# Assembly Language and Computer Architecture Lab

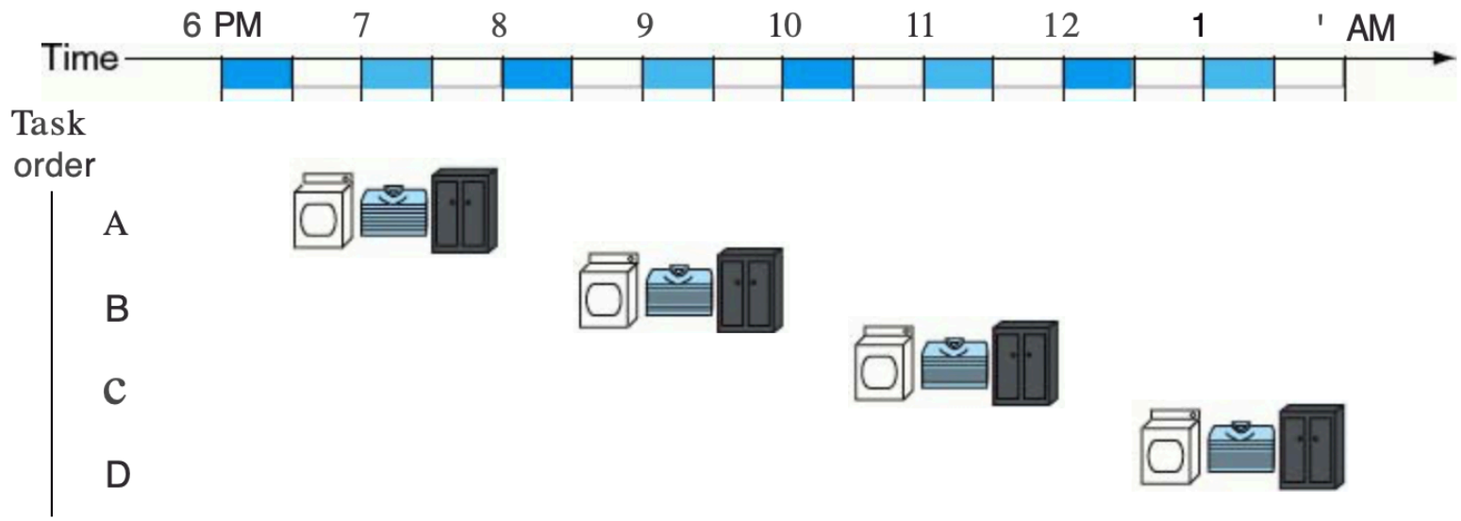
Nguyen Thi Thanh Nga
Dept. Computer engineering
School of Information and Communication Technology

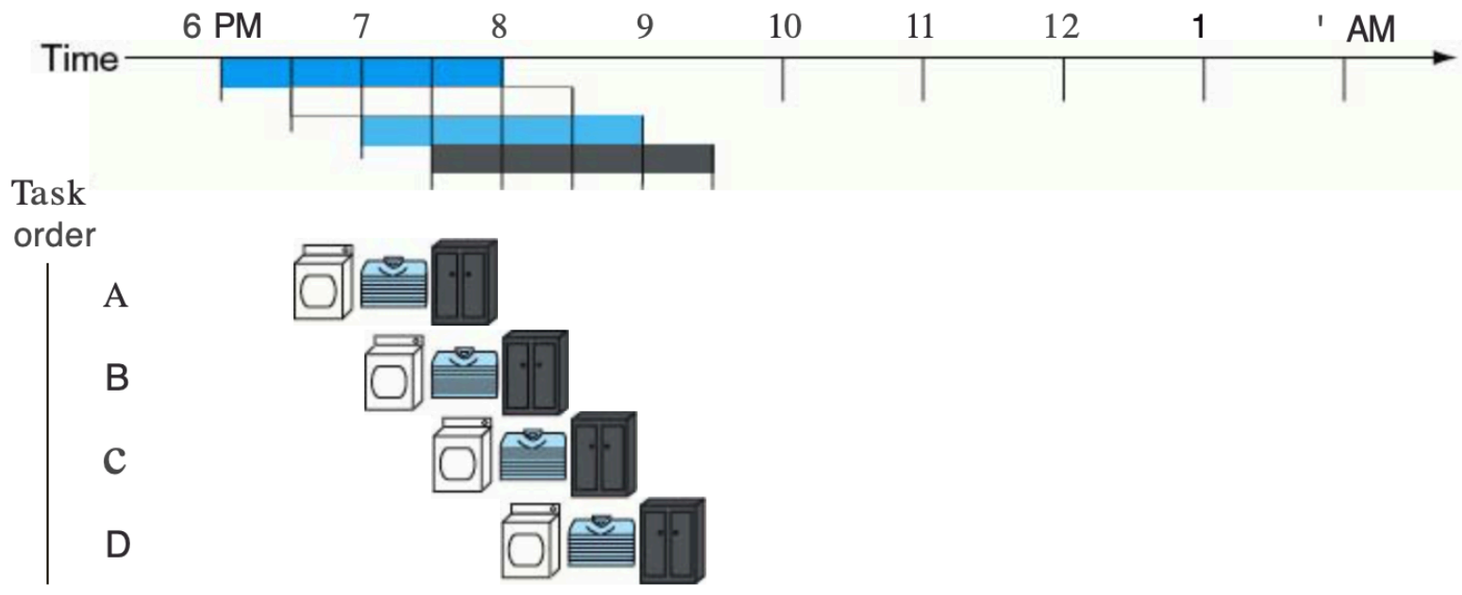
# Week 7

- Pipeline architechture
- Syscall introduction
- Some frequent syscall services
  - Syscall 1, 4, 5, 8, 10, 11, 12, 17, 31, 33, 50, 54, 55, 56, 59

# Pipelining

- Pipelining is a technique where multiple instructions are overlapped during execution.



# Pipelining

- Pipeline is divided into stages and these stages are connected with one another to form a pipe like structure.

- Instructions enter from one end and exit from another end.

- Pipelining increases the overall instruction throughput.

# Pipelining

When executed, the MIPS instructions are divided into five stages:

- **Stage 1 (Instruction Fetch)**
  In this stage the CPU reads instructions from the address in the memory whose value is present in the program counter.

- **Stage 2 (Instruction Decode)**
  In this stage, instruction is decoded and the register file is accessed to get the values from the registers used in the instruction.

- **Stage 3 (Instruction Execute)**
  In this stage, ALU operations are performed.

- **Stage 4 (Memory Access)**
  In this stage, memory operands are read and written from/to the memory that is present in the instruction.

- **Stage 5 (Write Back)**
  In this stage, computed/fetched value is written back to the register present in the instructions.

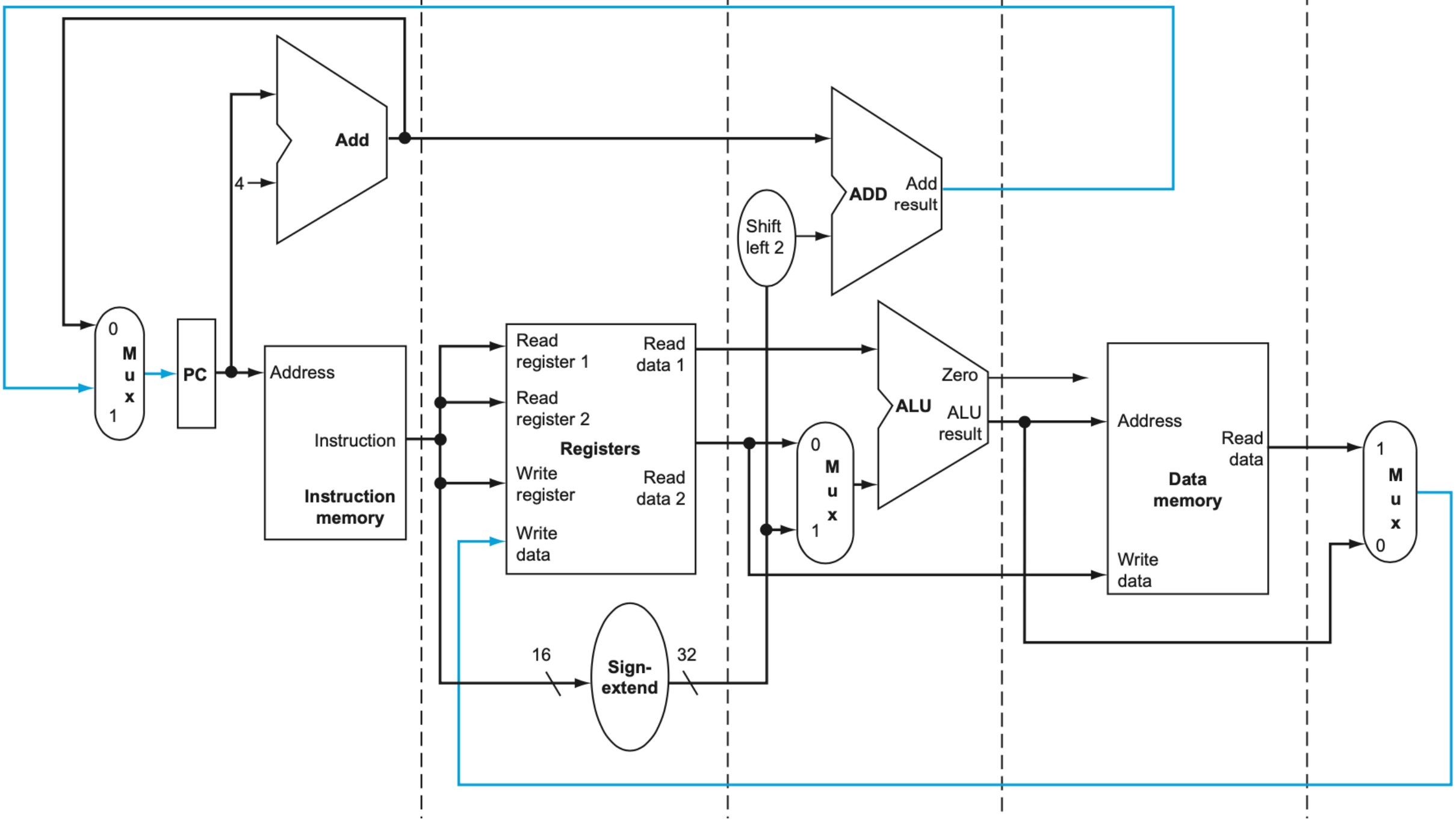
1. Intruction Fetch (IF)

