ARM-7 Assembly: Example Programs

CSE 2312
Computer Organization and Assembly Language Programming
Vassilis Athitsos
University of Texas at Arlington

Overview

- We are now ready to look at several types of ARM-7 instructions.
- The goal is not to cover every single instruction and feature.
- The goal is to learn enough instructions and see enough examples to be able to write some interesting code.

Hello World in Assembly

```
.globl _start
_start:
        ldr r0,=0x101f1000
        @ ASCII codes stored
        @ at [r0] get printed
        mov r1, #104        @ 'h'
        str r1,[r0]
        mov r1, #101        @ 'e'
        str r1,[r0]
        mov r1, #108        @ 'l'
        str r1,[r0]
        mov r1, #108        @ 'l'
        str r1,[r0]
        mov r1, #111        @ 'o'
        str r1,[r0]
```

```
        mov r1, #32         @ ' '
        str r1,[r0]
        mov r1, #119        @ 'w'
        str r1,[r0]
        mov r1, #111        @ 'o'
        str r1,[r0]
        mov r1, #114        @ 'r'
        str r1,[r0]
        mov r1, #108        @ 'l'
        str r1,[r0]
        mov r1, #100        @ 'd'
        str r1,[r0]

my_exit:  @do infinite loop at the end
        b my_exit
```

Hello World in Assembly, Version 2.

```
.globl _start
_start:
        ldr r0,=0x101f1000
        @ ASCII codes stored
        @ at [r0] get printed
        mov r1, #'h'      @ 'h'
        str r1,[r0]
        mov r1, #'e'      @ 'e'
        str r1,[r0]
        mov r1, #'l'      @ 'l'
        str r1,[r0]
        mov r1, #'l'      @ 'l'
        str r1,[r0]
        mov r1, #'o'      @ 'o'
        str r1,[r0]
```

```
        mov r1, #' '      @ ' '
        str r1,[r0]
        mov r1, #'w'      @ 'w'
        str r1,[r0]
        mov r1, #'o'      @ 'o'
        str r1,[r0]
        mov r1, #'r'      @ 'r'
        str r1,[r0]
        mov r1, #'l'      @ 'l'
        str r1,[r0]
        mov r1,#'d'       @ 'd'
        str r1,[r0]

my_exit:  @do infinite loop at the end
        b my_exit
```

Hexadecimal Numbers

- Hexadecimal numbers are numbers written using base-16 representation.
- Note: we use the assembler format for numbers.
  - We put # before a number.
- Example:
  - #123 is $(123)_{10}$, i.e., 123 in decimal.
  - #0x7b is $(7b)_{16}$, i.e., number 7b in hexadecimal.
  - #123 = #0x7b
- In the assembler environment we use, a lot of times it is much easier to use hexadecimal values.
  - You will see plenty of examples today.
- Thus, it is good to do a bit of a review in advance.

Hexadecimal Examples

- How do we write these numbers in hex?
  - Hex is short for hexadecimal.
- #5 = #0x???
- #9 = #0x???
- #10 = #0x???
- #11 = #0x???
- #12 = #0x???
- #13 = #0x???
- #14 = #0x???
- #15 = #0x???
- #16 = #0x???
- #17 = #0x???

Hexadecimal Examples

- How do we write these numbers in hex?
  - Hex is short for hexadecimal.
- #5 = #0x5
- #9 = #0x9
- #10 = #0xa (or #0xA)
- #11 = #0xb (or #0xB)
- #12 = #0xc (or #0xC)
- #13 = #0xd (or #0xD)
- #14 = #0xe (or #0xE)
- #15 = #0xf (or #0xF)
- #16 = #0x10
- #17 = #0x11

Hexadecimal Examples

- How do we write these numbers in hex?
  - Hex is short for hexadecimal.
- #20 = #0x???
- #25 = #0x???
- #26 = #0x???
- #31 = #0x???
- #32 = #0x???
- #33 = #0x???
- #100 = #0x???
- #1000 = #0x???

