CS 2110 Lab 20
DMA and State Machines

The TAs :)

November 5, 2018

1 Recitation Outline

1. DMA
2. GBA State Machines

2 DMA

2.1 What is DMA?

DMA stands for **D**irect **M**emory **A**ccess. It's essentially access to memory that doesn't go through the CPU (where our code is).

Why is this useful? Well say we write some code to change the video buffer, so we're manipulating memory from the CPU. Well that code is going to get compiled into a lot of assembly instructions, and if you remember from the LC3, load and store instructions to memory take more than one clock cycle. This means that it's going to be a time-intensive process to change memory from our code, and with the GBA, anything time intensive will lead to tearing!

So what if we had a separate piece of hardware for this? One that didn't require compiling code or running on our CPU's datapath, so it could do this job really fast. That's where DMA comes in!

2.2 GBA's DMA Controller

The GBA has a peice of hardware called the DMA Controller. When enabled, the CPU's execution is paused, the DMA controller copies memory, and afterwards the CPU's execution is resumed.

The DMA controller works into the GBA architecture somewhat like this:

$$\begin{tikzcd}\boxed{\text{CPU}} \arrow[r, leftrightarrow] & \boxed{\text{Memory}} \arrow[r, leftrightarrow] & \boxed{\text{DMA Controller}}\end{tikzcd}$$

2.3 Channels and Registers

The DMA controller has 4 channels for different purposes. We only worry about the last channel, which is used for general purpose copies of memory.

This channel has 3 important registers we need to set:

1. Source
2. Destination
3. Control

2.3.1 Source Register

This is going to be the source address of what we are copying. In general, we usually have two cases for this:

1. Drawing an Image Rectangle
2. Drawing a Color Rectangle

If we're drawing an image on the screen, we need to copy the image's buffer in memory into the video buffer in memory (or into a portion of the video buffer if the image isn't filling the whole screen). So, the source address would be the image's buffer.

If we're drawing a colored rectangle on the screen, the source address would be where that color is stored in memory. If we think of this as a function with a parameter for the color, this would be the address of the color parameter, `&color`. There is a slight problem with this, however, that arises with C itself. Essentially, if we only ever use the address of a variable (in this case color), and never use the value of the variable, the compiler tries to be smart and efficient and never actually initializes the variable, so we would always draw the color black (value of 0).

So how do we solve this? We use the `volatile` keyword we mentioned before, which tells the compiler to not try to optimize this variable. So, when using DMA with a color, the color parameter must **ALWAYS** be declared volatile.

2.3.2 Destination Register

This is going to be the destination address of where we are copying to. Since we almost always use DMA to draw onto the screen, this will be somewhere in the video buffer array. For example, if we wanted to draw starting at the top left of the image, this would just be the starting address of the video buffer array, which would be the `videoBuffer` variable.

2.3.3 Control Register

The control register defines the behavior of the DMA controller. This usually entails the following:

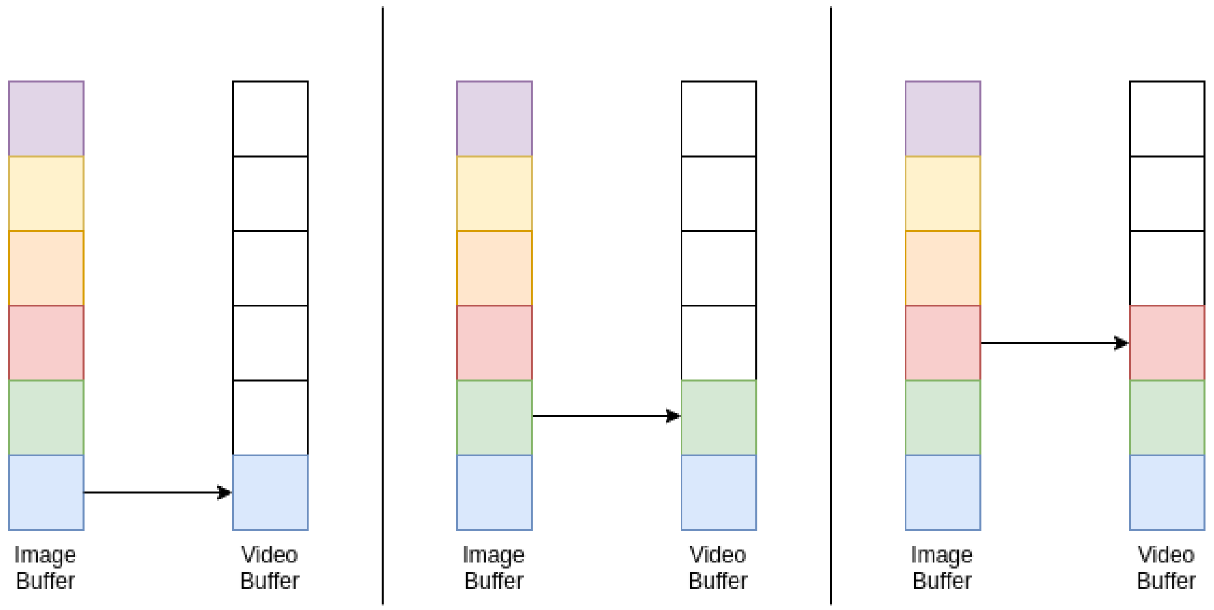
1. On/Off
2. How many elements to copy over
3. How to iterate through the destination
4. How to iterate through the source

