

Flask web evaluation for QtRvSim

Jakub Pelc

16.3.2024

Faculty of Electrical Engineering, Czech Technical University in Prague

Goals of this project

The main aim of this project is to provide a public way for students (and general public) to test their knowledge of the RISC-V assembly.

Registered users can submit solutions to the problems displayed on the frontpage and get immediate feedback on their solution. The solution performance is measured (in cycles needed to execute the submission), and local scoreboard is displayed for each task.

The project needed to be rather simple, for it to allow easy modularity and optional modification in the future. It can be expanded by more features, language support, or task types.

We are not saving the email addresses of the users (due to GDPR), but during the registration process, the users are required to provide an email address for verification purposes. So how is that achieved?

The email address is saved as a salted SHA-256 hash. This way, we can verify the email address, but we cannot reverse the hash to obtain the original email address.

This also allows us to send the user a password reset link, without having to store the email address in readable format. User always need to provide the email address, which is checked against the hash in the database.

A little bit about Flask

Flask is a micro web framework written in Python. It provides a simple way to create web applications.

As opposed to Django, Flask is not an all-inclusive framework. It is designed to be simple and easy to use.

To start creating a web application, the only thing you need to do is to install Flask and start writing your Python code.

To run the local webserver, the command `flask run` is used.

```
from flask import Flask
```

```
app = Flask(__name__)
```

```
@app.route("/")
```

```
def hello():
```

```
    return "<p>Hello, World!</p>"
```

Loading template files

We can utilize Flask to load HTML templates, instead of writing all of the HTML code in the `app.py` file.

These template file are to be located in the `templates` directory.

For this example, we will use the standard HTML.


```
from flask import Flask
from flask import render_template
from flask import request

app = Flask(__name__)
app.secret_key = 'e4ed89f02f3aa07a4309dbfff'

@app.route("/")
def index():
    return render_template('index.html')

@app.route("/register", methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        return "Registered with username: " + \
            request.form['username'] + " and password: " + \
            request.form['password']
    return render_template('register.html')
```

This is still not ideal, now we need to create a complete HTML page for every route.

We will now use the Jinja2 templating engine to create a base template, and then extend it for every route.

We can also add static files, such as CSS stylesheets, or images.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" href="{ url_for('static', filename='favicon.png') }}" type="image/x-icon">
  <title>{% block title %}{% endblock title %}</title>
  <link rel="stylesheet" type="text/css" href="{ url_for('static', filename='style.css') }}">
</head>
<body>
  <main class="main-content">
    {% block content %}{% endblock content %}
  </main>
</body>
</html>
```

```
{% extends "base.html" %}
```

```
{% block title %}
```

```
The Register Page
```

```
{% endblock title %}
```

```
{% block content %}
```

```
<h2>Register</h2>
```

```
<form method="POST" action="/register">
```

```
<input type="text" name="username" placeholder="Username">
```

```
<input type="password" name="password" placeholder="Pass">
```

```
<input type="submit" value="Register">
```

```
</form>
```

```
{% endblock content %}
```

The session system allows us to store information about the user across multiple requests.

We can also pass variables to the template which we will use while rendering the page.

This can be demonstrated by creating a simple login system.

```
from flask import Flask, render_template, session, redirect, url_for
from markupsafe import escape

app = Flask(__name__)
app.secret_key = 'e4ed89f02f3aa07a4309dbfff'

@app.route("/")
def index():
    return render_template('index.html')

@app.route("/name/<name>")
def name(name):
    session['user'] = escape(name)
    return redirect(url_for('index'))

@app.route('/personal')
def personal():
    logged_in = session.get('logged_in', False)
    return render_template('personal.html', logged_in=logged_in)

@app.route("/login")
def login():
    session['logged_in'] = True
    return redirect(url_for('index'))

@app.route("/logout")
def logout():
    session.clear()
    return redirect(url_for('index'))
```

Communication with the database

In the web application, a PostgreSQL database is used.

Only a few tables are needed to store the information about the users, tasks, submissions and results.

PostgreSQL triggers are used to automatically update the best score and source code.

Users Table

Field	Type	Length	Default
id	int	32	AUTO_INCREMENT
username	varchar	128	None
password	varchar	128	None
email	varchar	128	None
salt	varchar	128	None
verification_code	varchar	128	None
user_verified	tinyint	1	0

Submissions Table

Field	Type	Length	Default
id	int	64	AUTO_INCREMENT
userid	int	64	None
taskid	int	64	None
file	text	64	None
evaluated	tinyint	1	0
time	datetime	None	current_timestamp()

Results Table

Field	Type	Default
userid	bigint	PRIMARY
taskid	bigint	PRIMARY
result_file	text	NULL
last_source	text	NULL
best_source	text	NULL
score_last	integer	-1
score_best	integer	-1
time	timestamp with time zone	CURRENT_TIMESTAMP
result	smallint	-1