Hexadecimal Examples

- How do we write these numbers in hex?
  - Hex is short for hexadecimal.
- #20 = #0x14
- #25 = #0x19
- #26 = #0x1a (or #0x1A)
- #31 = #0x1f (or #0x1F)
- #32 = #0x20
- #33 = #0x21
- #100 = #0x64 Why? Because 100 = 6*16 + 4.
- #1000 = #0x3e8 (or #0x3E8)

  Why? Because 1000 = $3*16^2 + 14*16^1 + 8$.

Hexadecimal Examples

- How do we write these hex numbers in decimal?
- #0x8 = ???
- #0xa = ???
- #0xd = ???
- #0xf = ???
- #0x40 = ???
- #0xa0 = ???
- #0xd3 = ???
- #0xe4a = ???

Hexadecimal Examples

- How do we write these hex numbers in decimal?
- #0x8 = #8
- #0xa = #10
- #0xd = #13
- #0xf = #15
- #0x40 = #64 $64 = 4*16^1 + 0*16^0$
- #0xa0 = #160 $160 = 10*16^1 + 0*16^0$
- #0xd3 = #211 $211 = 13*16^1 + 3*16^0$
- #0xe4a = #3658 $14*16^2 + 4*16^1 + 10*16^0$

Fun with Hexadecimals

- 0xdeadbeef = ???

Fun with Hexadecimals

- 0xdeadbeef = 3735928559.
- This was (and is) a popular code for printing out an error, in cheap and small LED-based hexadecimal displays (that perhaps can only print out a characters).

Conversion Tool: Google

- Try these searches on Google:
  - 0xe4a to decimal
  - 2014 in hex

Printing Some Numbers

```
.globl _start
_start:
        ldr r4,=0x101f1000      @ r4 := 0x 101f 1000.
        @ Any ASCII code stored on r4 gets printed

        mov r0, #1
        add r0, r0, #48
        str r0, [r4]
```

- What does this program do?

Printing Some Numbers

```
.globl _start
_start:
        ldr r4,=0x101f1000      @ r4 := 0x 101f 1000.
        @ Any ASCII code stored on r4 gets printed

        mov r0, #1
        add r0, r0, #48
        str r0, [r4]
```

- What does this program do?
  - It prints "1".

Printing Some Numbers

```
.globl _start
_start:
        ldr r4,=0x101f1000      @ r4 := 0x 101f 1000.
        @ Any ASCII code stored on r4 gets printed

        mov r0, #1
        add r0, r0, 48
        str r0, [r4]
```

- What does this program do?

Printing Some Numbers

```
.globl _start
_start:
        ldr r4,=0x101f1000     @ r4 := 0x 101f 1000.
        @ Any ASCII code stored on r4 gets printed

        mov r0, #1
        add r0, r0, 48
        str r0, [r4]
```

- What does this program do?
  - It does not compile (48 should be #48).
  - If you type "make", you get:
  - test1.s:7: Error: shift expression expected -- `add r0,r0,48'

Printing Some Numbers

```
.globl _start
_start:
        ldr r4,=0x101f1000      @ r4 := 0x 101f 1000.
        @ Any ASCII code stored on r4 gets printed

        mov r0, #1
        add r0, r0, #48
        str r0, [r4]
```

- How do we modify this program to print "2" instead of "1"?

Printing Some Numbers

```
.globl _start
_start:
        ldr r4,=0x101f1000      @ r4 := 0x 101f 1000.
        @ Any ASCII code stored on r4 gets printed

        mov r0, #2
        add r0, r0, #48
        str r0, [r4]
```

- How do we modify this program to print "2" instead of "1"?

