GNU Assembly Programming

Humayun Kabir
Professor, CS, Vancouver Island University, BC, Canada

GNU Assembly Programming: Outline

- Need for Assembly Programming
- Elements of Assembly
- GNU Assembly Program Directives

Need for Assembly Programming

- Writing computer booting codes.
- Writing code for the machines where no compiler exists.
- Writing interrupt handler codes.
- Writing low-level locking codes for multi-threaded programs.
- Writing the compiler code generator.
- Writing code for the computer that has very limited memory and the compiler generated codes are not small and efficient enough.
- Writing codes to access low-level architectural features of a specialized processor.

Hello World Assembly Program

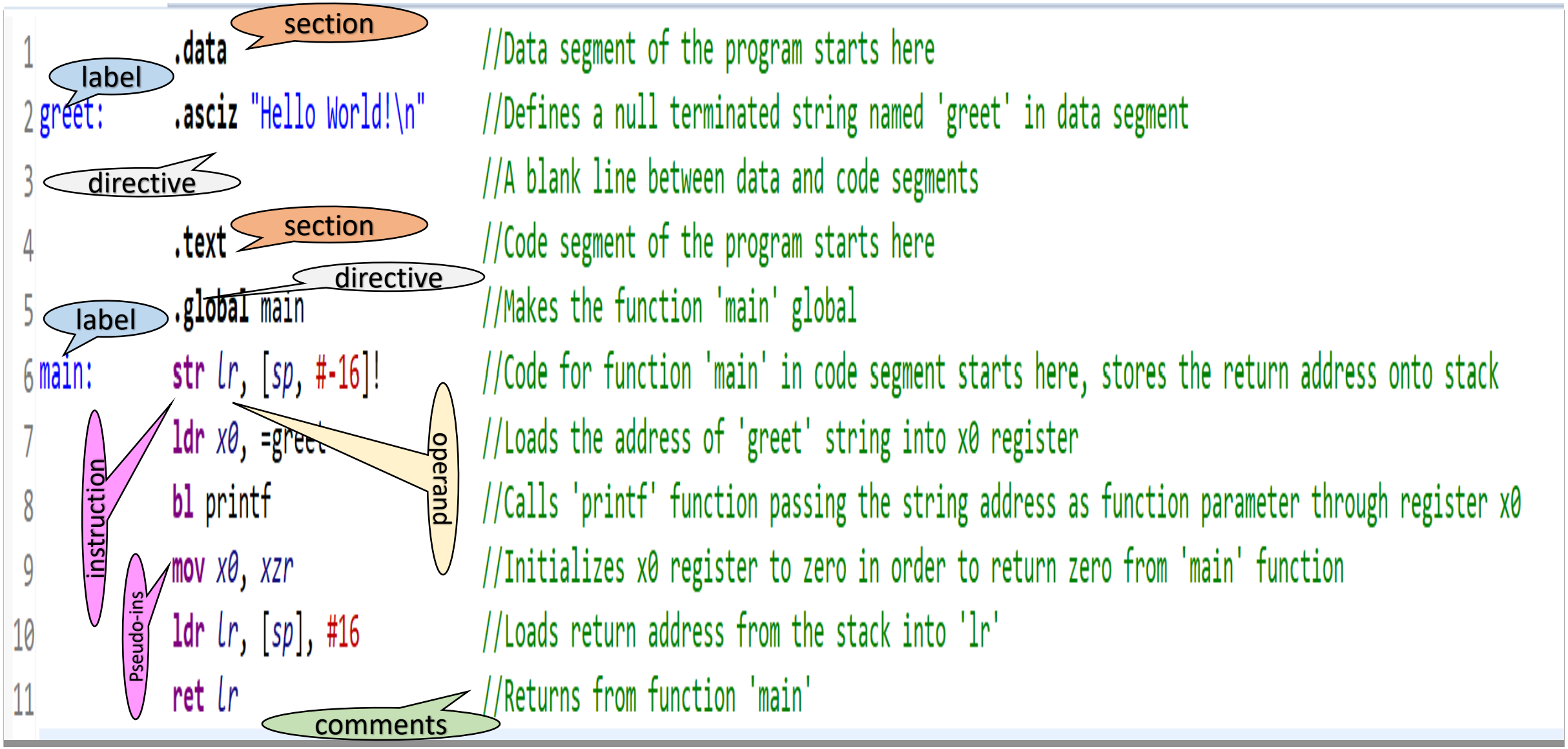
```
1           .data                   //Data segment of the program starts here
2 greet:    .asciz "Hello World!\n" //Defines a null terminated string named 'greet' in data segment
3                                   //A blank line between data and code segments
4           .text                   //Code segment of the program starts here
5           .global main            //Makes the function 'main' global
6 main:     str lr, [sp, #-16]!     //Code for function 'main' in code segment starts here, stores the return address onto stack
7           ldr x0, =greet          //Loads the address of 'greet' string into x0 register
8           bl printf               //Calls 'printf' function passing the string address as function parameter through register x0
9           mov x0, xzr             //Initializes x0 register to zero in order to return zero from 'main' function
10          ldr lr, [sp], #16       //Loads return address from the stack into 'lr'
11          ret lr                  //Returns from function 'main'
```

