ARM Instruction Set Architecture (I)

Lecture 5

Yeongpil Cho

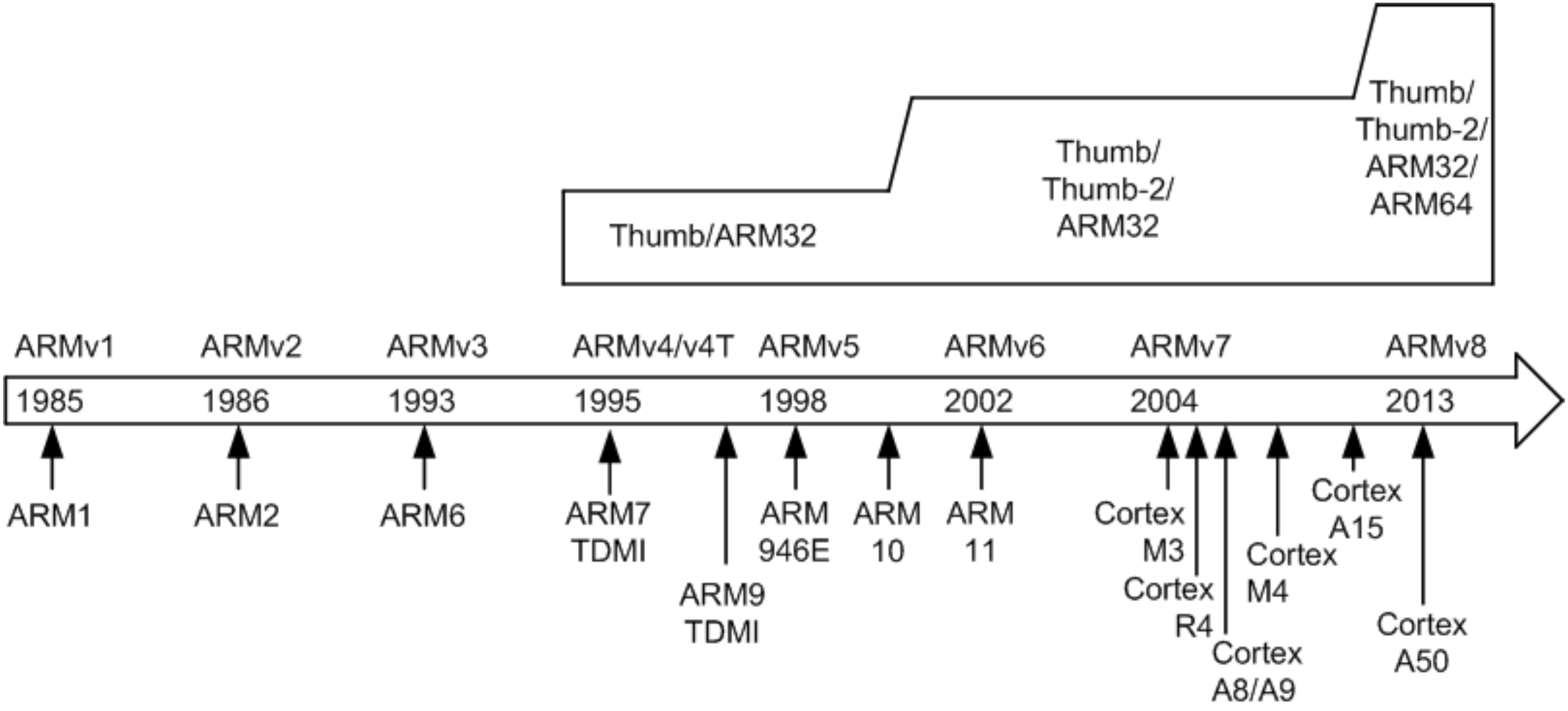
Hanynag University

Topics

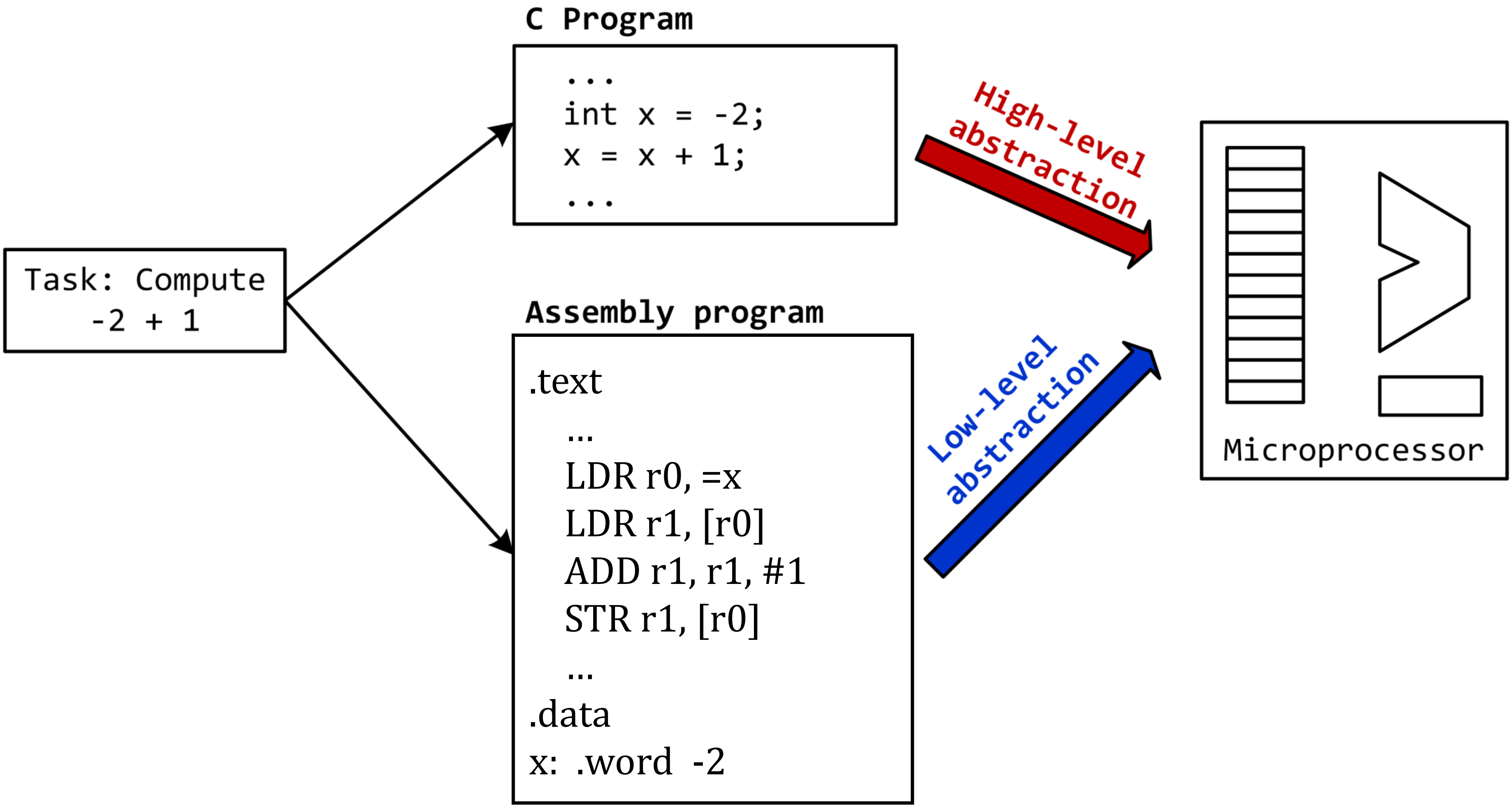
- Structure of ARM Assembly Code
- Assembler Directives

ARM Instruction Set Architecture

History of ARM Instruction Sets



From C to Assembly



Task: Compute -2 + 1

C Program

```
...
int x = -2;
x = x + 1;
...
```

Assembly program

```
.text
    ...
    LDR r0, =x
    LDR r1, [r0]
    ADD r1, r1, #1
    STR r1, [r0]
    ...
.data
x:  .word  -2
```

