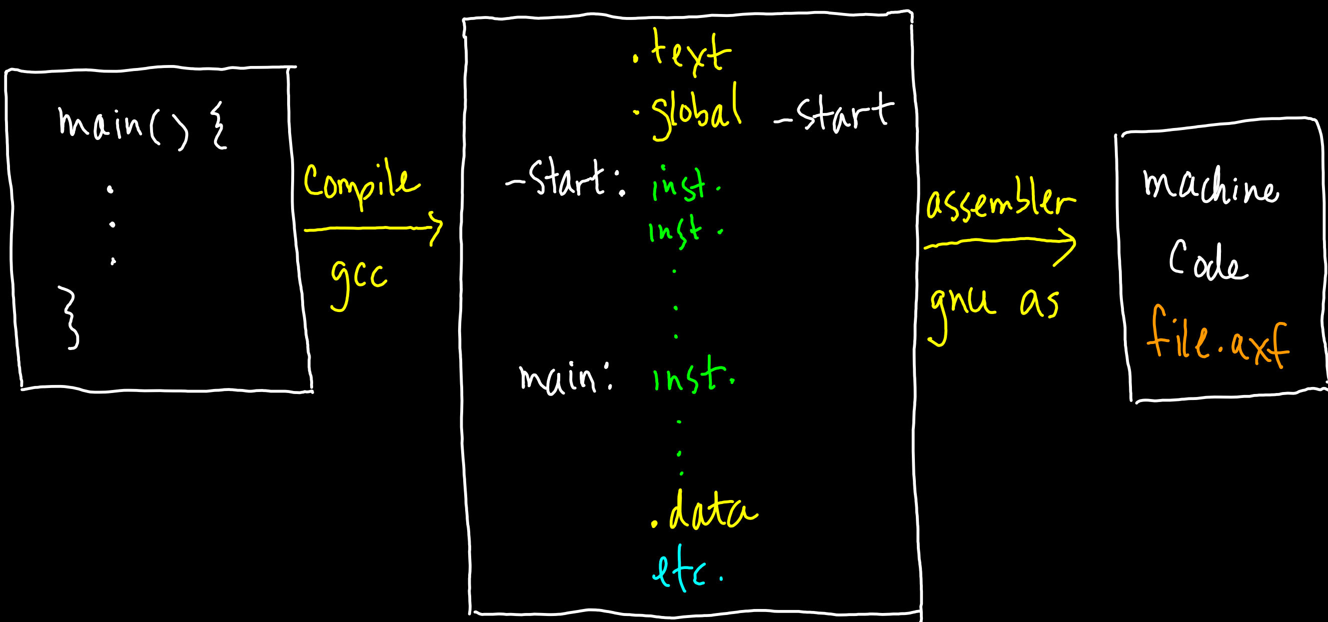
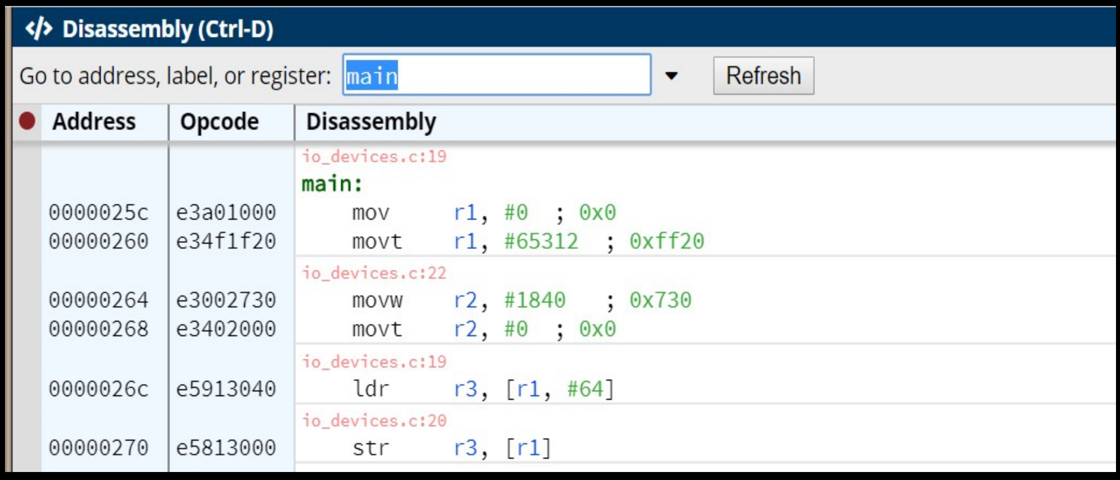
Using C Code

- when you compile C code, the compiler first generates Assembly code, and then machine code.



