Programming 2 (SS 2023)
Saarland University
Faculty MI
Compiler Design Lab

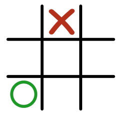
Sample Solution 2
Arithmetic and MIPS

Prof. Dr. Sebastian Hack
Pascal Lauer, M. Sc.
Marcel Ullrich, B. Sc.

The calendar indicates which script chapters you should study in conjuction with each lecture. The exercises are designed to enhance your understanding of the lecture material and prepare you for the mini-tests and the final exam. Additional exercises can be found at the end of each chapter in the script.
The difficulty of an exercise on the sheet is determined by the number of annotated 'X' and 'O' marks in the tic-tac-toe field, with four levels (1-4) increasing by one mark per level.

Arithmetic

Exercise 2.1:



We interpret the bit sequences as unsigned and signed numbers. Complete the following table.

$b \in \mathbb{B}^8$	$\langle b \rangle_{16}$	$\langle b \rangle$	$[b]$
01101111			
	0xab		
		127	
			-128
11001001			
	0xff		
		64	
			127

Solution

$b \in \mathbb{B}^8$	$\langle b \rangle_{16}$	$\langle b \rangle$	$[b]$
01101111	0x6f	111	111
10101011	0xab	171	-85
01111111	0x7f	127	127
10000000	0x80	128	-128
11001001	0xc9	201	-55
11111111	0xff	255	-1
01000000	0x40	64	64
01111111	0x7f	127	127

Exercise 2.2:

1. State the decimal representation for the following binary numbers:

   (a) $\langle 1010\rangle$

   (b) $[1010]$

   (c) $\langle 010\rangle$

   (d) $[010]$

   (e) $\langle 1111\rangle$

   (f) $[1111]$

   (g) $[01111]$

2. Compute:

   (a) $\langle 111\rangle + \langle 1110\rangle$

   (b) $\langle 111 + 1110\rangle$

   (c) $\langle 111 +_4 1110\rangle$

   (d) $[1110] - [1100]$

   (e) $[1110 - 1100]$

   (f) $[1110 -_4 1100]$

   (g) $[100110] + [10000]$

   (h) $[100110 + 10000]$

   (i) $[100110 +_5 10000]$

   (j) $[100110 +_6 10000]$

3. Compute

   - $[1101] + [0011]$,
   - $[1101 + 0011]$ and
   - $[1101 +_4 0011]$.

   What do you notice? How can you explain these results?

Solution

1. (a) $\langle 1010\rangle = 10$

   (b) $[1010] = -6$

   (c) $\langle 010\rangle = 2$

   (d) $[010] = 2$

   (e) $\langle 1111\rangle = 15$

   (f) $[1111] = -1$

   (g) $[01111] = 15$

2. (a) $\langle 111\rangle + \langle 1110\rangle = 7 + 14 = 21$

   (b) $\langle 111 + 1110\rangle = \langle 10101\rangle = 21$

   (c) $\langle 111 +_4 1110\rangle = \langle 0101\rangle = 5$

   (d) $[1110] - [1100] = -2 - -4 = 2$

(e) $[1110 - 1100] = [1110 + 0100] = [10010] = -14$

(f) $[1110 -_4 1100] = [0010] = 2$

(g) $[100110] + [10000] = -26 + -16 = -42$

(h) $[100110 + 10000] = [110110] = -10$

(i) $[100110 +_5 10000] = [10110] = -10$

(j) $[100110 +_6 10000] = [110110] = -10$

3. - $[1101] + [0011] = -3 + 3 = 0$
   - $[1101 + 0011] = [10000] = -16$
   - $[1101 +_4 0011] = [0000] = 0$

   The numbers are congruent modulo 16 and we witness an overflow using $+_4$.

