Lecture #4 – February 4, 2004

Introduction to the ARM Architecture

Registers

The ARM (Advanced Risc Machines) chip comes in several versions, but the one that we will use in this class is one of the more common, and more powerful, variants. It has sixteen 32-bit integer registers, named `R0` through `R15`, twelve of which are considered to be "general purpose" registers. The other four have special uses: `R15` is the ***program counter***, `R14` is the ***link register*** (used by subroutines, which we will discuss later), `R13` is the ***stack pointer***, and `R12` is the "instruction pointer" register, also used by subroutines. The special uses of `R13`, `R14`, and `R15` (also named `SP`, `LR`, and `PC`, respectively) are enforced by the hardware, but `R12` (named `IP`) is used as the instruction pointer only by agreed-upon programmer's conventions and has no hardware enforcement on its use.

The ARM chip also has eight ***floating-point registers*** for use in single precision or double precision arithmetic, but we will not consider those registers at this time.

In comparison to the 6502 and the 8088, having 16 general purpose registers may seem like a luxury, but as we will see no one ever has enough registers! On the 8088, arithmetic may be performed between two registers, between a register and a memory location, or between a memory location and a register (all operand pairs must be the same size). On the ARM, however, arithmetic is *always* performed between registers, and *never* involves a memory location. The only instructions that reference memory are instructions that load a register (or several registers) from memory, or store a register (or several registers) into memory.

Instructions and Addressing

The ARM can address 2^{32} bytes (4 gigabytes) of memory, so absolute memory addresses are 32 bits in length. Instructions are *always* 32 bits (4 bytes) in length, and must be aligned to addresses divisible by 4. Variables in memory that occupy four bytes must also be aligned to addresses divisible by 4. In general, assemblers will insure that instructions and data are so aligned, thus relieving the programmer of one more worry. There are only forty or so unique instruction mnemonics, depending on the version of the ARM chip.

Having both 32-bit addresses and 32-bit instructions leads to the situation where an absolute address cannot fit into any instruction (and neither can all possible constant values). Instead, absolute addresses are either stored in a register or the *offset relative to some register* is stored as part of an instruction. We will look at both of these cases at some point in the future.

The `LDR` instruction loads a 32-bit register from a 32-bit memory location (called a ***word*** on the ARM), and `STR` stores a register into memory. For example, `LDR R0,A` loads `R0` from memory at the address indicated by symbol `A`, and `STR R3,B` stores the contents of `R3` into memory at the address indicated by symbol `B`. Supposedly, you can choose whether variables are to be represented in memory in little-endian or in big-endian form, but all the ARM software I've seen so far uses little-endian exclusively. There are `LDRB` and `STRB` instructions that load and store individual 8-bit bytes, and on recent versions of the ARM chip there are also `LDRH` and `STRH` instructions to load and store 16-bit ***half-words***. Words in memory must be aligned to addresses divisible by 4, and half-words must be aligned to addresses divisible by 2.

Arithmetic Instructions

There are sixteen "basic" arithmetic instructions. (On some versions of the ARM chip there are a few other special instructions, including several different integer multiplication instructions, but those will not be considered here. There are no integer division instructions at all.) The sixteen basic arithmetic instructions are:

```
ADD – Add without Carry       Dest := Op1 + Op2
ADC - Add with Carry          Dest := Op1 + Op2 + C
SUB - Subtract without Carry  Dest := Op1 - Op2
SBC - Subtract with Carry     Dest := Op1 - Op2 + C - 1
RSB – Reverse Sub w/o Carry   Dest := Op2 - Op1
RSC - Reverse Sub with Carry  Dest := Op2 - Op1 + C - 1
AND - Logical AND             Dest := Op1 AND Op2
ORR - Logical OR              Dest := Op1 OR Op2
EOR - Logical EOR             Dest := Op1 EOR Op2
BIC - Bit Clear               Dest := Op1 AND (Not Op2)
CMP - Compare                 Op1 - Op2
CMN - Compare Negative        Op1 + Op2
TST - Bit Test                Op1 AND Op2
TEQ - Test Equal              Op1 EOR Op2
MOV - Move                    Dest := Op2
MVN - Move Negative           Dest := Not(Op2)
```

In all of these instructions, the destination (`Dest`) and the first operand (`Op1`) must be one of the sixteen registers. The second operand (`Op2`) may be a register, possibly shifted in some way, or a constant, also called an ***immediate value***. As each such instruction contains two operands and a destination (possibly all the same register) the ARM is considered to be a ***three address*** machine.

In ARM assembly language, the destination register is always written first. The instruction `ADD R4,R2,R7` means `R4 := R2 + R7`, and `ADD R0,R0,#1` means `R0 := R0 + 1`, where # indicates the presence of a constant value embedded into the instruction rather than a register name. The instruction `ADD R0,R0,R0` adds `R0` to itself and puts the result back into `R0`.

The SUB (subtract) and RSB (reverse subtract) instructions differ only in which of the two operands is subtracted from the other. As an example, `SUB R2,R4,R7` means `R2 := R4 - R7`, but `RSB R2,R4,R7` means `R2 := R7 - R4`. You might think that the reverse subtract instruction is not really needed, since you can rewrite the example using the regular subtract instruction as `SUB R2,R7,R4`, but because second operands may be constants (as well as other special forms) the reverse subtract is both useful and convenient. For example, `RSB R2,R4,#1` means `R2 := 1 - R4`, which cannot be done with a regular subtract instruction without first placing the constant 1 into an otherwise unused register.

The logical instructions AND, ORR, and EOR all perform bit-wise Boolean operations on all 32 bits of their operands, in parallel. For AND, a bit in the result will be 1 only if both operand bits are equal to 1, and will be 0 if either or both operand bits are 0. For ORR (inclusive OR) a bit in the result will be 1 if either or both operand bits are 1, and will be 0 only if both operand bits are 0. For EOR (exclusive OR) a bit in the result will be 1 if the operand bits are different, and will be 0 if the operand bits are the same.

Op1	Op2	AND	ORR	EOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

The four instructions CMP, CMN, TST, and TEQ have no destination for their results. The CMP instruction is identical to the SUB instruction, CPN is identical to ADD, TST is identical to AND, and TEQ is identical to EOR, except that in each case the result of the computation is discarded. Discarding the results may seem silly, but before each result is discarded it is used to set one or more of the ***status flags*** in the ***program status register***, or ***PSR***. These status flags are used to control the flow of a program with ***conditional branch instructions***, which will be covered in detail later. The remaining twelve arithmetic instructions optionally may be used to set the status flags, or not, as a side effect of the instructions and under the choice of the programmer.

Example

We can approach the problem of adding two memory variables together and storing the result back into memory, as in the high-level instruction `c := a + b`, in just a few ways, as shown below:

```
LDR  R0,A          LDR  R0,A          LDR  R0,A
LDR  R1,B          LDR  R1,B          LDR  R1,B
ADD  R2,R0,R1      ADD  R0,R0,R1      ADD  R1,R0,R1
STR  R2,C          STR  R0,C          STR  R1,C
```

The forms differ only in their use of the registers. The form on the far left uses three separate registers, necessary if the two original operands are to be used in other

calculations. The forms in the middle and on the right minimize the use of the registers by putting the sum back into one of the operand registers. These forms are preferred if the operands are not to be used in other calculations.

Conclusions

The ARM is a register-intensive chip. As we will learn over the next few lectures, much of the task of programming the ARM involves knowing what information is in which register, how to use the special registers `R12` through `R15`, and what instruction addressing modes simulate which functions from a high-level language.