Notes: If you have multiple C source-code files, then each one is compiled separately to create an object file (*.o). These files are combined by the linker to create file.axf.

_start is not main. Instead, it is the common startup code sequence. It initializes sp, and then performs various steps needed to start up a C program (sets initial variable values, sets uninitialized global variables to 0, ...). In CPUlator, or the Monitor Program, you can search for main to find the start of your code:



Disassembly (Ctrl-D)

Go to address, label, or register: main Refresh

Address	Opcode	Disassembly
		io_devices.c:19
		main:
0000025c	e3a01000	mov r1, #0 ; 0x0
00000260	e34f1f20	movt r1, #65312 ; 0xff20
		io_devices.c:22
00000264	e3002730	movw r2, #1840 ; 0x730
00000268	e3402000	movt r2, #0 ; 0x0
		io_devices.c:19
0000026c	e5913040	ldr r3, [r1, #64]
		io_devices.c:20
00000270	e5813000	str r3, [r1]

) ⊛

ff200040 (SW)

(LEDR)

Reading / Writing to I/O Devices

- in Assembly code to read SW, and write LEDR:

```
        LDR   R1, =0xFF200000    ) (*) or   mov   R1, #0
                                            movt  R1, #0xFF20
Loop:   LDR   R3, [R1, #0x40]   // read 0xFF200040 (SW)
        STR   R3, [R1]          // write to LEDR
        B     Loop
```

- in C code:

```c
volatile int *LEDR_ptr = 0xFF200000;
volatile int *SW_ptr = 0xFF200040;
int value;
while (1) {
    value = *SW_ptr;        ) (*)
    *LEDR_ptr = value;
}
```

(*) using *ptr is called pointer dereferencing. This is how we read/write specific addresses.

volatile: ensures that the variable will always be accessed by using its address. Without volatile the SW_ptr location might only be read once (i.e., moved outside the while loop). You should always use volatile for I/O pointers.

- write C code to display SW on LEDR and HEX3-0.

SW: $\underbrace{\uparrow\uparrow}_{seg7[3]}\ \underbrace{\downarrow\downarrow\downarrow\uparrow}_{seg7[1]}\ \underbrace{\uparrow\uparrow\uparrow\uparrow}_{seg7[15]}$ (31F)

00000000	01001111	00000110	01110001	0xFF200020 HEX3-0
	3	1	F	

char seg7[] = {0x3F, 0x06, 0x5B, 0x4F, ..., 0x71}; // 0,1,...,F HEX patterns

```c
int main() {
    int value;
    volatile int *LEDR_ptr = 0XFF200000;
    volatile int *SW_ptr = 0XFF200040;
    volatile int *HEX3_0_ptr = 0XFF200020;
    while (1) {
        value = *SW_ptr; // read switches
        *LEDR_ptr = value;
        *HEX3_0_ptr = seg7[value & 0xF] |
                      seg7[value >> 4 & 0xF] << 8 |
                      seg7[value >> 8] << 16;
    }
```

* *

- when compiled we get:

```
0000025c <main>:
25c: e3a01000    mov   r1, #0
260: e34f1f20    movt  r1, #0xff20        // address of LEDR

264: e3002730    movw  r2, #0x730         // address of seg7[] array
268: e3402000    movt  r2, #0

26c: e5913040    ldr   r3, [r1, #0x40]    // read sw
270: e5813000    str   r3, [r1]           // write 2 LEDR

274: e7d2c443    ldrb  r12, [r2, r3, asr #8]
278: e203000f    and   r0, r3, #15
27c: e7d20000    ldrb  r0, [r2, r0]
280: e180080c    orr   r0, r0, r12, lsl #16
284: e7e33253    ubfx  r3, r3, #4, #4
288: e7d23003    ldrb  r3, [r2, r3]
28c: e1803403    orr   r3, r0, r3, lsl #8
290: e5813020    str   r3, [r1, #0x20]    // write to HEX3_0

294: eafffff4    b  26c <main+0x10>
```