Exercise 2.3:

Complete the following table by extending the given bit sequences b to a sequence b' such that b' has length 6. Ensure that the condition on top of the corresponding column holds.

b	$\langle b \rangle = \langle b' \rangle$	$[b] = [b']$	$\langle b \rangle = [b']$
0010			
1001			
0111			
1111			
1011			

Solution

b	$\langle b \rangle = \langle b' \rangle$	$[b] = [b']$	$\langle b \rangle = [b']$
0010	000010	000010	000010
1001	001001	111001	001001
0111	000111	000111	000111
1111	001111	111111	001111
1011	001011	111011	001011

Exercise 2.4:

As discussed in the lecture, you can express the multiplication of a bit sequence with 2 using a left shift. Analogously it is possible to express division by 2 using shifts.

1. For any given bit sequence b, state a bit sequence b' such that $\langle b' \rangle = \langle b \rangle \cdot 4$.

b	$\langle b' \rangle = \langle b \rangle \cdot 4$	$\langle b' \rangle = \langle b \rangle \cdot 8$
0001001		
0000011		
0001110		

2. Complete the following table.

b	$\langle b' \rangle = \lfloor \frac{\langle b \rangle}{4} \rfloor$	$[b'] = \lfloor \frac{[b]}{4} \rfloor$
101101		
010011		
101010		
111111		

Solution

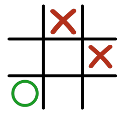
1. Shift the bit sequences two times or three times, respectively, to the left:

b	$\langle b' \rangle = \langle b \rangle \cdot 4$	$\langle b' \rangle = \langle b \rangle \cdot 8$
0001001	0100100	1001000
0000011	0001100	0011000
0001110	0111000	1110000

2. Shift the bit sequence two times to the right. For signed bit sequences the most significant bit stays the same and depending on it, the sequence is padded with 0 or 1.

b	$\langle b' \rangle = \lfloor \frac{\langle b \rangle}{4} \rfloor$	$[b'] = \lfloor \frac{[b]}{4} \rfloor$
101101	001011	111011
010011	000100	000100
101010	001010	111010
111111	001111	111111

Exercise 2.5:



1. State an expression of bit operations that produces the bit sequence

$$0^{n-(j-i+1)} a_j \ldots a_i$$

given a bit sequence $a = a_{n-1} \ldots a_0$, where $i < j < n$. In other words, the expression should *extract* the bit sequence $a_j \ldots a_i$.

2. State an expression that computes

$$a_{i-1} \ldots a_0 b_{n-1} \ldots b_i$$

given bit sequences $a = a_{n-1} \ldots a_0$, $b = b_{n-1} \ldots b_0$ and an index $i \in [0, n-1]$. Use only bit operations on bit sequences of length n.

3. Give an expression that multiplies a bit string a with 4, 5 and 31, respectively.

Solution

1. Since we have $n > j > i$, the bit sequence has the form $a_{n-1} \ldots a_j \ldots a_i \ldots a_0$. To extract $a_j \ldots a_i$ we have to remove the bits left of a_j and move the sequence to the very right. To remove the bits left of a_j, we do a left shift by $n-1-j$. Then do a right shift by $n-1-j+i$ (undo the left shift plus the shift by i). The final expression is:

$$\underbrace{(a \ll_n (n-1-j))}_{\text{result of left shift}} \gg_n (\underbrace{(n-1-j)}_{\text{undo left shift}} + \underbrace{i}_{\text{shift to the end}})$$

2. a is shifted to the left by $n-i$ and b is shifted to the right by i. Now we can compose a and b using $|_n$, because both are padded by zeros on right or left, respectively. The expression is:

$$(a \ll_n n-i) \;|_n\; (b \gg_n i)$$

3. The idea is to disassemble the desired multiplication into a sum of multiplications by powers of 2, since these can be computed using bit shifts.