Printing Some Numbers

```
.globl _start
_start:
        ldr r4,=0x101f1000     @ r4 := 0x 101f 1000.
        @ Any ASCII code stored on r4 gets printed

        mov r0, #2
        add r0, r0, #48
        str r0, [r4]
```

- How do we modify this program to print numbers from 2 to 8?

```
.globl _start
_start:
        ldr r4,=0x101f1000
        @ ASCII codes stored
        @ at [r4] get printed

        mov r0, #2
        add r0, r0, #48
        str r0, [r4]

        mov r0, #3
        add r0, r0, #48
        str r0, [r4]

        mov r0, #4
        add r0, r0, #48
        str r0, [r4]
```

```
        mov r0, #5
        add r0, r0, #48
        str r0, [r4]

        mov r0, #6
        add r0, r0, #48
        str r0, [r4]

        mov r0, #7
        add r0, r0, #48
        str r0, [r4]

        mov r0, #8
        add r0, r0, #48
        str r0, [r4]

my_exit:  @do infinite loop at the end
        b my_exit
```

Printing Numbers 2 to 8

- To print numbers from 2 to 8, it makes sense to use a loop.
- Notice:
  - labels
  - cmp
  - bgt

```
.globl _start
_start:
        ldr r4,=0x101f1000
        @ ASCII codes stored
        @ at [r4] get printed

        mov r0, #2

my_loop:
        cmp r0, #8
        bgt my_exit
        add r1, r0, #48
        str r1, [r4]
        add r0, r0, #1
        b my_loop

my_exit:  @do infinite loop at the end
        b my_exit
```

Printing Numbers 0 to 15 in Hex

- What do we want to print?

0123456789ABCDEF

- The code on the right prints what we want.

```
.globl _start
_start:
        ldr r4,=0x101f1000        @
ASCII codes stored
        @ at [r4] get printed

        mov r0, #0

my_loop:
        cmp r0, #0xf
        bgt my_exit

        cmp r0, #10
        addlt r1, r0, #48
        addge r1, r0, #55
        str r1, [r4]
        add r0, r0, #1
        b my_loop

my_exit:  @do infinite loop at the end
        b my_exit
```

Printing 2-Digit Hex Numbers

- The code in the next slide prints all numbers from 0x98 to 0xA5
- One number per line.

```
.globl _start
_start:
        ldr r4,=0x101f1000
        @ ASCII codes stored
        @ at [r4] get printed

        mov r0, #0x98

my_loop:
        cmp r0, #0xA5
        bgt my_exit

        lsr r1, r0, #4
        and r1, r1, #0x0000000f
        cmp r1, #10
        addlt r1, r1, #48
        addge r1, r1, #55
        str r1, [r4]
```

```
        lsr r1, r0, #0
        and r1, r1, #0x0000000f
        cmp r1, #10
        addlt r1, r1, #48
        addge r1, r1, #55
        str r1, [r4]

        mov r1, #13
        str r1, [r4]
        mov r1, #10
        str r1, [r4]

        add r0, r0, #1
        b my_loop

my_exit:  @do infinite loop at the end
        b my_exit
```

Printing a 32-bit Number in Hex

- Step 1: standard initialization:

```
.globl _start
_start:
        ldr r4, =0x101f1000
        @ ASCII codes stored
        @ at [r4] get printed
```

Printing a 32-bit Number in Hex

- Step 2: store some number to r0.
- Note: the mov instruction does not allow us to use arbitrary 32-bit constants, we can only use 8-bit constants. (Why?)

```
@ set r0 := 0x12ad730f
mov r0, #0x12
lsl r0, r0, #8
add r0, r0, #0xad
lsl r0, r0, #8
add r0, r0, #0x73
lsl r0, r0, #8
add r0, r0, #0x0f
```

Printing a 32-bit Number in Hex

- Step 2, shorter (but not faster) version: store some number to r0.
- Use the ldr pseudoinstruction

```
@ set r0 := 0x12ad730f
ldr r0, =0x12ad730f
```