High-level abstraction

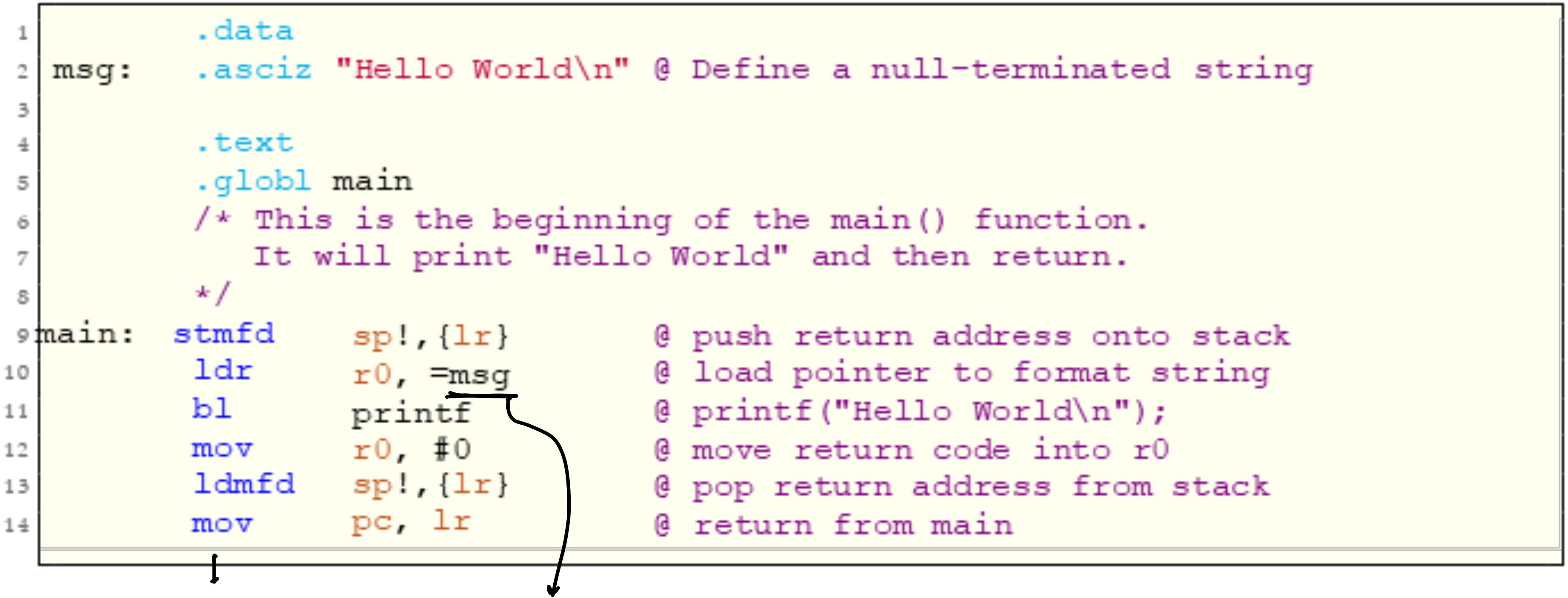
Low-level abstraction

Microprocessor

Two Pass Assembly

- Most assemblers read the source file twice.
- First Pass:
  - Build a symbol table.
  - Calculate and record values (to be used in the second pass) for each symbol.
  - Some symbols may not have a known value.
    - They are unresolved.
    - The linker will have to fix them.
- Second Pass:
  - Generate the object file.
  - Use the symbol table to provide values when needed.
  - Add information to the object file to tell the linker about any symbols that are unresolved.

Example Code

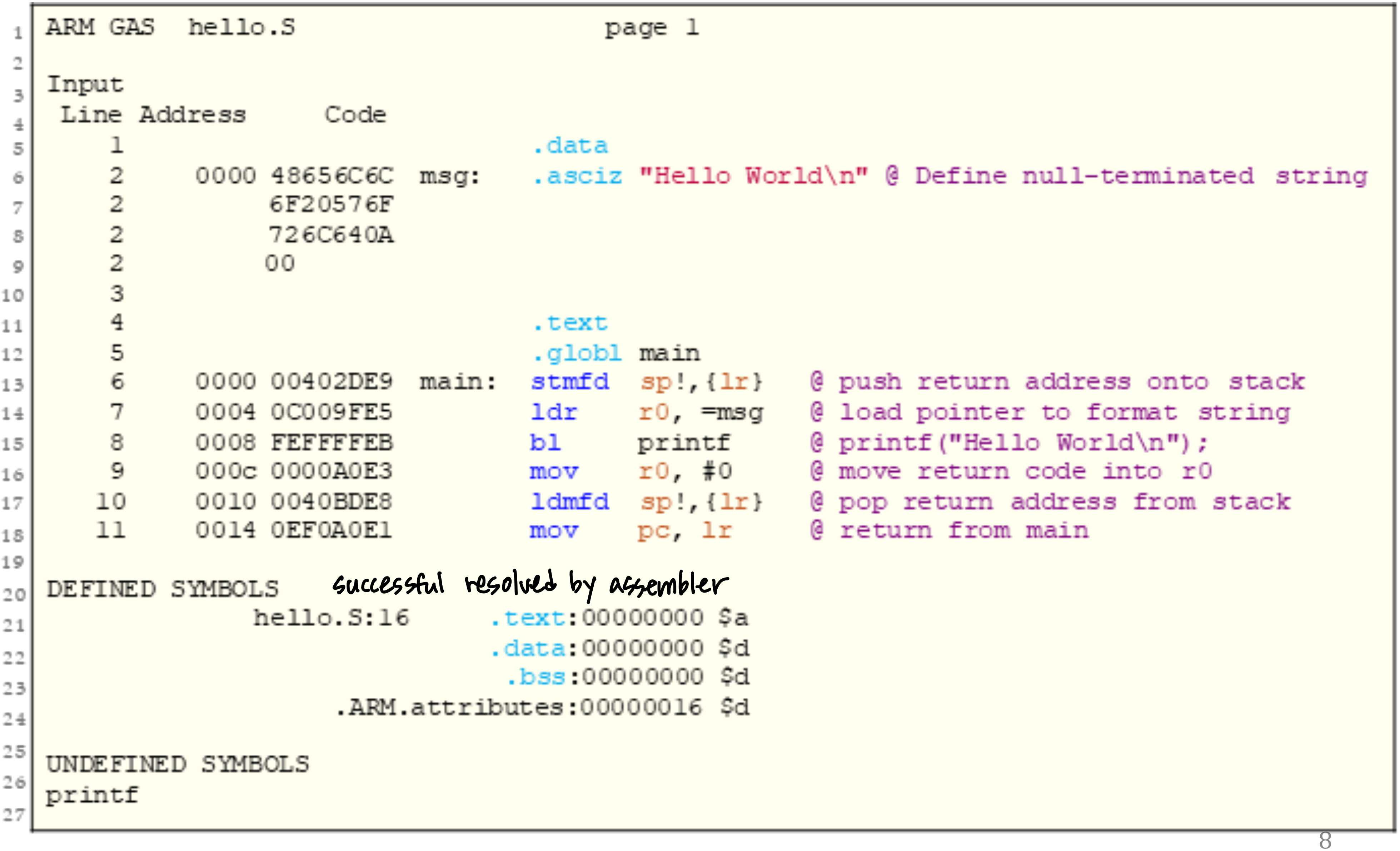


```
        .data
msg:    .asciz "Hello World\n" @ Define a null-terminated string

        .text
        .globl main
        /* This is the beginning of the main() function.
           It will print "Hello World" and then return.
        */
main:  stmfd    sp!,{lr}        @ push return address onto stack
        ldr     r0, =msg        @ load pointer to format string
        bl      printf          @ printf("Hello World\n");
        mov     r0, #0          @ move return code into r0
        ldmfd   sp!,{lr}        @ pop return address from stack
        mov     pc, lr          @ return from main
```

ARM instruction

real memory location으로 interpret (?) 필요

Assembly Listing

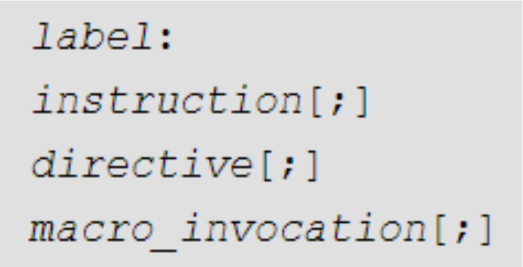


```
ARM GAS  hello.S                                page 1

Input
 Line Address      Code
    1                              .data
    2     0000 48656C6C  msg:     .asciz "Hello World\n" @ Define null-terminated string
    2          6F20576F
    2          726C640A
    2          00
    3
    4                              .text
    5                              .globl main
    6     0000 00402DE9  main:    stmfd  sp!,{lr}     @ push return address onto stack
    7     0004 0C009FE5            ldr    r0, =msg     @ load pointer to format string
    8     0008 FEFFFFEB            bl     printf       @ printf("Hello World\n");
    9     000c 0000A0E3            mov    r0, #0       @ move return code into r0
   10     0010 0040BDE8            ldmfd  sp!,{lr}     @ pop return address from stack
   11     0014 0EF0A0E1            mov    pc, lr       @ return from main

DEFINED SYMBOLS
               hello.S:16     .text:00000000 $a
                              .data:00000000 $d
                               .bss:00000000 $d
                   .ARM.attributes:00000016 $d

UNDEFINED SYMBOLS
printf
```

successful resolved by assembler

Syntax of Assembly

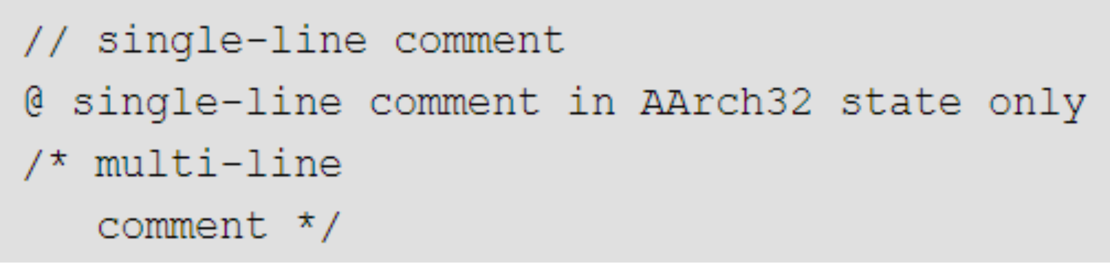
- Basic syntax



```
label:
instruction[;]
directive[;]
macro_invocation[;]
```

  - Label statements end after the ":"
  - The other statements end at the first newline or ";"
  - Directives start with a period "."