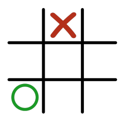
   - $a \ll_n 2$
   - $5 = 4 + 1 = 2^2 + 1$, thus $(a \ll_n 2) +_n a$
   - $31 = 16 + 8 + 4 + 2 + 1 = 2^3 + 2^2 + 2^1 + 2^0$, thus

$$(a \ll_n 4) +_n (a \ll_n 3) +_n (a \ll_n 2) +_n (a \ll_n 1) +_n a.$$

   Alternatively, use $-_n$ $(31 = 32 - 1)$:

$$(a \ll_n 5) -_n a$$

Exercise 2.6:



Proof for all bit strings $b_{n-1} \ldots b_0 \in \mathbb{B}^n$:

$$b_{n-1} = 1 \text{ if and only if } [b] < 0.$$

Solution

Let $b \in \mathbb{B}^n$.

$$\begin{aligned} [b] &= -\langle b_{n-1} \rangle \times 2^{n-1} + \langle b_{n-2} \cdots b_0 \rangle \\ &= -2^{n-1} + \langle b_{n-2} \cdots b_0 \rangle && | b_{n-1} = 1 \\ &< 0 && | \langle b_{n-2} \cdots b_0 \rangle < 2^{n-1} \end{aligned}$$

If $[b] < 0$ then we have $bn - 1 = 1$ since b_{n-1} is the only negative summand.

MIPS

Exercise 2.7:

This exercise deals with important terms and concepts from the script. Answer the following questions with the knowledge from the script.

1. What is the difference between `$1` and `1` in a MIPS instruction?
2. Which categories of MIPS instructions exist?
3. Why are there four addition instructions `add`, `addi`, `addu` and `addiu`?
4. What is `.text`? Which other directives have you learned or can you find in the script?
5. What purpose does for example `blub:` serve in MIPS? Which disadvantages do absolute values have when used instead of labels in instructions?

Solution

1. `$1` designates a register, whereas `1` is a constant.
2. The lecture discussed four categories in total:
   - Computational instructions: With these instructions, computational operations on registers are realized. For example: `addu`, `subu`, `and`, `slt`.
   - Memory instructions: These transfer data between the memory and the registers. For example: `lw`, `lh`, `sb`.
   - Branch instructions: With these instruction, the program's next instruction is influenced by conditionally changing the program counter. Other instructions than the one after the current instruction can be targeted and jumped to. For example `beq`, `bne`, `jr`.
   - Pseudo instructions: Instructions which exist to make the programmer's life easier. They can be expressed as several primitive instructions. For example: `li`, `la`, `not`.
3. There are two main reasons as to why these instructions exist. An `i` (immediate) is used when a register and a constant is used instead of two registers. A `u` means that the result of an addition shall be interpreted as unsigned, in which case a signed overflow does not throw an exception in contrast to `add`. If no overflow occurs, `addu` and `add` behave equally. If we for example assume 4-bit integers, then we can represent the numbers 0-15 using unsigned interpretation, and the numbers -8 to 7 using signed interpretation.

					unsigned	signed
	1	0	0	1	9	-7
+	0	0	1	1	3	3
=	1	1	0	0	12	-4

   → No signed overflow, so `add` and `addu` behave equally.

						unsigned	signed
		0	1	1	0	6	6
+		0	0	1	1	3	3
	0	1	1	0			
=		1	0	0	1	9	-7

→ Signed overflow ($6 + 3 = -7$), resulting in an exception of `add`. With unsigned interpretation, the result is correct and `addu` does not raise an exception. A signed overflow can easily be identified using the last two carry bits (read from right to left). If the bits differ, a signed overflow occured, otherwise this is not the case.

						unsigned	signed
		1	1	1	1	15	-1
+		1	1	0	0	12	-4
	1	1	0	0			
=		1	0	1	1	11	-5

→ No signed overflow, so `add` and `addu` behave the same and no exception is raised. However, when inspecting the unsigned result, one notices that an unsigned overflow has occured ($15+12 = 11$). Thus, when working with unsigned numbers, one has to manually identify such overflows. This is however easy check, as an unsigned overflow occured if the result is smaller than one of the operands (unsigned comparison!). Here, we have $11 < 15$, so an overflow occured.

						unsigned	signed
		1	0	0	1	9	-7
+		1	0	1	0	10	-6
	1	0	0	0			
=		0	0	1	1	3	3

→ Signed overflow, causing `add` to raise an exception, which `addu` does not raise. However, an unsigned overflow also occurs.

