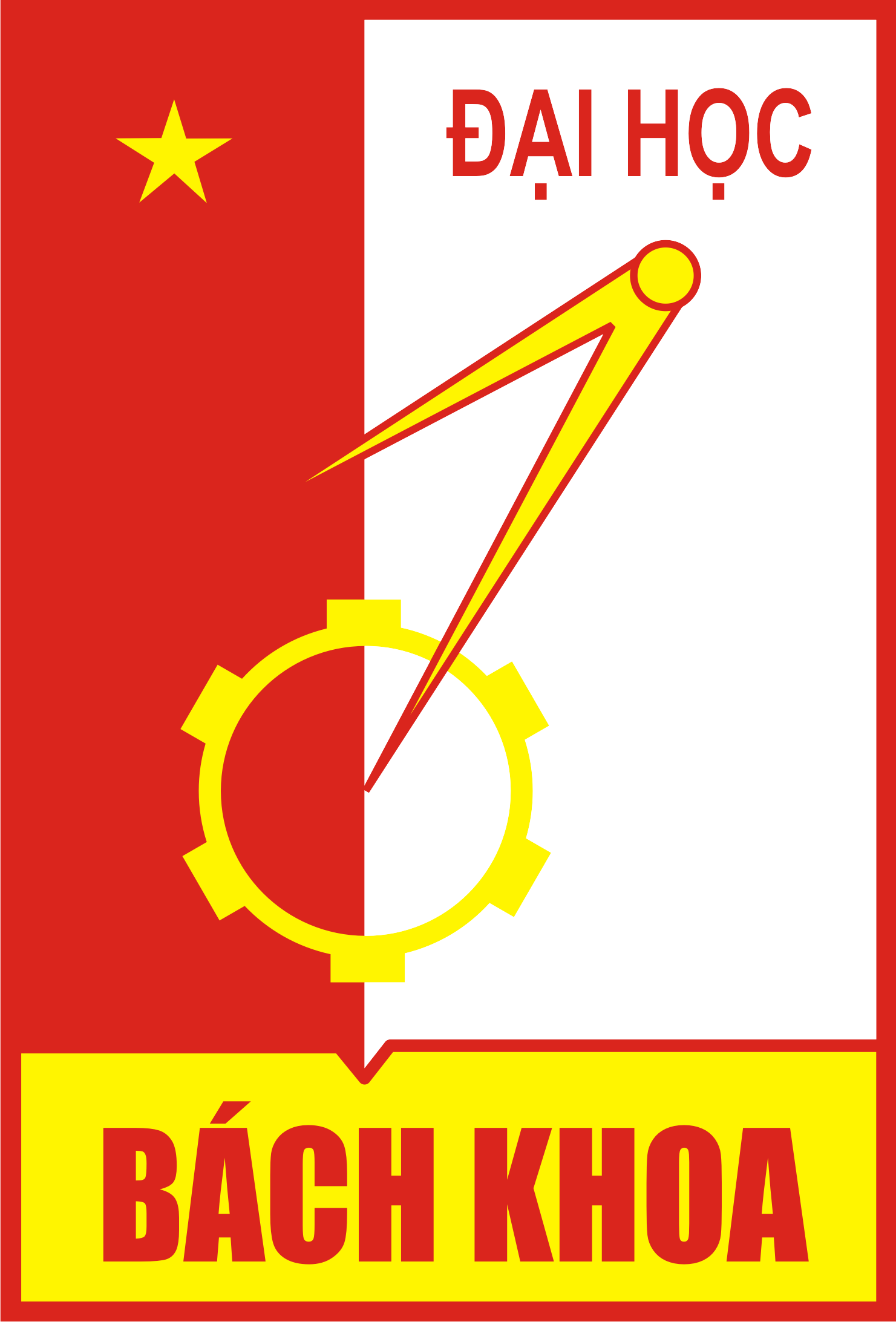
Hanoi University of Science and Technology  
School of Information and Communication Technology



**Final Project Report**

Assembly Language and Computer Architecture Laboratory

|  |  |
| --- | --- |
| **LECTURER:** | **Prof. Nguyen Thi Thanh Nga** |
| **CLASS ID:** | **113932** |
| **GROUP:** | **1** |

|  |  |
| --- | --- |
| ***Student Name*** | ***Student ID*** |
| Phi Hoang Long | 20184288 |
| Nguyen Huy Hoang | 20184265 |
| Le Ba Vinh | 20184331 |

*Hanoi, 6/2020*

***Topic 3* : Typing speed and accuracy checking**

|  |  |
| --- | --- |
| Student name: | Nguyễn Huy Hoàng |
| Student ID: | 20184265 |

1. Project description
   1. **Topic**

Check user input against a given string and output the number of correct characters to 7-leds.

