Instructor notes.

**Slide 3**: it's important to note that ARM does not manufacture anything. They design processor architectures and other companies license those architectures and manufacture physical silicon based on those designs. ARM is basically an intellectual property company and **designs** computer processors.

**Slide 4**: The actual CPU used in a Raspberry Pi 3 is built by Broadcom, a company that licenses ARM designs. They manufacture the BCM2837 which uses the ARM cortex a-53 architecture and as you can see this particular processor has four cores, some additional peripherals, and runs at 1.2 gigahertz. It is particularly worth noting from a heterogeneous perspective that this particular part includes a floating point co-processor (VFP) and SIMD engine (NEON). In addition, almost all ARM general purpose processor designs employ some version of the Thumb, an alternate instruction encoding scheme designed to save code space and reduce power consumption. The second set of slides contains more information about NEON and Thumb.

**Slide 5**: ARM has 32-bit and 64-bit versions of their architecture, but we will discuss specifically the 32-bit version. There are 16 32-bit registers on ARM labeled R0 through R15. Three of those are reserved for special purposes as shown on the slide. The pipeline visible to the programmer is three stages, although the actual pipeline is typically much deeper and varies by architecture version. ARM uses a load/store model; all data has to be loaded into memory before it can be used using an explicit load operation, and likewise data is stored via a store operation. Predicated or conditional instructions are available, and the version used on the Raspberry Pi has a floating point unit called the VFP, and a SIMD unit called NEON.

**Slide 6**: Let’s look at the general format of an ARM instruction, specifically the arithmetic and logic operations. ARM assembly can be a little bit confusing as there are a few different versions. (For example, the KEIL? emulator.) We will be using the syntax used by the GNU toolchain as you would use on a Raspberry Pi. <Dest> register is the destination of the computation. The second two parameters are the source operands. The destination must always be a register. Operand one is always a register. Operand two can be a register an immediate value with certain restrictions or a shifted value.

**Slide 7**: This is a list of some common arithmetic operations, not all of them but you can see add and subtract. There's also a reverse subtract instruction, RSB, that subtracts operand 1 from operand 2. This is important if operand 2 is an immediate value. The ADDC instruction works like regular ADD but also adds the contents of the carry bit to the result. The MUL instruction has a few restrictions as noted on the slide. ARM also provides an MLA (multiply and accumulate) instruction, to support DSP algorithms.

**Slide 8**: These are the bitwise instructions and, or, exclusive or, bit clear, and move negative. These follow the same pattern as the arithmetic instructions, with the destination operand first followed by the source operands. The bit-clear instruction is **dependent** on the order of its operands. In short, if we consider Operand1 and Operand2 as a sequence of bits, let O1*n* be the bit in position *n* of operand1 and O2*n* be the bit in position n of Operand 2. Let D*n* be bit *n* in the destination. If O2*n* == 0 then D*n* = O1*n* else (O2*n* == 1) D*n* = 0. That is, if the bit *n* in Operand 2 is a 0, then copy the corresponding bit in Operand1 to the Destination. Otherwise, bit *n* in operand 2 is 1, put a 0 in the destination. Logically this instruction does this:   
<dest> = <Operand1> AND (NOT <Operand1>)  
MVN stands for Move negative and effectively performs a bitwise NOT of operand1 and puts the result in <dest>

**Slide 9:** ARM allows shifted operands and you can see in the first example and an operation with the second operand R2 is going to be shifted left by 3 before being added to R1. This does not change the value in R2 . ARM provides logical and arithmetic shift left and logical and arithmetic shift right. ASL is rather redundant as it is the same as logical shift left. Assemblers typically only provide one of these. Most assemblers seem to allow a standalone shift operation as seen in the example. This is usually encoded as a MOV with a shifted second operand.

**Slide 10:** The MOV, or move instruction, implements a data copy operation. Operand 2 (notice there is no Operand1) is copied into the destination register. In the general load operation, the first operand is the destination of the load second operand contains the address from data will be loaded. Notice the second operand appears in square brackets. The STR instruction seems a little backwards. STR take the contents of the first operand and stores it into memory at the memory address pointed to by the second operand. In the example on the slide, the value in R3 will be stored at the address pointed at by R5.

**Slide 11:** ARM provides several addressing modes. The first is just regular indirect addressing. The operand in square brackets is a register containing the memory address from which data will be loaded. In pre-indexed addressing you'll notice the two operands appearing in the square brackets. These two values are added together to compute the effective address. In the first example the value 8 will be added to the contents of R1 and that will be the address from which data is loaded. With pre-indexed addressing, the memory address registers (in square brackets) are not changed by the instruction. Adding the ‘!’, or write-back, after the square brackets forces the memory address register to be updated with the computed effective address. So in the last example on the slide, after the memory address is computed, R1 + 8, that value will be stored back into R1.