while

ubfx r3, r3, #4, #4

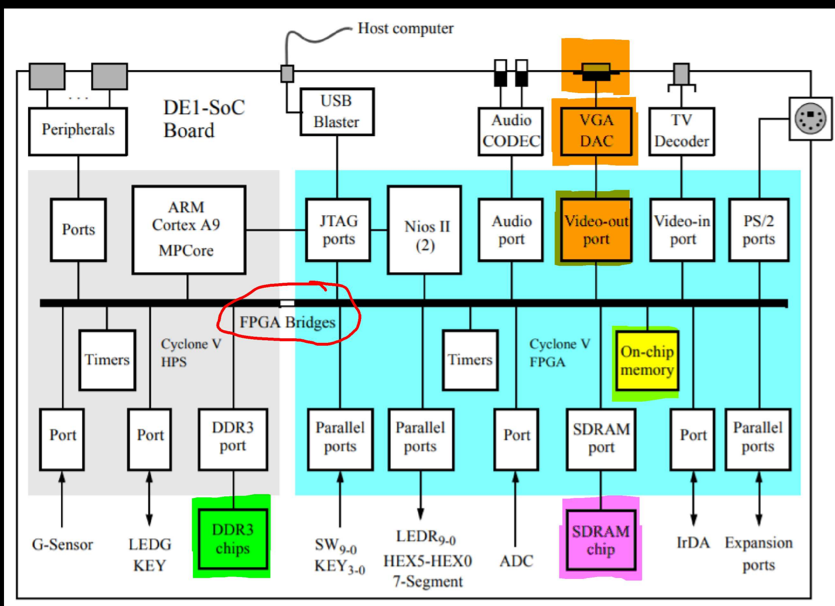
is equivalent to:

lsr r3, #4

* and r3, #0xF

Lab 7 Prep: Using a VGA display

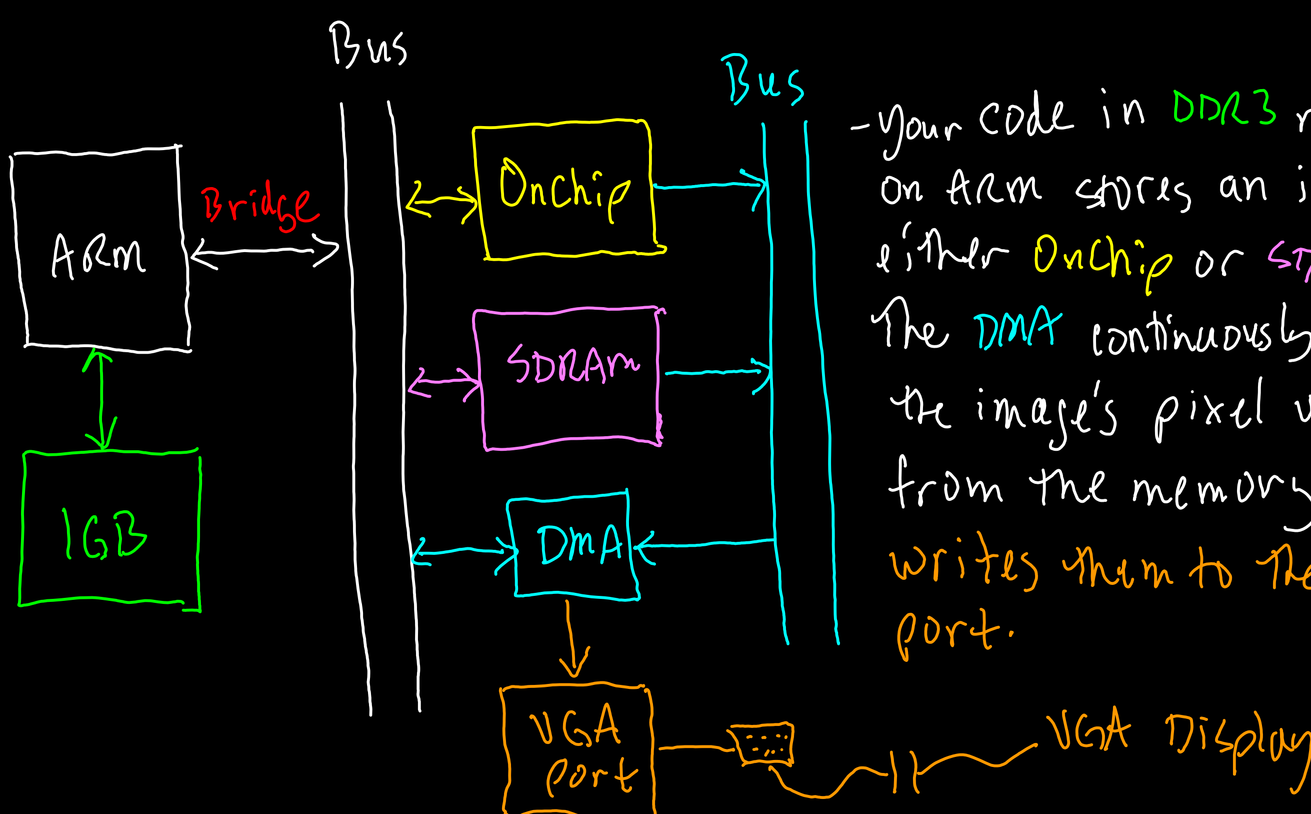
- The DE1-SoC computer has a VGA output port. It continuously reads pixel colors from a memory (called the Pixel Buffer), and displays them on a VGA screen. The image in memory is 320x240 pixels. We use two different memory locations for the pixel buffer: Onchip memory (in the FPGA), and SDRAM