2. Instruction Decode (ID)

3. Execute (EX)

4. Memory (MEM)

5. Write Back (WB)

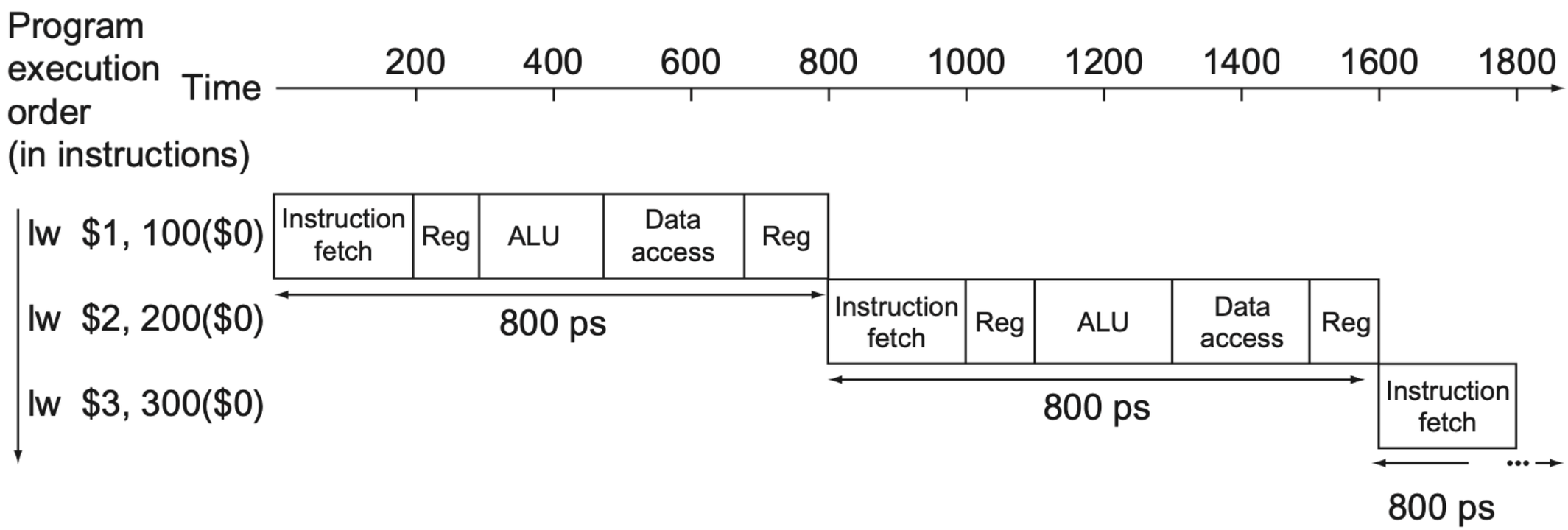


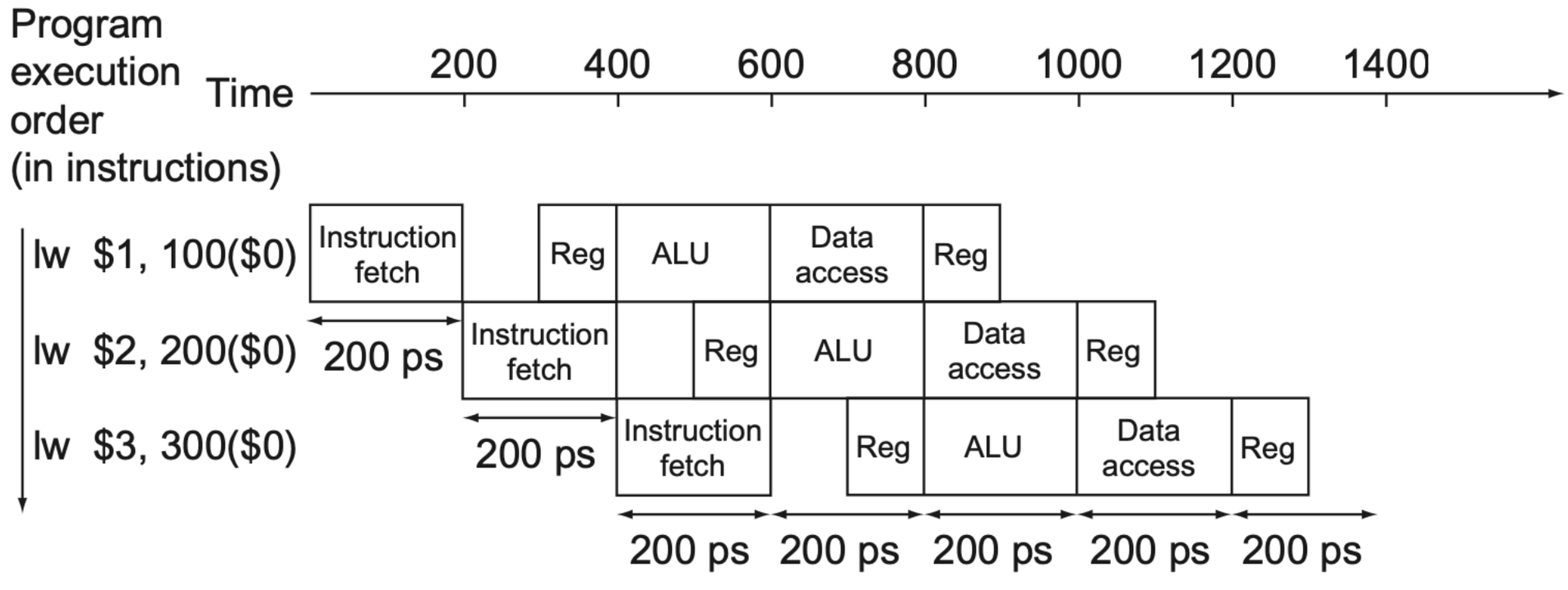
# Pipelining

- Consider the processor with 8 basic instructions, corresponding to the time for each stage as follows:

| Instruction class | Instruction fetch | Register read | ALU operation | Data access | Register write | Total time |
|---|---|---|---|---|---|---|
| Load word (`lw`) | 200 ps | 100 ps | 200 ps | 200 ps | 100 ps | 800 ps |
| Store word (`sw`) | 200 ps | 100 ps | 200 ps | 200 ps | | 700 ps |
| R-format (`add`, `sub`, `AND`, `OR`, `slt`) | 200 ps | 100 ps | 200 ps | | 100 ps | 600 ps |
| Branch (`beq`) | 200 ps | 100 ps | 200 ps | | | 500 ps |

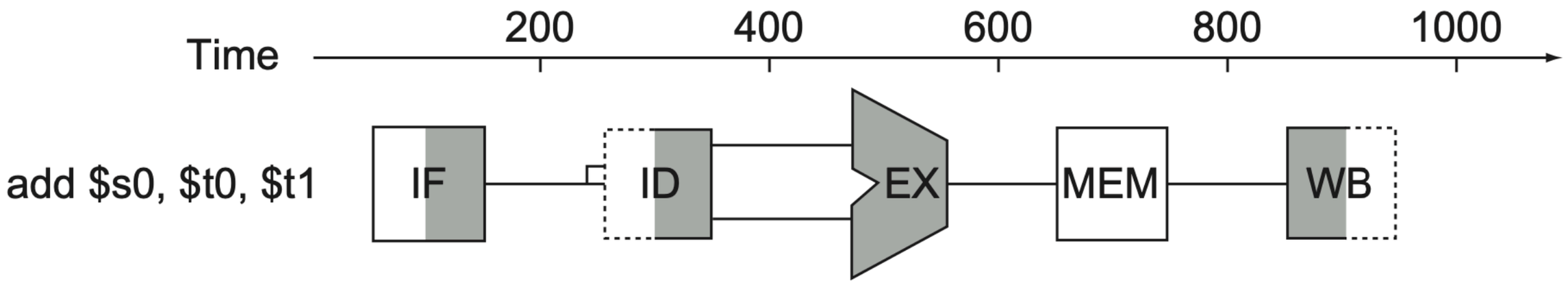
# Pipelining


Program execution order (in instructions)
Time
200
400
600
800
1000
1200
1400
1600
1800
lw $1, 100($0)
lw $2, 200($0)
lw $3, 300($0)
Instruction fetch
Reg
ALU
Data access
Reg
800 ps
800 ps
800 ps


Program execution order (in instructions)
Time
200
400
600
800
1000
1200
1400
lw $1, 100($0)
lw $2, 200($0)
lw $3, 300($0)
Instruction fetch
Reg
ALU
Data access
Reg
200 ps
200 ps
200 ps
200 ps
200 ps
200 ps
200 ps

# Pipelining

- The convention presents five stages of execution of an instruction:




- The do nothing command is highlighted white and vice versa.

- The right half filled with black is a read operation, and the left half filled with black is a write operation.

# Pipelining

- Hazards can occur when applying pipeline.

- Hazard is a state in which the next instruction cannot be executed (or executed but will be wrong), usually for one of three reasons:

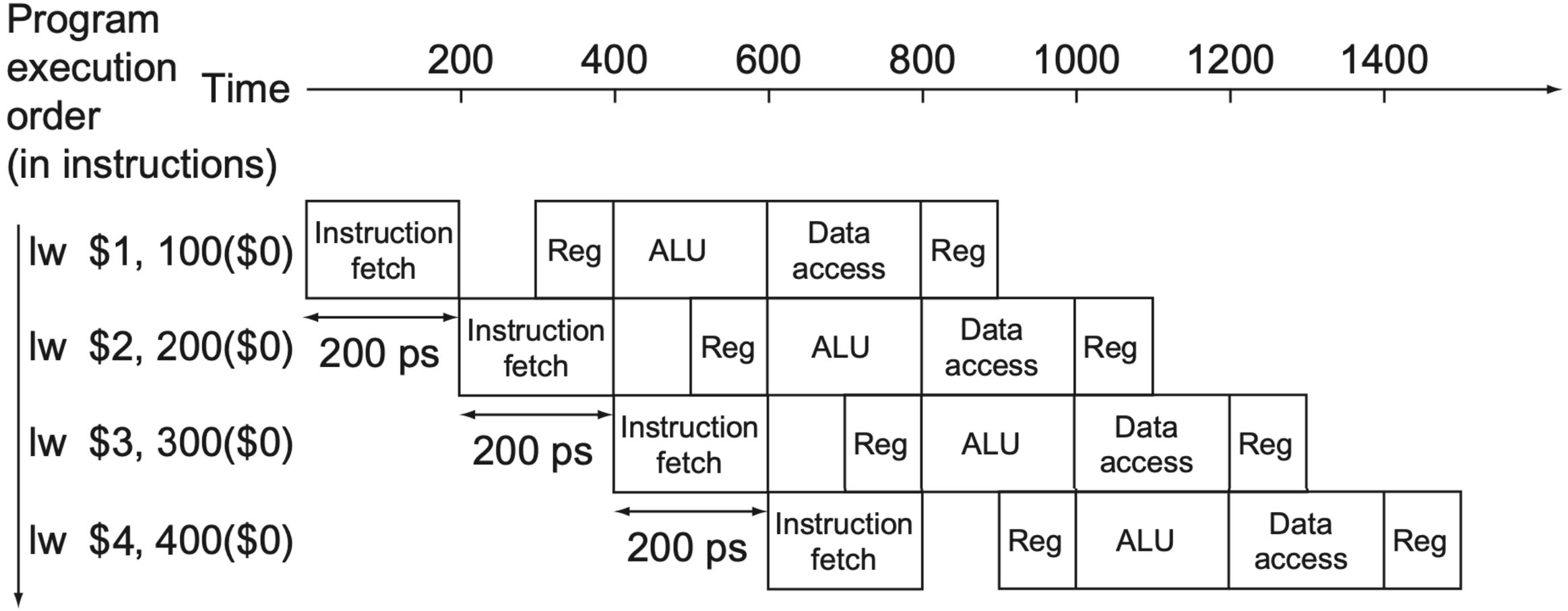
  - Structural hazard

  - Data hazard

  - Control hazard

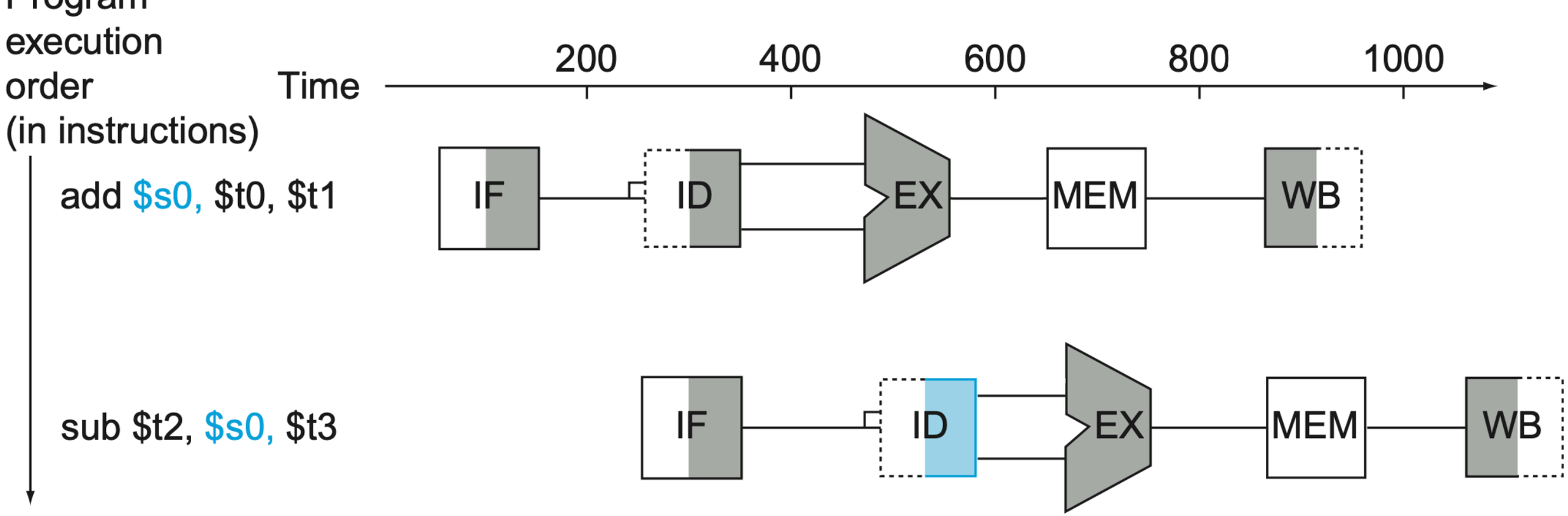
# Structural hazard

- When two (or more) instructions that are already in pipeline need the same resource.
- Structural hazards are sometime referred to as resource hazards.



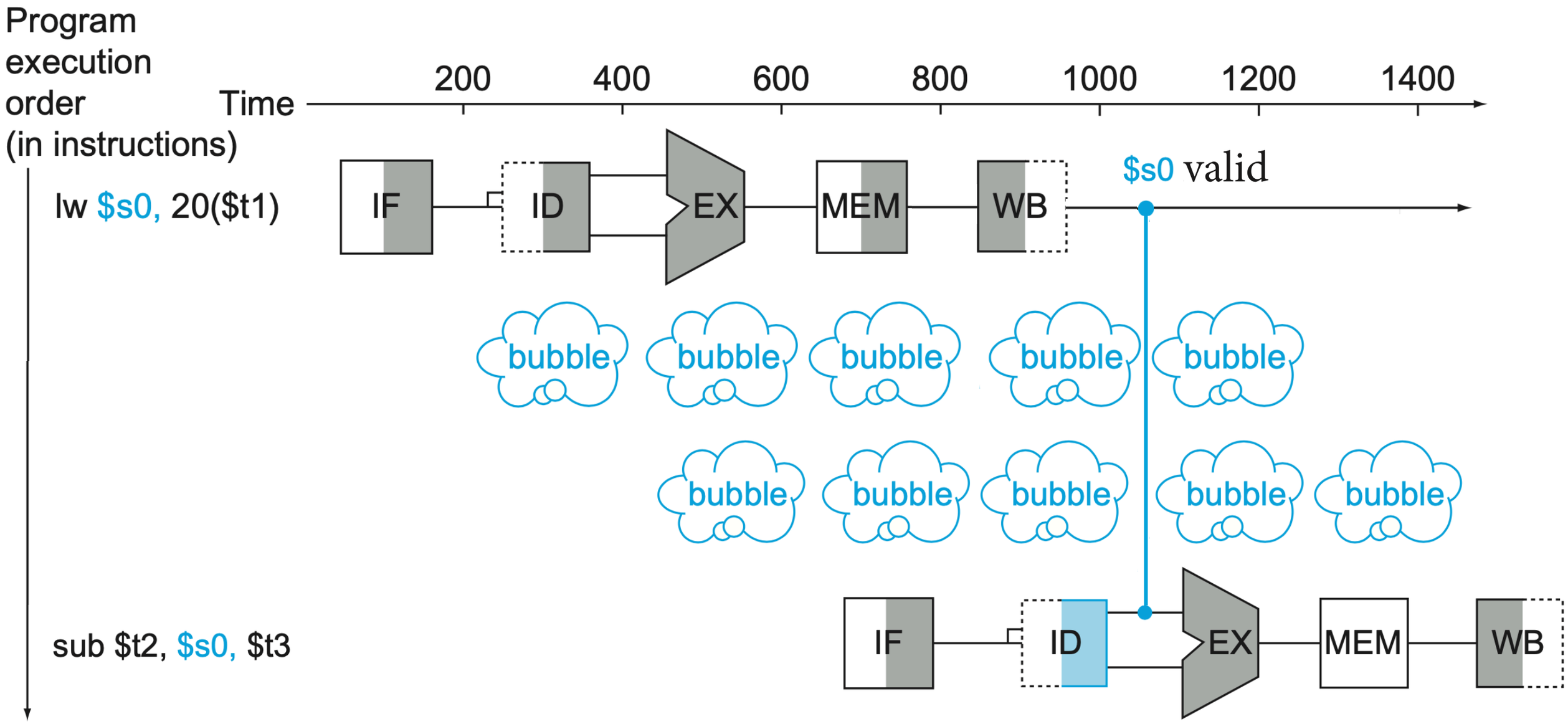
# Data hazard

- When an expected instruction cannot be executed in its pipeline because the data that this instruction needs is not available yet.



# Data hazard

- Wait for 2 more clock cycles:



# Control hazard

- When the pipeline makes wrong decisions on branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded.

- The term branch hazard also refers to a control hazard.

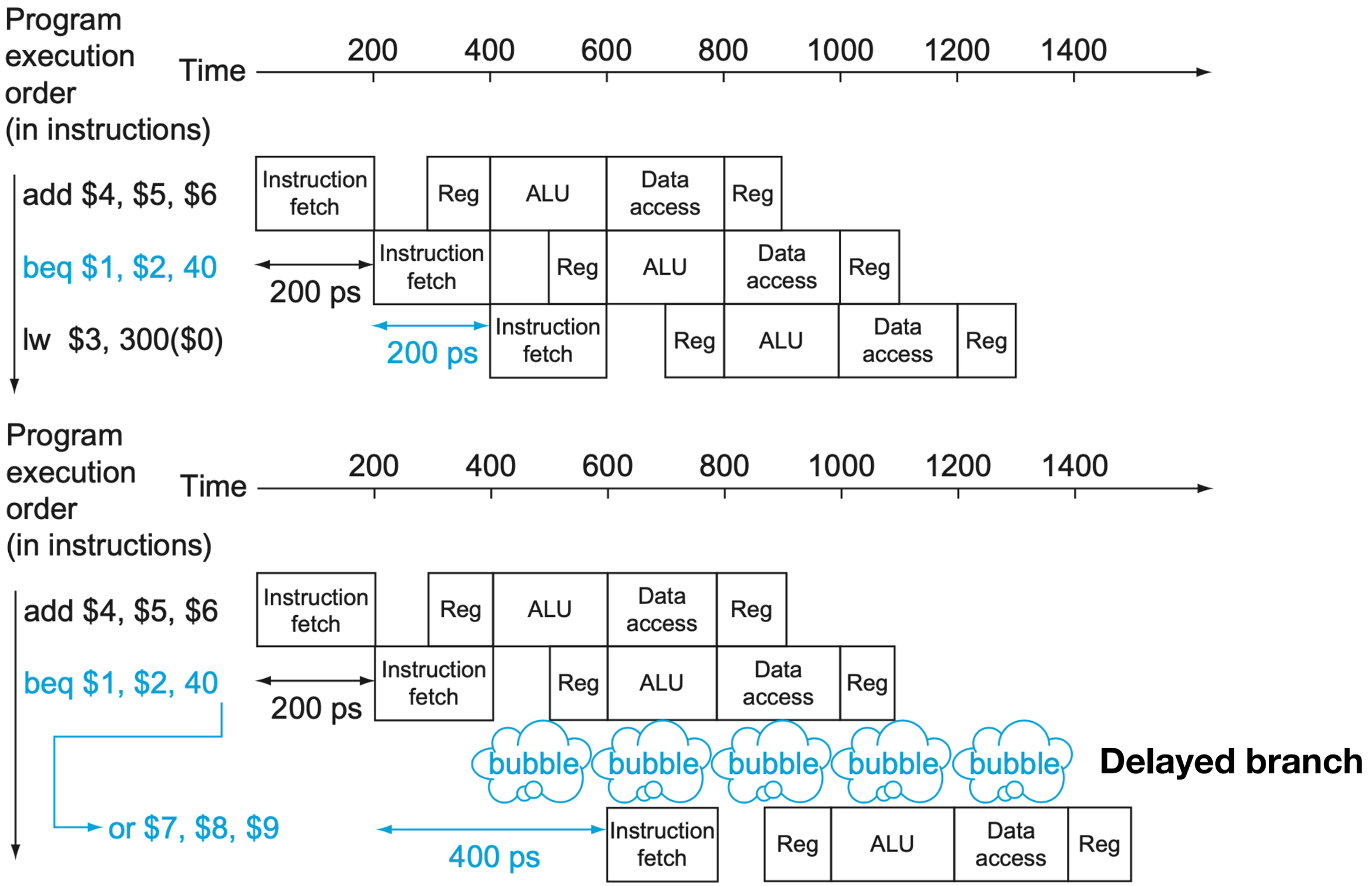
- Consider the following program segment:

```
add $4, $5, $6

beq $1, $2, label

lw $3, 300($0)
```

# Control hazard



# SYSCALL

- A number of system services, mainly for input and output, are available for use by your MIPS program.
- MIPS register contents are not affected by a system call, except for result registers as specified in the table below.

