ARM Assembler Workbook

CS160 Computer Organization
Version 1.1
October 27th, 2002
Revised Fall 2005

ARM University Program
Version 1.0
January 14th, 1997

Introduction

Aim

This workbook provides the student with a basic, practical understanding of how to write ARM assembly language modules.

Pre-requisites

The student (that is: YOU) should be familiar with the following material:

- The ARM Instruction Set
- The ARM Command Line Toolkit Workbook

Before you continue, recall that the reference `/assembler/session1/armex.s` should be read as `~/cs160/arm/assembler/session1/armex.s`.

Building the example code

For the command line:

To build a file using the ARM assembler, issue the command:

```
armasm -g code.s
```

The object code can then be linked to produce an executable:

```
armlink code.o -o code
```

This can then be loaded into `armsd` and executed:

```
armsd code
```

Note: The assembler's `-g` option adds debugging information so that the assembly labels are visible within the debugger.

Some exercises require you to compile a piece of C code. To do so, use the ARM C compiler:

```
armcc -g -c arm.c
```

In such exercises, you will also need to link your code with the ARM C library which can be found in the `lib` subdirectory of the toolkit installation. Thus, type:

```
armlink arm.o code.o $ARMLIB/armlib.32l -o arm
```

where the l (lowercase L) suffix indicates a little-endian version of the library. There is also a big-endian version called `armlib.32b`.

Session 1: Structure of an ARM Assembler Module

The following is a simple example that illustrates some of the core constituents of an ARM assembler module. See `/assembler/session1/armex.s`.

```
      AREA ARMex, CODE, READONLY    ; name this block of code
      ENTRY                         ; mark first instruction
                                    ; to execute
start
      MOV   r0, #10                 ; Set up parameters
      MOV   r1, #3
      ADD   r0, r0, r1              ; r0 = r0 + r1
stop  SWI   0x11                    ; Terminate
      END                           ; Mark end of file
```

Description of the module

1) The `AREA` directive

Areas are chunks of data or code that are manipulated by the linker. A complete application will consist of one or more areas. This example consists of a single area which contains code and is marked as being read-only. A single `CODE` area is the minimum required to produce an application.

2) The `ENTRY` directive

The first instruction to be executed within an application is marked by the `ENTRY` directive. An application can contain only a single entry point and so in a multi-source-module application, only a single module will contain an `ENTRY` directive. Note that when an application contains C code, the entry point will often be contained within the C library.

3) General layout

The general form of lines in an assembler module is:

```
<label> <whitespace> <instruction> <whitespace> ; <comment>
```

The important thing to note is that the three sections are separated by at least one whitespace character (such as a space or a tab). Actual instructions never start in the first column, because they must be preceded by whitespace, even if there is no label. All three sections are optional and the assembler will also accept blank lines to improve the clarity of the code.

4) Code description

The application code starts executing at routine of the label `start` by loading the decimal values 10 and 3 into registers `r0` and `r1`. These registers are then added together and the result placed back into `r0`. The application then terminates using the software interrupt `0x11`, at label `stop`, which causes control to return back to the debugger.

5) The `END` directive

This directive causes the assembler to stop processing this source file. Every assembly language source module must therefore finish with this directive.

Exercise 1.1 - Running the example

Build the example file `armex.s` and load it into the debugger as described in the introduction.

Set a breakpoint on `start` and begin execution of the program. Once the breakpoint is reached, single-step through the code and display the registers after each step. You should be able to see the register contents being updated. Continue until the program terminates normally.

Exercise 1.2 - Extending the example

Modify the example so that it produces the sum (+), the difference (–) and the product (×) of the two values originally stored in `r0` and `r1`. Build the modified program and verify that it executes correctly using the debugger.