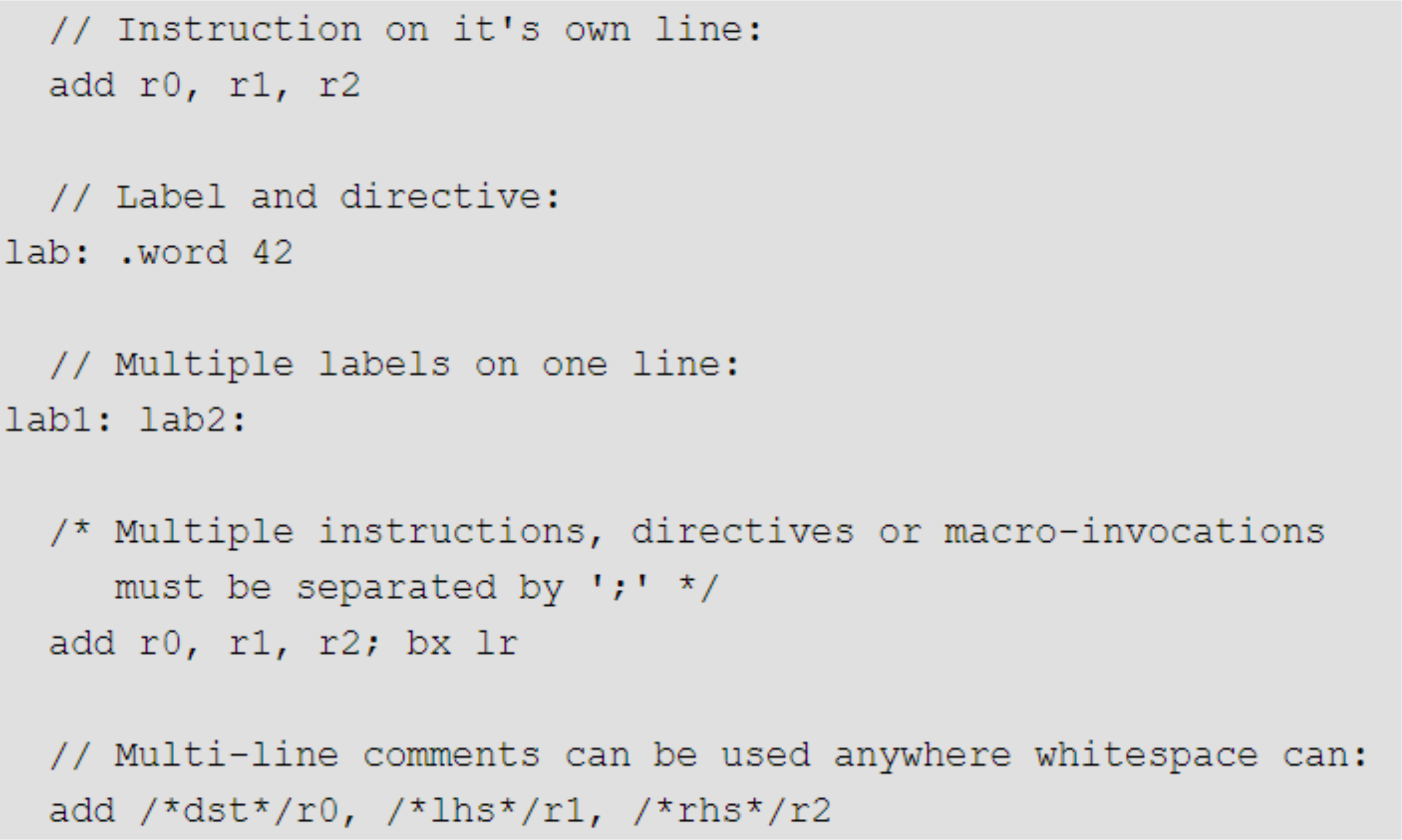
- Comments



```
// single-line comment
@ single-line comment in AArch32 state only
/* multi-line
   comment */
```

Syntax of Assembly

- Examples



```
  // Instruction on it's own line:
  add r0, r1, r2

  // Label and directive:
lab: .word 42

  // Multiple labels on one line:
lab1: lab2:

  /* Multiple instructions, directives or macro-invocations
     must be separated by ';' */
  add r0, r1, r2; bx lr

  // Multi-line comments can be used anywhere whitespace can:
  add /*dst*/r0, /*lhs*/r1, /*rhs*/r2
```

Assembly Expressions

- Expressions consist of one or more integer literals or symbol references, combined using operators.
- Expression can be used as instruction operands or directive argument.
- Assembler evaluates all expressions.

Assembly Expressions

- Constants
  - Decimal Integer
  - Hexadecimal integer, prefixed with 0x
  - Octal integer, prefixed with 0
  - Binary integer, prefixed with 0b

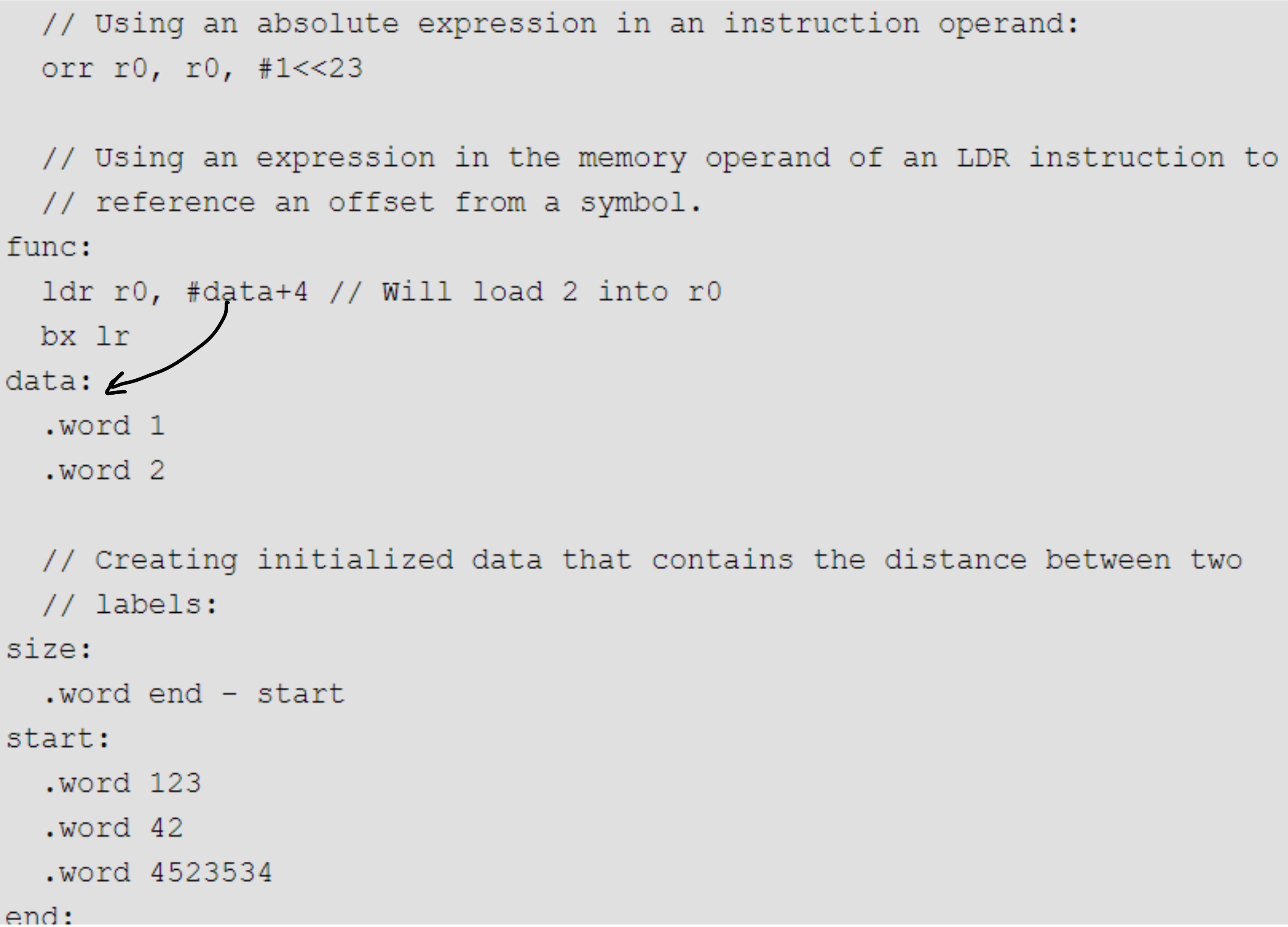
  - Negative numbers can be represented using the unary operator, -
- Symbol References
  - Symbols do not need to be defined in the same assembly language source file, to be referenced in expressions.
  - The period symbol (.) is a special symbol that can be used to reference the current location in the output file.

