CS 251, ARM Introduction

Stephen Mann

This document is meant as a brief introduction to the ARM assembly (the Patterson and Hennessy book uses a subset of ARM that they call "LEG"; we're really reviewing a subset of LEG here, but we'll refer to it as "ARM"). Some parts of how the ARM instructions execute have been simplified, so this document should not be considered a definitive source for ARM. Chapter 2 of the course text describes ARM in more detail, and shows how to convert C-code into ARM assembly. You may want to read Chapter 2; however, you will not need to know the ARM language in the amount of detail given in Chapter 2 for CS 251.

1 Basics

In ARM, there are 32 registers in which you can read and store data. These registers are typically referred to as X0, X1, ..., X31. Basically, you use these registers the way you would use integer variables in a program.

Note that register X31 always contains the value 0. If you write a different value to register X31, that write is ignored and register X31 continues to contain the value 0. Register X31 is also referred to as XZR.

ARM has several formats of instructions including R-Format, I-Format, D-Format, and B-Format. The format mostly refers to how many and what type the operands are (an operand is like an arugment to a function, although where you store the result of an operation is also considered an operand). An R-Format instruction has three operands, each of which is a register. An example of an R-format command is

```
ADD X1, X2, X3
```

which adds the contents of register X2 to the contents of register X3 and stores the result in register X1. The order of the operands (with the register you write to being first) may seem odd, but think of replacing the first comma with '=' and the second comma with '+', so in in this example, we have `ADD X1=X2+X3`.

An I-Format instruction has two register operands and a third operand that is an actual number. An example of an I-Format command is

```
ADDI X1, X2, #200
```

which adds the immediate value 200 to the contents of register X2 and stores the result in register X1. In general, in ARM, # preceeds a constant the is stored in the assembly

language instruction. Note that the maximum value of these constants is somewhat limited, and while some constants are signed numbers, the constant for the `ADDI` command can only be a non-negative integer. Details on the size of these constants appears later in this document.

The B-Format instructions are used for branching and the D-format are used for reading and writing to memory; both will be discussed later.

2 Memory

In addition to the 32 registers, an ARM program has access to a Random Access Memory (RAM) that is in theory is 2^{64} bytes in size (in a real ARM implementation, an ARM program has access to a smaller amount of memory). Each byte of memory can be accessed with a number from 0 to $2^{64} - 1$. However, the memory is grouped in 4-byte blocks called *words* or 8-byte blocks called *double-words*, and most memory accesses are to words or double-words of memory rather than bytes. Thus, most memory accesses are to addresses that are multiples of four or eight.

Both the ARM program and data for the program are stored in this memory. Accessing the data in memory will be discussed in a later section; in this section, we will describe how a program is stored in memory.

Each program instruction is one word (four bytes) in length, and an instruction address is a multiple of four. When writing a program, we sometimes write the memory address of an instruction, followed by the instruction:

Memory Address	Instruction
100:	`ADD X1, X2, X3`
104:	`SUB X1, X3, X5`
108:	`ADDI X2, X12, #16`

(`SUB` is the subtraction command; the subtraction command in this example computes $X3 - X5$ and stores the result in register X1).

We often don't need actual address numbers, and will instead write a symbolic label for the addresses of important instructions (e.g., instructions that we will branch to):

```
start:  ADD X1, X2, X3
        SUB X1, X3, X5
        ADDI X2, X12, #16
```

In this document, we will only use the former style of listing instructions (i.e., with a memory address associated with each instruction).

3 Control Flow

In ARM, there are no conditional statements like `if`; there are no loop constructs like `for` or `while`. Instead, program control is handled by updating a special register (the PC) using goto-like commands (branch), some of which are conditional goto's (`CBZ`, `CBNZ`). This section introduces these concepts in more detail for ARM assembly. Chapter 2 of the text book gives examples showing how to convert loops, etc., into ARM assembly instructions.

There is a special register called the *program counter* (or PC) that stores the address of the instruction that is currently executing. When we execute a command that is not a branch instruction, the PC is automatically incremented by 4, pointing it to the next command in sequence. Thus in the first code example above, the PC will initially be 100, and the command `ADD X1,X2,X3` will get executed. At the same time that this `ADD` command is executed, 4 will be added to the PC, making its value 104. Thus, the next instruction to get executed in this example will be `SUB X1,X3,X5`, with the PC being updated to 108 while this subtraction command is executed.

There are two types of branch instructions. The simplest is the *branch* instruction, `B`. The branch instruction is a B-Format command. It has a single argument that is a 26-bit number, which represents a word offset. When a branch instruction is executed, four times this number is added to the PC (the multiplication by 4 is because this number represents a word address, while the PC stores a byte address). So if the PC starts as 100, then in the following program,

```
100:  B #3
104:  ADD X1, X2, X3
108:  SUB X1, X3, X5
112:  ADDI X2, X12, #16
```

we will first execute `B #3`, which sets the PC to 112 ($100+4\times3$). So the second instruction executed is `ADDI X2, X12, #16`. Note that the constant used by `B` is a signed number, so the branch command can jump backwards in the program.

