CS 2261: Media Device Architecture - Day 2

Reminder

- Be in this room at 4:30 tomorrow for Lab 0
    - Bring your computer!

- No TA office hours <u>this week</u>!

Overview

- Data
- Von Neumann Architecture
- The GBA
-

Data (well, integers for now)

- Binary “There are 10 kinds of people in the world, those that understand binary and those that don't.”
  - 1 10 100 101 1111
- Base 10
  - 1 2 4 5 15
- If each 0 and 1 is a “bit”, how large a number can we represent with 1 bit? 2 bits? 3?
- Addition in binary:
  - Carry the 1 (a bunch)!

Binary Numbers (just the counting ones for now)

- For N bits, we can represent 2^(N-1) as a maximum value.
  - 2^N total numbers can be represented, including 0

- 8-bits, 0 - 255
- 16-bits, 0 - 65,535
- 32-bits, 0 - 4,294,967,295 -- still not enough to represent all the people on Earth.
- 64-bits, 0 - 18,446,744,073,709,551,615 (quintillions)

What about negative integers?

- Signed Binary Numbers
  - Sign bit?
    - For an 8-bit number, let's use the first bit to show if the number is positive or negative:
      - 00000011 = 3
      - 10000011 = -3
      - What about 00000000 vs 10000000?
        - -0 and +0?!?
  - What's the range of numbers we can now represent?
    - 11111111 - 01111111 = -127 to +127 -- this is 255 numbers (one less than before), and it has a weird +0 vs -0.
  - Adding these numbers is now also really odd.

Better Signed Binary Integers

- Can we make things a little less hard on ourselves (especially at the hardware layer)?
  - One's complement
    - Flip all the bits to make a negative number (still starts with 1)
      - 00000101 = 5   11111010 = -5
      - Added together the old fashioned way: 11111111 = -0 (so close!)
      - We still have +0 and -0 (still just 255 numbers -127 - 127)
  - Two's Complement
    - Flip all the bits -- then add 1
      - 00001101 = 13 -> 1110010 + 1 -> 1110011 = -13
      - How about the other way?   1110011 -> 0001100 + 1 -> 0001101
      - 00000000 = 0 -> 11111111 + 1 -> 00000000
        - No more +/-0!
      - Range: 11111111 - 01111111 (a full 256 possible numbers)
    - Addition Stays simple (believe it or not)

Other Bases

- Octal

01, 02, 03, 04, 05, 06, 07, 010, 011, 012, 013, 014, 015, 016, 017, 020.

  - Not confusing at all, right?
  - Traditionally (in C and many other languages), written with a leading 0
  - 034 = ???
  - Don’t accidentally use these!

- Hexadecimal
  - 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0xA, 0xB, 0xC, 0xD, 0xE, 0xF, 0x10
  - Less easily confused with decimal numbers, thank god!
  - Very commonly used when writing memory addresses, or raw values from computer memory. In that case, it might not even represent a number!
    - Each digit here represents 4 bits
      - 0x0 = 00000000, 0x0F = 00001111   0xFA = 11111010

Typical C Integer Data Types

- 8 bit “character” (char) -- ASCII character set
- 16 bit “short”
- 32 bit “integer”   (int or long) -- quirk
- 64 bit “long long”

These all come in both “regular” and “unsigned” versions.

Note: C data type sizes are all technically variable depending on the target platform.