Assembly Expressions

- Operators
  - Unary Operators: +, -, ~
  - Binary Operators: +, -, *, /, %
  - Binary Logical Operators: &&, ||
  - Binary Bitwise Operators: &. |, ^, >>, <<
  - Binary Comparison Operators: ==, !=, <, >, <=, >=

Assembly Expressions

- Examples



```
  // Using an absolute expression in an instruction operand:
  orr r0, r0, #1<<23

  // Using an expression in the memory operand of an LDR instruction to
  // reference an offset from a symbol.
func:
  ldr r0, #data+4 // Will load 2 into r0
  bx lr
data:
  .word 1
  .word 2

  // Creating initialized data that contains the distance between two
  // labels:
size:
  .word end - start
start:
  .word 123
  .word 42
  .word 4523534
end:
```

Assembly Directives

- String definition
- Data definition
- Alignment
- Space-filling
- Org
- Conditional
- Macro
- Section
- Type
- Symbol Binding
- Instruction Set Selection Directives

String definition directives

- Allocates one or more bytes of memory in the current section, and defines the initial contents of the memory from a string literal.

- **.ascii** "string“
  - **.ascii** does not append a null byte to the end of the string.

same

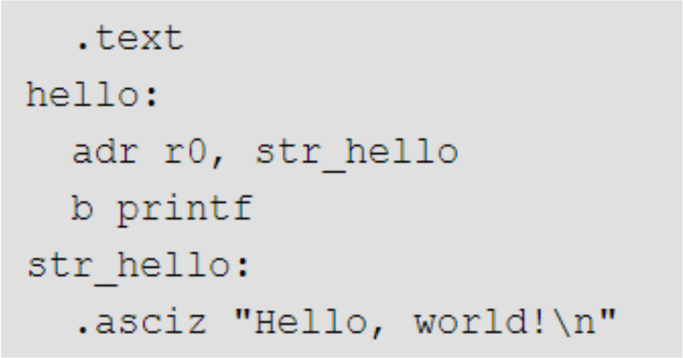
- **.asciz** "string"
- **.string** "string“

주로 이걸 써야함 (?)

  - **.asciz** and **.string** append a null byte to the end of the string.

String definition directives

- Examples

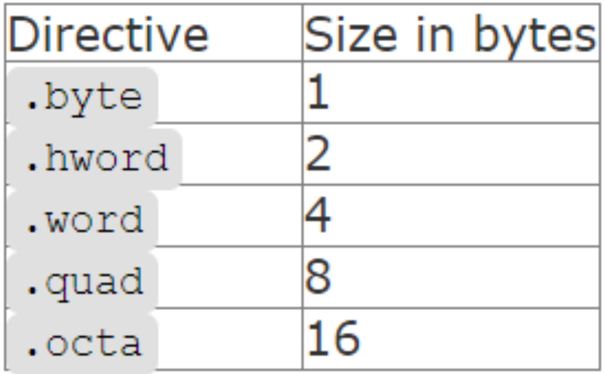


```
  .text
hello:
  adr r0, str_hello
  b printf
str_hello:
  .asciz "Hello, world!\n"
```

Data definition directives

- These directives allocate memory in the current section, and define the initial contents of that memory.

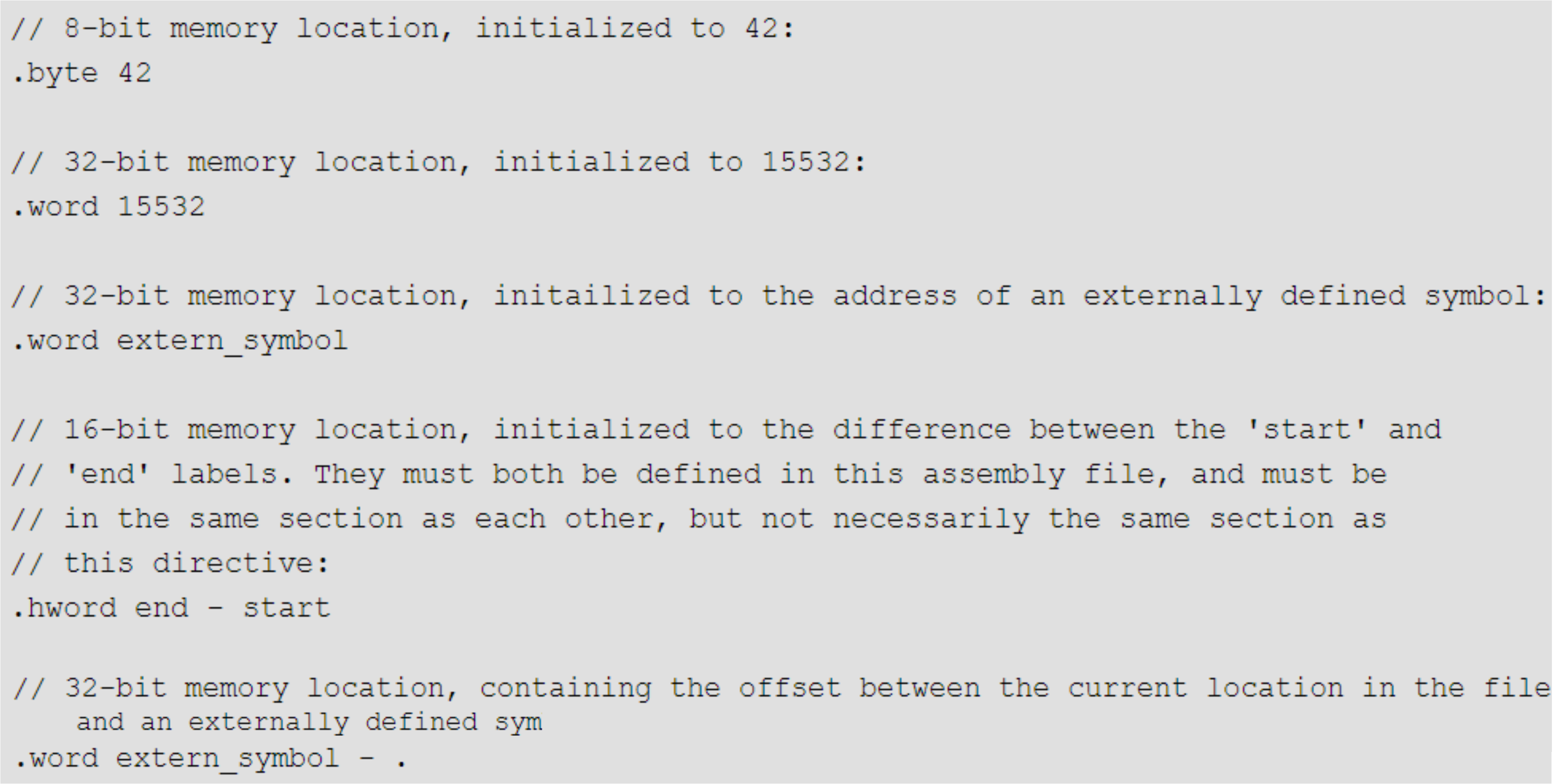
- **.byte** expr[, expr]…
- **.hword** expr[, expr]…
- **.word** expr[, expr]…
- **.quad** expr[, expr]…
- **.octa** expr[, expr]…
  - If multiple arguments are specified, multiple memory locations of the specified size are allocated and initialized to the provided values in order.