# How to use SYSCALL system services

- Load the service number in register **$v0**.
- Load argument values, if any, in **$a0**, **$a1**, **$a2**, or **$f12** as specified.
- Issue the SYSCALL instruction.
- Retrieve return values, if any, from result registers as specified.

# Table of Frequently Available Services

- Refer to the following Table of available services.

- http://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html

# 1. Print decimal integer

- Print an integer to standard output (the console)

*Argument(s):*

$v0 = 1
$a0 = number to be printed

*Return value:*

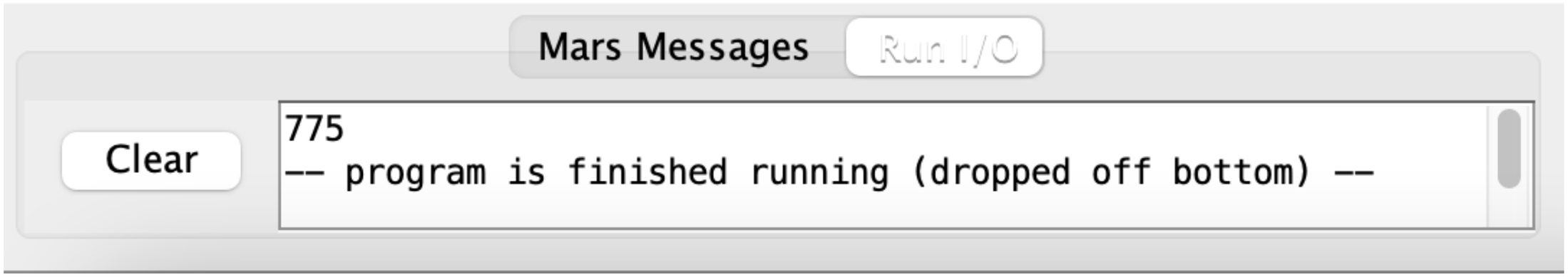
none

*Example:*

```
li $v0, 1          # service 1 is print integer
li $a0, 0x307      # the interger to be printed is 0x307
syscall            # execute
```

# 1. Print decimal integer

- Print an integer to standard output (the console)
- Result:



# 2. MessageDialogInt

- Show an integer to an information-type message dialog.

*Argument(s):*

$v0 = 56
$a0 = address of the null-terminated message string
$a1 = int value to display in string form after the first string *Return value:*

none

*Example:*

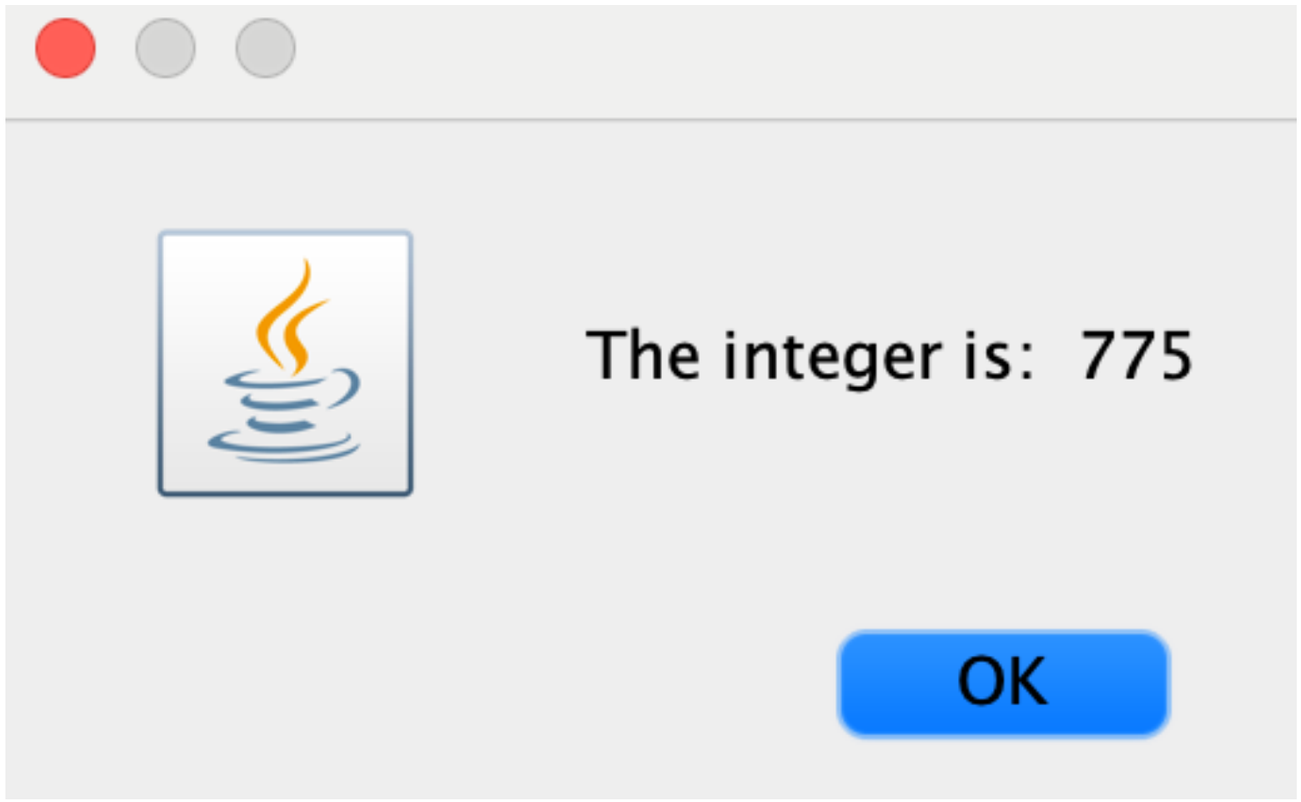
```
.data
message: .asciiz "The integer is:  "

.text
        li $v0, 56
        la $a0, message
        li $a1, 0x307   # the interger to be printed is 0x307
        syscall         # execute
```

# 2. MessageDialogInt

- Show an integer to an information-type message dialog.

- Result:



# 3. Print string

- Formatted print to standard output (the console).

*Argument(s):*

$v0 = 4

$a0 = value to be printed

*Return value:*

none

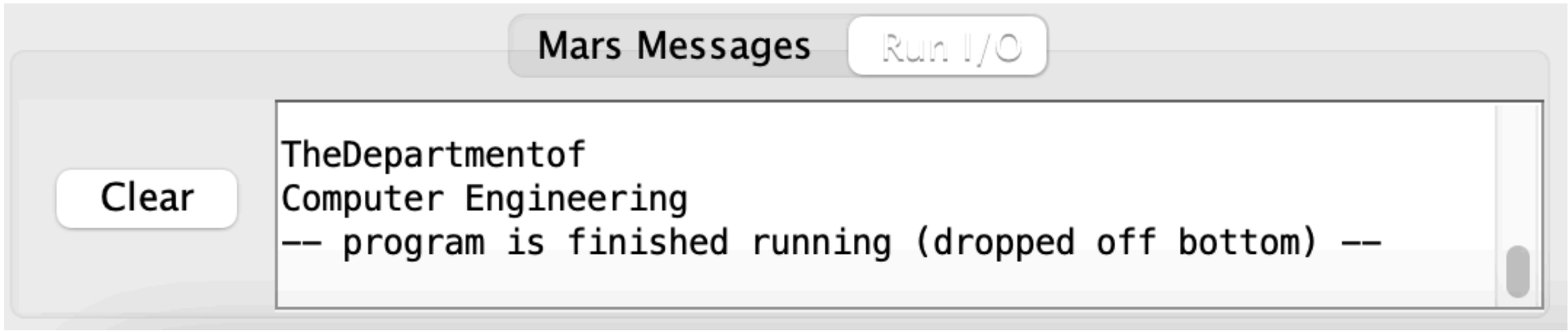
*Example:*

```
.data
message: .asciiz "TheDepartmentof \nComputer Engineering"

.text
        li $v0, 4
        la $a0, message
        syscall
```

# 3. Print string

- Formatted print to standard output (the console).
- Result:



# 4. MessageDialogString

- Show a string to a information-type message dialog

*Argument(s):*

$v0 = 59
$a0 = address of the null-terminated message string
$a1 = address of null-terminated string to display

*Return value:*

none

*Example:*

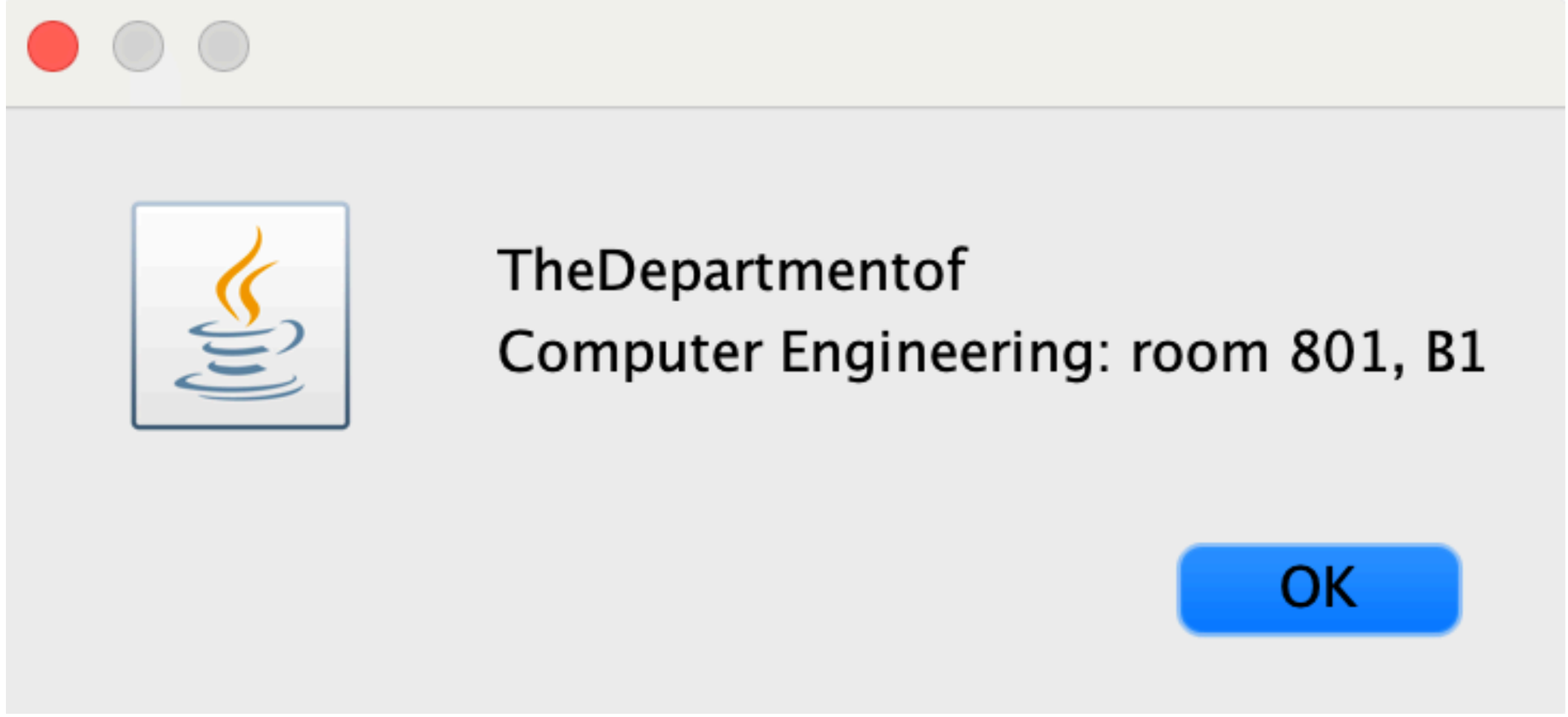
```
.data
message: .asciiz "TheDepartmentof \nComputer Engineering:"
address: .asciiz " room 801, B1"

.text
        li $v0, 59
        la $a0, message
        la $a1, address
        syscall
```

# 4. MessageDialogString

- Show a string to a information-type message dialog.

