CS2261 Media Device Architecture – Recitation 4

Announcements:

* Submit assignments as .zip files (not .rar or any other type of zip file)
* Lab on Thursday
* Lab 1 grades to be released soon

Number Bases:

* Numbers are symbols representing quantities, thinking of them more abstractly will help in this class
  + The base is the number of symbols
  + Examples:
    - Base-10: decimal, 10 symbols, numbering 0-9.
    - Base-16: hexadecimal, 16 symbols, from 0-9 then A-F, where F represents 15
    - Base-2: binary, 2 symbols, 0 and 1
  + Typically, we arrange the symbols from right to left
  + Every time you change the symbol to the left, you mean that you ran out of symbols to the right
  + The new symbol is worth: number of symbols \* worth of symbol to the right
    - Ex. Base-10 (decimal)
      * The base is 10, so we need 10 symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
      * When we run out (nine), we start over and add one to the left: 10
      * That means 1\*101 + 0\*100
    - Ex. Base-16 (hexadecimal)
      * The base is 16, so we need 16 symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
      * When we run out (fifteen), we start over and add one to the left: 10
      * That means 1\*161 + 0\*160 (sixteen)
    - Ex. Base-2 (binary)
      * The base is 2, so we need 2 symbols: 0, 1
      * When we run out (one), we start over and add one to the left: 10
      * That means 1\*21 + 0\*20 (two)

Bitwise Operators

* Operate not on the values, but the bits that represent them
  + When you declare an int in C, the GBA allocates 4 bytes to store it as a binary value
* Ex. int a may equal 58, but it really equals 000000000000000000000000001110102

|  |  |  |
| --- | --- | --- |
| Operator | Name |  |
| & | And | Only true when both are true, false otherwise |
| | | Or | Only false when both are false, true otherwise |
| ^ | Xor (Exclusive or) | False when both true or both false, true when not matching |
| ~ | Complement | Inverts the bits |
| << | Left-bitshift | Shifts bits to the left |
| >> | Right-bitshift | Shifts bits to the right |

Boolean

* In C, there is no boolean type.
  + 0 represents false, and any other number represents true (1, 10000, 17, -18)
* while (1) {
  + Since 1 is true, that’s an infinite loop. You could replace that 1 with any number that’s not 0
  + We don’t want the GBA game to suddenly end

Adding binary:

* Like adding regular numbers. If you add two 1’s in a column, write 0 and carry the 1 to the next column
  + 0111 1110 0010
  + 0100 0101 1101
  + + ----------------------
  + 1100 0011 1111

Bitwise logical operators

* + Ex. 35 & 50 = 34 35 | 50 = 51 35 ^ 50 = 17
    - 00100011 00100011 00100011
    - 00110010 00110010 00110010
    - ---------------------------------------------------------------------------------------
    - 00100010 00110011 00010001

Bitwise not (complement) ~

* Ex. ~35 = 220

You convert the decimal into binary and change every 0 to a 1, every 1 to a 0

35 = 00100011

~35 = 11011100 = 220

Bitwise shifting operators (<< >>)

* Left-bitshift (<<)
  + Shifts the bits to the left by a certain number of places
  + Pads the right side with that number of 0s
* Right-bitshift (>>)
  + Shifts the bits to the right by a certain number of places
  + Pads the left side with that number of 0s
  + Ex. 43 << 4 = 176 43 >> 3 = 5
* Losing bits with shifting
  + When you shift bits, some bits will be lost
  + They “fall off the end” either to the left (<<) or right (>>)
    - Ex. 00101011 << 4 = 10110000 (43<<4 = 176)
  + If 43 is a char (stored in only 8 bits), the 0010 is lost to make room for the 0000
  + If 43 is a short (stored in 16 bits, thus 0000000000101011), the initial 0000 is lost
    - Ex. 00101011 >> 3 = 00000101 (43>>3 = 5)
  + No matter how many bits 43 is stored in, the 011 is lost to make room for the 000
  + Any quantity shifted by the number of bits it is stored in (or more) equals 0
    - Ex. The 8-bit char 00101011 left-bitshifted by 8 is 00000000
    - 43<<8 = 0 if 43 is stored in 8 bits

Using bitwise operators with flags

* A flag is a sequence of bits used to change something in a register
  + Ex. Using BG2\_ENABLE or MODE3 to change REG\_DISPCTL
* Since each bit in a register means something specific, we only want to affect the specific bits we want to change
  + Ex. The 10th bit in REG\_DISPCTL controls whether background 2 is enabled
* Creating a flag: use << to shift the bit to the desired location
  + Ex. BG2\_ENABLE is 1<<10, thus 0000010000000000
  + Ex. MODE3 is 3<<0, thus 0000000000000011 (you can leave off the <<0)
* Setting a bit with a flag: use | to set that bit in the register
  + Ex. 0000000000000000 REG\_DISPCTL
  + BG2\_ENABLE
    - * 0000000000000011 MODE3
      * 0000010000000011 REG\_DISPCTL | BG2\_ENABLE | MODE3

More flag operations

* Checking if a flag is set: use & to see if those bits are on (if they are 1)
  + Ex. if (REG\_DISPCTL & BG2\_ENABLE) {
  + If BG2\_ENABLE has been set, this equals 1024 (true). If not, this equals 0 (false)
* Turning a flag off: Use &~ to clear those bits from the register
* Ex. 0000010000000000 BG2\_ENABLE
  + - 1111101111111111 ~BG2\_ENABLE
    - 0000010000000011 REG\_DISPCTL
    - 0000000000000011 REG\_DISPCTL & ~BG2\_ENABLE
* Even more flag operations
* Toggling a flag: use ^ to set the bits or clear them if already present
* Ex. 0000000000000011 REG\_DISPCTL
* BG2\_ENABLE
  + - 0000010000000011 REG\_DISPCTL ^ BG2\_ENABLE
* BG2\_ENABLE
  + - 0000000000000011 REG\_DISPCTL ^ BG2\_ENABLE

Using the Preprocessor

* The Preprocessor “language”
* The preprocessor does most of its work with statements written specifically for it.
* Called “preprocessor directives”
* Begin with #, then a keyword, then a space
  + #keyword “WYSIWYG”
* The rest of the statement is (mostly) WYSIWYG (What You See Is What You Get) (even spaces) until the end of the line.
* Remember: all of this runs and completes before the compilation step starts.
* Basic code:
  + #define “WYSIWYG”
  + #include
    - This one is like importing