Directive	Size in bytes
`.byte`	1
`.hword`	2
`.word`	4
`.quad`	8
`.octa`	16

Data definition directives

- Examples



```
// 8-bit memory location, initialized to 42:
.byte 42

// 32-bit memory location, initialized to 15532:
.word 15532

// 32-bit memory location, initailized to the address of an externally defined symbol:
.word extern_symbol

// 16-bit memory location, initialized to the difference between the 'start' and
// 'end' labels. They must both be defined in this assembly file, and must be
// in the same section as each other, but not necessarily the same section as
// this directive:
.hword end - start

// 32-bit memory location, containing the offset between the current location in the file
    and an externally defined sym
.word extern_symbol - .
```

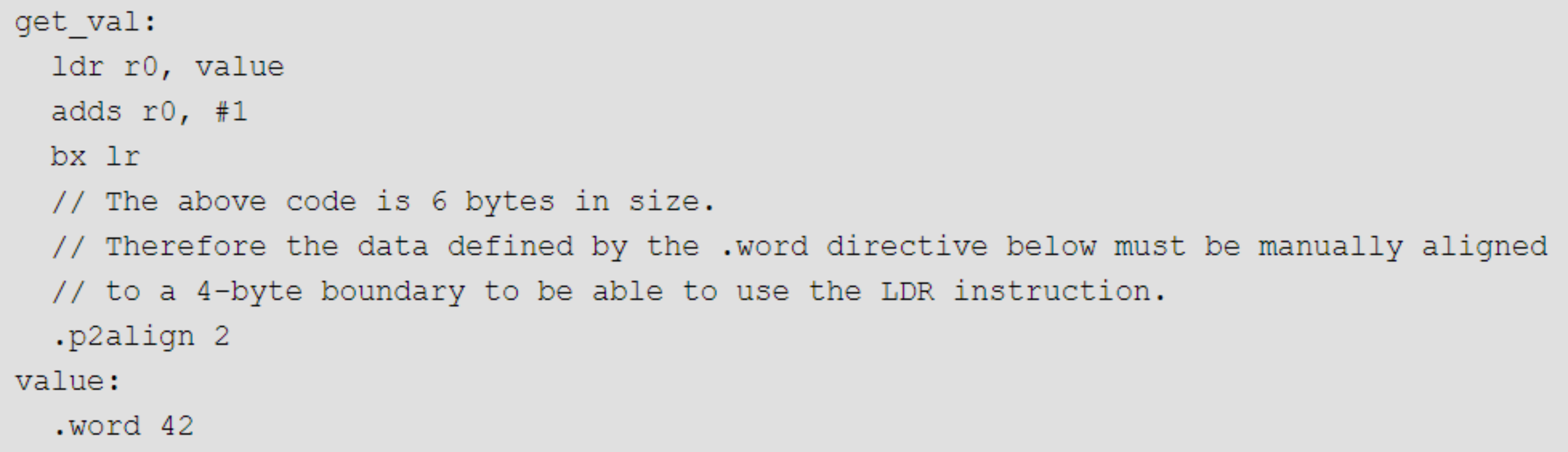
Alignment Directives

- The alignment directives align the current location in the file to a specified boundary.

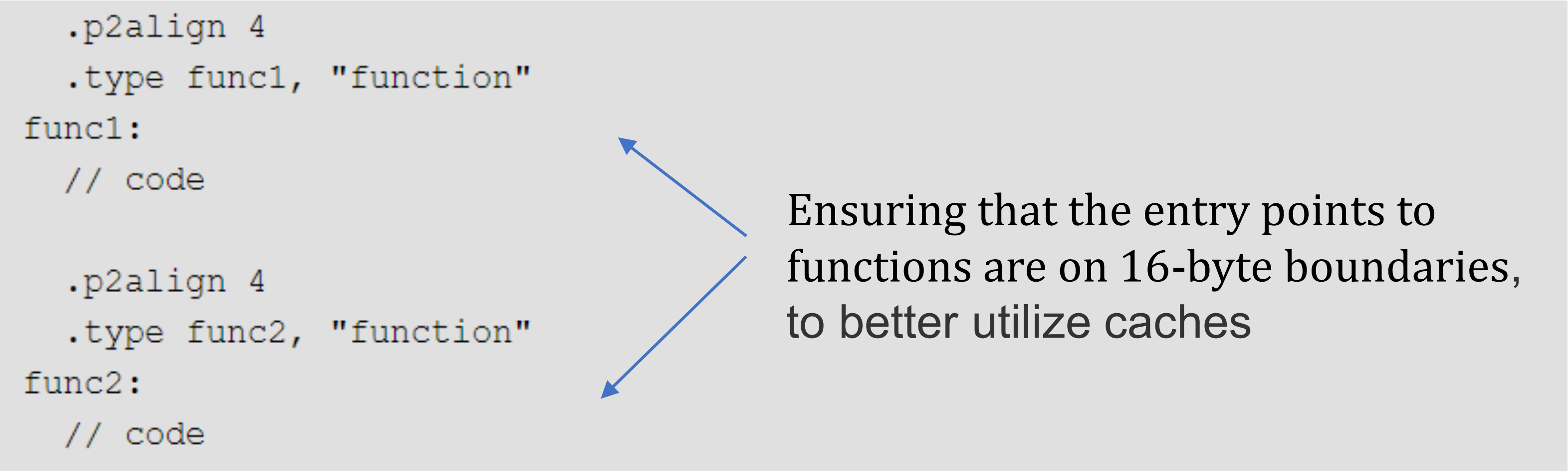
- **.balign** num_bytes [, fill_value]
- **.p2align** exponent [, fill_value]
- **.align** exponent [, fill_value]
  - *num_bytes*
    - This parameter specifies the number of bytes that must be aligned to.
    - The number must be a power of 2.
  - *exponent*
    - This parameter specifies the alignment boundary as an exponent.
    - The actual alignment boundary is $2^{exponent}$
  - *fill_value*
    - The value to fill any inserted padding bytes with. This value is optional.

Alignment Directives

- Examples



```
get_val:
  ldr r0, value
  adds r0, #1
  bx lr
  // The above code is 6 bytes in size.
  // Therefore the data defined by the .word directive below must be manually aligned
  // to a 4-byte boundary to be able to use the LDR instruction.
  .p2align 2
value:
  .word 42
```



```
  .p2align 4
  .type func1, "function"
func1:
  // code

  .p2align 4
  .type func2, "function"
func2:
  // code
```

Ensuring that the entry points to functions are on 16-byte boundaries, to better utilize caches

Space-filling Directives

- **.space** count [, value]
  - The **.space** directive emits count bytes of data, each of which has value value.
- **.fill** count [, size [, value]]
  - The **.fill** directive emits count data values, each with length size bytes and value value.

Org Directives

- The **.org** directive advances the location counter in the cu rrent section to new-location.

- **.org** new_location [, fill_value]
  - *new_location*
    - must be one of:
    - An absolute integer expression, in which case it is treated as the nu mber of bytes from the start of the section.
    - An expression which evaluates to a location in the current section. This could use a symbol in the current section, or the current locati on ('.').
  - *fill_value*
    - This is an optional 1-byte value.

