CS258-HW2

Team Members:
1- Achref Rebai
2- Krishnan Iyer
3- Mojtaba AlShams

Q2.

We have the following pipeline:

- Instruction fetch (35 picoseconds)
- Instruction decode (400 picoseconds)
- Operand fetch (200 picoseconds if in register, 500 picoseconds if in cache)
- Operand fetch (200 picoseconds if in register, 500 picoseconds if in cache)
- Execution unit (50—400 picoseconds) • Result store (200 picoseconds)

When utilizing all stages in the pipeline, the bottleneck is typically determined by the stage where the data resides. If the data is in the cache, then the bottleneck may occur at the operand fetch unit. Conversely, if the data is in a register, the bottleneck may shift to the instruction decode stage. Setting an appropriate clock frequency is crucial in balancing performance and power consumption. It's important to find a frequency that is low enough to prevent excessive power usage while still allowing each component to operate optimally. In other words, setting the frequency too high can lead to many cycles of stalling in some components, resulting in wasted energy. Conversely, setting the frequency too low may lead to inefficient use of time, as it could exceed the required time for the bottleneck component, ultimately reducing throughput.

So here in this case consider this pipeline:

Time (pico secs)	**35**	**400**	**200**	**200**	**50 - 400**	**200**
Pipeline Stage	**IF**	**ID**	**REG**	**REG**	**ALU**	**REG**

Now as an example let's consider an arbitrary frequency of 1/300 ps. So that means every stage will take at least one instruction cycle and so the difference between what's needed and the clock cycle would be unused wait time. In this case wait time will be

Stage	**Wait time**
Instruction Fetch	300 - 35 = 265
Instruction Decode	300*2 - 400 = 200

Register fetch (best case)	300 - 200 = 100
Register fetch (best case)	300 - 200 = 100
ALU(best case)	300 - 50 = 250
Register store	300 - 200 = 100

As you can see the total wait time would be 865ps in best case and worst case it will be 815ps. The idea here is to minimize the unused wait time even if it takes multiple clock cycles for each stage in the pipeline.

We can formulate the same by the equation below.

Total unused wait time $= \frac{\Sigma(ceil(s_t{*}f) - (s_t{*}f))}{f}$

For example, if clock is 200 and in case of Instruction fetch time i.e 35 ps, let's compute unused time by above equation

$(s_t * f)$ = 35/200 = 0.175

$ceil(s_t * f)$ = 1

$ceil(s_t * f) - (s_t * f)$ = 1 - 0.175 = 0.825

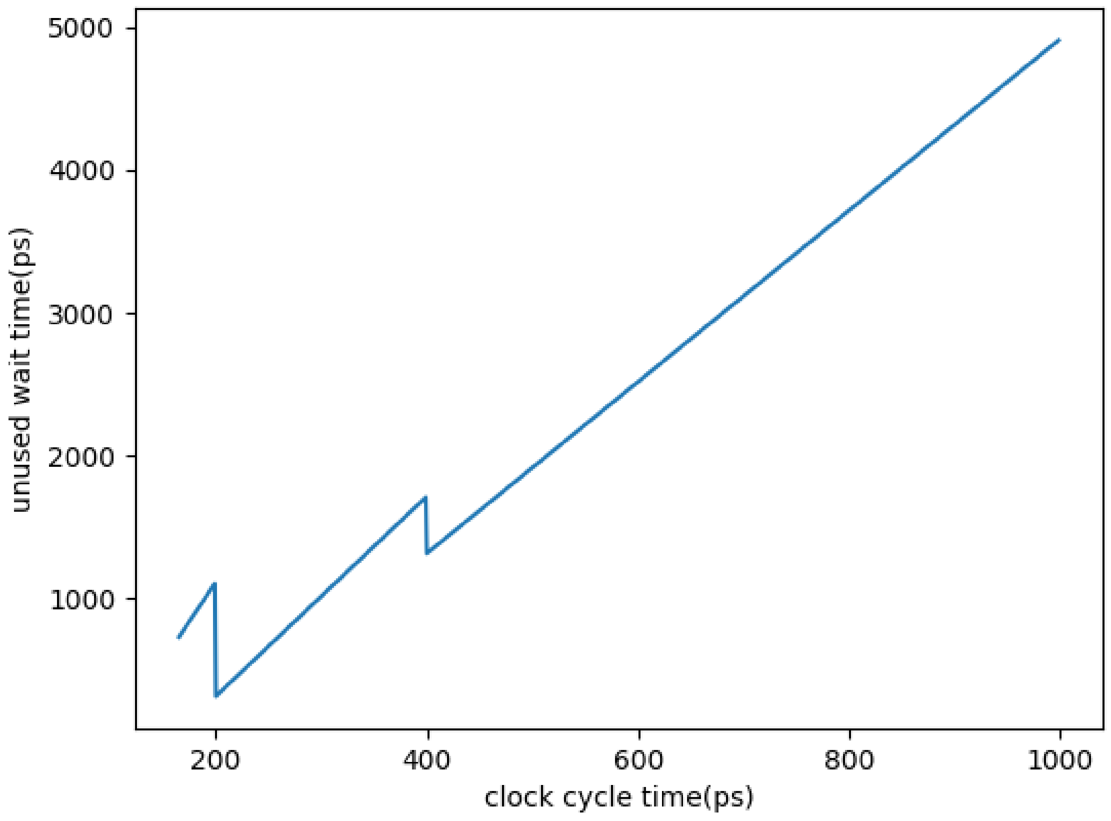
$\frac{ceil(s_t{*}f) - (s_t{*}f)}{f}$ = 0.825 * 200 = 165 i.e 200 - 35

