CS 2261: Media Device Architecture - Week 2

Announcements/Clarifications

- Lab01 coming soon
- Final project demo is during final exam period
  - There will be an option to Demo before the final exam period for volunteers.
  - More exact format / timing still to come.
- Milestone roadmap coming soon (hopefully by the end of the week).
- Quiz 1: Wednesday 9/12/18 (not Monday!)

Overview

- Memory
- Bitmaps
  - GBA Video Mode 3
- First GBA Program
  - Light up a single pixel!
- SetPixel
- C
  - Program Structure
  - Compilation Basics

Memory

- RAM
  - Random Access Memory
  - Short-term
  - Fast
  - Lost when device loses power
- ROM
  - Read Only Memory
  - Permanent (can’t be overwritten)
    - Stays the same after power cycle
  - Game cartridge is mostly ROM
    - It has a little RAM for saving games (with a battery!)

How would we set a bit?

- To set bit 3 of x where x is an 8 bit quantity
  - `x = x | (1 << 3)`

- To set bit n of x where x is an 8 bit quantity
  - `x = x | (1 << n)`

Note: Use | and not +

How would we clear a bit?

- To clear bit 3 of x where x is an 8 bit quantity
  - `x = x & ~(1 << 3)`
- To clear bit n of x where x is an 8 bit quantity
  - `x = x & ~(1 << n)`

Note: Use & and not -

How would we test a bit?

- Assume bits are numbered like this:
  - 7 6 5 4 3 2 1 0 (rightmost bit is 1, leftmost is 128)
- To test bit 3 of x where x is an 8 bit quantity
  - `if( x & (1 << 3) )`
    `// where 3 is the bit number we want to check`
- If the desired bit is set, the expression will evaluate to true (i.e. In C, not zero)

GBA Video Output

- 240 x 160 pixel display -- 38400 pixels
  - Pixel: Picture Element - single dot on a screen
- 6 Display modes
  - Bitmaps modes - 3, 4, 5
  - Tiled modes - 0, 1, 2
- Let's consider bit mapped mode first
  - Every pixel corresponds directly to one or more bits in (video) memory, at a specified location, per-pixel
  - If each pixel were a single bit?
    - On/Off (black or white) only. Monochrome is boring.
    - We don't even have addresses to single bits! (only 8 at a time)!

Better Bitmaps

- More bits per pixel!
  - 2? -- 4 values. Maybe 100%, 66%, 33%, 0%?
    - You could get some crude grayscale, I guess.
  - 3? -- 8 values. Better greyscale? Primary/secondary colors via one bit each!?
  - 4? -- 16 values. Now we can do 16 colors or 16 shades of gray.
- How about a single byte?
  - 8 bits -- 256 values. That's a lot of shades of grey to work with -- we could even do okay color now.
- If we want color, we need to divide the bits into groups for each color.
- The specific mapping of bits to pixels represents a display mode

Bit Vectors

- Storing multiple values (as individual bits or groups of bits) is known as a bit vector
- Bit vectors are used to save space, and can offer speed benefits as well, when properly used.
- Bit vectors are used extensively in GBA programming especially with I/O
- A key to using bit vectors is the bitwise operations

Mode 3 (easiest to explain)

- 16 bit quantity to describe color
  - Assigned as a short int type (syntax: 'short')
- For mode 3, GBA reserves block of memory that is 38400 shorts -- (75kB of the total 96kB VRAM)
- Describing color in 16 bits:
  - 5 bits for blueness, 5 bits for greenness, and 5 bits for redness (Are we missing something?)
  - Bit 15 is always 0 (so it's really 15-bit color?)
  - How many levels of each color can we have?
    - 2^5 = 32
  - How many colors?
    - 32x32x32 = 2^15= 32768

How can we put three values in one variable?

- Color values in Mode 3 of the GBA are stored as 3 (Red, Green and Blue) five-bit values packed into one short.
  - XBBBBBGGGGGRRRRR
  - 15-bits of color, 5 per color (0 - 31, per color channel)

How can we put three values in one variable?