* 1. **Requirement**

Number of correct characters is updated on the 7-leds every interval

of time counter.

* 1. **Advanced idea**

Input is refreshed everytime user press *Enter.*

1. Variables and Constants
   1. **Constants**

- MASK\_CAUSE\_COUNTER: mask to identify if counter is the cause of interupt.

- MASK\_CAUSE\_KEYBOARD: mask to identify if pressed keyboard is the cause of interupt.

- KEY\_READY: address which hold value showing if any key is pressed: 1 for true, 0 otherwise.

- KEY\_CODE: address which hold ASCIIZ value of key pressed.

- DISPLAY\_READY: address which hold value showing if screen is ready to display: 1 for true, 0 otherwise.

- KEY\_CODE: address which hold ASCIIZ value to be displayed.

- SEVENSEG\_LEFT: load value to display left 7-led.

- SEVENSEG\_RIGHT: load value to display right 7-led.

- COUNTER: 1 to active counter, 0 to disable.

* 1. **Variables**

- target\_string: string used to compare correct characters.

- length: length of target\_string.

- input\_string: string which stores value which the user inputs.

- max\_input\_length: max length of input that input\_string will hold.

- display\_num: value to be displayed on 7-leds.

- last\_display: value currently displayed on 7-leds.

1. Functions
   1. **Main**
   2. **Used registers**

- $k0: store KEY\_CODE value.

- $k1: store KEY\_READY value.

- $s0: store DISPLAY\_CODE value.

- $s1: store DISPLAY\_READY value.

- $t1: hold pressed key value.

* 1. **Logic**

Infinite loop waiting for interrupt(s).

* 1. **Interrupt Handler**
  2. **Used registers**

- $t1: hold pressed key value.

- $t2, $t3, $t4, $a0, $a1, $t8, $t9: hold temporary data, condition results.

* 1. **Logic**

- Find the cause of interruption and navigate to the apppropriate handler

- If the counter is interrupted: re-display 7-leds value if changed

- If the keyboard is interrupted:

+ If the key pressed has ASCIIZ value of 10 (*Enter*): refresh display, input\_string.

+ Else: append to input\_string, compare with target\_string and update display\_num.