So we intend to minimize total unused wait time for below cases and we calculate total wait time for easy clock cycle from 165ps to 1000 ps with a step of 1 and considering minimum value from all those values.

- Case 1 : Operands take 200 ps and execution unit take 50 ps
- Case 2 : Operands take 500 ps and execution unit take 400 ps

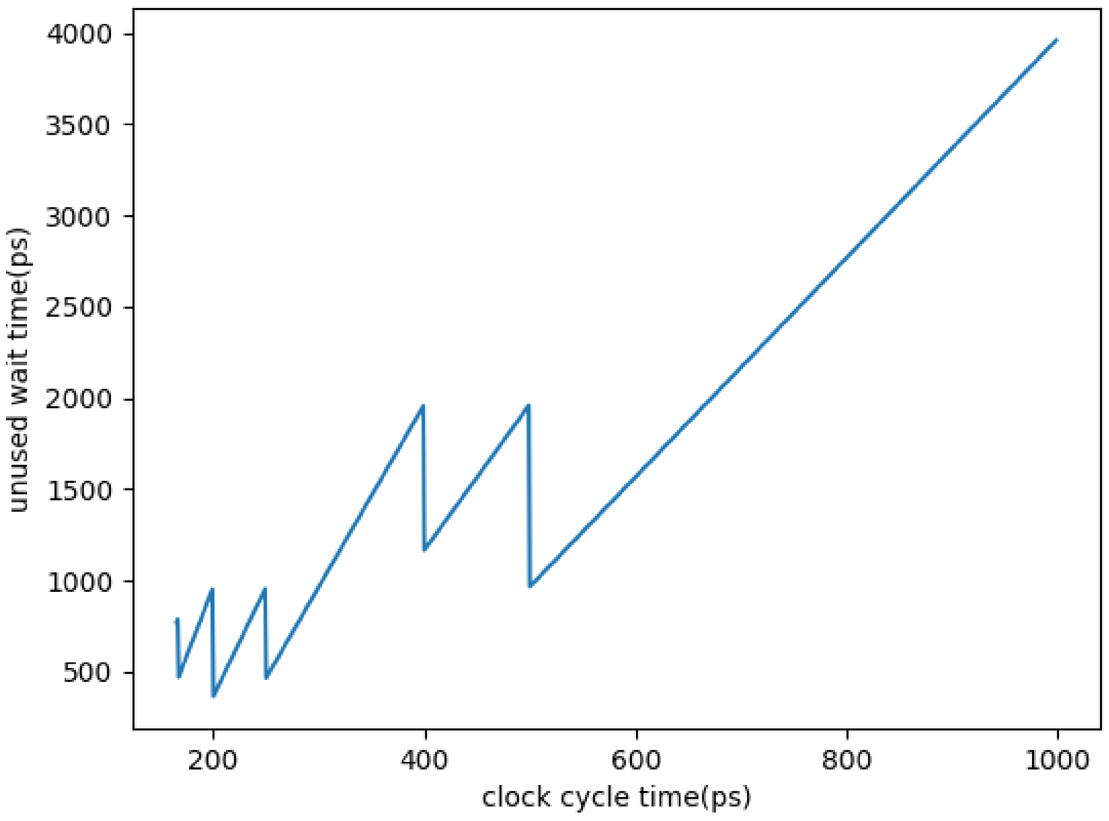
For Case 1 : we got the following results