4. The directive `.text` marks the start of the code segment. Another special directive is `.data`, which marks the start of the data segment. There are also the directives `.ascii`, `.asciiz`, `.byte`, `.half` and `.word`, which all store objects of a specific size in memory. To allocate storage, the directive `.space` can be used. With `.align`, the next data entry is aligned. Lastly, `.globl` makes a symbol visible for other files.

5. Labels are used to mark points in the code to which branch instructions can jump. They serve as an aid for the programmer to make the code more readable. If a program is written without labels, the program becomes hard to maintain. For example, if an instruction is added in code which uses absolute values for jump targets, then it may be that the addresses are incorrect after the instruction is inserted.

Exercise 2.8:

Do your first steps with MARS:

1. Load the number 5 into the register `$t0`.
2. Load the number 11 into the register `$t1`.
3. Store the result of the addition of both numbers in the register `$t2`.
4. Subtract the number in register `$t0` from the number in register `$t1` and store the result in register `$t3`.
5. Subtract the same numbers in reverse order and store the result in register `$t4`.
6. Print the last result.
7. Override the registers `$t2 - $t4` with the value `0` without using `li`. Find 2 different possibilities.

Solution

1. `li $t0 5`
2. `li $t1 11`
3. `add $t2 $t1 $t0`
4. `sub $t3 $t1 $t0`
5. `sub $t4 $t0 $t1`
6. ```
   li $v0 1
   move $a0 $t4
   syscall
   ```
7. Three options:
   - `move $t2 $zero`
   - `addiu $t3 $zero 0`
   - `sub $t4 $t4 $t4`
   ```

Exercise 2.9:

Consider the following Mips program:

```
.text
# 6 in $a0
# 3 in $a1
start:
  beq $a0  $zero a0iszero
loop:
  beq $a1  $zero a1iszero
  bgt $a0 $a1    mid
  sub $a1 $a1    $a0
  b      loop
mid:
  sub $a0 $a0    $a1
  b      loop
a0iszero:
  add $a0 $a0    $a1
a1iszero:
  and $v0 $v0    $zero
  or   $v0 $v0    $a0
```

Since your Mips interpreter is currently not available and you need to know what this programs computes, you have to write down an execution protocol. The code has been loaded to the address 0x00400000 in memory.

1. Complete the following execution protocol. State the content for every relevant register for every step until the program terminates.

Step	`$v0`	`$a0`	`$a1`	`pc`
1.		6	3	0x00400000
2.	...	...	...	...

2. Give a mathematical definition for the function computed by the program:

$$f(a,b) = \begin{cases} \dots \end{cases}$$

Solution

1. The complete execution protocol:

Step	`$v0`	`$a0`	`$a1`	`pc`
1.		6	3	0x00400000
2.		6	3	0x00400004
3.		6	3	0x00400008
4.		6	3	0x00400014
5.		3	3	0x00400018
6.		3	3	0x00400004
7.		3	3	0x00400008
8.		3	3	0x0040000c
9.		3	0	0x00400010
10.		3	0	0x00400004
11.		3	0	0x00400020
12.	0	3	0	0x00400024
13.	3	3	0	0x00400028

2. Euclidean algorithm to compute the greatest common divider:

$$\mathrm{gcd}(a,b) = \begin{cases} a & \text{wenn } b = 0 \\ b & \text{wenn } a = 0 \\ \mathrm{gcd}(a-b,b) & \text{wenn } a > b \\ \mathrm{gcd}(a,b-a) & \text{wenn } b \geq a \end{cases}$$

Exercise 2.10:

Take a look at the following MIPS program. Try to solve this exercise without using MARS.

```
 1 main:
 2   li $a1 5
 3   li $a2 10
 4   li $a3 7
 5
 6   sltu $t1 $a1 $a2
 7   beqz $t1 case2
 8
 9 case1:
10   sltu $t1 $a2 $a3
11   beqz $t1 reta2
12   b reta3
13
14 case2:
15   sltu $t1 $a3 $a1
16   beqz $t1 reta3
17   b reta1
18
19 reta1:
20   addu $a0 $0 $a1
21   b end
22
23 reta2:
24   addu $a0 $0 $a2
25   b end
26
27 reta3:
28   addu $a0 $0 $a3
29   b end
30
31 end:
32   li $v0 1
33   syscall
34   li $v0 10
35   syscall
```

1. What value is stored in register `$a0` after execution of this code?
2. What does the program compute for general values in `$a1`, `$a2` and `$a3` (assuming the program loads different values into the registers in lines 2-4)?
3. Two of the instructions can be stripped from the program without changing the semantics (for arbitrary values of `$a1`, `$a2` and `$a3`). Which lines can be removed?

Solution

1. 10
2. The maximum of the three numbers.
3. The branches in line 17 and 29, since they only jump to the next instruction.

Exercise 2.11:

The MIPS assembler offers a variety of pseudo-instructions. These are helpful instructions which are not implemented in hardware, but can easily be expressed as one or two existing instructions. Specify the implementations of the pseudo-instructions `blt`, `bgt`, `ble`, `neg`, `not`, `bge`, `li`, `la`, `lw`, `move`, `sge` and `sgt`.

Solution

- `blt $t8 $t9 label`:

```
slt $at $t8 $t9
bne $at $zero label
```

- `bgt $t8 $t9 label`:

```
slt $at $t9 $t8
bne $at $zero label
```

- `ble $t8 $t9 label`:

```
slt $at $t9 $t8
beq $at $zero label
```

- `neg $t8 $t9`:

```
sub $t8 $zero $t9
```

- `not $t8 $t9`:

```
nor $t8 $t9 $zero
```

- `bge $t8 $t9 label`:

```
slt $at $t8 $t9
beq $at $zero label
```

- `li $t8 i`: An instruction can contain an immediate that is at most 16 bits long. Hence, the constant must be split in two halves and copied in the register when it is larger.

$$i = \langle b_{31} \cdots b_0 \rangle$$
$$i_u = \langle b_{31} \cdots b_{16} \rangle$$
$$i_l = \langle b_{15} \cdots b_0 \rangle$$

If $i < 2^{16} - 1$:

```
ori $t8 $zero i
```

If $i \geq 2^{16} - 1$:

```
lui $at iu
ori $t8 $at il
```

- `la $t8 i($t9)`:

```
ori $at $zero i
add $t8 $t9 $at
```

- `lw $t8 add`: Depending on the size of the address, it must be loaded in two steps, similar to `li`.
- `move $t8 $t9`:

```
addu $t8 $zero $t9
```

- `sge $t8 $t9 $t1`:

```
slt $at $t9 $t1
xori $t8 $at 1
```

- `sgt $t8 $t9 $t1`:

```
slt $t8 $t1 $t9
```

Exercise 2.12:

Translate the following function written in pseudo code into a MIPS function:

```
EUCLID(a,b)
  while b != 0
    h = a mod b
    a = b
    b = h
  return a
```

Solution

How to solve this task:

1. First, define the function `EUCLID`. It gets two arguments.
2. The function uses in total 3 variables. In Mips they will be stored in registers. We assign registers for each variable:

`a`	`$a0`
`b`	`$a1`
`h`	`$t0`

   Since `a` and `b` are arguments they have to be in the corresponding registers. `h` is a local variable, thus it should reside in a register for temporary values.
3. Since there is a loop in the program we have to write code that ensures that the "body" of the loop is executed repeatedly as long as the condition at the "head" of the loop is satisfied. You can either put the check of the loop condition in front of or behind the body of the loop. In the latter case you have to jump there before the initial execution of the loop.
4. Within the body of the loop we have to implement the modulo computation using `rem`, as well as the re-assignments of `a` and `b` using `move`.
5. Finally, it is important to return from the function. It is essential to move the return value to the designated register `$v0`. Otherwise a function calling "EUCLID" has no way to retrieve the result.

