SIMP 1.0 API

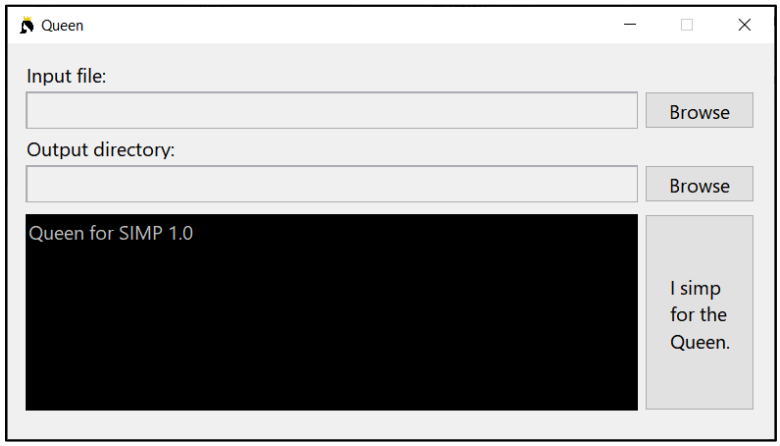
A higher level language that translates directly into MIPS assembly code via the Queen interpreter.

Table of Contents:

“Too difficult to write MIPS code? Don’t worry, it’s simple.”

Building Your First SIMP Program

Double click on “Queen.jar” to run the Queen interpreter.



To the right of “Input file”, click on the “Browse” button. A file selection window will appear. Navigate to the “demo” folder and select any of the demo files.

Here you will notice that all SIMP programs end with the .simp extension.

To the right of “Output directory”, click on the “Browse” button. Another file selection window will appear, and navigate to the folder that you would like to output the .asm file to. You may select the same demo folder.

Click on “I simp for the Queen.” If you see the words “Translation complete!” in the text box below, it means that the program has been successful translated and contains no syntax errors. Otherwise, the text box will show “error on line: x” where x is the line that contains faulty code.

Navigate to your specified output folder and there should now be a .asm file with the same name as the .simp file. Open this file in any MIPS assembler (i.e. MARS) and the file should contain valid MIPS code.

*Note: Queen does not detect any memory leaks or register overflow errors. A SIMP program that successfully compiles in Queen does not guarantee that it will work as expected in a MIPS assembler.

Registers and Syntax

You are allowed to use the following registers in SIMP:

- $0 (read only, cannot be written to)
- $v0, $v1
- $a0, $a1, $a2, $a3
- $t0, $t1, $t2, $t3, $t4, $t5, $t6, $t7, $t8, $t9
- $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7, $s8
- $gp
- $sp
- $fp
- $ra

Register $v1 is used for many of the instructions in SIMP, do not modify it unless you know what you are doing.

SIMP code is translated from top to bottom line by line.

Some of the instructions in SIMP, particularly arithmetic operations, have two ways of writing.

For example, addition can be written as:

```
a = b + c
```

or

```
a = b add c
```

The former syntax style is resembles more closely to modern high-level programming languages, while the latter syntax style resembles more closely to MIPS assembly code.

Choose whichever syntax is more comfortable for you.

In this documentation:

- reg = register
- imm = immediate
- exp = Boolean expression

Math

Equate (reg **a**, reg/imm **b**)

```
a = b
```

Negate (reg **a**, reg/imm **b**)

```
a = !b
a = not b
```

Addition (reg **a**, reg **b**, reg/imm **c**)

```
a = b + c
a = b add c
a += c
```

Subtraction (reg **a**, reg **b**, reg/imm **c**)

```
a = b - c
a = b sub c
a -= c
```

Multiplication (reg **a**, reg **b**, reg/imm **c**)

```
a = b * c
a = b mul c
a *= c
```

Integer Division (reg **a**, reg **b**, reg/imm **c**)

```
a = b / c
a = b div c
a /= c
```

Modulo (reg **a**, reg **b**, reg/imm **c**)

```
a = b % c
a = b mod c
```

Shift Left Logical (reg **a**, reg **b**, reg/imm **c**)

```
a = b << c
a = b sll c
```

Shift Right Logical (reg **a**, reg **b**, reg/imm **c**)

```
a = b >> c
a = b srl c
```

Shift Right Arithmetic (reg **a**, reg **b**, reg/imm **c**)

```
a = b >>! c
a = b sra c
```

Bitwise AND (reg **a**, reg **b**, reg/imm **c**)

```
a = b & c
a = b and c
```

Bitwise OR (reg **a**, reg **b**, reg/imm **c**)

```
a = b | c
a = b or c
```

Bitwise XOR (reg **a**, reg **b**, reg/imm **c**)

```
a = b ^ c
a = b xor c
```

Boolean (reg **a**, bool **exp**)

```
a = exp
```

The following Boolean expressions are valid: (reg **b**, reg/imm **c**)

```
b == c
b != c
b > c
b >= c
b < c
b <= c
```

You cannot combine multiple operations into one expression.
Ex. `a = (b + c) > (d - e)` is invalid.

Data

The third parameter is optional. If unused, it will be defaulted to zero offset.

Load address (reg **a**, label **b**)
```
a.la(b)
```

Load word (reg **a**, reg **b**, *optional* imm **c**)
```
a.lw(b)
a.lw(b + c)
a.lw(b - c)
```

Store word (reg **a**, reg **b**, *optional* imm **c**)
```
a.sw(b)
a.sw(b + c)
a.sw(b - c)
```

Load byte (reg **a**, reg **b**, *optional* imm **c**)
```
a.lb(b)
a.lb(b + c)
a.lb(b - c)
```

Store byte (reg **a**, reg **b**, *optional* imm **c**)
```
a.sb(b)
a.sb(b + c)
a.sb(b - c)
```

Push word in register to stack (reg **a**)
```
stack.push(a)
```

Pop word from stack into register (reg **a**)
```
stack.pop(a)
```

Conditionals

'if' block

```
if exp:
    // code
end
```

'while' block

```
while exp:
    // code
end
```

`exp` may be replaced with any of the Boolean expressions.
All conditional statement blocks must end with the `end` keyword.

Functions

Function definition

```
func name():
     // code
end
```

All functions must end with the `end` keyword. To jump back to the code before calling the function, the `return` keyword must be used.

```
func name():
     // code
     return
end
```

A register/immediate may be returned as well, which will be placed into register $v0.

```
return a
```

To call a function without jumping back, use the `j` keyword.

```
j name()
```

To call a function that returns a value, use the `jal` keyword.

```
jal name()
```

Note that using `j` on a function that returns a value will cause problems.

Extras

Exit the program (syscall 10)

```
exit
```

Line comment

```
// comment
```

Comments will be preserved in the translated code.

Inline assembly

```
asm: add $t0, $t1, $t2
```

Queen does not check if the inline assembly is valid MIPS code.

Limitations

- Only words can be pushed to the stack using stack.push(). Pushing and pulling bytes must be done manually using .sb() and .lb().
- Floats and doubles are not supported.
- “else” for if statements are not supported.
- Because $v1 is used by Queen, return values can only be up to 32 bits.
- SIMP code currently can only be written in a plain text editor without syntax highlighting.
- Queen does not generate the most efficient MIPS code.