Min unused time = 315 ps for a clock of 200 ps.



For Case 2 : we got the following results

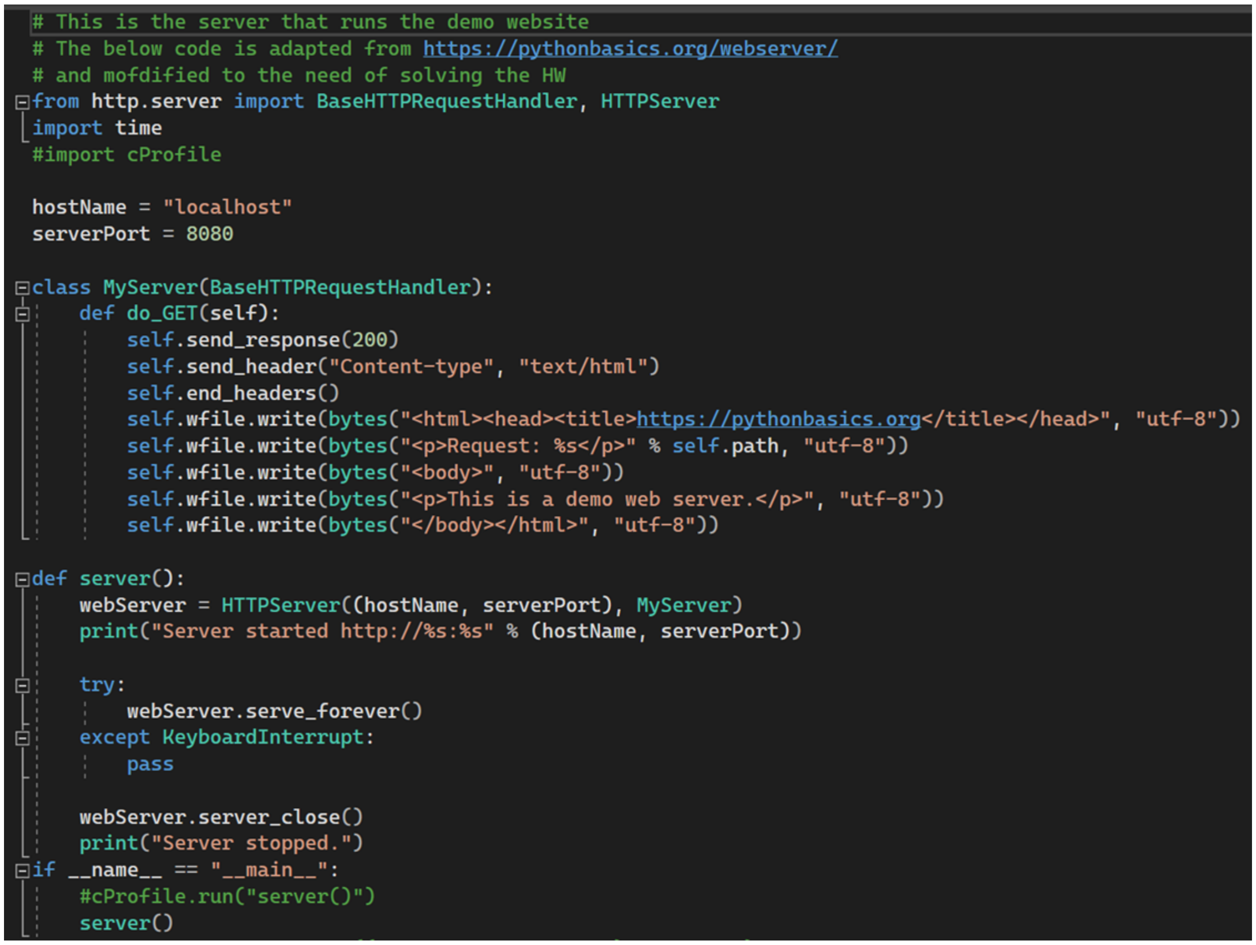
Min unused time = 365 ps for a clock of 200 ps.