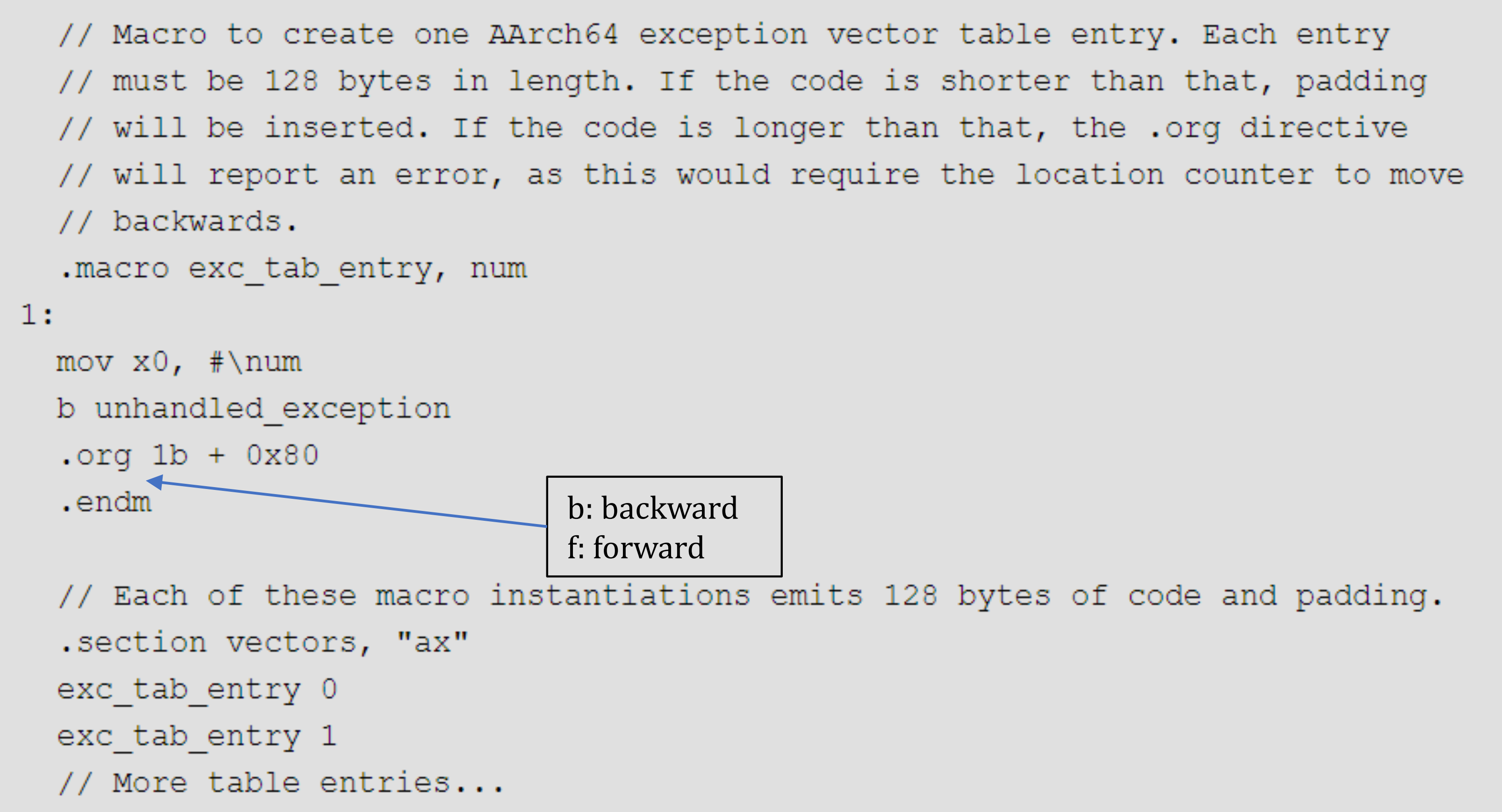
Org Directives

- Operation
  - The **.org** directive can only move the location counter forward, not backward.

  - By default, the **.org** directive inserts zero bytes in any locations that it skips over.
  - This can be overridden using the optional *fill_value* argument, which sets the 1-byte value that will be repeated in each skipped location.

Org Directives

- Examples

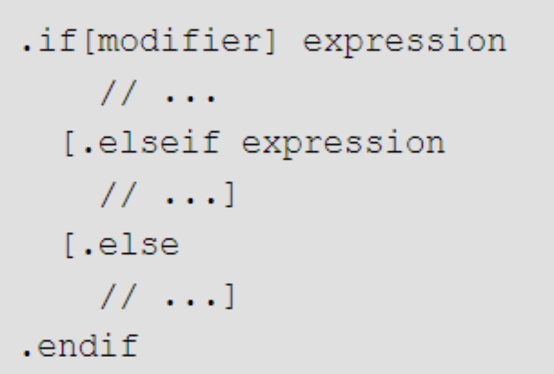


```
  // Macro to create one AArch64 exception vector table entry. Each entry
  // must be 128 bytes in length. If the code is shorter than that, padding
  // will be inserted. If the code is longer than that, the .org directive
  // will report an error, as this would require the location counter to move
  // backwards.
  .macro exc_tab_entry, num
1:
  mov x0, #\num
  b unhandled_exception
  .org 1b + 0x80
  .endm

  // Each of these macro instantiations emits 128 bytes of code and padding.
  .section vectors, "ax"
  exc_tab_entry 0
  exc_tab_entry 1
  // More table entries...
```

b: backward
f: forward

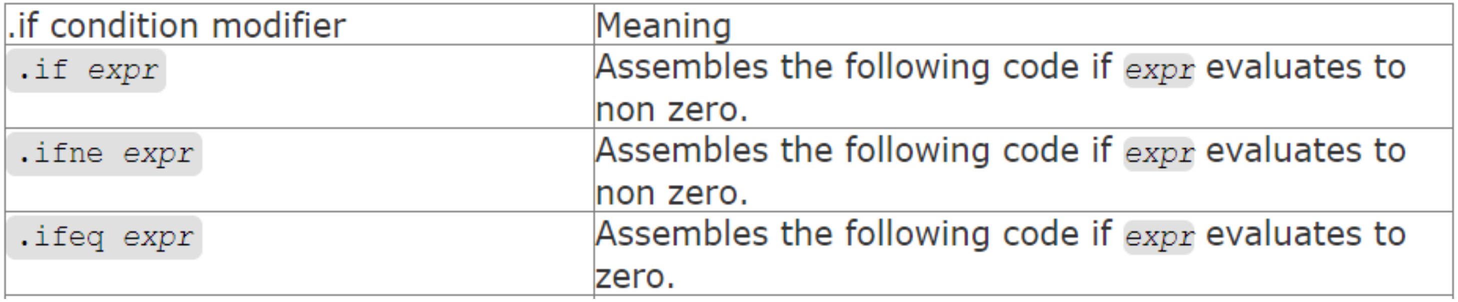
Conditional Assembly Directives

- These directives allow you to conditionally assemble sequences of instructions and directives.
- Syntax



```
.if[modifier] expression
    // ...
  [.elseif expression
    // ...]
  [.else
    // ...]
.endif
```

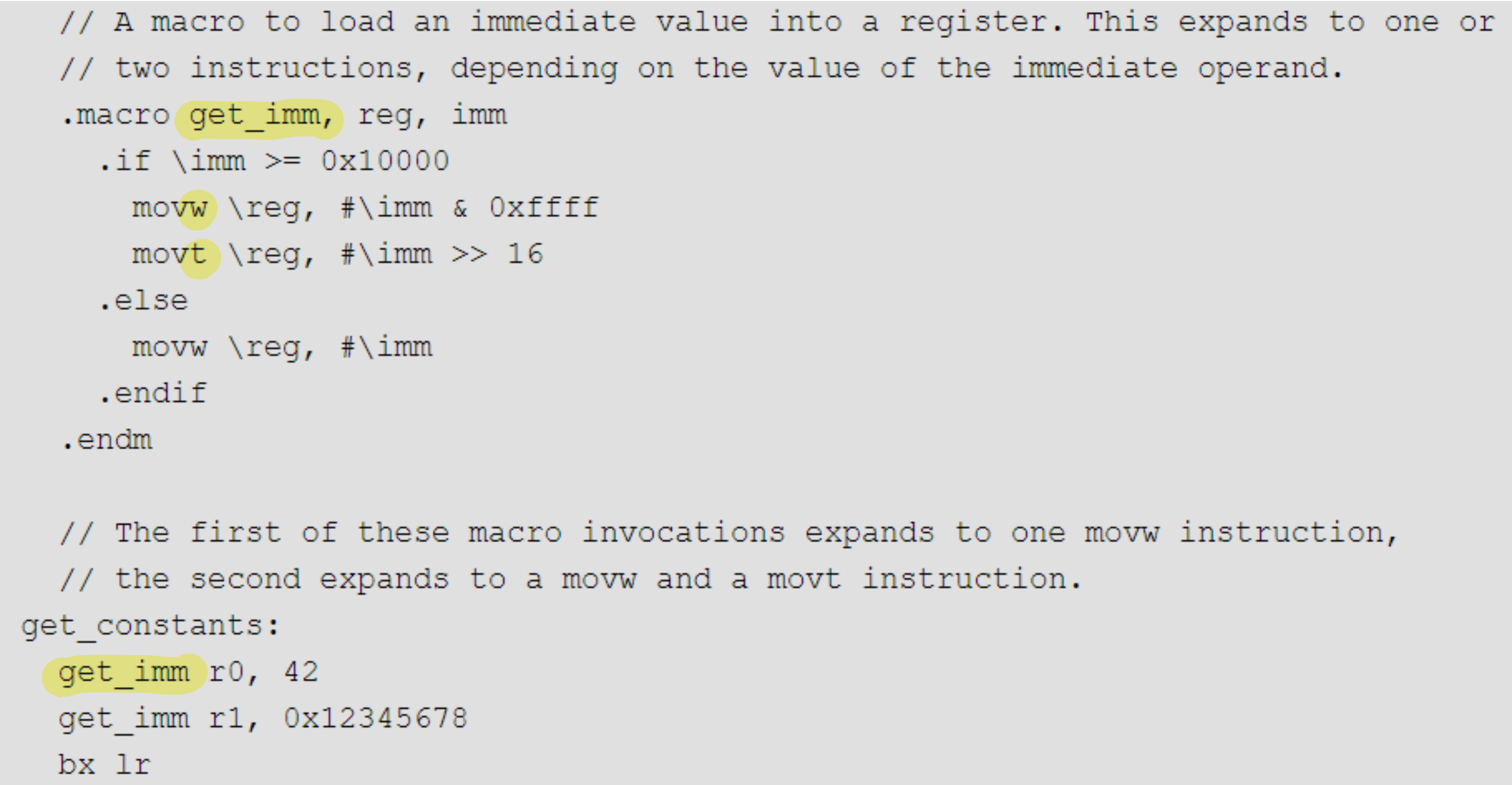
  - You should note that all directives are evaluated by assembler!
    - Condition will not be checked at run-time!
  - Modifiers decide how to check conditions



.if condition modifier	Meaning
`.if expr`	Assembles the following code if `expr` evaluates to non zero.
`.ifne expr`	Assembles the following code if `expr` evaluates to non zero.
`.ifeq expr`	Assembles the following code if `expr` evaluates to zero.