- Result:



# 5. Read integer

- Get a integer from standard input (the keyboard).

*Argument(s):*

$v0 = 5

*Return value:*

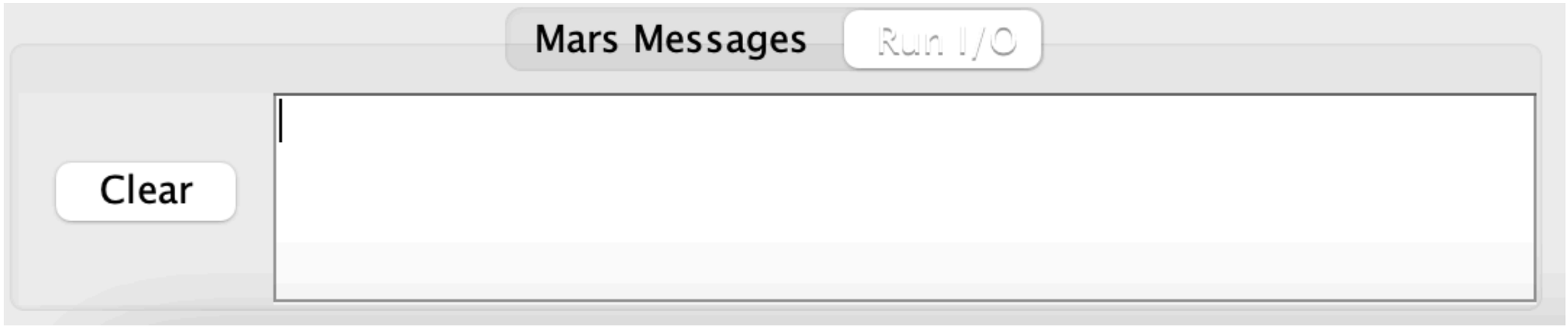
$v0 = contains integer read

*Example:*

```
li $v0, 5
syscall
```

# 5. Read integer

- Get a integer from standard input (the keyboard).
- Result:



# 6. InputDialogInt

- Show a message dialog to read a integer with content parser.

*Argument(s):*

$v0 = 51
$a0 = address of the null-terminated message string

*Return value:*

$a0 = contains int read
$a1 contains status value
0: OK status
-1: input data cannot be correctly parsed
-2: Cancel was chosen
-3: OK was chosen but no data had been input into field

# 6. InputDialogInt

- Show a message dialog to read a integer with content parser.

- Example:

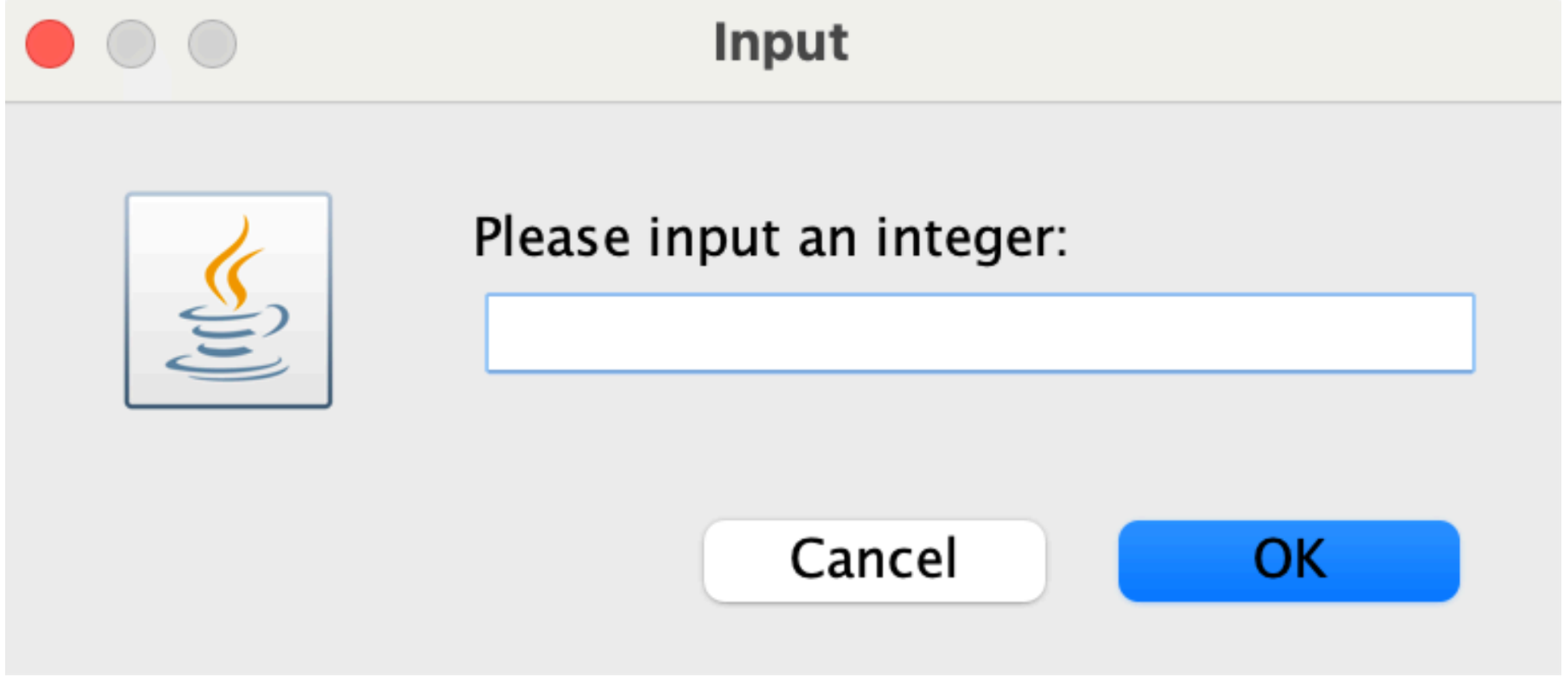
```
.data
message: .asciiz "Please input an integer: "

.text
        li $v0, 51
        la $a0, message
        syscall
```

# 6. InputDialogInt

- Show a message dialog to read a integer with content parser.

- Result:



# 7. Read string

- Get a string from standard input (the keyboard).

*Argument(s):*

$v0 = 8
$a0 = address of input buffer
$a1 = maximum number of characters to read

*Return value:*

none

*Remarks:*

For specified length n, string can be no longer than n-1.

- If less than that, adds newline to end.
- In either case, then pads with null byte

Just in special cases:

If n = 1, input is ignored and null byte placed at buffer address.

If n < 1, input is ignored and nothing is written to the buffer.

# 7. Read string

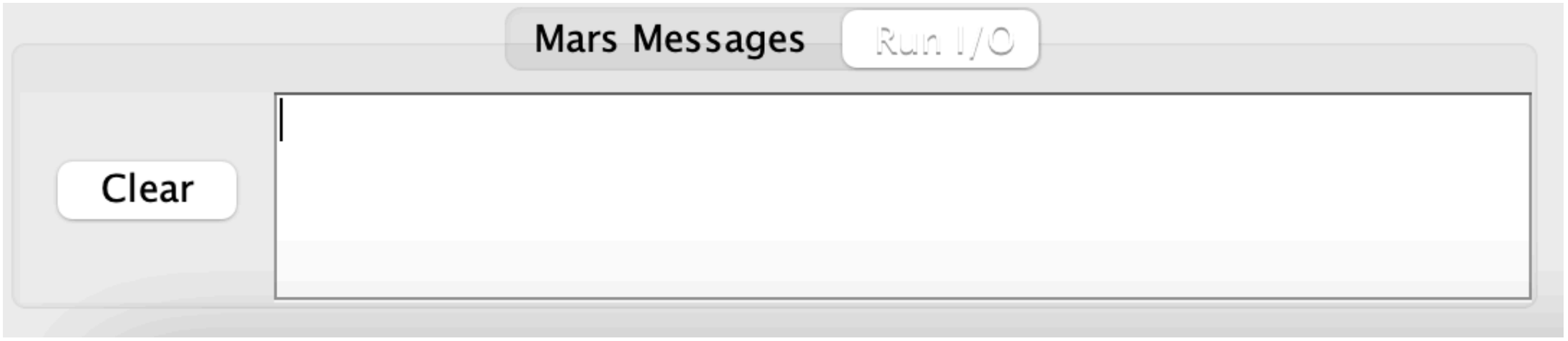
- Get a string from standard input (the keyboard).
- Example:

```
.data
message: .space 100      # Buffer 100 bytes for the input string

.text
        li $v0, 8
        la $a0, message
        li $a1, 100
        syscall
```

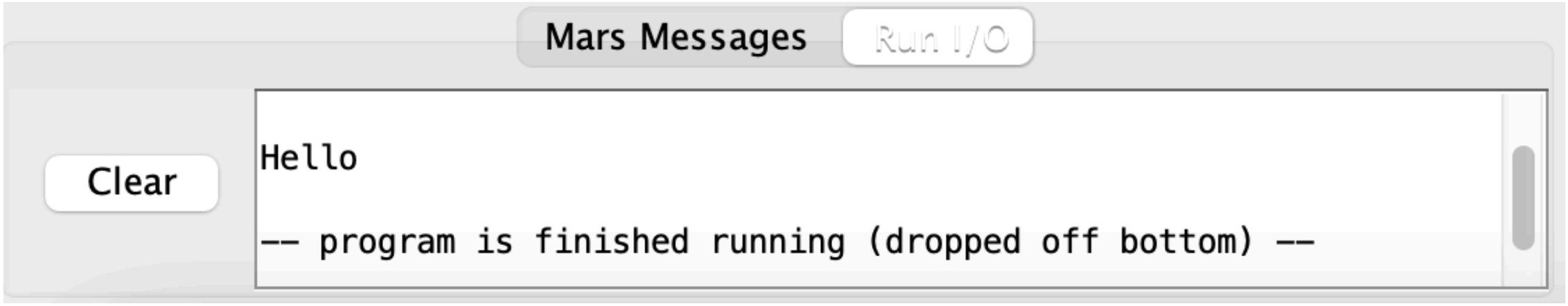
# 7. Read string

- Get a string from standard input (the keyboard).
- Example:



# 7. Read string

- Get a string from standard input (the keyboard).
- Input a string:



# 7. Read string

- Get a string from standard input (the keyboard).
- Result:

| Address | Value (+0) | | | | Value (+4) | | | |
|---|---|---|---|---|---|---|---|---|
| 0x10010000 | l | l | e | H | \0 | \0 | \n | o |
| 0x10010020 | \0 | \0 | \0 | \0 | \0 | \0 | \0 | \0 |

# 8. InputDialogString

- Show a message dialog to read a string with content parser.