* 1. **Push**
  2. **Used registers**
  3. **Logic**

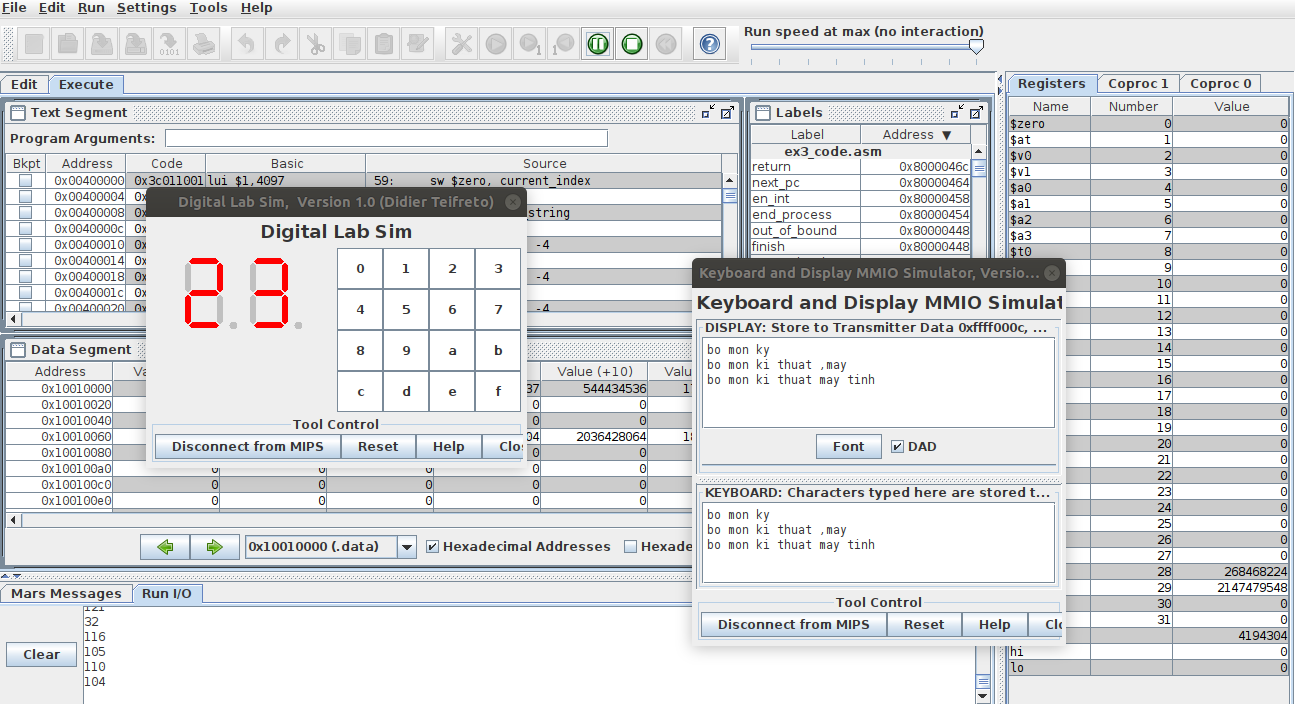
- Push value in register to stack.

* 1. **Pop**
  2. **Used registers**
  3. **Logic**

- Pop value from stack to register.

* 1. **get\_length\_reg**
  2. **Used registers**
  3. **Logic**

- Get the length of an input string, whose address is stored in a register.

1. Simulation Sample Result

***Topic 5* : Infix and postfix expressions**

|  |  |
| --- | --- |
| Student name: | Lê Bá Vinh |
| Student ID: | 20184331 |

1. Project description
   1. **Topic**

A program to print postfix expression from infix expression inputted from user then calculate the result with postfix expression.

* 1. **Requirement**

- Operands (numbers) are integer in range of 0 to 99

- Operators are ‘+’, ‘-‘, ‘\*’, ‘/’

* 1. **Advanced idea**

- Support ‘(‘ and ‘)’

- Program can ask user to either continue inputting new expression or cancel to exit program.

1. Variables and registers used
   1. **Variables**

- Infix (256): store string infix expression from user

- Postfix (256): a stack of postfix elements from infix expression

- Operator (256): a stack of operators, used when creating postfix stack

- Result (256): a stack to calculate the result with 2 operands scanned from postfix expression (the result will be the bottom element in the stack)

* 1. **Registers**

- $s3 stores the first number to calculate

- $s4 stores the second number to calculate

- $s5 is the code of precedence of the checking-operand

- $s6 is the code of precedence of the top operand

if $s6 = 1 then '+' / '-'

if $s6 = 2 then '\*' / '/'

- $s7 is the digit of operand (1 or 2)

- $t0, $t1, $t2 are indexes

- $t4 is status code to check error input

if $t4 = 0: previous scan is operator except '(' and ')'

if $t4 = 1: previous scan is operand

- $t5 stores scanning character in Infix string and Postfix stack or store temporary result when calculating

- $t6 stores operator loaded from Operator stack

- $t7 stores address holding character in Infix string or Postfix stack

- $t8 stores address holding top operand or operator in Postfix stack

- $t9 stores address holding top operand in Operator stack

1. Algorithms
   1. **Main**

*\* Operators are encoded: operator := operator + 100*

|  |  |
| --- | --- |
| Step 1 | Get infix expression from user |
| Step 2 | Initiate value for registers |
| Step 3 | Scan infix expression (operands and operators)   * Convert number from string to value * Check operator * If the character is neither a number nor an operator, input is invalid -> back to Step 1 * While the stack of operators is non-empty, pop every element then push them to the postfix stack * If there is ‘(‘ left on the stack, input is invalid -> back to Step 1 |
| Step 4 | Print the postfix expression. Then add ‘\n’ to the postfix stack |
| Step 5 | Calculate the result with the postfix stack and result stack   * If divisor (if exists) = 0, result is invalid -> back to Step 1 * else go to step 6 |
| Step 6 | Print the result and ask user whether to continue program or stop |