Conditional Assembly Directives

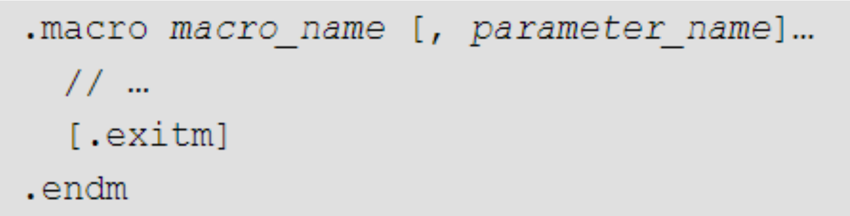
- Examples



```
  // A macro to load an immediate value into a register. This expands to one or
  // two instructions, depending on the value of the immediate operand.
  .macro get_imm, reg, imm
    .if \imm >= 0x10000
      movw \reg, #\imm & 0xffff
      movt \reg, #\imm >> 16
    .else
      movw \reg, #\imm
    .endif
  .endm

  // The first of these macro invocations expands to one movw instruction,
  // the second expands to a movw and a movt instruction.
get_constants:
  get_imm r0, 42
  get_imm r1, 0x12345678
  bx lr
```

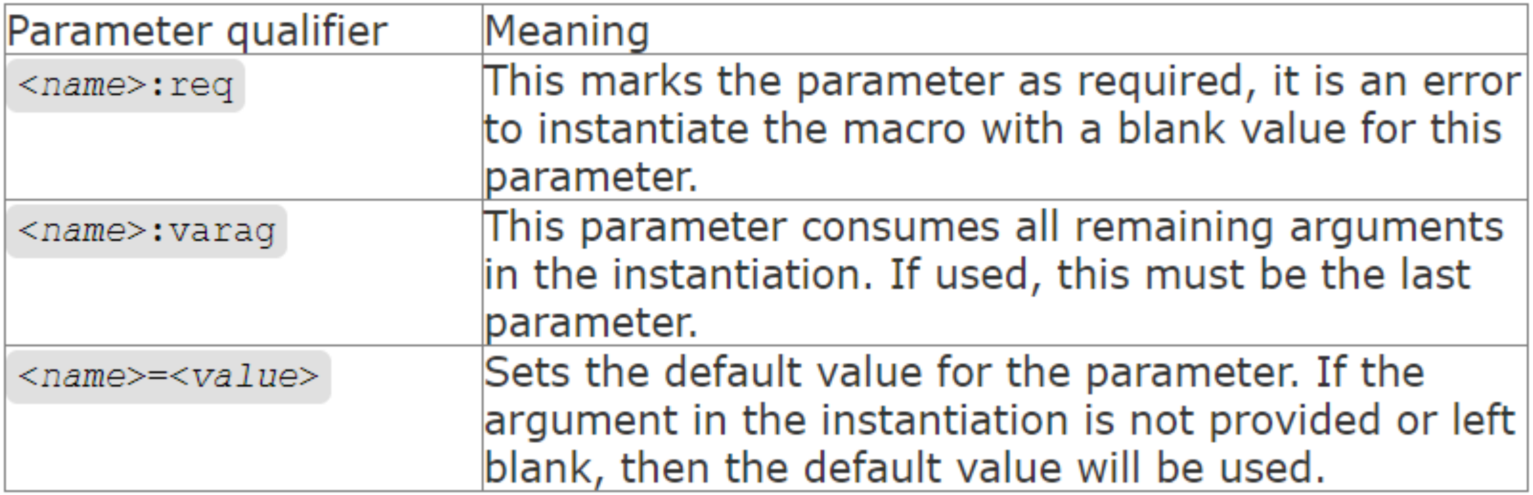
Macro Directives

- Syntax



```
.macro macro_name [, parameter_name]…
  // …
  [.exitm]
.endm
```

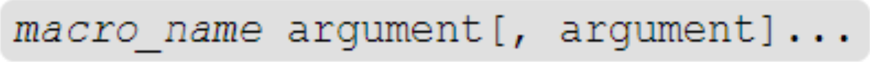
  - *macro_name*
    - The name of the macro.
  - *parameter_name*
    - Inside the body of a macro, the parameters can be referred to by their name, prefixed with \. When the macro is instantiated, parameter references will be expanded to the value of the argument.



Parameter qualifier	Meaning
`<name>:req`	This marks the parameter as required, it is an error to instantiate the macro with a blank value for this parameter.
`<name>:varag`	This parameter consumes all remaining arguments in the instantiation. If used, this must be the last parameter.
`<name>=<value>`	Sets the default value for the parameter. If the argument in the instantiation is not provided or left blank, then the default value will be used.

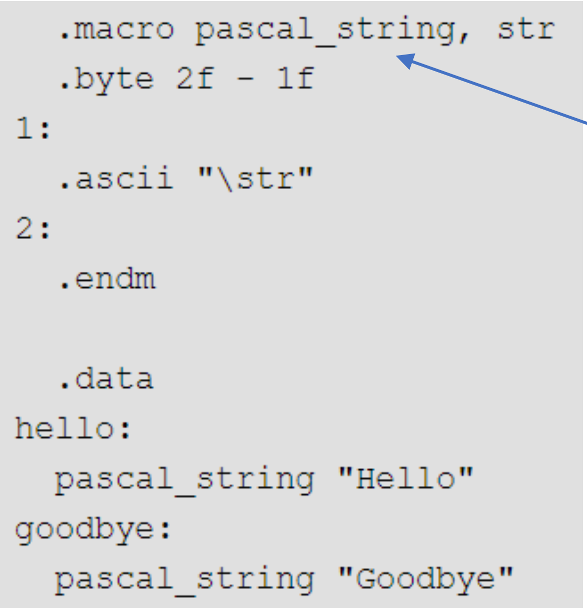
Macro Directives

- Operation
  - The **.macro** directive defines a new macro with name *macro_name*. Once a macro is defined, it can be instantiated by using it like an instruction mnemonic:



```
macro_name argument[, argument]...
```

- Examples



```
  .macro pascal_string, str
  .byte 2f - 1f
1:
  .ascii "\str"
2:
  .endm

  .data
hello:
  pascal_string "Hello"
goodbye:
  pascal_string "Goodbye"
```

pascal-style strings are prefixed by a length byte, and have no null terminator

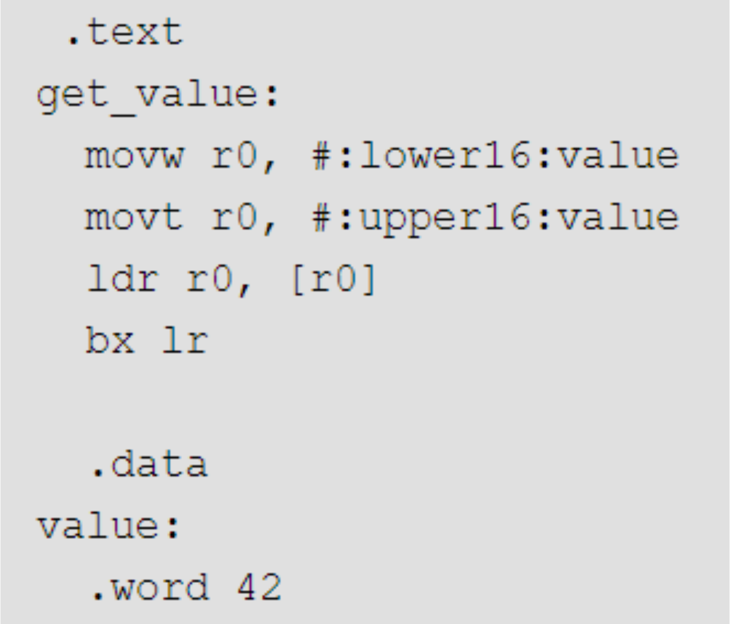
Section Directives

- The section directives instruct the assembler to change the ELF section that code and data are emitted into.

- **.section** name [, "flags" [, %type [, entry_size] [, group_name [, linkage]] [, link_order_symbol] [, unique, unique_id]]]
- **.text**
- **.data**
- **.rodata**
- **.bss**
  - **.section** directive switches the current target section to the one described by its arguments.
  - The rest of the directives (**.text**, **.data**, **.rodata**, **.bss**) switch to one of the built-in sections.