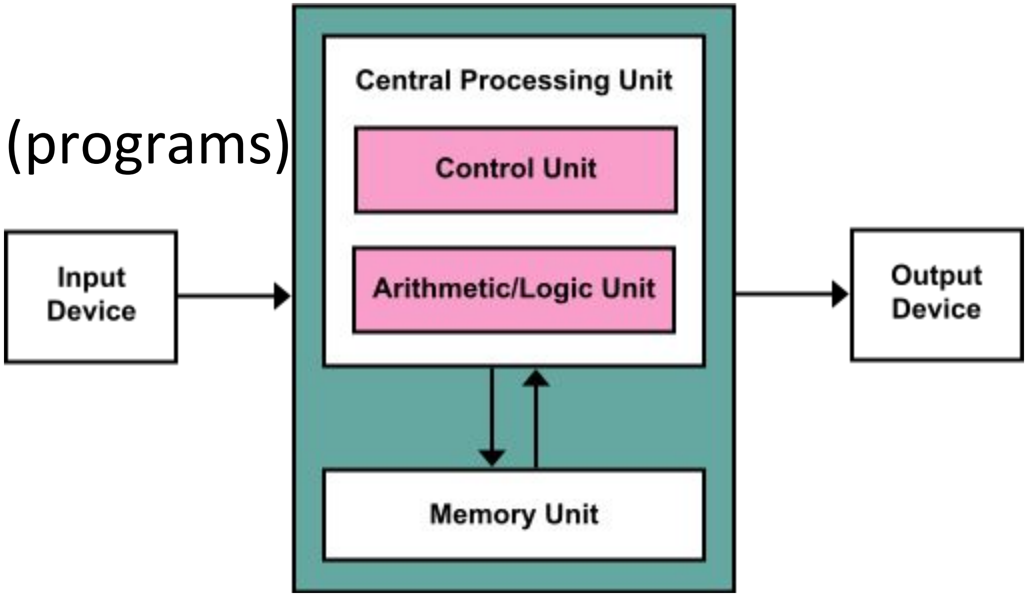
8 bits <= char <= short (>=16 bits) <= int (>= 16 bits) <= long (>= 32 bits) <= long long (>= 64 bits)

John von Neumann (1903 - 1957)

- Hungarian-American
  - Major mathematician, physicist, computer scientist most people have never heard of.
  - Said to have invented the merge sort algorithm in 1945.
  - While working on the EDVAC in1945 (one of the first binary computers -- 5.5 KiB of memory, 56 kW of power!), Von Neumann proposed enhancements to its design, featuring a “stored-program” concept, leading to what we now call the Von Neumann architecture.

The Von Neumann Architecture

- Central Processing Unit (CPU)
  - Control Unit
    - Instruction Register (current instruction being performed)
    - Program Counter (a.k.a. instruction pointer)
  - Arithmetic Logic Unit
  - Memory (RAM)
    - Data and Instructions (programs)
  - Input
  - Output



Von Neumann Benefits

- General purpose computing
  - Before storing programs as data, “reprogramming” a computer was often extremely difficult and required physical modifications.
- Makes Assemblers, Compilers, Linkers, Loaders, Interpreters all possible.
- Allows code to be self-modifying (both awesome and terrible)