*Argument(s):*

$v0 = 54
$a0 = address of the null-terminated message string
$a1 = address of input buffer
$a2 = maximum number of characters to read

*Return value:*

$a1 contains status value
0: OK status
-2: OK was chosen but no data had been input into field.
No change to buffer.
-3: OK was chosen but no data had been input into field
-4: length of the input string exceeded the specified maximum. Buffer contains the maximum allowable input string plus a terminating null.

# 8. InputDialogString

- Show a message dialog to read a string with content parser.

- Example:

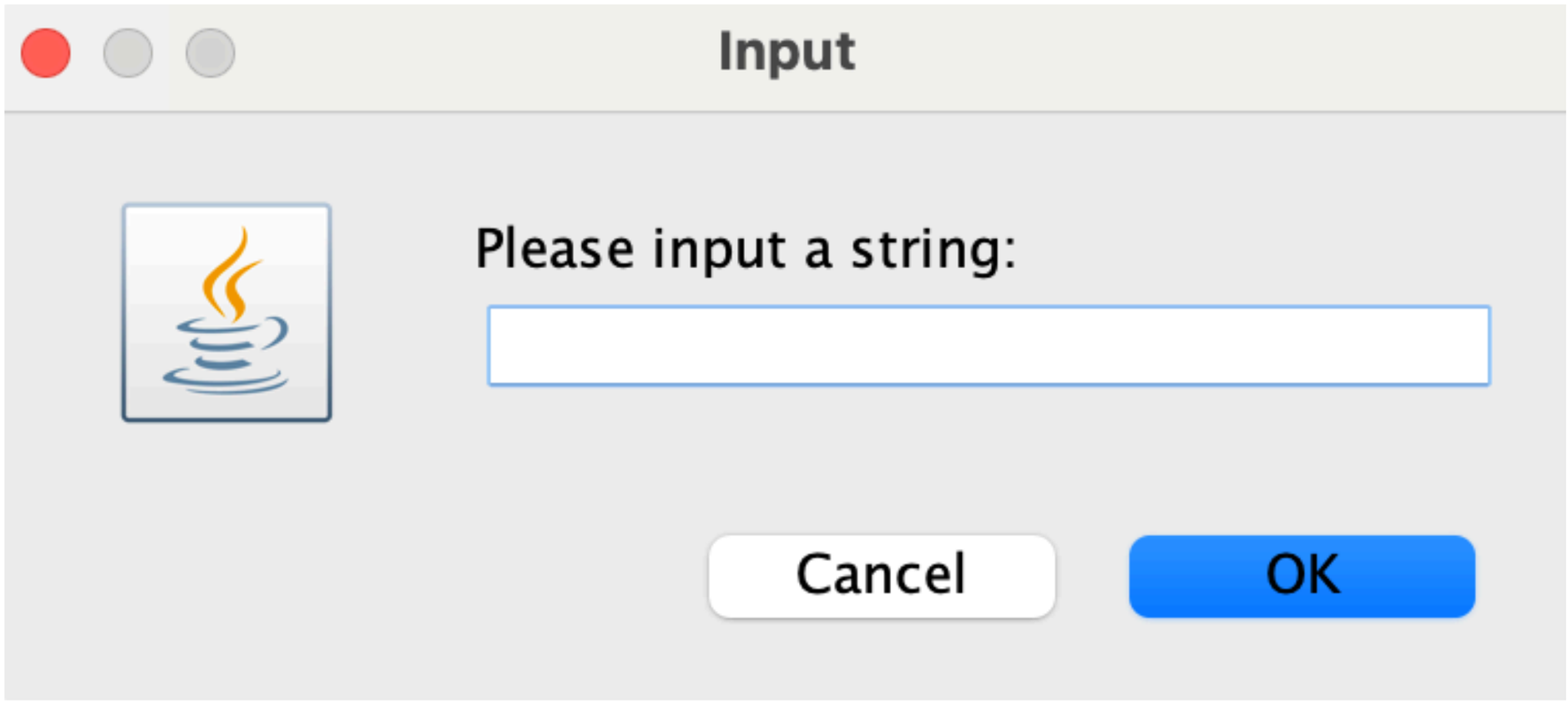
```
.data
message: .asciiz "Please input a string: "
string: .space 100

.text
        li $v0, 54
        la $a0, message
        la $a1, string
        la $a2, 100
        syscall
```

# 8. InputDialogString

- Show a message dialog to read a string with content parser.

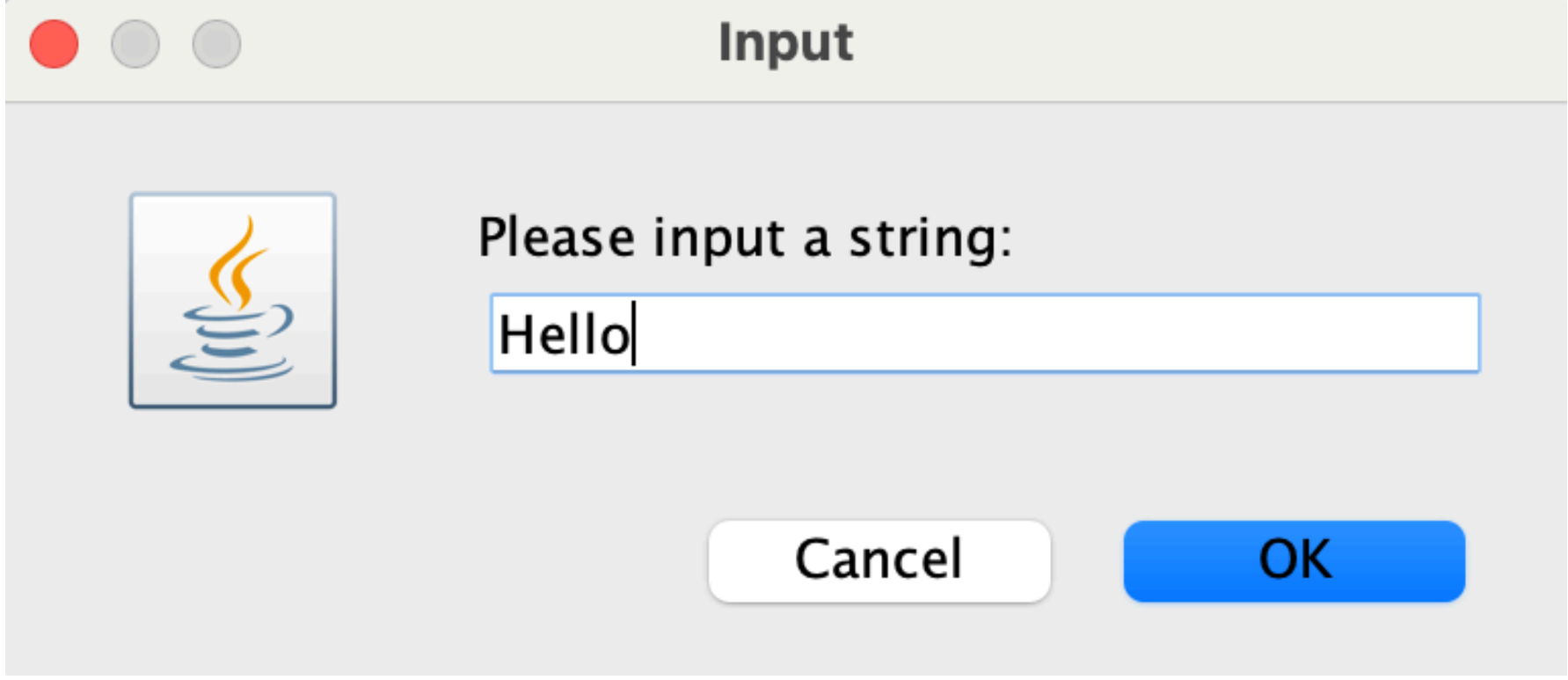
- Result:



# 8. InputDialogString

- Show a message dialog to read a string with content parser.

- Input a string:



# 8. InputDialogString

- Show a message dialog to read a string with content parser.

- Result:

Data Segment

| Address | Value (+0) | Value (+4) | Value (+8) | Value (+c) | Value (+10) | Value (+14) | Value (+18) | Value (+1c) |
|---|---|---|---|---|---|---|---|---|
| 0x10010000 | a e l P | i e s | t u p n | s a | n i r t | \0 : g | l l e H | \0 \0 \n o |
| 0x10010020 | \0 \0 \0 \0 | \0 \0 \0 \0 | \0 \0 \0 \0 | \0 \0 \0 \0 | \0 \0 \0 \0 | \0 \0 \0 \0 | \0 \0 \0 \0 | \0 \0 \0 \0 |
| 0x10010040 | \0 \0 \0 \0 | \0 \0 \0 \0 | \0 \0 \0 \0 | \0 \0 \0 \0 | \0 \0 \0 \0 | \0 \0 \0 \0 | \0 \0 \0 \0 | \0 \0 \0 \0 |
| 0x10010060 | \0 \0 \0 \0 | \0 \0 \0 \0 | \0 \0 \0 \0 | \0 \0 \0 \0 | \0 \0 \0 \0 | \0 \0 \0 \0 | \0 \0 \0 \0 | \0 \0 \0 \0 |

| Value (+18) | Value (+1c) |
|---|---|
| l l e H | \0 \0 \n o |
| \0 \0 \0 \0 | \0 \0 \0 \0 |

# 9. Print character

- Print a character to standard output (the console).

*Argument(s):*

$v0 = 11
$a0 = character to print (at the lowest significant byte)

*Return value:*

none

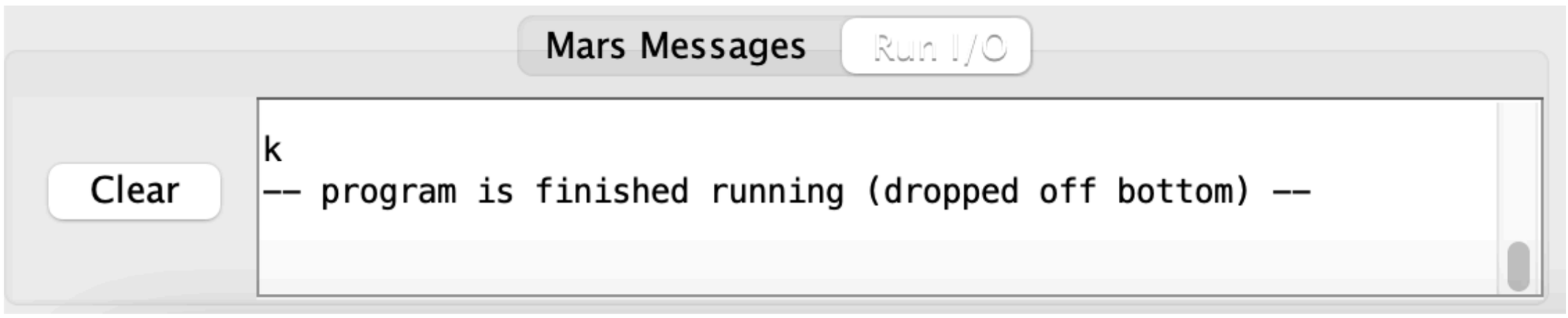
*Example:*

```
li $v0, 11
li $a0, 'k'
syscall
```

# 9. Print character

- Print a character to standard output (the console).