Section Directives

- Examples
  - Splitting code and data into the built-in **.text** and **.data** sections



```
 .text
get_value:
  movw r0, #:lower16:value
  movt r0, #:upper16:value
  ldr r0, [r0]
  bx lr

  .data
value:
  .word 42
```

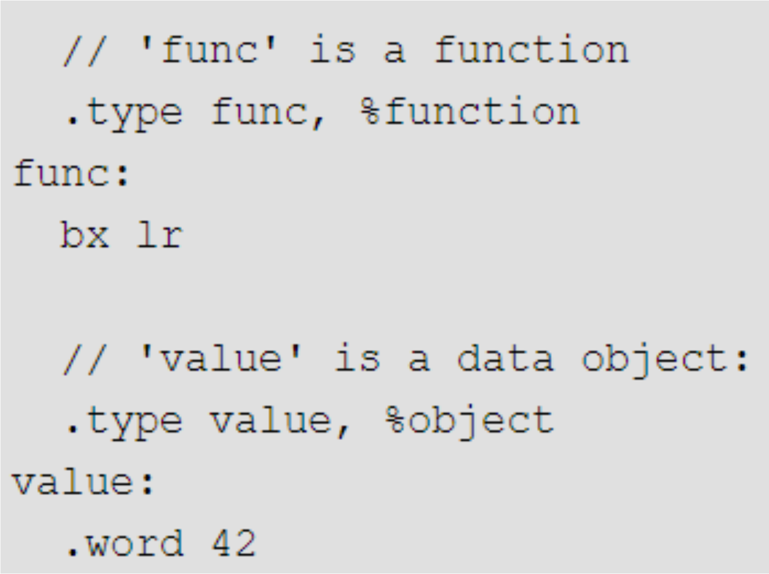
Type Directive

- The default type of a symbol in an object file is the assembly-time type of the symbol.
  - Symbolic constants and undefined symbols → @notype
  - Labels and common symbols → @object
  - Function names → @function
- The .type directive explicitly sets the type of a symbol.

- **.type** symbol, %type
  - *%type*
    - The following types are accepted:
    - %function
      - a function name
    - %object
      - a data object
    - %tls_object
      - a thread-local data object.

Type Directive

- Examples



```
  // 'func' is a function
  .type func, %function
func:
  bx lr

  // 'value' is a data object:
  .type value, %object
value:
  .word 42
```

Symbol Binding Directives

- These directives modify the ELF binding of one or more symbols.

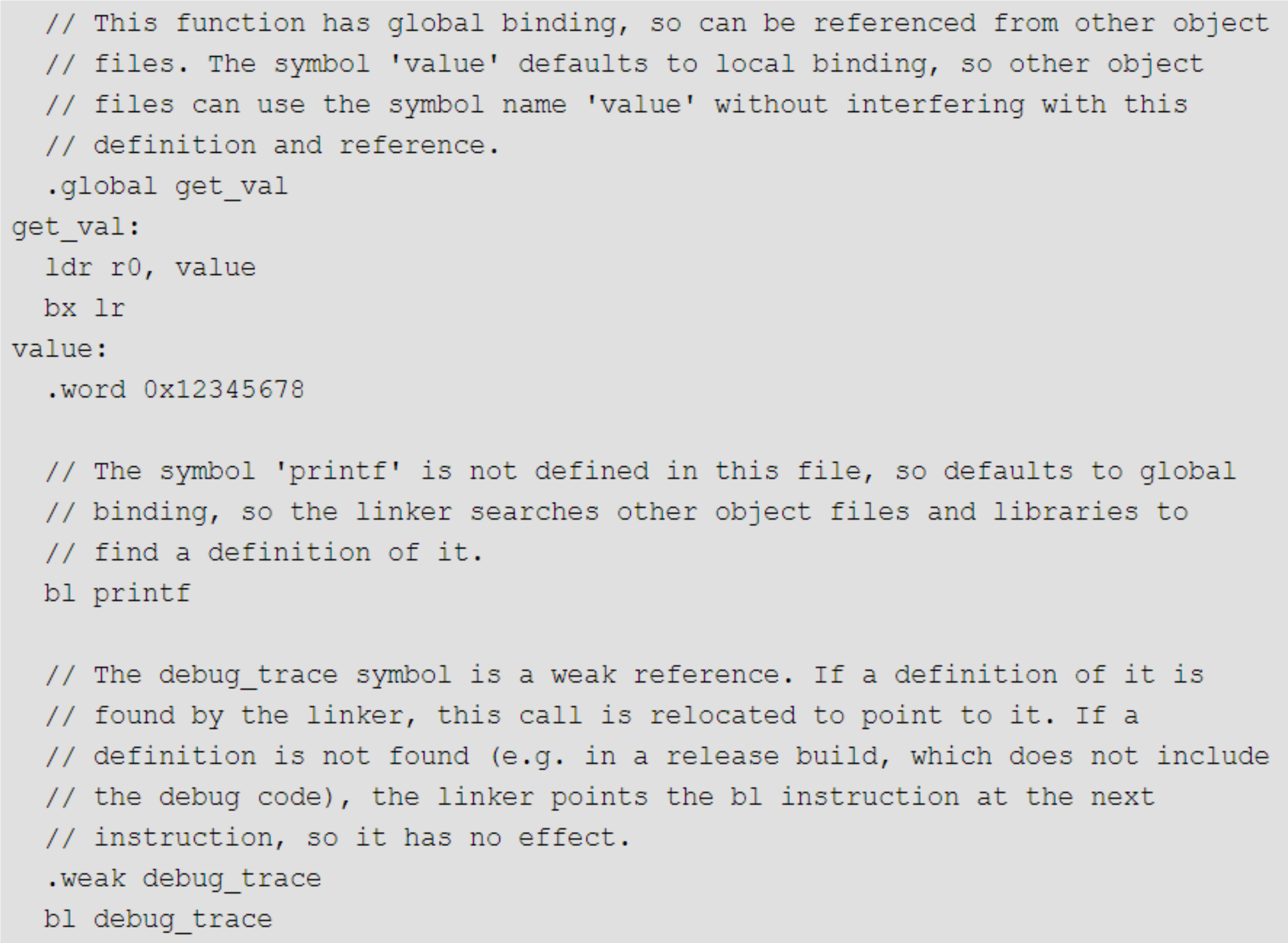
- **.global** symbol[, symbol]…
  - These symbols are visible to all object files being linked, so a definition in one object file can satisfy a reference in another.
- **.local** symbol[, symbol]…
  - These symbols are not visible outside the object file they are defined or referenced in, so multiple object files can use the same symbol names without interfering with each other.
- **.weak** symbol[, symbol]…
  - These symbols behave similarly to global symbols, with these differences:
    - If a reference to a symbol with weak binding is not satisfied (no definition of the symbol is found), this is not an error.
    - If multiple definitions of a weak symbol are present, this is not an error. If a definition of the symbol with strong binding is present, that definition satisfies all references to the symbol, otherwise one of the weak references is chosen.

Symbol Binding Directives

- Operation
  - The symbol binding directive can be at any point in the assembly file, before or after any references or definitions of the symbol.
  - If the binding of a symbol is not specified using one of these directives, the default binding is:
    - If a symbol is not defined in the assembly file, it has global visibility by default.
    - If a symbol is defined in the assembly file, it has local visibility by default.

Symbol Binding Directives

- Examples



```
  // This function has global binding, so can be referenced from other object
  // files. The symbol 'value' defaults to local binding, so other object
  // files can use the symbol name 'value' without interfering with this
  // definition and reference.
  .global get_val
get_val:
  ldr r0, value
  bx lr
value:
  .word 0x12345678

  // The symbol 'printf' is not defined in this file, so defaults to global
  // binding, so the linker searches other object files and libraries to
  // find a definition of it.
  bl printf

  // The debug_trace symbol is a weak reference. If a definition of it is
  // found by the linker, this call is relocated to point to it. If a
  // definition is not found (e.g. in a release build, which does not include
  // the debug code), the linker points the bl instruction at the next
  // instruction, so it has no effect.
  .weak debug_trace
  bl debug_trace
```

Instruction Set Selection Directives

- .arm
  - The .arm directive instructs the assembler to interpret subsequent instructions as A32 instructions.
- .thumb
  - The .thumb directive instructs the assembler to interpret subsequent instructions as T32 instructions, using the UAL syntax.
- .thumb_func
  - This directive specifies that the following symbol is the name of a Thumb encoded function.
- .syntax [unified | divided]
  - This directive sets the Instruction Set Syntax.
  - divided (default for compatibility with legacy)
    - ARM and Thumb instructions are used separately
  - unified
    - Enables UAL (Unified Assembly Language) syntax
    - Necessary for Thumb2 instructions