

# More on Assembly

CSE 2312

Computer Organization and Assembly Language Programming

Vassilis Athitsos

University of Texas at Arlington

# Pseudo-Instructions

- A pseudo-instruction is something that:
  - When you read assembly code, it looks like a regular instruction.
  - When it is translated to binary code, it is translated to multiple actual CPU instructions.
- Pseudo-instructions are handy.
  - We can write code that is shorter, easier to read, easier to test.
- However, if we care about performance:
  - We must always be aware of the difference between an actual instruction and a pseudo-instruction.
  - We should know how the pseudo-instruction is translated.

# The LDR Pseudo-instruction

- `ldr rd, [rn]`:
- `ldr rd, [rn, #constant]`
  - **ldr** is a regular instruction, that we have been using.
- However, `ldr` can also be used as a pseudo-instruction:
- `ldr rd, =constant.`
  - In this usage, `constant` is a 32-bit number, written in decimal or hexadecimal.
- Example: `ldr r5, =2014.`
  - Sets `r5 = 2014.`

# The LDR Pseudo-instruction

- Example: `ldr r5, =2014`.
  - Sets `r5 = 2014`.
- Why do we need this pseudo-instruction?
- Why can't we just write: `mov r5, #2014`
- `mov r5, constant` is a real instruction, that gets translated to a single CPU instruction.
- For that to be possible, constant must obey certain rules (must be 8 bits, possibly shifted to the left).

# The LDR Pseudo-instruction

- The ldr pseudo-instruction allows us to replace bulky code like:

```
mov r5, #0x7f
lsl r5, r5, #8
add r5, r5, #0xff
lsl r5, r5, #8
add r5, r5, #0xff
lsl r5, r5, #8
add r5, r5, #0xf0
```

- with a single line:

```
ldr r5, =7fffffff0
```

# The PUSH/POP Pseudo-instructions

- A typical function preamble looks like this:

```
sub sp, sp, #20
```

```
str lr, [sp, #0]
```

```
str r0, [sp, #4]
```

```
str r4, [sp, #8]
```

```
str r5, [sp, #12]
```

```
str r6, [sp, #16]
```

- This can all be replaced with this pseudo-instruction:

```
push {r0, r4, r5, r6, lr}
```

# The PUSH/POP Pseudo-instructions

```
push {r0, r4, r5, r6, lr}
```

- The push pseudo-instruction is translated as follows:
  - Decrement the stack pointer as much as is needed to make room for the list of registers that is provided.
  - Save the provided list of registers into the stack.

# The PUSH/POP Pseudo-instructions

- Similarly, a typical function wrapup looks like this:

```
ldr lr, [sp, #0]
```

```
ldr r0, [sp, #4]
```

```
ldr r4, [sp, #8]
```

```
ldr r5, [sp, #12]
```

```
ldr r6, [sp, #16]
```

```
add sp, sp, #20
```

```
bx lr
```

- This can all be replaced with this pseudo-instruction:

```
pop {r0, r4, r5, r6}
```



# The PUSH/POP Pseudo-instructions

- This line will produce a compiler warning:

```
push {lr, r0, r4, r5, r6}
```

- The warning says:

```
test1.s:20: Warning: register range not in ascending order
```

- The compiler wants you to order the list of registers in ascending order.
- Register lr is really register r14, so should come after the other registers in the list:

```
push {r0, r4, r5, r6, lr}
```

# Assembly Directives

- A directive is a line that does not specify an instruction, but provides other pieces of information.
- For the time being, we will cover these directives:
  - equ
  - ascii
  - asciz
  - byte
  - word

# The EQU directive

```
.equ IO_ADDRESS, 0x101f1000
```

```
... (possible other code in between)
```

```
ldr r4, =IO_ADDRESS
```

- The equ directive allows us to give names to constants.
- This helps make the code more readable.
- It also helps with editing the code faster.
  - To change the value of the constant, we just need to change the .equ line that assigns a name to the constant.
  - Otherwise, we need to change the value of the constant in every single line that uses the constant.