condition in the beginning

```
EUCLID:
loop:
beqz $a1 end
rem $t0 $a0 $a1
move $a0 $a1
move $a1 $t0
b loop
end:
move $v0 $a0
jr $ra
```

condition at the end:

```
EUCLID:
b condition
loop:
rem $t0 $a0 $a1
move $a0 $a1
move $a1 $t0
condition:
bnez $a1 loop
end:
move $v0 $a0
jr $ra
```

Exercise 2.13: Copy > 7

Define a MIPS function that copies all elements > 7 from an input buffer to an output buffer and returns the number of copied elements.
Every elements is 2 bytes large and is interpreted as unsigned.
You can assume that both buffers are large enough.
To do this, complete the following code:

```
# Arguments:
#   $a0: Address of the input buffer.
#   $a1: Number of elements in the input buffer.
#   $a2: Address of the output buffer.
# Return:
#   $v0: Number of copied elements
copy_greater_seven:
```

Solution

Intuitively, we want to write the following C code in MIPS (Assumption: 1 Byte = 8 Bit):

```
#include <stdint.h> /* Header for uint16_t (unsigned, 16 bit integer) */

int copy_greater_seven(uint16_t* input_buffer /* $a0 */,
                       int number_of_elements /* $a1 */,
                       uint16_t* output_buffer /* $a2 */)
{
    register int copied = 0; /* $v0 */

    while(number_of_elements){
        register uint16_t cur = *input_buffer; /* $t0 */
        if(cur > 7){
            *output_buffer = cur;
            // Because one entry is 2 bytes in size, go ++ 2 bytes forward
            output_buffer ++;
            copied ++;
        }
        // Because one entry is 2 bytes in size, go ++ 2 bytes forward to
        // check next element in new iteration
        input_buffer ++;
        number_of_elements --;
    }
    return copied;
}
```

MIPS code (Related C code is added to the right of the instructions):

```
# Arguments:
#   $a0: Address of the input buffer.
#   $a1: Number of elements in the input buffer.
#   $a2: Address of the output buffer.
# Return:
#   $v0: Number of copied elements
copy_greater_seven:

    # $v0 Set number of already copied elements to 0
    and $v0 $v0 $zero # register int copied = 0;

    copy_greater_seven_loop:

        # if $a1 == 0 (looked at all values), jump to the end
        beqz $a1 copy_greater_seven_done # while(number_of_elements) {

        # ------------------------------------------------------
        #      There are still elements in the input buffer,
        #               that have to be looked at
        # ------------------------------------------------------

        # Load current element into t0
        lhu $t0 ($a0) # register uint16_t cur = *input_buffer;
        # Skip copying element if <= 7
        ble $t0 7 copy_greater_seven_loop_copy_done # if(cur > 7) {

            # ---------------------------------------------
            # Current element is > 7 (has to be copied)
            # ---------------------------------------------

            # Write element to output buffer
            sh $t0 ($a2)    # *output_buffer = cur;
            # Go forward one element (= 2 bytes) in the output buffer
            addiu $a2 $a2 2 # output_buffer ++;
            # Increase number of already copied elements by 1
            addiu $v0 $v0 1 # copied ++;

        copy_greater_seven_loop_copy_done: # }

        # --------------------------------------------
        #       Element was copied, if required
        # --------------------------------------------

        # Look at next address in the output buffer
        addiu $a0 $a0 2 # input_buffer ++;
        # Decrease number of remaining elements by 1
        subiu $a1 $a1 1 # number_of_elements --;
        # Jump back to the beginning of the loop
        b copy_greater_seven_loop # }

    copy_greater_seven_done:

    # ---------------------------------------------------
    #     Every required element was copied, finished!
    # ---------------------------------------------------

    # Jump back to the caller
    jr $ra # return copied;
```

Exercise 2.14:

Take a look at the MIPS program below.