So for both case 1 and case 2, a clock 200 ps produces optimum results. Hence the optimum frequency to operate above pipeline would be $\frac{1}{200\,ps} = 5GHz$

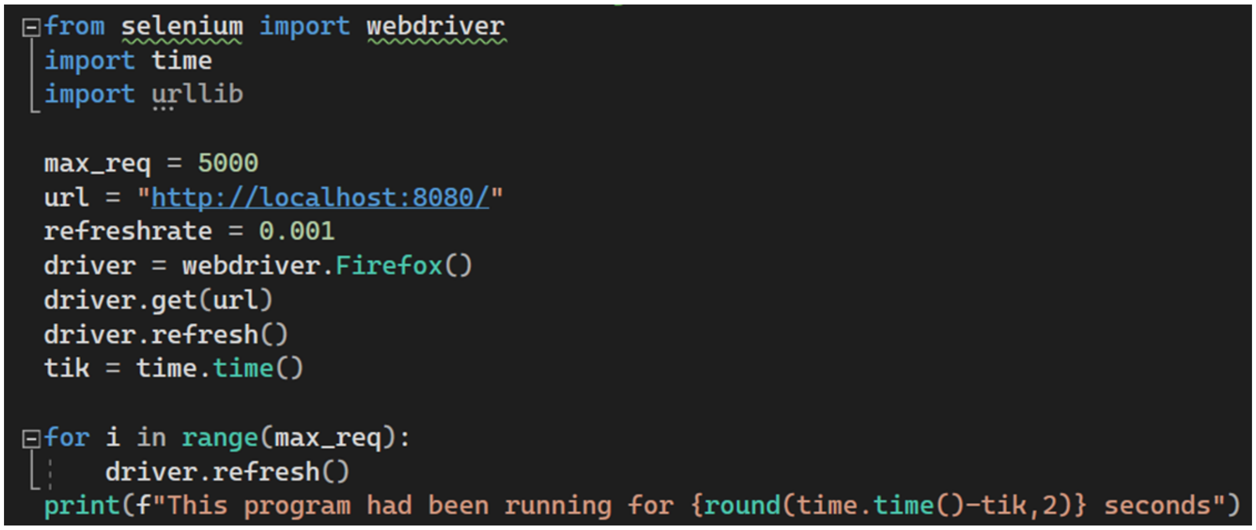
Q3.

A website server was generated using the python library “http.server” using the python file “server_demo.py”. A screenshot of the program is in Figure.1. Another python file, “requester.py” shown in Figure.2, was used to continuously send requests and estimate the response time of all the requests. In this experiment, 5000 requests were sent to the server in a for loop that refresh the website (i.e. request it over and over). These requests first were sent without profiling the server and then sent again while profiling is active to find the dilation time.



```python
# This is the server that runs the demo website
# The below code is adapted from https://pythonbasics.org/webserver/
# and mofdified to the need of solving the HW
from http.server import BaseHTTPRequestHandler, HTTPServer
import time
#import cProfile

hostName = "localhost"
serverPort = 8080

class MyServer(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.end_headers()
        self.wfile.write(bytes("<html><head><title>https://pythonbasics.org</title></head>", "utf-8"))
        self.wfile.write(bytes("<p>Request: %s</p>" % self.path, "utf-8"))
        self.wfile.write(bytes("<body>", "utf-8"))
        self.wfile.write(bytes("<p>This is a demo web server.</p>", "utf-8"))
        self.wfile.write(bytes("</body></html>", "utf-8"))

def server():
    webServer = HTTPServer((hostName, serverPort), MyServer)
    print("Server started http://%s:%s" % (hostName, serverPort))

    try:
        webServer.serve_forever()
    except KeyboardInterrupt:
        pass

    webServer.server_close()
    print("Server stopped.")
if __name__ == "__main__":
    #cProfile.run("server()")
    server()
```

