Lecture #8 – February 13, 2004

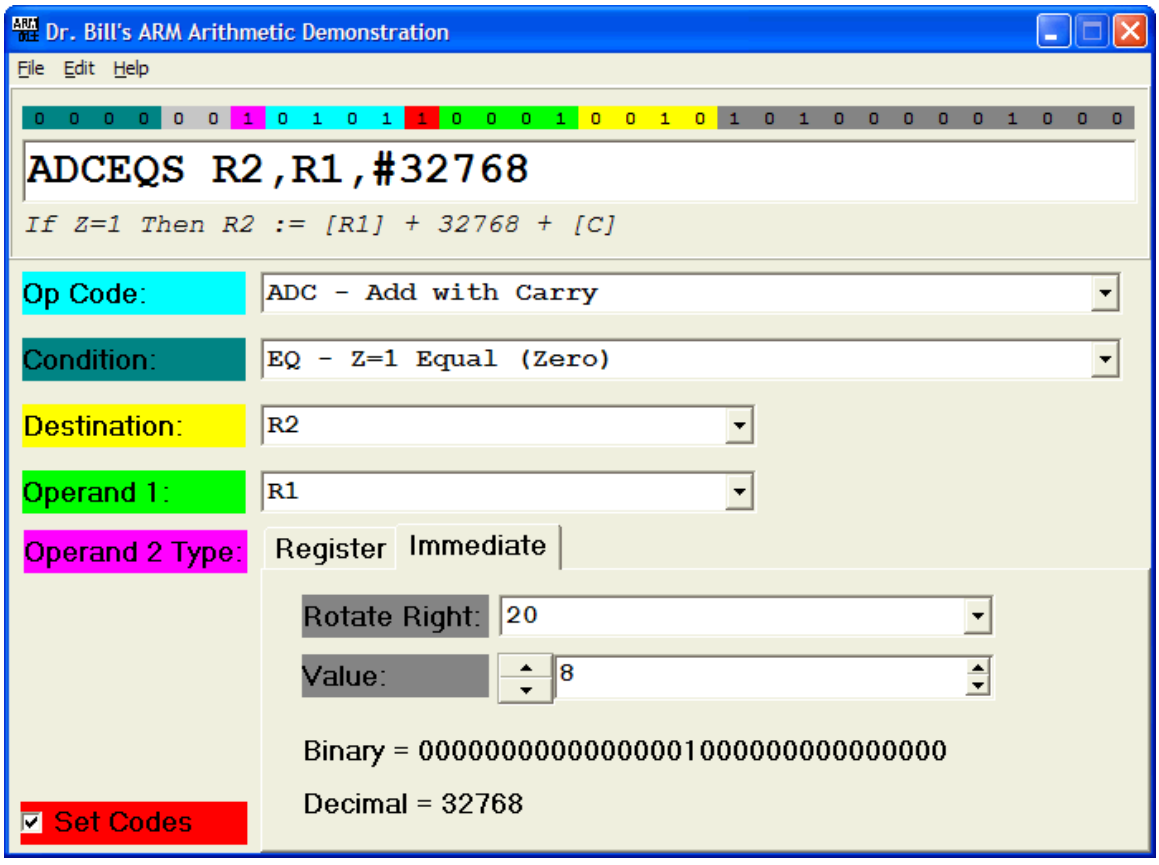
Instruction Formats and the ARMulator

Instruction Formats

It is appropriate to examine how a small group of ARM instructions are translated into binary in order to understand the larger picture of instruction encoding in general. To facilitate this process, there is a program available which illustrates how the group of arithmetic instructions is translated into binary. This program is available at:

`http://www.cs.umass.edu/~verts/cmpsci201/ARM_Arithmetic_Distribution_V1_0.zip`

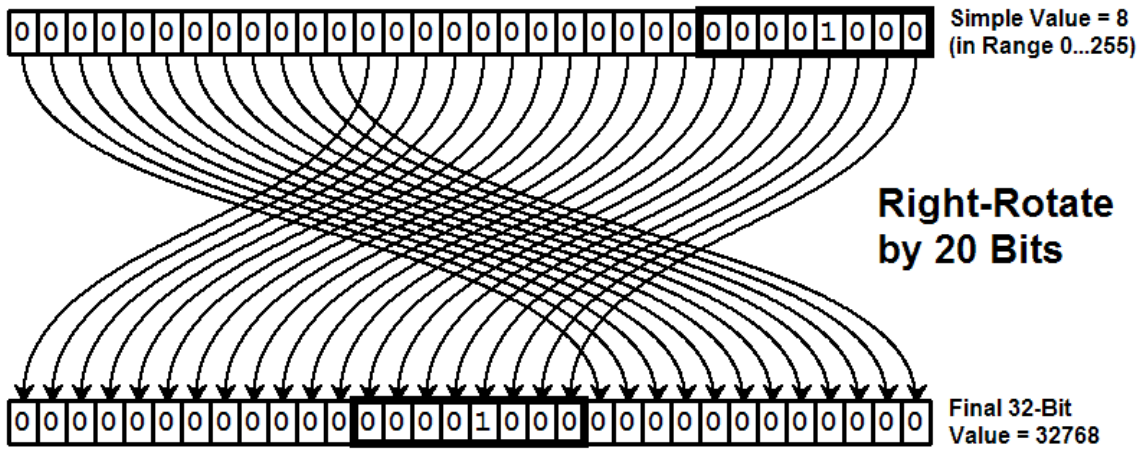
Unpacking the `.ZIP` archive reveals a single `.EXE` program of 399K. With this program you first select the OpCode, the destination register, the first source operand register, and the second operand. The second operand can be either a register (possibly shifted by a constant or the contents of a fourth register) or a constant (formed by a number between 0 and 255, right-rotated by an even number between 0 and 30). In most arithmetic instructions the destination and one or both source registers may be the same physical register. The program also shows the effects of selecting a conditional execution mode or the flag that sets the condition codes. An example is shown below:



Notice that the actual ARM instruction that would be written by an assembly language programmer is shown in the white edit box (`ADCEQS R2,R1,#32768`) and that the equivalent high-level form is shown directly below. For this particular instruction the high-level form `If Z=1 Then R2 := [R1] + 32768 + [C]` tells us that only if the `Z` flag is set to 1 then the contents of register `R1`, the constant 32768, and the contents of the carry bit are added together and the sum will be sent to register `R2`. The `S` suffix on the instruction also tells us that the result of the addition will be used to modify the status bits in the program status register.

The number 32768 is one of a small number of constants that can be encoded into an ARM instruction. Since all instructions are exactly 32 bits in length, and must contain the OpCode, condition flags, destination register, etc., there are very few bits left over for encoding constants. In particular, there are exactly twelve bits available for creating constants. With twelve bits you can create any of 4096 separate constants (2^{12}=4096). One approach would be to treat those bits as an unsigned number between 0 and 4095, but the designers of the ARM chose a different approach to spread the existing patterns more evenly over the space of 2^{32} possible constants. Eight of the twelve bits are used to form a simple number between 0 and 255, and the other four bits are used to determine a ***rotation factor***. Four bits is not enough to select all possible rotations between 0 and 31 (you need five bits to do that), so the four bits are used to select the *even* rotation factors between 0 and 31 (i.e., rotations of 0, 2, 4, 6, …, 28, and 30 bits). The simple number is then right-rotated within a 32-bit word by the amount of the rotation factor to form the final constant.

In our example, the simple 8-bit value 8 is right-rotated by 20 bits to form the final 32-bit number 32768. (Note that within 32 bits, a right-rotation of 20 bits is equivalent to a left-rotation of 12 bits.) This process is shown below:



The trick to forming constants in this manner is to first insure that the desired constant has all of its 1-bits constrained to an 8-bit block, shown by the thick line in the diagram. If the 1-bits fit into an 8-bit block *and* the count of 0-bits to the right of the block is an even number, then the constant can be formed.

All arithmetic instructions on the ARM form constants this way, except that the `MVN` (move-negative) instruction moves the one-complement of the number into the destination register. This doubles the number of definable constants to include those constants which are all 1-bits, but which may contain a small block of 0-bits (the block of 0-bits must fit into an 8-bit block and the count of 1-bits to the right of the block must be an even number).

The ARMulator

The software available for the class consists of three major components: the assembler, the linker, and the emulator (or debugger). Programs are written in a simple text editor such as Windows Notepad, then saved to a file with a `.s` file extension.

The ***assembler*** reads in the `.s` file, and then creates a `.o` file containing the translated binary version and a `.ali` file containing an assembly listing of the program. The `.o` file will not be created if the `.s` file contains errors; in such cases the `.ali` file is used to locate the error. Programmers fix the errors by editing the `.s` file and reassembling the program until the `.o` file is created and the `.ali` file contains no errors.

The next step is to link the program into a final executable version. The ***linker*** reads in the `.o` file created by the assembler, and then creates a `.map` file containing symbol information and a file with no extension. The file with no extension contains the final binary version of the program.

The ***emulator/debugger*** is used to load in the final binary program for execution. The image is loaded into the emulator, various views of the registers and output console windows are set up, and then the image is executed until an `SWI &11` instruction is encountered.

Running the program again often requires that the image be reloaded into memory. There are a number of bugs in the emulator, many of which are subtle and a few that require the emulator to be restarted from scratch.