* 1. **Convert number from string to value**

Scan each byte from infix string to $t5 then check if it was a number or not

* If it is a number, store it to $s3 then continue scanning (if the number has 2 digits, $s3 = 1st digit, $s4 = 2nd digit)
* If not, then push ($s3 = num) to the postfix stack, $t4 = 1 then check the operator

**3. Check operator**

If ‘(‘ is detected: push it to the operator stack

If ‘)’ is detected:

* While the operator stack is not empty AND the (t9 = top) != ‘(‘, pop operator stack and push each element to the postfix stack
* Pop the operator stack (which is ‘(‘) and discard it

If an operator is detected:

* If the operator stack is empty or if the top element is ‘(‘: push it to the operator stack
* If not:

+ While the operator stack is not empty AND

the top of the operator stack != ‘(‘ AND

precedence of operator <= precedence of the operator stack’s top (s5 <= s6),

pop the operator stack and put it to the postfix stack

+ Push it to the operator’s stack

**4. Calculate the result**

Scan the postfix from the start and load it to t5 until reach ‘\n’ character

* If it is a number (< 100): put it to result’s stack
* If it is an operator (> 100): decode it and calculate the temporary result from the result stack using the operator

After that, the last value in the result stack is the result of the expression and print it to console.

*\* Result can take up to 32bits*

III. Simulated Sample Result

- Input an infix expression:

A screenshot of a computer

Description automatically generated

- Show result then ask user:

A screenshot of a computer

Description automatically generated

***Topic 7* : MIPS basic instruction syntax checking**

|  |  |
| --- | --- |
| Student name: | Phí Hoàng Long |
| Student ID: | 20184288 |

# Project description

* 1. **Topic**

A program to simulate a specific task of a MIPS compiler, which is opcode syntax checking (basic instructions only).

* 1. **Requirements**
* User enters a mips instruction.

*Ex : beq $t1, $t2, label*

* Check if input opcode exists.
* If input opcode exists, continue to check its operands’ validation behind.
* If input mips instruction is valid, print out the cycles in which it is executed.

# Algorithms

* 1. **Main**
     1. **Registers used**
* $a0: input string, opcode cycles
* $a1: classified group of opcodes
* $a2: opcode
* $a3: string of operands
* $v0: flag -> 0/1
  1. **If-else check**

*Note: Opcode and the number of cycles is found by using “check\_substring\_appearance” defined in Utils below.*

|  |  |
| --- | --- |
| **Step 1** | Read input string from user |
| **Step 2** | Split string into 2 parts: *Opcode* & *Operands* |
| **Step 3** | Use branch instructions to find a possible *Opcode* in every classified group |
| **Step 4** | If *Opcode* is not found, then branch to the next label to find it. (brute force search)  If *Opcode* is not found in any groups, then print an invalid message to console |
| If *Opcode* is found, start checking the *Operands* part by using the global variables and validation-checking functions defined in resource file |
| **Step 5** | If operands part is valid, print valid valid message console, followed by the its number of cycles, which could be found by using the global variables and functions defined in resource and utils file (*reg\_char\_first\_pos*) |

* 1. **Utils**
  2. **Purpose**

To define subprograms, which will be supportively used for the miniprojects.