- your code & data are in the DDR3 (1GB)
- But the video-out VGA port uses either the Onchip memory (256kB) or the SDRAM (64MB)

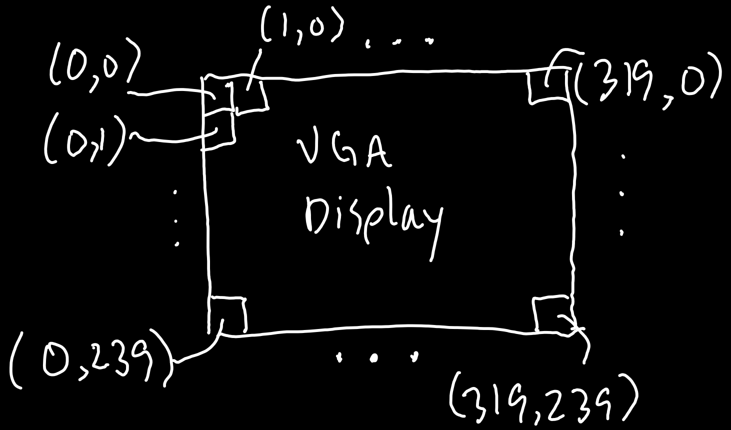
Onchip: C8000000 – C803FFFF

SDRAM: C0000000 – C3FFFFFF



- Your code in DDR3 running on ARM stores an image in either Onchip or SDRAM. The DMA continuously reads the image's pixel values from the memory and writes them to the VGA port.

DMA: Direct Memory Access Controller



- Each pixel takes 2 bytes:

15–11	10–5	4–0
red	green	blue

- The address of pixel (x,y) is

31–17	17–10	9–1	0
Base	y	x	0

∴ address pixel = Base + (y<<10) + (x<<1);

- The default pixel buffer location is in the OnChip memory:

(0,0) : C8000000

(1,0) : C8000000 + (0<<10) + (1<<1) = C8000002

(0,1) : C8000000 + (1<<10) + 0 = C8000400

(1,1) : C8000402

⋮

(319,239): C8000000 + (239<<10) + (319<<1) = C803BE7E

Example: to make the bottom-right pixel full green, write the 16-bit color 00000111111 00000 = $(07E0)_{16}$ to C803BE7E

Part 1: drawing a line

- we cannot draw an exact line; we can only color pixels close to the line

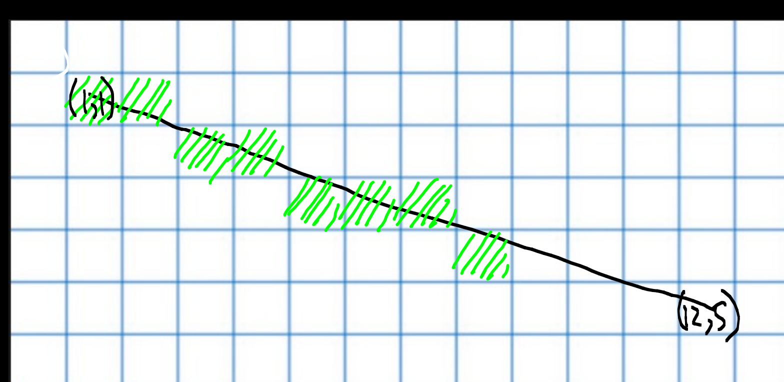
Bresenham's Algorithm

$\Delta x = 12 - 1 = 11$

$\Delta y = 5 - 1 = 4$

error $= -\frac{\Delta x}{2} = -\frac{11}{2} = -5$

```
for (x=1, y=1; x<13; ++x){
    draw_pixel (x, y)
    error = error + Δy
    if (error ≥ 0) {
        y = y + 1
        error = error - Δx
    }
}
```



x=1	x=2	x=3	x=4	x=5	x=6	x=7	x=8
-1	3	-4	0	-7	-3	1	
	2		3			4	
	-8		-11			-10	

Part 2: Synchronize with VGA Timing

The DMA Controller is a memory-mapped I/O device:

Address	31 … 24	23 … 16	15 … 12	11 … 8	7	6	5 … 2	1	0	
0xFF203020	C8000000 front buffer address									Buffer register
0xFF203024	C8000000 back buffer address									Backbuffer register
0xFF203028	240 Y		320 X							Resolution register
0xFF20302C	m	n	Unused	BS	SB		Unused	A	S	Status register

the DMA controller continuously reads pixel values starting at the address in this register.