- Result:



# 10. Read character

- Get a character from standard output (the keyboard).

*Argument(s):*

$v0 = 12

*Return value:*

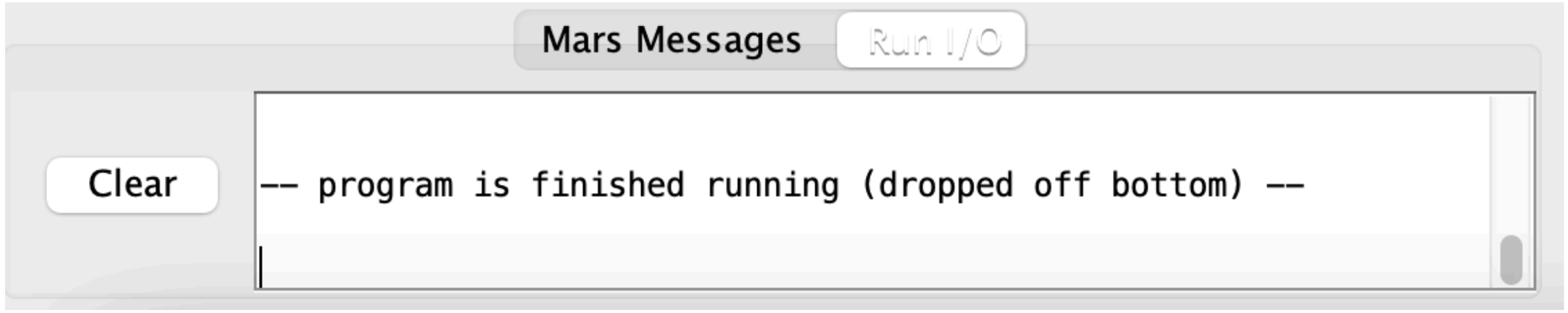
$v0 contains character read

*Example:*

```
li $v0, 12
syscall
```

# 10. Read character

- Get a character from standard output (the keyboard).
- Result:



# 11. ConfirmDialog

- Show a message bog with 3 button: Yes | No | Cancel

*Argument(s):*

$v0 = 50
$a0 = address of the null-terminated message string

*Return value:*

$a0 = contains value of user-chosen option
0: Yes
1: No
2: Cancel

# 11. ConfirmDialog

- Show a message bog with 3 button: Yes | No | Cancel.

- Example:

```
.data
message: .asciiz "Do you agree?"

.text
        li $v0, 50
        la $a0, message
        syscall
```

# 11. ConfirmDialog

- Show a message bog with 3 button: Yes | No | Cancel.

- Result:

# 12. MessageDialog

- Show a message bog with icon and button OK only.

*Argument(s):*

$v0 = 55
$a0 = address of the null-terminated message string
$a1 = the type of message to be displayed:
0: error message, indicated by Error icon
1: information message, indicated by Information icon
2: warning message, indicated by Warning icon
3: question message, indicated by Question icon
other: plain message (no icon displayed)

*Return value:*

none

# 12. MessageDialog

- Show a message bog with icon and button OK only.

- Example:

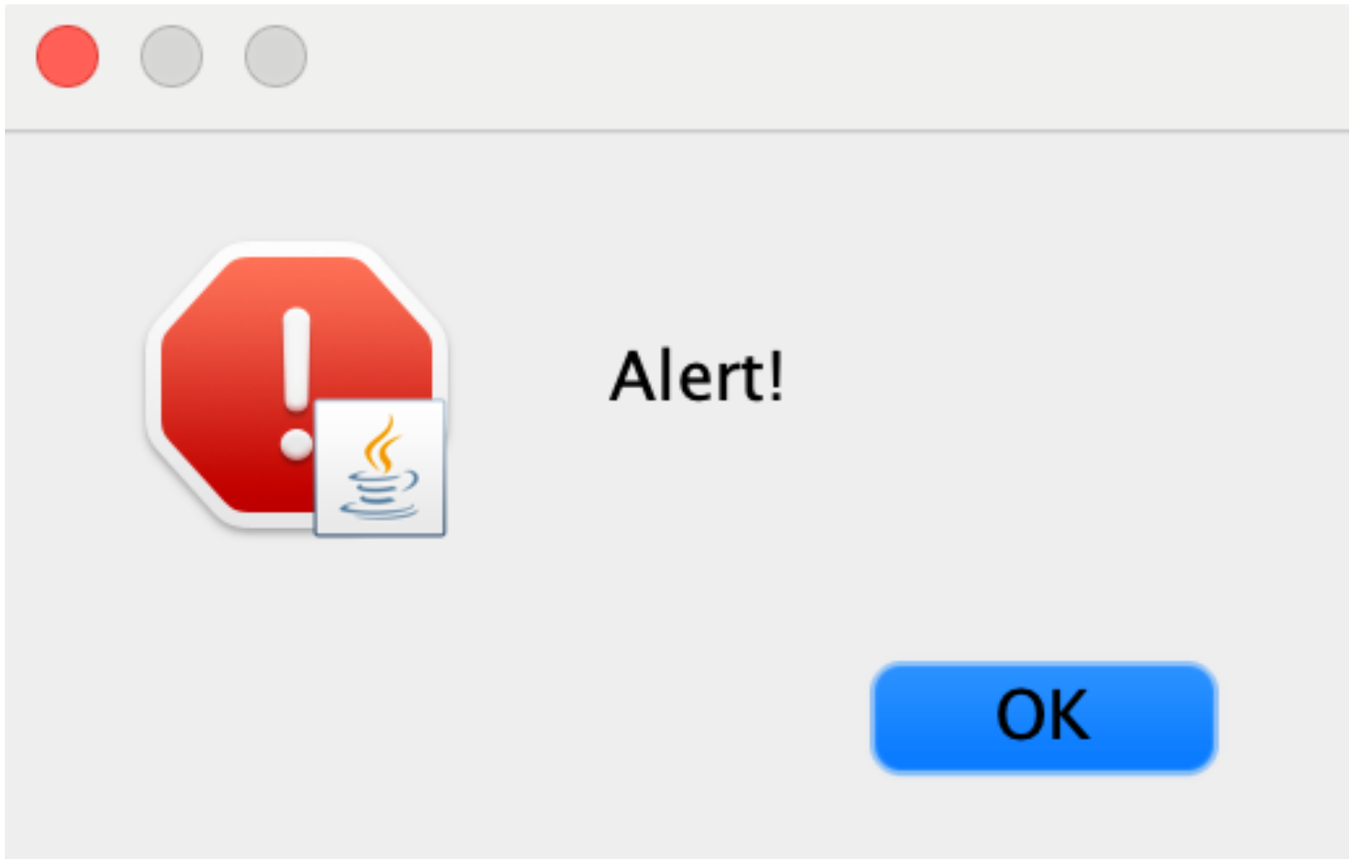
```
.data
message: .asciiz "Alert!"

.text
        li $v0, 55
        la $a0, message
        syscall
```

# 12. MessageDialog

- Show a message bog with icon and button OK only.

- Result:



# Using SYSCALL system services 31 and 33: MIDI output

- These system services are unique to MARS, and provide a means of producing sound.

- MIDI output is simulated by your system sound card, and the simulation is provided by the javax.sound.midipackage.

  - Service 31 will generate the tone then immediately return.

  - Service 33 will generate the tone then sleep for the tone's duration before returning. Thus it essentially combines services 31 and 32.

- This service requires four parameters as follows:

# Using SYSCALL system services 31 and 33: MIDI output

**pitch ($a0)**

- Accepts a positive byte value (0-127) that denotes a pitch as it would be represented in MIDI
- Each number is one semitone / half-step in the chromatic scale.
- 0 represents a very low C and 127 represents a very high G (a standard 88 key piano begins at 9-A and ends at 108-C).
- If the parameter value is outside this range, it applies a default value 60 which is the same as middle C on a piano.
- From middle C, all other pitches in the octave are as follows:
  - 61 = C# or Db
  - 62 = D
  - 63 = D# or Eb
  - 64 = E or Fb
  - 65 = E# or F
  - 66 = F# or Gb
  - 67 = G
  - 68 = G# or Ab
  - 69 = A
  - 70 = A# or Bb
  - 71 = B or Cb
  - 72 = B# or C
- To produce these pitches in other octaves, add or subtract multiples of 12.

# Using SYSCALL system services 31 and 33: MIDI output

**duration in milliseconds ($a1)**

- Accepts a positive integer value that is the length of the tone in milliseconds.
- If the parameter value is negative, it applies a default value of one second (1000 milliseconds).

# Using SYSCALL system services 31 and 33: MIDI output

**instrument ($a2)**

- Accepts a positive byte value (0-127) that denotes the General MIDI "patch" used to play the tone.
- If the parameter is outside this range, it applies a default value 0 which is an *Acoustic Grand Piano*.
- General MIDI standardizes the number associated with each possible instrument (often referred to as *program change* numbers), however it does not determine how the tone will sound. This is determined by the synthesizer that is producing the sound. Thus a *Tuba* (patch 58) on one computer may sound different than that same patch on another computer.
- The 128 available patches are divided into instrument families of 8:

| | | | |
|---|---|---|---|
| 0-7 | Piano | 64-71 | Reed |
| 8-15 | Chromatic Percussion | 72-79 | Pipe |
| 16-23 | Organ | 80-87 | Synth Lead |
| 24-31 | Guitar | 88-95 | Synth Pad |
| 32-39 | Bass | 96-103 | Synth Effects |
| 40-47 | Strings | 104-111 | Ethnic |
| 48-55 | Ensemble | 112-119 | Percussion |
| 56-63 | Brass | 120-127 | Sound Effects |

- Note that outside of Java, General MIDI usually refers to patches 1-128. When referring to a list of General MIDI patches, 1 must be subtracted to play the correct patch. For a full list of General MIDI instruments, see www.midi.org/about-midi/gm/gm1sound.shtml. The General MIDI channel 10 percussion key map is not relevant to the toneGenerator method because it always defaults to MIDI channel 1.

# Using SYSCALL system services 31 and 33: MIDI output

**volume ($a3)**

- Accepts a positive byte value (0-127) where 127 is the loudest and 0 is silent. This value denotes MIDI velocity which refers to the initial attack of the tone.
- If the parameter value is outside this range, it applies a default value 100.
- MIDI velocity measures how hard a *note on* (or *note off*) message is played, perhaps on a MIDI controller like a keyboard. Most MIDI synthesizers will translate this into volume on a logarithmic scale in which the difference in amplitude decreases as the velocity value increases.
- Note that velocity value on more sophisticated synthesizers can also affect the timbre of the tone (as most instruments sound different when they are played louder or softer).