The second type of branch instruction is a *conditional branch*. An example of this CB-Format instruction is `CBZ` (compare and branch on zero):

```
CBZ X1, #100
```

Similar to the branch instruction, the constant field represents a word offset, and thus we must multiply it by 4 to convert it to a relative offset, which gets added to the PC. Note that this constant value can be negative. In this example `CBZ` instruction, the value in register X1 is compared to zero. If the value in X1 is zero, then the PC is updated to PC+(4×100). If the values are not equal, then the PC will be set to the next instruction (i.e., the PC will be set to PC+4). The command `CBNZ` is similar, except that it branches if the value in the register is non-zero.

Consider the following code, which is a loop between instructions 108 and 116:

```
      100:  ADD X1, XZR, XZR
      104:  ADDI X2, XZR, #6
Loop  108:  ADDI X1, X1, #5
      112:  SUBI X2, X2, #1
      116:  CBNZ X2, #-2
      120:  ADD X4, X6, X8
```

Assuming the PC starts at 100, the first instruction sets register X1 to 0. The second instruction (address 104) sets register X2 to 6. The third instruction (address 108) adds 5 to register X1. The fourth instruction (address 112) subtracts 1 from the register X2.

The fifth instruction (address 116) is the one that interests us. This instruction compares the value in register X2 to zero. The first time through the loop, register X2 will be 5 (it was set to 6 on line 104 and had one subtracted from it on line 112). Since the contents of register X2 are not equal to zero, the PC is updated to PC$-(2 \times 4) = 108$. The execution of the code would return to line 108 and would then execute the loop five more times; on the last time through the loop, register X2 is equal to 0 and rather than branching back to line 108, we will execute line 120.

Note that in ARM code, `B`, `CBZ`, and `CBNZ` are often written using labels rather than offsets, so you may see code like `CBNZ Loop` rather than `CBNZ #-2` in the above code example. However, for most of this class, you should use the `CBNZ #-2` form of the branch and conditional branch instructions.

4 Memory Access

32 registers (one of which is always 0!) are clearly not enough to store the data used by any reasonable program. So there are ARM instructions to allow a program to access data in RAM. Memory accesses are handled in ARM with one of two D-Format instructions: `LDUR` and `STUR`. (There are actually a few more memory access commands, but we won't be using them.)

Load Unscaled Register (`LDUR`) reads a double-word (8 bytes) from memory and stores it in a register. `LDUR`'s format is

100: LDUR X1, [X2, #100]

which has the effect of reading memory at address 100+X2 (written as `M[100+X2]`) and storing the result in register X1. Note that this is a double-word access to memory, so "100+X2" must be a multiple of 8.

Store Unscaled Register (`STUR`) takes the value of a register and writes it to memory. Its format is

100: STUR X1,[X2, #100]

which has the effect of writing the value in register X1 to memory at address 100+X2. Again, this is a double-word access to memory, so "100+X2" must be a multiple of 8.

As an example, suppose we want to add 56 to the value already stored in memory at address 120. The following ARM code will do that:

```
100:  LDUR X1, [XZR, #120]
104:  ADDI X1, X1, #56
108:  STUR X1, [XZR, #120]
```

5 Constants

Several ARM instructions have constants embedded in the instruction. These constants are different sizes (i.e., number of bits) depending on the instruction. Further, while most of these constants are signed numbers, for some instructions they are unsigned. The following table is a summary of these constants. Note that a signed constant will use one of the bits for the sign of the number.

Instruction	Bits in constant	Signed/Unsigned
ADDI/SUBI	12 bits	Unsigned
LD/ST	9 bits	Signed
B	26 bits	Signed
CBZ	19 bits	Signed

For example, the offset used by LDUR and STUR must be in the range $[-256, 255]$.

6 Simulator

We have written a simple simulator to demonstrate the execution of ARM programs in class; we have also made the simulator available for you to test your programs if you want. The simulator runs on the linux.student.cs machines and should be executed from a terminal (xterm, putty, etc.).

The simulator only executes a small number of ARM commands, only has 128 double-words of memory, and only displays 32 double-words of memory. Displayed on the screen are the values of all 32 registers, 32 double-words of RAM, and the contents of the PC. The values in memory can be displayed in decimal, hexidecimal, or as if the value in memory were ARM instructions.

To enter a program into the simulator, set the PC to the word of memory at which you wish the program to start (by typing `PC=0`, for example), and then start typing ARM command, hitting enter after each command. You should see each command appear in memory each after you hit enter.

To execute a single instruction of the program, set the PC to the word of memory that contains the instruction you wish to execute, and hit enter. Hitting enter will execute the instruction, as well as add four to the PC. In this way, you can execute a program by repeatedly pressing enter. You can execute multiple instructions by using the **run** command.

There are also commands to read and write a program to disk, set the registers, set the value of a double-word of memory, and clear the registers and memory. Type the **help** command to get a summary of the commands the simulator can execute.

The simulator knows register names X0,X1,...,X31, and also recognizes XZR, although it converts XZR to X31. You are not required to use upper case letters to specify ARM instructions or registers; the simulator will convert the case for you. You also are not required to type the '#' in front of numeric constants; the simulator will insert it for you.

The simulator will display a 32-double-word block of memory that the PC is currently on. To change the block of memory displayed, change the PC to a word in the block you would like displayed.