Session 2: Loading Values into Registers

The following is a simple ARM code example that attempts to load a set of values into registers. See `/assembler/session2/value.s`.

```
      AREA Value, CODE, READONLY    ; name this block of code
      ENTRY                         ; mark first instruction
                                    ; to execute
start
      MOV   r0, #0x1                ; = 1
      MOV   r1, #0xFFFFFFFF         ; = -1 (signed)
      MOV   r2, #0xFF               ; = 255
      MOV   r3, #0x101              ; = 257
      MOV   r4, #0x400              ; = 1024

stop  SWI   0x11                    ; Terminate
      END                           ; Mark end of file
```

Exercise 2.1 - What is wrong with the example?

Pass the example file `value.s` through `armasm`.

What error messages do you get and why?

[Hint: Look at the sections on using immediate values and loading 32-bit constants in the ARM Programming Techniques document.]

Exercise 2.2 - Producing a correct version of the example

Copy the example file as `value2.s` and edit this so as to produce a version that will be successfully assembled by `armasm`.

[Hint: Make use of `LDR Rn,=const` where appropriate.]

After assembling and linking, load the executable into the debugger. Set a breakpoint on `start` and begin execution of the program. Once the breakpoint is reached, display the registers. Single-step through the code until you reach `stop`, taking careful note of what instruction is being used for each load command. Look at the updated register values to see that the example has executed correctly and then execute the rest of the program to completion.

Session 3: Loading Addresses into Registers

The following is a simple ARM code example that copies one string over the top of another string. See `/assembler/session3/copy.s`.

```
      AREA  Copy, CODE, READONLY
      ENTRY             ; mark the first instruction to call
start LDR   r1, =srcstr ; pointer to first string
      LDR   r0, =dststr ; pointer to second string
strcopy                 ; copy first string over second
      LDRB  r2, [r1],#1 ; load byte and update address
      STRB  r2, [r0],#1 ; store byte and update address;
      CMP   r2, #0      ; check for zero terminator
      BNE   strcopy     ; keep going if not
stop
      SWI   0x11        ; terminate

      AREA  Strings, DATA, READWRITE
srcstr      DCB "First string - source",0
dststr      DCB "Second string - destination",0

      END
```

Notable features in the module

1) `LDR Rx, =label`

This is a pseudo-instruction that can be used to generate the address of a label. It is used here to load the addresses of `srcstr` and `dststr` into registers. This is done by the assembler allocating space in a nearby literal pool (portion of memory set aside for constants) for the address of the required label. The instruction placed in the code is actually an `LDR` instruction that will load the address in from the literal pool.

2) `DCB`

“Define Constant Byte” is an assembler directive to allocate one or more bytes of memory. It is a therefore a useful way to create a string in an assembly language module.

Exercise 3.1 - Running the example

Build the example file `copy.s` using `armasm` and load it into the debugger as described in the introduction.

Set a breakpoint on `start` and begin execution of the program. Once the breakpoint is reached, single-step through the code up to `strcpy`. Watch the addresses of the two strings being loaded into `r0` and `r1`, noting the instructions used to generate those addresses. Now set two additional breakpoints, one on `strcpy` and the other on `stop`.

Now restart execution of the program. Each time the program reaches a breakpoint, look at the updated string contents. Repeat this process until execution completes.

Session 4: Assembler Subroutines

Exercise 4.1 - Converting `copy.s` to use a subroutine

This file `copy.s` in `/assembler/session4` is the same program as that used in Exercise 3.1. Convert this version so that the code between `strcpy` and `stop` becomes a subroutine that is called by the main program using a `BL <label>` instruction. The subroutine should return using a `MOV pc,lr` instruction.

Build the converted `copy.s` using `armasm` and load it into the debugger. Follow the execution as per Exercise 3.1 to ensure that the converted `copy.s` has the same result as the original.

Session 5: Calling the Assembler from C

ARM defines an interface to functions called the ARM Procedure Call Standard (APCS). This interface specifies that the first four arguments to a function are passed in registers `r0` to `r3` (any further parameters being passed on the stack) and a single-word result is returned in `r0`. Using this standard it is possible to mix calls between C and assembler routines.