# The ASCII and BYTE Directive

string\_hello:

```
.ascii "hello world"
```

```
.byte 0x00
```

- The above lines of code define a memory location that can be referred by the rest of the code as string\_hello.
- This memory location has the following contents:
  - The ASCII codes for the characters in "hello world", followed by:
  - A byte with content 0x00.

# Example of Usage

```
ldr r4, =IO_ADDRESS          @ r0 := 0x 101f 1000
lrd r5, =string_hello
```

print\_str:

```
ldrb r6,[r5]
cmp r6,#0x00                 @ '\0' = 0x00: null character?
beq print_done              @ if yes, quit
str r6,[r4]                  @ otherwise, write character
add r5,r5,#1                 @ go to next character
b print_str                  @ repeat
```

print\_done:

# The ASCIZ Directive

- Strings typically have a '\0' character (ASCII code 0) at their end.
- Instead of having to specify manually the null character at the end of the string, we can use the .asciz directive.
- For example, instead of writing

string\_hello:

```
.ascii "hello world"
```

```
.byte 0x00
```

- We can just write:

string\_hello:

```
.asciz "hello world"
```

# The WORD Directive

- The byte directive specifies a byte of memory.
- The word directive specifies a word of memory.
- A word is 4 bytes in the ARM-7 architecture.
- For example, consider this line of code in C:

```
int * array1 = {2014, 1914, 1814, 1714};
```

- In assembly, you can write:

```
array1:
```

```
    .word 2014
```

```
    .word 1914
```

```
    .word 1814
```

```
    .word 1714
```

# Negative Numbers

- In most modern architectures, negative numbers are represented using what is called "two's complement".
- Suppose that your registers hold N bits.
- The two's complement representation of number -X is:  $2^N - X$ .



# Negative Numbers: Examples

- The two's complement representation of number  $-X$  is:  $2^N - X$ .
- In ARM-7,  $N = 32$ .
- How is number  $-1$  represented:
  - In binary?
  - In hexadecimal?

# Negative Numbers: Example

- The two's complement representation of number  $-X$  is:  $2^N - X$ .
- In ARM-7,  $N = 32$ .
- How is number  $-1$  represented:
- In binary:
  - $2^{32}-1 = 1\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 - 1$   
=  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111$
- In hexadecimal:
  - $2^{32}-1 = 1\ 0000\ 0000 - 1$   
=  $0x\ ffff\ ffff$
- If you call `print_number` with  $-1$ , you see `0xffffffff`.

# Input

- We have already seen how to print:
  - What we have to do is store each ASCII code that we want to print into a specific memory address: 0x101f1000
- To get text input from the user, we do something similar:
  - We load an ASCII code from the same memory address: 0x101f1000
- However, input is a little bit more complicated than output.
- How does the program know that we have something to print?
  - Easy: when the program hits an instruction that stores something to the designated address.
- How does the program know that the user has entered text?
  - Not so easy: the program hits an instruction that loads something from the designated address.
  - However, is there something at that address? Has the user typed something yet?

# Input

- To read a character of text, we will follow a simple approach (more sophisticated/efficient approaches are available):
  - Wait until the user has entered a character.
  - When the user has entered the character, read the character.
- How can we know that the user has entered a character?
- There is a specific memory address, that holds a specific bit, that:
  - Is automatically set to 1 when the user enters some text.
  - Is automatically set to 0 when we read that text.
- This specific memory address is: 101f1018
- The bit at position 4 in that address is set to 1 when the user has entered data.
  - (Bit 0 is the least significant bit).

# Input

- To read a character of text, we will follow a simple approach (more sophisticated/efficient approaches are available):
  - Wait until the user has entered a character.
  - When the user has entered the character, read the character.
- How can we know that the user has entered a character?
- There is a specific memory address, that holds a specific bit, that:
  - Is automatically set to 1 when the user enters some text.
  - Is automatically set to 0 when we read that text.
- This specific memory address is: 101f1018
- The bit at position 4 in that address is set to 1 when the user has entered data.
  - (Bit 0 is the least significant bit).