Printing a 32-bit Number in Hex

- Step 3: Print each digit, using a loop.
- For each of the 8 digits, starting from the leftmost digit:
  - Shift bits to the right, so that the digit becomes the rightmost.
  - Isolate that digit by taking a bitwise AND with 0x0000000f.
  - Print the digit.

```
            mov r2, #28
my_loop:
            cmp r2, #0
            blt my_exit
            lsr r1, r0, r2
            and r1, r1, #0x0000000f
            cmp r1, #10
            addlt r1, r1, #48
            addge r1, r1, #55
            str r1, [r4]
            sub r2, r2, #4
            b my_loop
my_exit:  @do infinite loop at the end
            b my_exit
```

Printing a 32-bit Number in Hex

- For example: r0 := 0x12ad730f
- First (most significant) digit: 1.
- By how many bits do we need to shift to make this digit rightmost? ??? bits.
- Register r2 holds the number of bits we need to shift.
- We do the shift.
  - Note that we store the result on another register, not r0.
  - We still need the rest of the data on r0.
- Result: r1 := ???

```
            mov r2, #28

my_loop:
            cmp r2, #0
            blt my_exit

            lsr r1, r0, r2
            and r1, r1, #0x0000000f
            cmp r1, #10
            addlt r1, r1, #48
            addge r1, r1, #55
            str r1, [r4]

            sub r2, r2, #4
            b my_loop

my_exit:  @do infinite loop at the end
            b my_exit
```

Printing a 32-bit Number in Hex

- For example: r0 := 0x12ad730f
- First (most significant) digit: 1.
- By how many bits do we need to shift to make this digit rightmost? 28 bits.
- Register r2 holds the number of bits we need to shift.
- We do the shift.
- Result: r1 := 0x00000001.
- Now, we do bitwise AND between r1 and 0x0000000f.
- Result: 0x00000002.
- We have managed to isolate the digit, ready to print it.

```
        mov r2, #28
my_loop:
        cmp r2, #0
        blt my_exit
        lsr r1, r0, r2
        and r1, r1, #0x0000000f
        cmp r1, #10
        addlt r1, r1, #48
        addge r1, r1, #55
        str r1, [r4]
        sub r2, r2, #4
        b my_loop
my_exit:  @do infinite loop at the end
        b my_exit
```

Printing a 32-bit Number in Hex

- example: r0 := 0x12ad730f
- Second digit: 2.
- By how many bits do we need to shift to make this digit rightmost? ??? bits.
- Register r2 := r2 - 4, to hold the number of bits we need to shift.
- We do the shift.
- Result: r1 := ???
- Now, we do bitwise AND between r1 and 0x0000000f.
- Result: ???.

```
        mov r2, #28
my_loop:
        cmp r2, #0
        blt my_exit
        lsr r1, r0, r2
        and r1, r1, #0x0000000f
        cmp r1, #10
        addlt r1, r1, #48
        addge r1, r1, #55
        str r1, [r4]
        sub r2, r2, #4
        b my_loop
my_exit:  @do infinite loop at the end
        b my_exit
```

Printing a 32-bit Number in Hex

- example: r0 := 0x12ad730f
- Second digit: 2.
- By how many bits do we need to shift to make this digit rightmost? 24 bits.
- Register r2 := r2 - 4, to hold the number of bits we need to shift.
- We do the shift.
- Result: r1 := 0x00000012
- Now, we do bitwise AND between r1 and 0x0000000f.
- Result: 0x00000002.
- We have managed to isolate the digit, ready to print it.

```
        mov r2, #28
my_loop:
        cmp r2, #0
        blt my_exit
        lsr r1, r0, r2
        and r1, r1, #0x0000000f
        cmp r1, #10
        addlt r1, r1, #48
        addge r1, r1, #55
        str r1, [r4]
        sub r2, r2, #4
        b my_loop
my_exit:  @do infinite loop at the end
        b my_exit
```