- Color values in Mode 3 of the GBA are stored as 3 (Red, Green and Blue) five-bit values packed into one short.
  - XBBBBBGGGGGRRRRR
  - 15-bits of color, 5 per color (0 - 31, per color channel)

- We can code:

```
R | G<<5 | B<<10   OR   B<<10 | G<<5 | R
```

- Or as a preprocessor macro (more later)

```
#define RGB(R, G, B) ((R) | (G)<<5 | (B)<<10)
```

Light up a pixel in Mode 3

- There is a register for assigning display mode
  - “Display Control Register”
  - Location is at memory address 0x04000000
    - Sometimes shorthanded to x4000000 (people are lazy)
  - Writing them in binary would be too long for the human eye to parse well.
    - 0 4 0 0 0 0 0 0
    - 0000 0100 0000 0000 0000 0000 0000 0000
  - Writing them in decimal form would make reasoning about the underlying bits hard.

REG_DISPCNT (0x04000000)

Bit # (hex)	0F	0E	0D	0C	0B	0A	09	08	07	06	05	04	03	02	01	00
Bit # (dec)	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Bit Meanings:					BG3	BG2	BG1	BG0						Mode		

- Bits 0-2 bits are the mode:
  - Values: 0, 1, 2 -- tiled modes
  - Values: 3, 4, 5 -- bitmap modes
- Bits 8-11 are for backgrounds
  - We need to turn on BG2 when using bitmaps

REG_DISPCNT (0x04000000)

Bit # (hex)	0F	0E	0D	0C	0B	0A	09	08	07	06	05	04	03	02	01	00
Bit # (dec)	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Bit Meanings:					BG3	BG2	BG1	BG0						Mode		

- Setting mode 3 and GB2 bit-wise
  - We need 3 bits, to make the number 3: 0011
  - We need the 10th bit to be a 1: +0100 0000 0000

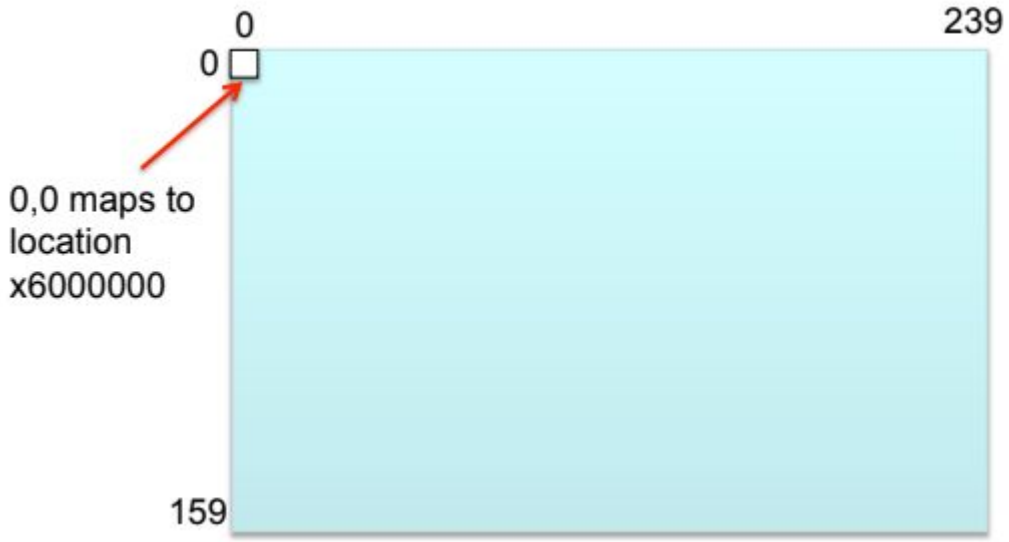
    0100 0000 0011

    0x 4 0 3

    aka 1027

Video Memory (Mode 3)

- Starts at location 0x06000000
  - Consists of 240 x 160 16-bit (short) integers



To make this pixel white, we need to set it to (binary) 0111 1111 1111 1111 = 0x7FFF = 32767

Program Time

- We want to activate Mode 3
  - Apply 1027 bits to address 0x04000000
- Then set the top-left pixel to white
  - Store 32767 at address 0x06000000

First GBA Program

- Just like Java, execution starts with function main
  - Always returns `int`
    - Return value means nothing on the GBA, but when C programs are run by other programs, returning anything other than zero, usually means an error occurred.
  - Game needs to run forever (until the GBA is turned off)
    - If you win/lose, it would just go back to the title screen
    - This case is true of a lot of dedicated hardware (thermostats, clocks, ovens, etc.). They're really only ever truly off with no power.