*Note: In this miniproject, I use macros to define every subprogram for better understanding.*

* 1. **Important functions**

1. ***check\_in\_reg\_range(%int\_register, %lower\_bound, %upper\_bound)***

* Purpose: to check if an integer stored in a register is in range [lower\_bound, upper\_bound]
* Return: *$v0 = 1 if true, 0 if false*

1. ***check\_label(%string\_reg)***

* Purpose: to check if a label stored in a register is valid or not
* Return: *$v0 = 1 if true, 0 if false*

1. ***check\_substring\_appearance(%string\_reg, %substring\_reg)***

* Purpose: to check if a substring belongs to a given string in which words are split by spaces

*Ex: substring “c” in string “a b c d e”*

* Return: *$v0 = 1 if found, 0 if not found*

1. ***convert\_num\_hex(%register\_string)***

* Purpose: to convert a valid string of 16-bit hexadecimal to decimal
* Return: *$v0 = num*

1. ***is\_hexa(%register\_string)***

* Purpose: to check if a string stored in a register is a valid 16-bit hexadecimal or not

*Ex: 0xAF14, 0x0032E6, 0x12345678*

* Return: *$v1 = 1, $v0 = convert\_num\_hex(%register\_string) if valid*

*$v1 = 0, $v0 = 0 if invalid*

1. ***is\_num(%register\_string)***

* Purpose: to check if a string stored in a register is a valid integer or not
* Return: *$v1 = 0, $v0 = 0 if invalid*

*$v1 = 1, $v0 = num(decimal) if valid*

1. ***reg\_char\_first\_pos(%char\_reg, %string\_reg)***

* Purpose: to find the first position of a character in a string, both stored in registers
* Return: *$v0 = position if found, -1 if not found*

1. ***split\_by\_literal\_separator(%string\_reg, %char)***

* Purpose: to split a string stored in a register into 2 substrings, using separator character
* Return: *if %char is found:*

*$a2 = the first half of the string split by this function*

*$a3 = the address of the second half of the string*

*(remove redundant spaces as many as possible)*

*else: $a2 = %string\_reg*

*$a3 = null*

*Ex: $t0 = “add $t1 , $t2, $t3 “*

*split\_by\_literal\_separator($t0, “ “)*

* *$a2 = “add”*
* *$a3 = “$t1 , $t2, $t3”*

**3. Resource**

* 1. **Global variables**

*Note: In this miniproject, I classify:*

*+ every opcode stored in a specific string named based on operands and number of cycles*

*+ every register based on type*

*Ex: cee : condition flag, even-float-register, even-float-register*

*es16\_i : even-float-register, signed\_16\_bit\_integer(integer\_register)*

* Abbreviation of operands list:

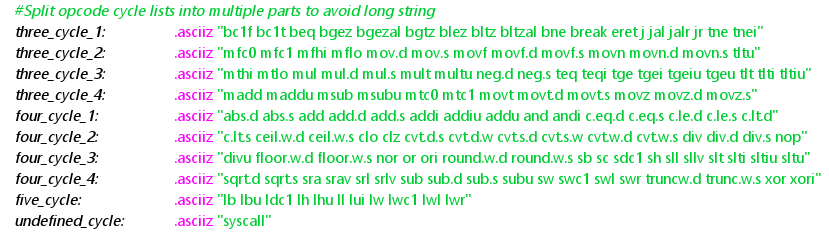
|  |  |
| --- | --- |
| **Abbreviation** | **Meaning** |
| c | Condition flag (0-7) |
| e | Even float register ($f0, $f2, …, $f30) |
| f | Float register ($f0, $f1, …, $f31) |
| i | Integer Register ($zero, $at, …, $ra, $0, $1, …, $31) |
| 8 | Coprocessor 0 ($8, $12, $13, $14) |
| l | Label |
| s16 | Signed 16-bit integer |
| u16 | Unsigned 16-bit integer |
| u5 | Unsigned 5-bit integer |
| none | No operands |

* Classified groups of opcodes based on operands:

A screenshot of a cell phone

Description automatically generated

* Classified groups of opcodes based on the number of cycles:



* Classified groups of registers based on type:

A picture containing table

Description automatically generated

* 1. **Operands validation-checking functions**
* **Function Format: *valid\_[group](%string\_of\_operands)***
* **Return: $v0 = 1 if valid, 0 if invalid**

*Note: For each classified groups of opcodes based on operands, I define a specific operand validation-checking function.*

As most of operand validation-checking functions share similarities, I only make a general description of algorithm based on the number of operands.