1. How do linked lists work in MIPS? Complete the following program, which should add up all the numbers contained in the linked list and print the result to the console.

```
.text
# $a0: address of first element

add_numbers:

    # TODO
```

2. Now you should implement an algorithm that checks if a linked list ends in a cycle. The algorithm you should use is called "the tortoise and the hare".

   "In each step, the tortoise advances one list element and the hare advances two elements. Now the claim is that the tortoise and the hare meet (both pointers are equal) if and only if the list ends in a cycle. It is straightforward that they never meet if there is no cycle. But why do they meet if there is a cycle?

   Assume that the list has n elements and ends in a cycle of l nodes. So, after $n - l$ steps, the tortoise reaches element 0 of the cycle. This element is the only element to which a pointer from outside the cycle points to. Assume that the hare is at element d of the cycle. Now, in each step the distance from hare to tortoise will *decrease* by one. (The hare is in fact moving one element away from the tortoise in each step, but because they run on the cycle, it is actually moving closer.) So, after $l - d$ steps they stand on the same element."

```
.text
.globl tortoise_and_hare

# $a0: address of first element

tortoise_and_hare:
    # $a0 Hare
    # $a1 Tortoise
    move  $a1 $a0
loop:

      # TODO

cyclic:
    li   $v0 1
    jr   $ra
not_cyclic:
    li   $v0 0
    jr   $ra
```

3. Change your algorithm so that the hare advances 3 instead of 2 steps each time. Also, you can write a main function to test your implementation with an example list.

4. Does that still work in all cases (every circle length)? Justify your answer. If you think this combination does work for all cases, can you find one where it does not?

Solution

1. The following code is correct:

```
.text
# $a0: address of first element

add_numbers:
    move  $t0  $a0
```

```
    li  $t2 0           # result
loop:
    lw  $t1 ($t0)       # element
    add $t2 $t2 $t1
    lw  $t0 4($t0)      # next address
    beqz  $t0 end
    b loop
end:
    li  $v0 1
    move  $a0 $t2
    syscall
```

2. Please have a look at the script, chapter 3.5.

3. The following code is correct:

```
.text
.globl tortoise_and_hare

# $a0: address of first element
tortoise_and_hare:
    # $a0 Hare
    # $a1 Tortoise
    move  $a1 $a0
loop:
    beqz  $a0 not_cyclic
    lw  $a0 4($a0)
    beqz  $a0 not_cyclic
    lw  $a0 4($a0)
    beqz  $a0 not_cyclic
    lw  $a0 4($a0)
    lw  $a1 4($a1)
    beq  $a0 $a1 cyclic
    b  loop
cyclic:
    li  $v0 1
    jr  $ra
not_cyclic:
    li  $v0 0
    jr  $ra

    .data
L1:
    .word 1
    .word L6
L2:
    .word 2
    .word L4
L3:
    .word 3
    .word L5
L4:
    .word 4
    .word L3
L5:
    .word 5
    .word 0
L6:
    .word 6
    .word L2

```

```
.text
.globl   main
main:
    la       $a0 L1
    jal      tortoise_and_hare
    move     $a0 $v0
    li       $v0 1
    syscall
    li       $v0 10
    syscall
```

4. Yes, it does. The algorithm does only work if the greatest common divisor of the step size of tortoise and hare is not a divisor (or 1) of the length of the circle. Since the greatest common divisor of 1 and 3 is 1, this condition is fulfilled.

   So yes, there exist combinations where the algorithm fails, e.g. if the step size of the tortoise is 2 and the one of the hare is 4, we will not be able to find cycles of even size.

Exercise 2.15:

Given the natural numbers from 1 to 49, sift out the prime numbers using the Sieve of Eratosthenes Algorithm discussed in Chapter 3.2. Start by filling a table with the numbers from 1 to 49.

Solution

This is the initial table.

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	32	33	34	35
36	37	38	39	40	41	42
43	44	45	46	47	48	49

No we cross out all the multiples of 2.