[label:] [directive or instruction] // comment

Elements of Assembly

- Sections
- Symbols
  - Labels
  - Operands
- Assembler directives
- Instructions and pseudo-instructions
- Comments

Elements of Assembly




```
1           .data                   //Data segment of the program starts here
2   greet:  .asciz "Hello World!\n" //Defines a null terminated string named 'greet' in data segment
3                                   //A blank line between data and code segments
4           .text                   //Code segment of the program starts here
5           .global main            //Makes the function 'main' global
6   main:   str lr, [sp, #-16]!     //Code for function 'main' in code segment starts here, stores the return address onto stack
7           ldr x0, =greet          //Loads the address of 'greet' string into x0 register
8           bl printf               //Calls 'printf' function passing the string address as function parameter through register x0
9           mov x0, xzr             //Initializes x0 register to zero in order to return zero from 'main' function
10          ldr lr, [sp], #16       //Loads return address from the stack into 'lr'
11          ret lr                  //Returns from function 'main'
```

GNU Assembler Directives

- All assembler directives begin with a period (.)
- The rest of the name is composed of letters, usually in lower case
- Instructs the GNU Assembler during its assembly process on
  - Controlling the Sections
  - Allocating space for Variables and Constants
  - Setting and manipulating symbols
  - Filling and Aligning memory space
  - Conditional assembly
  - Defining Macros

GNU Assembler Directives: Sections

.data *subsection*

Tells the assembler to assemble the following statements onto the end of the data subsection

.text *subsection*

Tells the assembler to assemble the following statements onto the end of the text subsection.

.bss *subsection*

Tells the assembler to assemble the following statements onto the end of the bss subsection.

.section *name*

Tells the assembler to create custom section. Rarely needed.

GNU Assembler Directives: Allocating Strings

.ascii "string"

Expects zero or more string literals separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

.asciz "string"

.string "string"

Like .ascii, but each string is followed by a zero byte. The “z” in .asciz stands for zero. .string is an alias for .asciz.

greet: .**asciz** *“Hello World!”*

Sets *“Hello World!”* to symbol **greet**.

GNU Assembler Directives: Allocating Integers

.word *expressions*

.long *expressions*

Takes zero or more expressions, separated by commas and emits 32-bit numbers for each expression value and the address is advanced accordingly. If no expression is given, the address is not advanced. .long and .word are the synonymous.

.hword *expressions*

.short *expressions*

Like .word but emits a 16-bit numbers for each expression. .hword and .short are synonymous.

.byte *expressions*

Like .word but emits 8-bit numbers for each expression.

GNU Assembler Directives: Allocating Floating Point

.float *flonums*

.single *flonums*

Assembles zero or more 4-byte single precision floating point numbers on ARM. .float and .single are synonyms.

.double *flonums*

Assembles zero or more 8-byte double precision floating point numbers on ARM.

GNU Assembler Directives: Setting and Manipulating Symbols

.equ *symbol, expression*

.set *symbol, expression*

This directive sets the value of *symbol* to *expression*.