The following is a simple C program that copies one string over the top of another string, using a call to a subroutine. See `/assembler/session5/strtest.c`.

```
#include <stdio.h>
extern void strcopy(char *d, char *s);

int main() {
  char *srcstr = "First string - source ";
  char *dststr = "Second string - destination ";

  printf("Before copying:\n");
  printf("  %s\n  %s\n",srcstr,dststr);
  strcopy(dststr,srcstr);
  printf("After copying:\n");
  printf("  %s\n  %s\n",srcstr,dststr);
  return (0);
}
```

Exercise 5.1 - Extracting `strcopy` from `copy.s`

Copy the file `copy.s` produced in Exercise 4.1 into `/assembler/session5`. Now modify the file so that it only contains the subroutine `strcopy`. Note that you will also need to remove the `ENTRY` statement as the entry point will now be in C. Also add `EXPORT strcopy` so that the subroutine is visible outside of the module.

Build the application using `armcc` for `strtest.c` and `armasm` for `copy.s`, linking with the ARM C library as detailed in the introduction. Load the executable into the debugger and ensure that it functions correctly.

Session 6: Jump Tables

The following is a simple ARM code example that implements a jump table. This file can be found in `/assembler/session6/jump.s`.

```
      AREA  Jump, CODE, READONLY    ; name this block of code

num   EQU   2           ; Number of entries in jump table

      ENTRY             ; mark the first instruction to call

start MOV   r0, #0      ; set up the three parameters
      MOV   r1, #3
      MOV   r2, #2
      BL    arithfunc   ; call the function
      SWI   0x11        ; terminate

arithfunc               ; label the function
      CMP   r0, #num    ; Treat function code as unsigned integer
      BHS   DoAdd       ; If code is >=2 then do operation 0.

      ADR   r3, JumpTable     ; Load address of jump table
      LDR   pc, [r3,r0,LSL#2] ; Jump to the appropriate routine

JumpTable
      DCD   DoAdd
      DCD   DoSub

DoAdd ADD   r0, r1, r2  ; Operation 0, >1
      MOV   pc, lr      ; Return

DoSub SUB   r0, r1, r2  ; Operation 1
      MOV   pc,lr       ; Return

      END         ; mark the end of this file
```

Description of the module

The function `arithfunc` takes three arguments. The first controls the operation carried out on the second and third arguments. The result of the operation is passed back to the caller routine in `r0`. The operations the function are

`0` : Result = `argument2 + argument3`

`1` : Result = `argument2 - argument3`

Values outside this range have the same effect as value `0`.

EQU

The `EQU` assembler directive is used to give a value to a label name. In this example it assigns `num` the value `2`. Thus when `num` is used elsewhere in the code, the value `2` will be substituted (similar to using `#define` to set up a constant in C).

ADR

This is a pseudo-instruction that can be used to generate the address of a label. It is thus similar to `LDR Rx,=label` encountered earlier. However rather than using a literal pool to store the address of the label, it instead constructs the address directly by using its offset from the current program counter. It should be used with care though as it has only a limited range (255 words for a word-aligned address and 255 bytes for a byte-aligned address). It is advisable to use it only for generating addresses to labels within the same area, as the user cannot easily control how far areas will be apart at link time.

An error will be generated if the required address cannot be generated using a single instruction. In such circumstances either an `ADRL` (which generates the address in two instructions) or `LDR Rx,=label` mechanism can be used.

DCD

This declares one or more words. In this case each `DCD` stores a single word - the address of a routine to handle a particular clause of the jump table. This can then be used to implement the jump using `LDR pc, [r3,r0,LSL#2]`.

LDR pc, [r3,r0,LSL#2]

This instruction causes the address of the required clause of the jump table be loaded into the program counter. This is done by multiplying the clause number by four (to give a word offset), adding this to the address of the jump table, and then loading the contents of the combined address into the program counter (from the appropriate `DCD`).

Exercise 6.1 - Running the example

Build the example file `jump.s` using `armasm` and load it into the debugger as described in the introduction.