1. *3 operands (Ex: add $t0, $t1, $t2)*

|  |  |
| --- | --- |
| **Step 1** | Load the suitable group of opcodes depending on its name to $t0  *Ex: Function “valid\_cee” needs Group “cee”* |
| **Step 2** | Split the string of operands into 2 parts with separator ‘,’ (comma), using function **split\_by\_literal\_separator** defined in Utils  *Ex: “$t0, $t1, $t2” -> split\_by\_literal\_separator($t0, ‘,’)*   * *$a2 = “$t0”* * *$a3 = “$t1, $t2”*   Based on the operand’s type, check if that operand exists, by using:   * “check\_substring\_appearance” if operand is a register * “is\_num”/”is\_hexa” && ”check\_int\_reg\_range” if operand is a number/offset * “check\_label” if operand if a label   If valid, go to the next step, else invalid |
| **Step 3** | Similar to Step 2, split the string of operands in $a3 to get the second operand by **split\_by\_literal\_separator($a3, ‘,’)**, then check the validation of it  *Ex: “$t1, $t2” -> split\_by\_literal\_separator($a3, ‘,’)*   * *$a2 = “$t1”* * *$a3 = “$t2”*   If valid, go to the next step, else invalid |
| **Step 4** | Repeat Step 3 for the third operand  *Ex: “$t2” -> split\_by\_literal\_separator($a3, ‘,’)*   * *$a2 = “$t2”* * *$a3 = null* |
| **Step 5** | Check if the fourth operand exists by getting the string length of $a3. If length > 0 return invalid, else valid |

1. *2 operands (Ex: bc1f 1, label)*

|  |  |
| --- | --- |
| **Step 1** | Load the suitable group of opcodes depending on its name to $t0  *Ex: Function “valid\_ee” needs Group “ee”* |
| **Step 2** | Split the string of operands into 2 parts with separator ‘,’ (comma), using function **split\_by\_literal\_separator** defined in Utils  *Ex: “1, label” -> split\_by\_literal\_separator($t0, ‘,’)*   * *$a2 = “1”* * *$a3 = “label”*   Based on the operand’s type, check if that operand exists, by using:   * “check\_substring\_appearance” if operand is a register * “is\_num”/”is\_hexa” && ”check\_int\_reg\_range” if operand is a number/offset * “check\_label” if operand if a label   If valid, go to the next step , else invalid |
| **Step 3** | Similar to Step 2, split the string of operands in $a3 to get the second operand by **split\_by\_literal\_separator($a3, ‘,’)**, then check the validation of it  *Ex: “label” -> split\_by\_literal\_separator($t0, ‘,’)*   * *$a2 = “$t1”* * *$a3 = “$t2”*   If valid, go to next step, else invalid |
| **Step 4** | Check if the third operand exists by getting the string length of $a3.  If length > 0 return invalid, else valid |

1. *1 operand (Ex: mfhi $t1)*

|  |  |
| --- | --- |
| **Step 1** | Load the suitable group of opcodes depending on its name to $t0  *Ex: Function “valid\_i” needs Group “i”* |
| **Step 2** | Split the string of operands into 2 parts with separator ‘,’ (comma), using function **split\_by\_literal\_separator** defined in Utils  *Ex: “$t1” -> split\_by\_literal\_separator($t0, ‘,’)*   * *$a2 = “$t1”* * *$a3 = null*   Based on the operand’s type, check if that operand exists, by using:   * “check\_substring\_appearance” if operand is a register * “is\_num”/”is\_hexa” && ”check\_int\_reg\_range” if operand is a number/offset * “check\_label” if operand if a label   If valid, go to the next step , else invalid |
| **Step 3** | Check if the second operand exists by getting the string length of $a3. If length > 0 return invalid, else valid |

1. *0 operand (Ex: syscall)*

|  |  |
| --- | --- |
| **Step 1** | Load the suitable group of opcodes depending on its name to $t0  *Ex: Function “valid\_none” needs Group “none”* |
| **Step 2** | Check if the first operand exists by getting the string length of $t0.  If length > 0 return invalid, else valid |

# Simulation Sample Result

A screenshot of a cell phone

Description automatically generated

A screenshot of a social media post

Description automatically generated