Figure 1: A screenshot to server_demo.py



```python
from selenium import webdriver
import time
import urllib

max_req = 5000
url = "http://localhost:8080/"
refreshrate = 0.001
driver = webdriver.Firefox()
driver.get(url)
driver.refresh()
tik = time.time()

for i in range(max_req):
    driver.refresh()
print(f"This program had been running for {round(time.time()-tik,2)} seconds")
```

Figure 2: A screenshot to requester.py

The server was profiled using the command line “python -m cProfile -o profiled_server.prof server_demo.py”. Also, “SnakeViz” was used to generate the profile outputs. A screenshot of the generated flame graph is presented below in figure.3. Figure.4 presents part of the default cProfile output. For the interactive flame graph, kindly run the following command in the terminal “snakeviz profiled_server.prof”. Please ensure that profiled_server.prof file already exists in the directory.

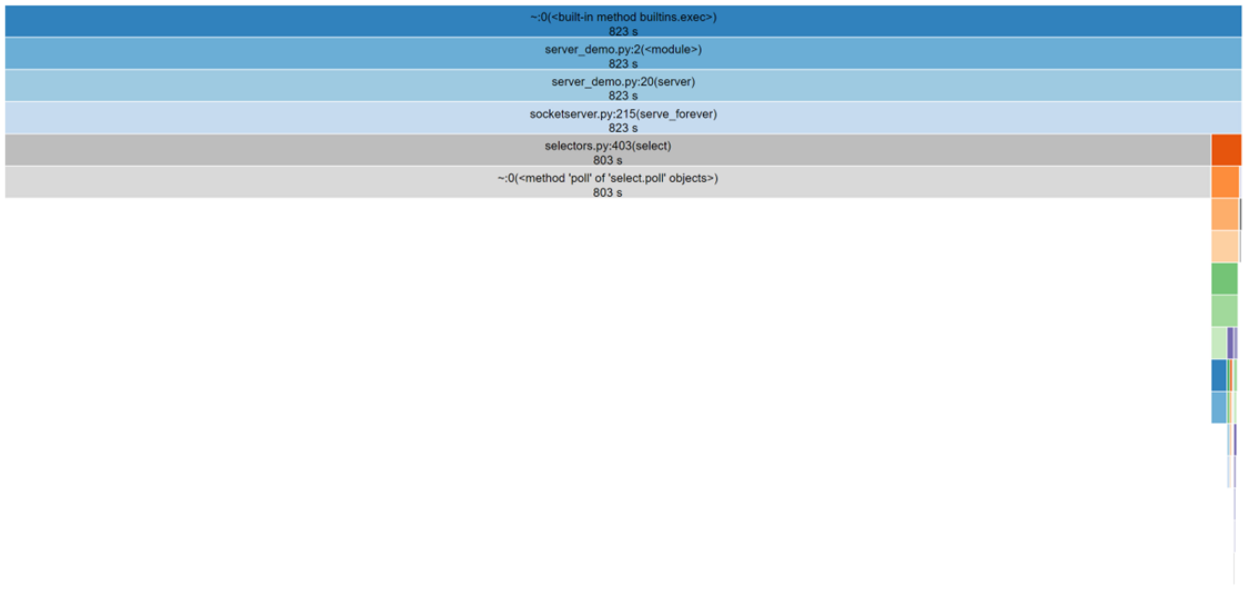
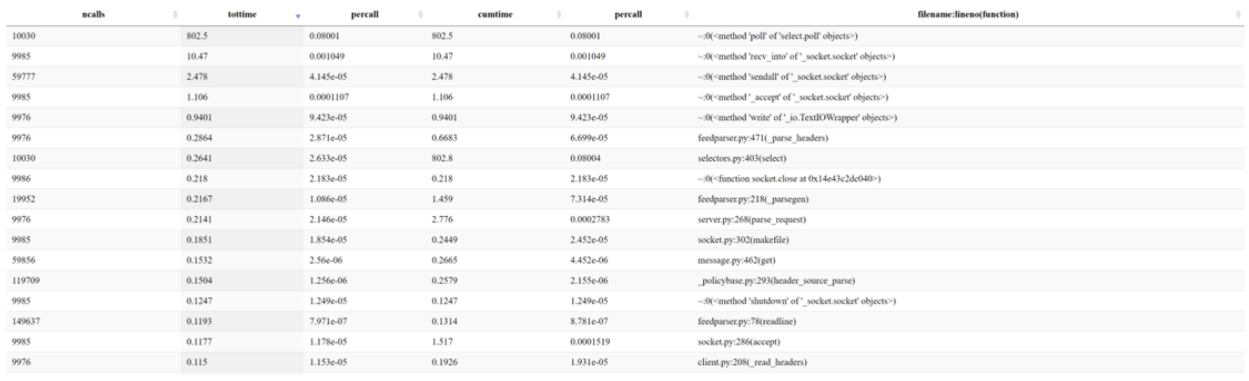