1	2	3		5		7
	9		11		13	
15		17		19		21
	23		25		27	
29		31		33		35
	37		39		41	
43		45		47		49

No we cross out all the multiples of 3.

1	2	3		5		7
			11		13	
		17		19		
	23		25			
29		31				35
	37				41	
43				47		49

We do not need to consider the multiples of 4 because every multiple of 4 is also a multiple of 2 and thus already crossed out. So now we cross out all the multiples of 5.

1	2	3		5		7
			11		13	
		17		19		
	23					
29		31				
	37				41	
43				47		49

We do not need to consider the multiples of 6 because every multiple of 6 is also a multiple of 2 and thus already crossed out. So now we cross out all the multiples of 7.

1	2	3		5		7
			11		13	
		17		19		
	23					
29		31				
	37				41	
43				47		

We know that we can stop crossing out as soon as we reach the square root of 49. Thus, the prime numbers between 1 and 49 are: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47

Exercise 2.16:

In the following table, 32-bit hexadecimal numbers are paired with their representations in Little and Big Endian. Fill in the missing table entries.

Hexadecimal	Little Endian	Big Endian
0x11223344		
	DE AD C0 DE	
		F0 05 BA 11
0xDECAFF01		
	C0 FF EE 00	
		BA 5E BA 11

Solution

Hexadecimal	Little Endian	Big Endian
0x11223344	44 33 22 11	11 22 33 44
0xDEC0ADDE	DE AD C0 DE	DE C0 AD DE
0xF005BA11	11 BA 05 F0	F0 05 BA 11
0xDECAFF01	01 FF CA DE	DE CA FF 01
0x00EEFFC0	C0 FF EE 00	00 EE FF C0
0xBA5EBA11	11 BA 5E BA	BA 5E BA 11

Exercise 2.17:

The following memory footprint of a MIPS program shows the content of the memory starting from address `0x10000000`. It contains the personal data of a student. In the following illustration, four MIPS words in hexadecimal representation are displayed per row, 16 bytes in total.

Memory address	Content of memory			
0x10000000	6e6c6e66	31303030	07d00c19	fffffde8
0x10000010	42464e49	00000003	00000000	00000000

1. Which values are stored if we assume the following interpretations? Remember that in MIPS, the first byte of a multi-byte number contains the least significant bits (little endian).

Offset	Interpretation
0-7	student-identification (8 ASCII characters)
8	Birthday: Day (unsigned, 1 byte)
9	Birthday: Month (unsigned, 1 byte)
10-11	Birthday: Year (unsigned, 2 bytes)
12-15	Semester fee (signed, 4 bytes)
16-18	Course of study (3 ASCII characters)
19	Graduation degree (1 ASCII character)
20	Term of studying (unsigned, 1 byte)

2. Write down the data definitions (`.data`) which lead to this memory layout. Validate the effect of the data definitions in MARS.

Solution

1. The values:

Offset	Interpretation	Values
0-7	student-identification (8 ASCII characters)	"fnln0001"
8	Birthday: Day (unsigned, 1 byte)	0x19 (25)
9	Birthday: Month (unsigned, 1 byte)	0x0c (12)
10-11	Birthday: Year (unsigned, 2 bytes)	0x07d0 (2000)
12-15	Semester fee (signed, 4 bytes)	0xfffffde8 (-536)
16-18	Course of study (3 ASCII characters)	"INF"
19	Graduation degree (1 ASCII character)	"B"
20	Term of studying (unsigned, 1 byte)	0x03 (3)

2. The data definitions:

```
.data
.ascii "fnln0001"      # sequence of 8 chars, 8 bytes
.byte   25             # unsigned integer, 1 byte
.byte   12             # unsigned integer, 1 byte
.half   2000           # unsigned integer, 2 bytes
.word   -536           # signed integer, 4 bytes
.ascii "INF"           # sequence of 3 chars, 3 bytes
.ascii "B"             # char, 1 byte
.byte   3              # unsigned integer, 1 byte
```