# 13. MIDI out

- Make a sound

*Argument(s):*

$v0 = 31
$a0 = pitch (0-127)
$a1 = duration in milliseconds
$a2 = instrument (0-127)
$a3 = volume (0-127)

*Return value:*

Generate tone and return immediately

# 13. MIDI out

- Make a sound
- Example:

```
li $v0, 31
li $a0, 42      #pitch
li $a1, 2000    #time
li $a2, 0       #musical instrusment
li $a3, 212     #volume
syscall
```

# 14. MIDI out synchronous

- Make a sound

*Argument(s):*

$v0 = 33
$a0 = pitch (0-127)
$a1 = duration in milliseconds
$a2 = instrument (0-127)
$a3 = volume (0-127)

*Return value:*

Generate tone and return upon tone completion

# 14. MIDI out synchronous

- Make a sound.
- Example:

```
li $v0, 33
li $a0, 42      #pitch
li $a1, 2000    #time
li $a2, 0       #musical instrusment
li $a3, 212     #volume
syscall
```

# 15. Exit

- Terminated the software.

  - Make sense that there is no EXIT instruction in the Instruction Set of any processors.

  - Exit is a service belongs to Operating System.

*Argument(s):*

$v0 = 10

*Return value:*

none

# 15. Exit

- Terminated the software.
  - Make sense that there is no EXIT instruction in the Instruction Set of any processors.
  - Exit is a service belongs to Operating System.
- Example:

```
li $v0, 10 #exit
syscall
```

# 16. Exit with code

- Terminated the software.

  - Make sense that there is no EXIT instruction in the Instruction Set of any processors.

  - Exit is a service belongs to Operating System.

*Argument(s):*

$v0 = 17
$a0 = termination result

*Return value:*

none

# 16. Exit with code

- Terminated the software.

  - Make sense that there is no EXIT instruction in the Instruction Set of any processors.

  - Exit is a service belongs to Operating System.

- Example:

```
li $v0, 17 # exit
li $a0, 3  # with error code = 3
syscall
```

# Assigment 7.1

- The following simple assembly program will display a welcome string.

  - We use **printf** function for this purpose.

  - Read this example carefully, pay attention to the way to pass parameters for **printf** function.

  - Read Mips Lab Enviroment Reference for details.

# Assigment 7.1

- The following simple assembly program will display a welcome string.

```
.data
test: .asciiz "Hello World"

.text
        li   $v0, 4
        la   $a0, test
        syscall
```

# Assigment 7.1

- Create a new project to implement the above program.

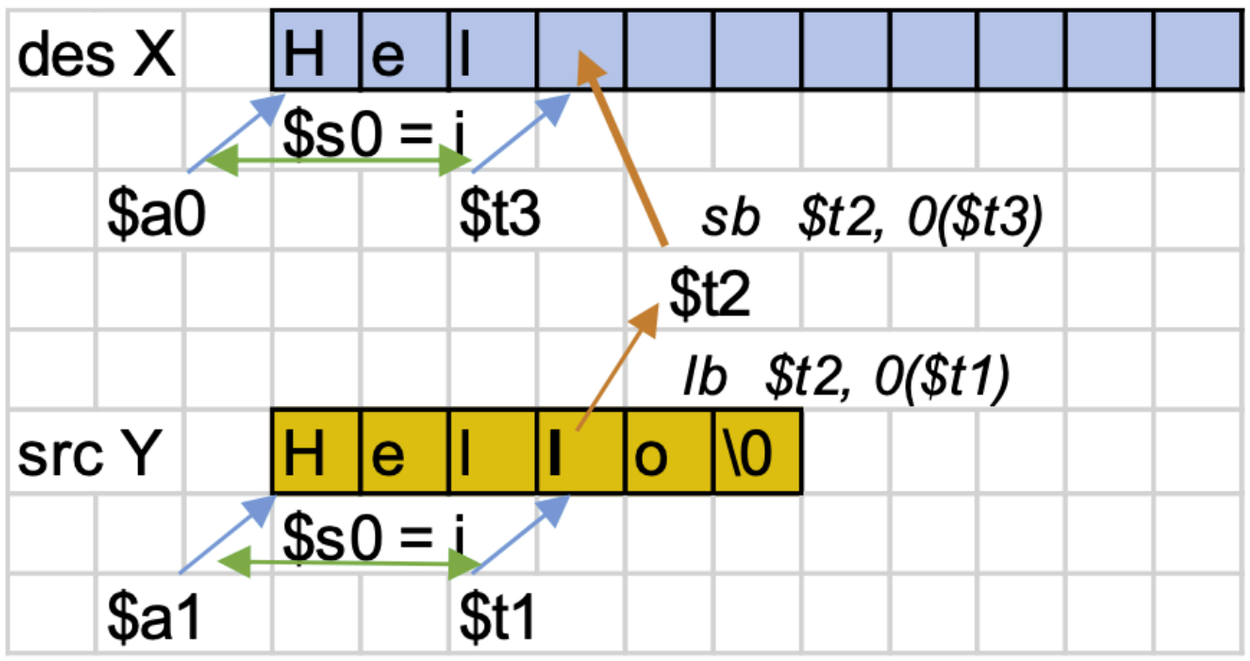
  - Compile and upload to simulator.

  - Run and observe the result.

  - Go to data memory section, check how test string are stored and packed in memory.

# Assigment 7.2

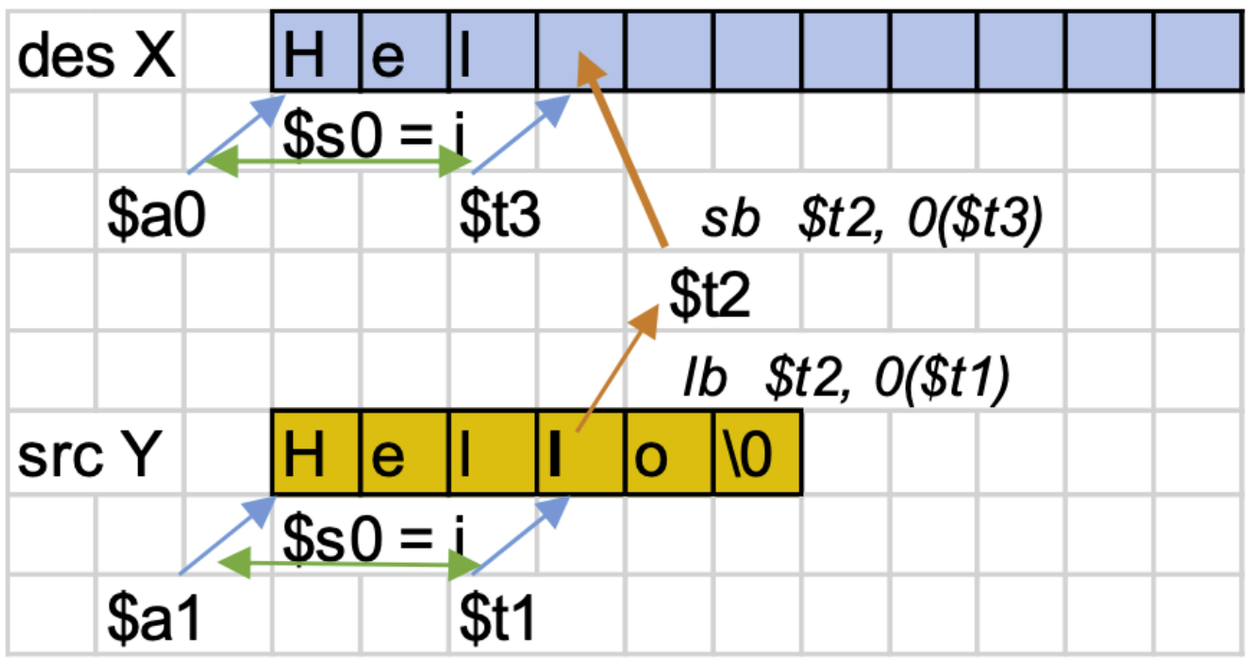
- Procedure **strcpy** copies **string y** to **string x** using the null byte termination convention of C.
- Read this example carefully, try to understand all of this code section.



# Assigment 7.2

- Procedure **strcpy** copies **string y** to **string x** using the null byte termination convention of C.

- Read this example carefully, try to understand all of this code section.



# Assigment 7.2

- Code:

```
.data
x: .space 1000              # destination string x, empty
y: .asciiz "Hello"          # source string y

.text
strcpy:
        add $s0,$zero,$zero             #s0 = i=0
L1:
        add $t1,$s0,$a1                 #t1 = s0 + a1 = i + y[0]
                                        # = address of y[i]
        lb $t2,0($t1)                   #t2 = value at t1 = y[i]
        add $t3,$s0,$a0                 #t3 = s0 + a0 = i + x[0]
                                        # = address of x[i]
        sb $t2,0($t3)                   #x[i]= t2 = y[i]
        beq $t2,$zero,end_of_strcpy     #if y[i]==0, exit
        nop
        addi $s0,$s0,1                  #s0=s0 + 1 <-> i=i+1
        j L1                            #next character
        nop
end_of_strcpy:
```

# Assigment 7.2

- Create a new project to print the sum of two register **$s0** and **$s1** according to this format:

"The sum of (**s0**) and (**s1**) is (**result**)"

# Assigment 7.3

- The following program count the length of a null-terminated string.
- Read this example carefully, analyse each line of code.

# Assigment 7.3

- Code:

```
.text
main:
get_string:                             # TODO
get_length:
        la $a0, string                  # a0 = Address(string[0])
        xor $v0, $zero, $zero           # v0 = length = 0
        xor $t0, $zero, $zero           # t0 = i = 0

check_char:
        add $t1, $a0, $t0               # t1 = a0 + t0
                                        #= Address(string[0]+i)
        lb   $t2, 0($t1)                # t2 = string[i]
        beq  $t2,$zero,end_of_str       # Is null char?
        addi $v0, $v0, 1                # v0=v0+1->length=length+1
        addi $t0, $t0, 1                # t0=t0+1->i = i + 1
        j    check_char
end_of_str:
end_of_get_length:
print_length:                           # TODO
```

# Assigment 7.3

- Create a new project to implement the above program.

  - Add more instructions to assign a test string for **y** variable, and implement **strcpy** function.

- Compile and upload to simulator.

- Run and observe the result.

# Assigment 7.4

- Accomplish the Assignment 7.3 with syscall function to:

  - Get a string from dialog
  - Show the length to message dialog

# Assigment 7.5

- Write a program that let user input a string.

  - Input process will be terminated when user press Enter or then length of the string exceed 20 characters.

  - Print the reverse string.

# Conclusions

Before you pass the laboratory exercise, think about the questions below:

- What the difference between the string in C and Java?
- In C, with 8 bytes, how many characters that we can store?
- In Java, with 8 bytes, how many characters that we can store?

# End of week 7