First GBA Program (main.c)

```
int main(){
  // Store 0x403 in location 0x04000000



  // Store 32767 in location 0x06000000

  while(1) {  // anything non-zero is true in C!
    // just keep doing nothing...
  }
  return 0;  // make some strict compilers happy
}
```

First GBA Program (main.c)

```
int main(){
  unsigned short *REG_DISPCNT = 0x04000000;  // pointer to addr.
  *REG_DISPCNT = 0x403;  // put value into the address

  // Store 32767 in location 0x06000000

  while(1) {  // anything non-zero is true in C!
    // just keep doing nothing...
  }
  return 0;  // make some strict compilers happy
}
```

Pointers?!?

```
int main(){
  unsigned short *REG_DISPCNT = 0x04000000;  // pointer to addr.
  *REG_DISPCNT = 0x403;  // put value into the address

  // Store 32767 in location 0x06000000

  while(1) {  // anything non-zero is true in C!
    // just keep doing nothing...
  }
  return 0;  // make some strict compilers happy
}
```

- What are these new pointer things?
  - A pointer is the C type used to *point* at an address in memory.
  - It's a variable that can hold an address (to nothing, to another variable, to a random place in memory)
- What is the * doing there?
  - Two different things!

Basic C Variable Syntax

- C is strongly typed, so every variable needs a type
- Declaration

```
typeName varName;  // declare a var of type
```

- Assignment

```
varName = someValue;  // assign var to value
```

- Combined:

```
typeName varName = someValue;  // both at once!
```

Examples:

```
int foo = 3;
int bar;
bar = foo;
```

Basic Pointer Syntax (part 1)

- Declaration:

```
unsigned short *ptr;
// pointer to unsigned short
```

- Assignment

```
ptr = 0x04000000;
```

```
unsigned short *ptr          = 0x04000000;
```

[type of thing pointed at] *[variable name] = [numeric address];

Really, `(unsigned short *)` is the type here, from C's perspective.

Basic Pointer Syntax (part 2)

- Dereferencing (the second use of *)
  *REG_DISPCNT = 0x403;
  - Used this way, the * in front of the variable name *dereferences* the pointer.
    - This means *ptr, is used to mean “the thing my pointer was pointing at).
    - Here, we’re setting that thing to 0x403.

- REG_DISPCNT is the pointer
  *REG_DISPCNT is the thing REG_DISPCNT is pointing at.

First GBA Program (main.c)

```
int main(){
  unsigned short *REG_DISPCNT = 0x04000000;  // pointer to addr.
  *REG_DISPCNT = 0x403;  // put value into the address

  unsigned short *pixel = 0x06000000;
  *pixel = 32767;

  while(1) {  // anything non-zero is true in C!
    // just keep doing nothing...
  }
  return 0;  // make some strict compilers happy
}
```

Demo

Compact Form of main.c

```
int main(){
  *(unsigned short *)0x04000000 = 0x403;
  *(unsigned short *)0x06000000 = 32767;

  while(1) {  // anything non-zero is true in C!
    // just keep doing nothing...
  }
  return 0;  // make some strict compilers happy
}
```

- (unsighted short *) is a cast here -- we're telling C we want to treat that int as a pointer to an unsigned short.
- We then immediately dereference the pointer and assign values to those locations in memory.

Pointers are Tricky

- Pointers have a type, but you can create two pointers with different types pointing to the same address in memory

  Ex:

```
short *foo = 0x06000000;
int *bar = (char *)foo;  // cast
```

Pointer types basically tell the C how to interpret the bytes at a given memory location when you perform a dereference.

How about not just white pixels?

```
#define RGB(R, G, B) ((R) | (G)<<5 | (B)<<10)

int main(){
  *(unsigned short *)0x04000000 = 0x403;

  *(unsigned short *)0x06000000 = RGB(31, 31, 31);
  *(unsigned short *)0x06000002 = RGB(31, 0, 0);
  *(unsigned short *)0x06000004 = RGB(0, 31, 0);
  *(unsigned short *)0x06000006 = RGB(0, 0, 31);

  while(1) {  // anything non-zero is true in C!
    // just keep doing nothing...
  }
  return 0;  // make some strict compilers happy
}
```

Demo again