```
import psycopg2
import os

db_config = {
    'user': os.getenv('DB_USER'),
    'password': os.getenv('DB_PASSWORD'),
    'host': os.getenv('DB_HOST'),
    'database': os.getenv('DB_DATABASE'),
    'port': os.getenv('DB_PORT'),
    'sslmode': 'require',
    'connect_timeout': 10
}
```

```
def connect():
    db = psycopg2.connect(**db_config)
    cursor = db.cursor()
    return (db, cursor)

def get_user(username):
    (db, cursor) = connect()
    cursor.execute('SELECT password FROM \
        users WHERE username = %s', (username,))
    user = cursor.fetchone()
    cursor.close()
    db.close()
    return user
```

Submission evaluation

Each of the submissions is being evaluated by a `qtrvsim_cli` python wrapper `qtrvsim.py`.

For each task, there is a `.toml` file, which is parsed using an evaluator script. The parsed data is used to create a new `QtRvSim` instance, which evaluates all the testcases in the task file.

The result, score, and the log is then displayed to the user.

```

from qtrvsim import QtRVSim

sim = QtRVSim(args="--d-regs --dump-cycles --cycle-limit 1000", submission_file="file.S")

ending_regs = {
    "a1": 2,
    "a2": 4,
    "a3": 6,
}

starting_mem = {
    "array_start": [2, 4],
}

ending_mem = {
    "array_start": [2, 4, 6],
}

sim.set_reference_ending_regs(ending_regs)

sim.set_starting_memory(starting_mem)

sim.set_reference_ending_memory(ending_mem)

#sim.set_private() #optional, if set to true, does not show errors

sim.run("Testcase 1")

print(sim.get_log())
print(sim.get_scores()["cycles"] if sim.get_result() == 0 else "-1")

sim.reset()

```

```

[task]
name = "Task"
template = "S_templates/template.S"

description = '''
# Description
The task description
'''

[arguments]
run = "--d-regs --dump-cycles --cycle-limit 1000"

[[testcases]]
name = "Testcase 1"
private = true

[[testcases.reference_regs]]
a1 = 2
a2 = 4
a3 = 6

[[testcases.starting_mem]]
array_start = [2, 4]

[[testcases.reference_mem]]
array_start = [2, 4, 6]

[score]
testcase = "Testcase 1"

```

The evaluator is also able to set a cache for the task, whose parameters are configurable as a part of the task. This is done by setting the maximum cache size for the task, users are then required to configure the cache parameters.

Serial input and output can also be used.

It is also possible, to create a task in C, but this also requires for a custom Makefile to be provided to the taskfile. If custom files need to be present at compile time, the can also be added.

[task]

```
name = "Cache example"  
template = "S_templates/cache.S"  
cache_max_size = 16
```

[arguments]

```
run = "--dump-cycles --read-time 10 --cycle-limit 5000 \  
      --write-time 10 --burst-time 2"
```

```

[task]
name = "C example"
template = "S_templates/example.c"
c_solution = true

[[testcases]]
name = "test1"

[[testcases.input_uart]]
uart = "111\n222\n"

[[testcases.reference_uart]]
uart = "333\n"

[score]
testcase = "test1"

[make]
Makefile=""
#provide a rule that will compile the solution into a binary `submission`
#please provide a clean rule, this is run after evaluation
clean:
    rm -f *.o *.a $(OBJECTS) $(TARGET_EXE) depend
"""

[[files]]
name = "crt0local.S"
code = """
/* minimal replacement of crt0.o which is else provided by C library */
"""

```

Mini Competition

Mini Competition

On the page

`eval.comparch.edu.cvut.cz`

you can try out your skills in RISC-V assembly.

For each task the best five users will acquire $(6 - p)$ points, where p is the place they finished (according to the cycles needed to execute their solution).

We offer FEE CTU merch for the best users, who submit their solutions before TODO:17.3. 15:00.

References

Links and references:

Flask

`eval.comparch.edu.cvut.cz`

`comparch.edu.cvut.cz`

QtRvSim repository

Web Eval repository

Jakub Pelc