.equiv *symbol*, *expression*

The .equiv directive is like .equ and .set, except that the assembler will signal an error if symbol is already defined.

.**equ** max *1024*

sets 1024 to symbol max.

.**equiv** max *2048*

does not set 2048 to symbol max since it has already been set before.

GNU Assembler Directives: Setting and Manipulating Symbols

.global *symbol*
.globl *symbol*

This directive makes the symbol visible to the linker. If you define *symbol* in your partial program, its value is made available to other partial programs that are linked with it.

```
.global main
        makes main symbol visible to the linker.
```

GNU Assembler Directives: Filling Memory Space

.skip *size*, *fill*

.space *size*, *fill*

This directive emits size bytes, each of value fill. Both size and fill are absolute expressions. If the comma and fill are omitted, fill is assumed to be zero. .space and .skip are equivalent.

.skip 4, 0 advances the location counter by **4** and pads the advanced bytes with **0**.

GNU Assembler Directives: Aligning Memory Space

.align *advance*, *fill*, *max*

- Pad the location counter (in the current subsection) to a particular storage boundary.
- *advance* specifies the number of low-order zero bits of the location counter must have set after advancement.
- *fill* gives the fill value to be stored in the padding bytes.
- *max* is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all.

Both .align and .skip directives can be used alternatively.

GNU Assembler Directives: Aligning Memory Space

.align 3, 0 advances the location counter until it's a multiple of **8** and if the location counter is already a multiple of 8, no change is needed. If advance is successful, pads the advanced bytes with **0**.

.align 2, 0 advances the location counter until it's a multiple of *4* and if the location counter is already a multiple of 4, no change is needed. If advance is successful, pads the advanced bytes with **0**.

.align 1, 0 advances the location counter until it's a multiple of *2* and if the location counter is already a multiple of 2, no change is needed. If advance is successful, pads the advanced bytes with **0**.

GNU Assembler Directives: Aligning Memory Space

.align 3, 0, 4 advances the location counter until it's a multiple of **8** and if the location counter is already a multiple of 8, no change is needed. If it requires more than **4** bytes to advance, it does not advance at all. If advance is successful, pads the advanced bytes with **0**.

GNU Assembler Directives: Conditional Assembly

.if *argument*
.else
.endif

.if marks the beginning of a section of code which is only considered part of the source program being assembled if the *argument* (which must be an absolute expression) is non-zero. The end of the conditional section of code must be marked by the .endif directive. Optionally, you may include code for the alternative condition, flagged by the .else directive.

GNU Assembler Directives: Conditional Assembly

```
.set ARM8 1

.if ARM8
ldr x0, [x0, #8]
.else
ldr r0, [r0, #4]
.endif
```

Assembles to
`ldr x0, [x0, #8]`

```
.set ARM8 0

.if ARM8
ldr x0, [x0, #8]
.else
ldr r0, [r0, #4]
.endif
```

Assembles to
`ldr r0, [r0, #4]`

GNU Assembler Directives: Include

.include "file"

- This directive provides a way to include supporting files at specified points in your source program. The code from file is assembled as if it followed the point of the .include. Quotation marks are required around file.
- When the end of the included file is reached, assembly of the original file continues.
- You can control the search paths used with the '-I' command line parameter.
- This is a good way to include files containing macros and other definitions. It is similar to including header files in C and C++.

GNU Assembler Directives: Macro

.macro *macname macargs ...*

Begin the definition of a macro called *macname*. If your macro definition requires arguments, specify their names after the macro name, separated by commas or spaces. You can supply a default value for any macro argument by following the name with ‘=default’.

.endm

Mark the end of a macro definition.

.exitm

Exit early from the current macro definition.

GNU Assembler Directives: Macro

Macro Definition

```
.macro SHIFT a,b
mov \a, #\b
.if \b<0
asr \a, \a, #-\b
.else
lsl \a, \a, #\b
.endif
.endm
```

Macro Expansions

```
SHIFT x1, 3
```

```
MOV     x1,#3
LSL     x1,x1,#3
```

```
SHIFT x4, -6
```

```
MOV     x4,#0xfffffffffffffffa
ASR     x4,x4,#6
```