Figure 3: Flame graph of the profiled server while serving 5000 requests

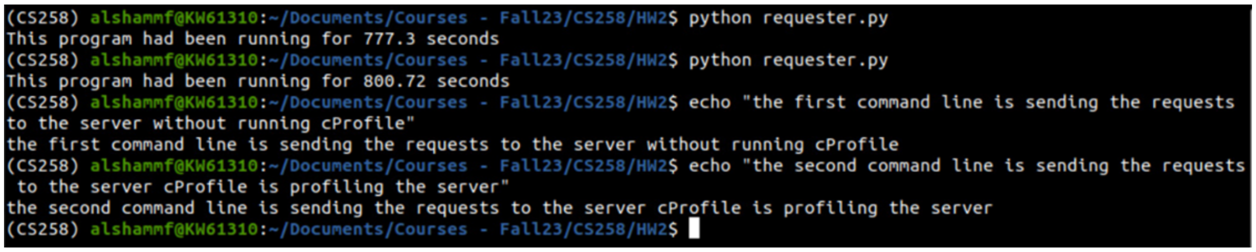


ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
10030	802.5	0.08001	802.5	0.08001	~:0(<method 'poll' of 'select.poll' objects>)
9985	10.47	0.001049	10.47	0.001049	~:0(<method 'recv_into' of '_socket.socket' objects>)
59777	2.478	4.145e-05	2.478	4.145e-05	~:0(<method 'sendall' of '_socket.socket' objects>)
9985	1.106	0.0001107	1.106	0.0001107	~:0(<method '_accept' of '_socket.socket' objects>)
9976	0.9401	9.423e-05	0.9401	9.423e-05	~:0(<method 'write' of '_io.TextIOWrapper' objects>)
9976	0.2864	2.871e-05	0.6683	6.699e-05	feedparser.py:471(_parse_headers)
10030	0.2641	2.633e-05	802.8	0.08004	selectors.py:403(select)
9986	0.218	2.183e-05	0.218	2.183e-05	~:0(<function socket.close at 0x14e43c2dc040>)
19952	0.2167	1.086e-05	1.459	7.314e-05	feedparser.py:218(_parsegen)
9976	0.2141	2.146e-05	2.776	0.0002783	server.py:268(parse_request)
9985	0.1851	1.854e-05	0.2449	2.452e-05	socket.py:302(makefile)
59856	0.1532	2.56e-06	0.2665	4.452e-06	message.py:462(get)
119709	0.1504	1.256e-06	0.2579	2.155e-06	_policybase.py:293(header_source_parse)
9985	0.1247	1.249e-05	0.1247	1.249e-05	~:0(<method 'shutdown' of '_socket.socket' objects>)
149637	0.1193	7.971e-07	0.1314	8.781e-07	feedparser.py:78(readline)
9985	0.1177	1.178e-05	1.517	0.0001519	socket.py:286(accept)
9976	0.115	1.153e-05	0.1926	1.931e-05	client.py:208(_read_headers)

Figure 4:Part of the default output of the profiled server while serving 5000 requests

When a flame graph is compared to the default output, one advantage of flame graphs over the table format is the effective visibility of the most time consuming functions and how functions are related in terms of calling dependencies. It can also summarize multiple insights (e.g. which function is more important to optimize) in a much smaller space compared to the table representation. Nevertheless, table representation is complementary to the graph. The programmer can use the table for further analysis after identifying which function(s) to be optimized and how it can be enhanced.