The 3rd and 4th behaviors are likely confusing at first, but are very important. For both the source and destination addresses, we can choose to:

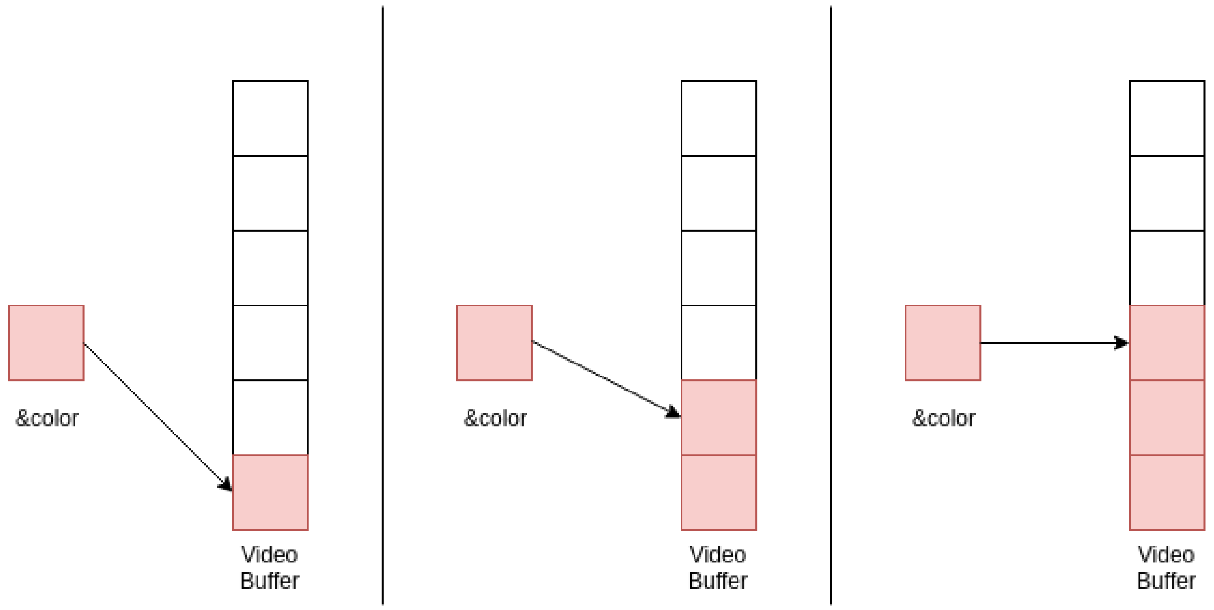
1. Increment the address with each element
2. Decrement the address with each element
3. Fix the address with each element (don't change it)

How would we use these though? Well let's look at two examples:

- Say we're drawing an image on the entire screen, we could say to increment the source address and increment the destination address. The first couple steps would look like this:



- What if we're drawing a colored rectangle though? In that case the source address should stay fixed for each element, but the destination address should increment. The first couple steps would look like this:



2.4 How to use DMA in actual code

So this is all great and fun, but how do we actually use the DMA controller from our code? Well in the header file we provide for all the GBA assignments, we have defined the following struct:

```
typedef struct
{
 const volatile void *src;
 const volatile void *dst;
 u32                 cnt;
} DMA_CONTROLLER;
```

We've also defined the following macro, which is the array of DMA Controllers, one for each channel:

```
#define DMA ((volatile DMA_CONTROLLER *) 0x040000B0)
```

So in our case, since we work with the fourth channel (index 3), we could access our DMA controller struct by doing `DMA[3]`, and access the source register, for example, with `DMA[3].src`.