Printing a 32-bit Number in Hex

- example: r0 := 0x12ad730f
- Third digit: a.
- By how many bits do we need to shift to make this digit rightmost? ??? bits.
- Register r2 := r2 - 4, to hold the number of bits we need to shift.
- We do the shift.
- Result: r1 := ???
- Now, we do bitwise AND between r1 and 0x0000000f.
- Result: ???

```
        mov r2, #28
my_loop:
        cmp r2, #0
        blt my_exit
        lsr r1, r0, r2
        and r1, r1, #0x0000000f
        cmp r1, #10
        addlt r1, r1, #48
        addge r1, r1, #55
        str r1, [r4]
        sub r2, r2, #4
        b my_loop
my_exit:  @do infinite loop at the end
        b my_exit
```

Printing a 32-bit Number in Hex

- example: r0 := 0x12ad730f
- Third digit: a.
- By how many bits do we need to shift to make this digit rightmost? 20 bits.
- Register r2 := r2 - 4, to hold the number of bits we need to shift.
- We do the shift.
- Result: r1 := 0x0000012a
- Now, we do bitwise AND between r1 and 0x0000000f.
- Result: 0x0000000a.
- We have managed to isolate the digit, ready to print it.

```
            mov r2, #28
my_loop:
            cmp r2, #0
            blt my_exit
            lsr r1, r0, r2
            and r1, r1, #0x0000000f
            cmp r1, #10
            addlt r1, r1, #48
            addge r1, r1, #55
            str r1, [r4]
            sub r2, r2, #4
            b my_loop
my_exit:  @do infinite loop at the end
            b my_exit
```

Infinite Loops

- Why do we always put an infinite loop at the end?
- Otherwise, some programs keep rerunning from the beginning.
- I have verified that even correct code can get into this behavior.
- This phenomenon is somewhat complicated to explain, but it is easy to fix.

Infinite Loops

- First of all, how to fix:

Short version:

- At the very end of your program (every program that you write), put these lines (or something equivalent):

```
the_end:
    b the_end
```

- These lines make sure that your program, when it reaches the end, stays there forever.
  - No extra output is produced.

Infinite Loops

Longer version:

- At the very end of your program (every program that you write), put the code on the right.
- This way, when you get to the end of the program, you see the word END printed.
  - You know that you reached the end of your program (as opposed to getting stuck in some infinite loop somewhere else, due to a bug).

```
        ldr r4,=0x101f1000
        mov r1, #’\r’
        str r1, [r4]
        mov r1, #’\n’
        str r1, [r4]
        mov r1, #'E'
        str r1, [r4]
        mov r1, #'N'
        str r1, [r4]
        mov r1, #'D'
        str r1, [r4]

the_end:
        b the_end
```

Infinite Loops

- The assembly programs that we write run in a very primitive environment.
- How does a program know when to stop?
- What does the CPU execute when the program stops?

Infinite Loops

- The assembly programs that we write run in a very primitive environment.
- How does a program know when to stop?
- What does the CPU execute when the program stops?
- These are issues that are typically handled by an operating system.
- In our case, the the program runs on a simulated machine with no operating system.
- When the program finishes, what is the CPU supposed to do?

Infinite Loops

- When the program finishes, what is the CPU supposed to do?
- The CPU just fetches the next instruction from memory.
- What is the next instruction?
- It is just whatever happened to reside in memory at that time.
- Thus, while you think that your program has finished executing, the program still executes meaningless instructions.
- However, at some point, the program may reach memory that you have used on your stack.
- Some of the data you have stored on the stack, when interpreted as instructions, executes a branch to the beginning of the program.

Infinite Loops

- In summary: correct code getting into an infinite loop is a problem that you may or may have not run across.
- If you have not run across it,do not worry about it.
- If you have, it may take hours trying to find the mistake where there isn't one.
- Using the suggested fixes (especially the one that prints END at the end) resolves this issue.
- Plus, using the suggested fixes ensures that if you do observe an infinite loop, the problem is with your code.
- If you mess up your stack (by adding or subtracting the wrong values, or restoring the value of lr from the wrong place) you may get all sorts of weird execution behavior.