In terms of time dilation, the below screenshot, shown in figure.5, indicates how much it differed in terms of time consumption when the server was being profiled during responding to the request stream. The profiler increased the response time of the 5000 requests from 777s to 800s with a relative time increase of 2.96%. It should be mentioned that the experimentation was conducted using an Intel Xeon CPU. Also, the profiled demo website was very simple. The process is expected to have larger time dilation if a weaker CPU is to be used and a more sophisticated website is being served by the server.



```
(CS258) alshammf@KW61310:~/Documents/Courses - Fall23/CS258/HW2$ python requester.py
This program had been running for 777.3 seconds
(CS258) alshammf@KW61310:~/Documents/Courses - Fall23/CS258/HW2$ python requester.py
This program had been running for 800.72 seconds
(CS258) alshammf@KW61310:~/Documents/Courses - Fall23/CS258/HW2$ echo "the first command line is sending the requests
to the server without running cProfile"
the first command line is sending the requests to the server without running cProfile
(CS258) alshammf@KW61310:~/Documents/Courses - Fall23/CS258/HW2$ echo "the second command line is sending the requests
 to the server cProfile is profiling the server"
the second command line is sending the requests to the server cProfile is profiling the server
(CS258) alshammf@KW61310:~/Documents/Courses - Fall23/CS258/HW2$ 
```

Figure 5: Screenshot of response time estimation with and without profiling.

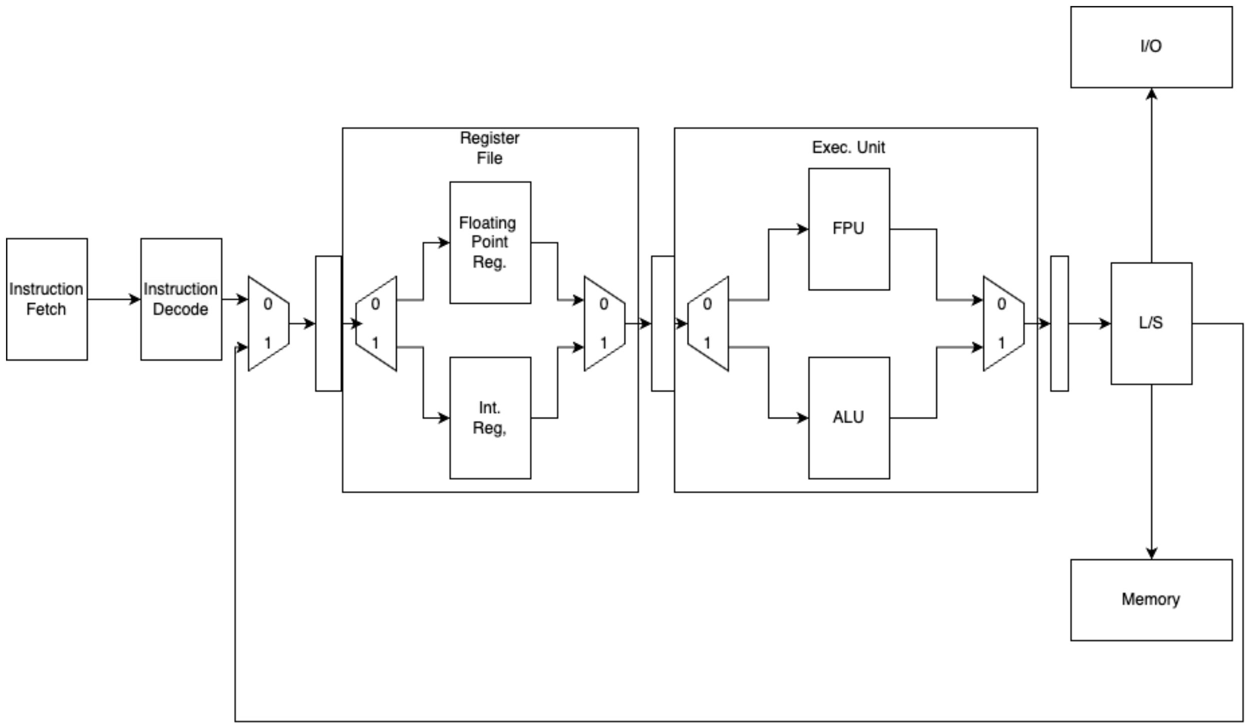
Q4.