Set a breakpoint on `arithfunc` and begin execution of the program. Once the breakpoint is reached, verify the contents of the registers to ensure that the parameters have been set up correctly. Now single-step through the code, ensuring that the correct jump is taken based on the value in stored in `r0`. When you return from `arithfunc` to the main program, verify that the correct result has been returned. Now tell the debugger to execute the rest of the program to completion.

Reload the program and execute up to the breakpoint on `arithfunc`. Check the registers to ensure that the parameters have been set up, but alter `r0` so that another action will be carried out by the jump table. Single-step through the program again and verify that the correct path is taken for the altered parameter.

Exercise 6.2 - Logical operations

Create a new module called `gate.s` based on `jump.s`, which implements the following operations depending on the value passed through `r0`:

- `0` : Result = `argument2 AND argument3`
- `1` : Result = `argument2 OR argument3`
- `2` : Result = `argument2 EOR argument3`
- `3` : Result = `argument2 AND NOT argument3` (bit clear)
- `4` : Result = `NOT (argument2 AND argument3)`

5 : Result = `NOT (argument2 OR argument3)`
6 : Result = `NOT (argument2 EOR argument3)`
Values outside this range should have the same effect as value `0`.

Add a loop to the main program that cycles through the each of these values. Build `gate.s` using `armasm` and verify that it functions correctly.

Session 7: Block Copy

The following is a simple ARM code example that copies a set of words from a source location to a destination. See `/assembler/session7/word.s`.

```
      AREA CopyBlock, CODE, READONLY      ; name this block of code

num   EQU   20          ; Set number of words to be copied

      ENTRY             ; mark the first instruction to call

start LDR   r0, =src    ; r0 = pointer to source block
      LDR   r1, =dst    ; r1 = pointer to destination block
      MOV   r2, #num    ; r2 = number of words to copy

wordcopy
      LDR   r3, [r0], #4      ; a word from the source
      STR   r3, [r1], #4      ; store a word to the destination
      SUBS  r2, r2, #1        ; decrement the counter
      BNE   wordcopy          ; ... copy more

stop  SWI   0x11              ; and exit


      AREA Block, DATA, READWRITE

src   DCD   1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst   DCD   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

      END
```

Exercise 7.1 - Running the example

Build the example file `word.s` using `armasm` and load it into the debugger as described in the introduction.

Set breakpoints on `wordcopy` and `stop`. Begin execution of the program. Once the breakpoint on `wordcopy` is reached, check the registers to ensure that they have been set correctly and examine (using the `examine` command) the `src` and `dst` blocks of memory. Restart the program – each time a breakpoint is reached, re-examine the `src` and `dst` blocks. Continue until the program runs to completion.

Exercise 7.2 - Using multiple loads and stores

Create a new module called `block.s` based on `word.s`, which implements the block copy using `LDM` and `STM` for as much of the copying as possible. A sensible number of words to transfer at one time is eight. The number of eight-word multiples in the block to be copied can be found (if `r2` contains the number of words to be copied) using:

```
MOVS  r3, r2, LSR #3    ; number of eight word multiples
```

The number of single-word `LDR`s and `STR`s remaining after copying the eight-word multiples can be found using:

```
ANDS  r2, r2, #7        ; number of words left to copy
```

Build `block.s` using `armasm` and verify that it functions correctly by setting breakpoints on the loop containing the code to perform eight-word multiple copies as well as the code to perform single word copies. Examine the `src` and `dst` blocks of memory once a breakpoint is reached.

Continue testing your code by modifying the number of words to be copied (specified in `num`) to be 7 and then 3.

Exercise 7.3 - Extending `block.s`

Copy the file `block.s` produced in Exercise 7.2 as `block2.s`. Extend this so that once the copying of eight-word multiples has completed, if there are four or more remaining words, four-word groups will be copied using `LDM` and `STM`. In other words your code will have three sections: copy eight-word groups, copy four-word groups, copy single words.

Test your code with `num` set to 20, 7, and 3.