Aside: Pointer Arithmetic in C

```
char *cp;   // pointer to a byte
int  *ip;   // pointer to a word
   :
cp=cp+1;    // adds 1 to cp, because char is a byte
ip=ip+1;    // adds 4 to ip, because int is a word
```

So, if ip = 0xFF200000, then (ip+2) = 0xFF200008.

Summary: in C if you increment a pointer, the amount added (in the assembly code generated by the compiler) depends on the type of the pointer (char *, int *, ...)

it is useful to know when the DMA ctrl has finished tranferring the last pixel on the screen. This is done using the S bit in the status register.

```
void wait_for_vsync() {
    volatile int * pixel_ctrl_ptr = 0xFF203020; // pixel controller
    register int status;

    *pixel_ctrl_ptr = 1; // start the synchronization process

    status = *(pixel_ctrl_ptr + 3);
    while ((status & 0x01) != 0) {
        status = *(pixel_ctrl_ptr + 3);
    }
}
```

(*) (**)

wait for S to become 0

(**) - when you write a 1 to the Buffer register in the DMA ctrl. this does not change the contents of the register. Instead, it serves as a request to synchronize with the VGA timing. It sets S to 1 in the Status register. We now wait for S to change back to 0. This happens when the DMA ctrl finishes with the last pixel.

- in Part 2 you use wait_for_vsync to "animate" a horizontal line (e.g. draw the line, then wait (S bit), then erase/redraw line one y coordinate up/down on the screen). Note: on a VGA monitor it takes exactly $\frac{1}{60}$ second to draw pixels (0,0) to (319,239).

Part 3: Animation

- We will configure the DMA ctrl. to use two pixel buffers that are in different memories (Onchip & SDRAM):

```
*(pixel_ctrl_ptr + 1) = 0xC0000000; //set Back buffer to SDRAM
```

Address	31 ... 24	23 ... 16	15 ... 12	11...8	7	6	5...2	1	0	
0xFF203020	C8000000		front buffer address							Buffer register
0xFF203024	C0000000		back buffer address							Backbuffer register
0xFF203028	240 Y		320 X							Resolution register
0xFF20302C	m	n	Unused	BS	SB		Unused	A	S	Status register

- The DMA ctrl. always displays the image in the front buffer. While this is happening (it takes $\frac{1}{60}$ sec) we can "draw" a new image in the Backbuffer memory. When you are ready to show the new image you can ask the DMA ctrl. to swap the front/back buffers.

(**) - When you request a VGA synchronization, this not only sets S=1, but also requests a buffer swap. The swap happens at the time that S becomes 0. you would now draw the next new image in the back buffer, which is currently Onchip memory. This process is used continuously to make an animation.

(Demo of Part 3)

- placing N objects on the screen with random colors, random Δx, Δy, and random starting locations:

```
int color_box[N], dx_box[N], dy_box[N], x_box[N], y_box[N];
```

- in general you can use the rand() C library function:

```
// to set dx_box, dy_box to -1 or 1 at random
for each box, i
    dx_box[i] = rand() %2 x2 -1;
    dy_box[i] = rand() %2 x2 -1;
```

modulo (2) = {0,1}

- if you had an array of 10 colors:

```
short color[10] = {0xFFFF, 0xF800, 0x07E0, 0x001F, ...};
    color_box[i] = color[rand() %10];
```

- similarly, you would set a random x_box, y_box.

```
// animation
while (1) {
    draw();
    wait_for_vsync();
    back_buffer = *(pixel_ctrl_ptr + 1);
}
```

```
draw() {
    erase_screen();
    for each box, i
        draw_box(i);
        draw_line(i, (i+1)%N);
    for each box, i
        x_box[i] += dx_box[i];
        y_box[i] += dy_box[i];
        // you must adjust Δx, Δy when a box "hits"
        // one of the sreen edges.
```

e.g. for $N=8$