Pipeline Properties:

Function	Comments
Single-ported integer file	Read or write one at a time. For two operands it will take 2 cycles.
Single-ported floating file	
Integer Adder	1 cycle
Float Mul	7 cycle
Float Add	4 cycle
Instruction Fetch (IF)	1 cycle
Instruction Decode (ID)	1 cycle

Assumptions:

- Single Issue
- No bypass between load/store unit and Execution Unit
- Either floating point register or integer register can be fetched at a time.
- Either floating point or integer operation can be executed at a time and not simultaneously
- Register renamed is committed once the previously dependent execution is done.



Normal Execution:

```
start:
     L1:  ld   f2, (r1)
     L2:  mul  f4, f2, f0
     L3:  ld   f6, (r2)
     L4:  add  f6, f4, f6
     L5:  st   f6, (r2)
     L6:  add  r1, r1, 8
     L7:  add  r2, r2, 8
     L8:  add  r3, -1
     L9:  bnz  r3, start
```

Cycle	IF	ID	REG	REG	ALU	MEM	REG
1	L1						
2	L2	L1					
3	L3	L2	L1, r1				
4	L4	L3		L1, f2			
5	L5	L4			L1		
6	L6	L5				L1	
7	L7	L6					L1, f2
8	L8	L7	L2, f0				
9	L9	L8	L3, r2	L2, f2			
16 (9+7)		L9		L3	L2, mul		
17					L3	L2	
18						L3	L2, f4
19							L3, f6
20			L4, f6				
21				L4, f4			
25 (21+4)					L4, add		
26						L4	
27							L4, f6
28			L5, r2				
29			L6, r1	L5, f6			
30			L7, r2	L6	L5		
31			L8, r3	L7	L6	L5	
32				L8	L7	L6	
33					L8	L7	L6, r1

34						L8	L7, r2
35							L8, r3
36			L9, r3				
37				L9			
38					bnz		

`IPC = 9/38 = 0.239`

Optimal Execution :

- With register renaming
- Out of order execution
- With branch prediction

```
start:
     L1:  ld   f2, (r1)
     L3:  ld   f6, (r2)
     L2:  mul  f4, f2, f0
     L6:  add  r1, r1, 8
     L4:  add  f6, f4, f6
     L5:  st   f6, (r2)
     L7:  add  r2, r2, 8
     L8:  add  r3, -1
     L9:  bnz  r3, start
```

Cycle	IF	ID	REG	REG	ALU	MEM	REG
1	L1						
2	L2	L1					
3	L3	L2	L1, r1				
4	L4	L3		L1			
5	L5	L4	L3, r2		L1		
6	L6	L5		L3, f6		L1	
7	L7	L6			L3		L1, f2

8	L8	L7	L2, f0			L3	
9	L9	L8	L6, r1	L2, f2			L3, f6
16 (9+7)	L1(2)	L9	L4, f6	L6	L2, mul		
17	L2(2)	L1(2)	L7, r2	L4, f4	L6, add	L2	
21 (17+4)	L3(2)	L2(2)	L8, r3	L7	L4, add	L6	L2, f4
22	L4(2)	L3(2)	L1(2), r1	L8	L7, add	L4	L6, r1
23	L5(2)	L4(2)	L3(2), r2	L1(2)	L8, add	L7	L4, f6
24	L6(2)	L5(2)	L5, r2	L3(2), f6	L1(2)	L8	L7, r2
25	L7(2)	L6(2)		L5, f6	L3(2)	L1(2)	L8, r3
26	L8(2)	L7(2)	L9, r3		L5	L3(2)	L1(2), f2
27	L9(2)	L8(2)	L2(2), f0	L9		L5	L3(2), f6
28		L9(2)	L6(2), r1	L2(2), f2	L9, bnz		

IPC = 11/28 = 0.392
Speed up = 0.392/0.239 = 1.6x