The control register is probably the trickiest to understand. The control register itself is 32 bits, and these bits are laid out similarly to how our 16 bit LC3 instructions were laid out. It has different ranges of bits that specify the different functions. For example bits 23-24 specify whether to increment, decrement, or fix the source address, and bits 0-15 specify how many elements to copy. You don't need to know these different ranges, so in the header file, we've provided a bunch of macros. Each is a mask over some part of the 32 bits that will set a functionality, and you just need to bitwise or all the masks you want to use together, along with the number of elements you're copying, to construct the value for the control register.

For example, if I wanted to copy 10 elements, increment the source and destination addresses, and turn the DMA controller on, I would do:

```
DMA[3].cnt = 10 | DMA_SOURCE_INCREMENT | DMA_DESTINATION_INCREMENT | DMA_ON;
```

Where `DMA_SOURCE_INCREMENT`, `DMA_DESTINATION_INCREMENT`, and `DMA_ON`, are all examples of the macros we've defined for you.

All together, if we wanted to write a function that colors the entire background of our GBA screen a certain color, it would look like this:

```
// Assuming videoBuffer and all the macros are defined
void colorBackground(volatile u16 color) {
    DMA[3].src = &color;
    DMA[3].dst = videoBuffer;
    DMA[3].cnt = 240 * 160 | DMA_SOURCE_FIXED | DMA_DESTINATION_INCREMENT | DMA_ON;
}
```

3 GBA State Machines

3.1 A GBA program as a state machine

With our gameboy programs, it's convenient to think of them as state machines, just like the ones we did in homework 4. Except in our programs, at the most basic level, our states might be:

1. The start screen
2. Some core gameplay
3. The game over screen

Thinking of it this way can help organize our code more logically. We can declare an enum that defines all the different states of our program and create a variable to keep track of our current state. Which in our example, might look like:

```
enum GBAState {
    STATE_START_SCREEN,
    STATE_GAMEPLAY,
    STATE_GAME_OVER,
};

enum GBAState state = STATE_START_SCREEN;
```

Then inside our main loop, we can switch what code gets executed for that frame depending on which state we're in:

```
switch(state) {
case STATE_START_SCREEN:
    // Do start screen stuff
    break;

case STATE_GAMEPLAY:
    // Do core gameplay stuff
    break;

case STATE_GAME_OVER:
    // Do game over stuff
    break;
}
```

Your lab assignment will mimic this structure.

3.2 Ensuring smooth transitions between states

A lot of times we may want something to happen when a user presses a button. Let's consider the example that when the user presses the A button, the state of the game should change to the next state. If you were to simply write:

```
if (KEY_DOWN_NOW(BUTTON_A)) {
    state = next_state;
}
```

Then since humans can't possibly press the A button for a single frame (a single iteration of our main loop), what will end up happening in our example is we'd go through multiple states from one button press. This isn't what we want, however. We only want to move to one new state for each button press. So how can we fix this?

If we have a variable that keeps track of the whether or not the A button was pressed down in the previous frame we can solve this! With two variables: the state of A last frame and the state of A this frame, we have 4 possiblities, so what should we do in each of them?

1. Last frame A was not pressed and this frame A is pressed:
   This is the case we actually want to change states in, so we should change states, and update our previous state variable.
2. Last frame A was pressed and this frame A is pressed:
   This is the case that was breaking our code previously. In this case we should do nothing.
3. Last frame A was pressed and this frame A is not pressed:
   In this case, we don't need to do anything other than update our previous state variable.
4. Last frame A was not pressed and this frame A is not pressed:
   In this case, we don't need to do anything.

With our new logic, we can see that the only time we transition states is when A goes from not pressed to pressed, which is exactly what we want!