Why the GBA

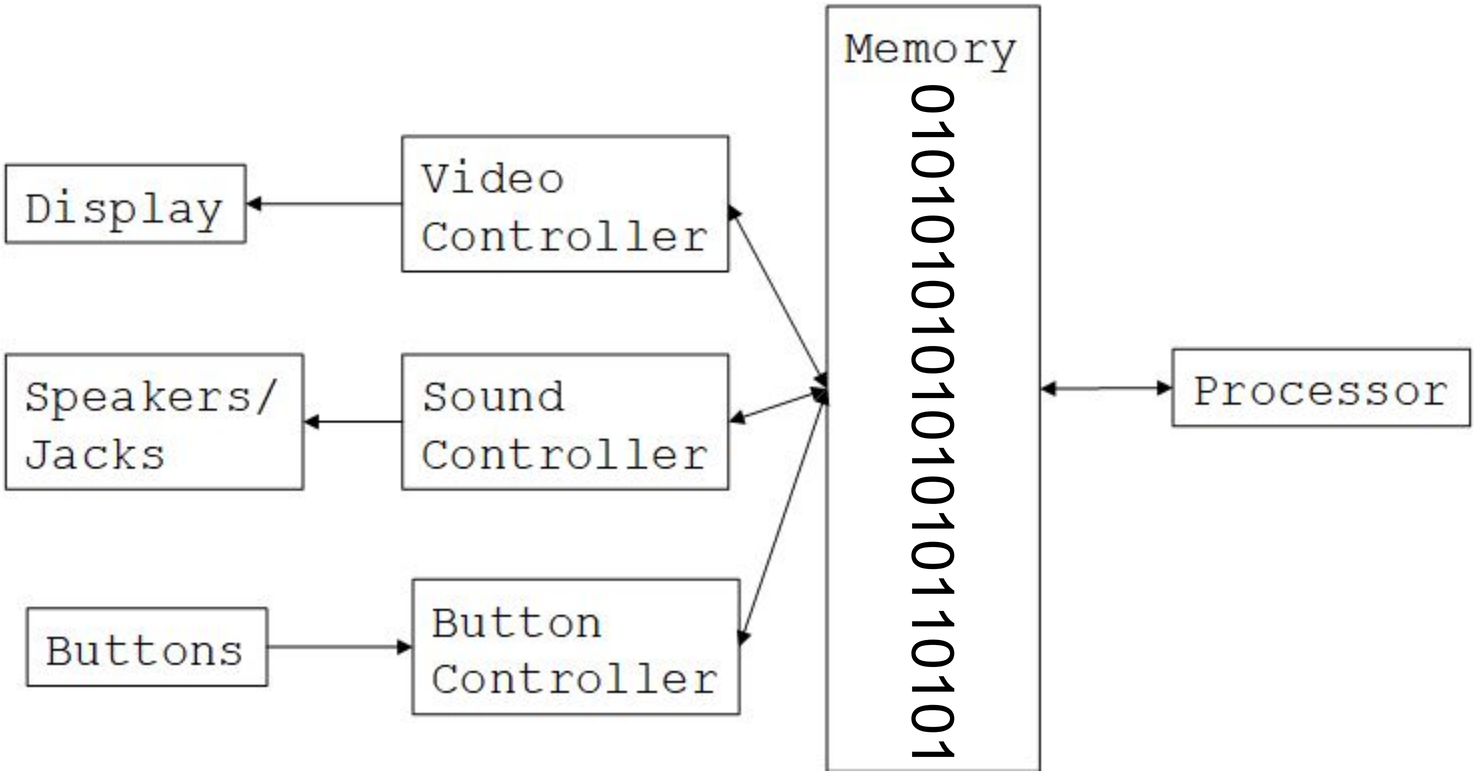
- No operating system
  - Programming is “on the [emulated] metal” - DIY!
  - Avoid OS abstractions and complications
- Slow and with limited resources
  - Forces programming decisions related to performance tradeoffs and storage limitations.
- It’s a Von Neumann Architecture
- It’s cheap (well, now free), and not fake (CS2110!)
- Fun!? (and now nostalgic)

GBA as Von Neumann

- Processor (16.8 MHz 32-bit ARM7TDMI)
  - Actually has a second CPU for playing old GameBoy games)
  - Can't (natively) do floating-point arithmetic.
- Memory (128 KiB Video RAM, 256 KiB general RAM)
  - This is where program can interface with I/O
- Input -- 10 buttons!
  - (Start, Select, A, B, Left, Right, Up, Down, Left shoulder, Right shoulder)
- Output
  - Screen (240 × 160 pixels ~0.04MP -- max 32,768 (2^15) colors)
  - Speakers (Dual 8-bit DAC for stereo sound)


CPU AGB B
JAPAN ARM
BSI
Nintendo
C/AGS-CPU-11
3343729

Machine Model



Memory: Bits, Bytes, Words

- Memory consists of individual bits which may have a value of 0 or 1
- BUT… The smallest quantity of bits we can access is 8. This is known as a byte (char's are 1 byte)
  - Bytes have addresses that increment by 1
- Other data items which consist of groups of bytes have addresses which increment depending on the number of bytes
  - e.g. Shorts are 2 bytes long so their addresses are multiples of 2
- Words are the size at which the processor operates
  - 32-bits / 4 bytes on the GBA

Addresses

- Addresses are usually expressed as hexadecimal numbers

- There are gaps in the address space

- Some areas of memory may be accessed as bytes, shorts and ints while others may only be accessed as shorts and ints
  - 8, 16, 32
  - 16, 32

How do we mess with bits

- Bitwise Operators
  - & bitwise and
  - | bitwise or
  - ~ bitwise complement
  - ^ bitwise xor
  - >> right shift
  - << left shift

C Operators (for completeness)

- A complete list:
  - https://en.cppreference.com/w/c/language/operator_precedence

    or
  - http://web.cse.ohio-state.edu/~babic.1/COperatorPrecedenceTable.pdf

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)

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 -

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
- Bit vectors are used extensively in GBA programming especially with I/O
- The key to bit vectors is the bitwise operations

How can we put three values in one variable?

- Color values are often stored in the GBA as 3 (Red, Green and Blue) five bit values packed into one unsigned 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 are often stored in the GBA as 3 (Red, Green and Blue) five bit values packed into one unsigned 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 macro

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