Internship Report
Duration: May-2021 to July-2021

Topic: Framework for SoC Simulation (QEMU)

Manas Modi (U19EC018)
B.Tech. (ECE)
SVNIT, Surat

Academic Supervisor:
Dr. Pinalkumar J. Engineer

Acknowledgment

Projects are the bridge between theoretical and practical learning. The internship opportunity I had was an excellent chance for learning and professional development. I consider myself to be very lucky to be a part of it.

I would like to express my gratitude to Dr. Pinalkumar J. Engineer, Department of Electronics Engineering, SVNIT, for providing me with the topic for this project, for guidance and direction during the same period. I would also like to thank Nagar Mitul Sudhirkumar for his advice on the same.

I would like to thank them for transmitting to me their passion for the area of embedded systems and for their time and patience in evaluating this report.

Introduction

Multi-Processor System-On-Chip (MPSoC) development with homogeneous and heterogeneous processor architectures has led to the use of virtual platforms for evaluating various systems and analyzing hardware/software proposals before their implementation stage. This project deals with the exploration of QEMU for emulating multicore processors. QEMU was chosen because it is fast, open-source, and supports numerous processor architectures. It supports the emulation of many ARM, AVR, Sparc32, MIPS, x86, and other machines. It has three modes of operation, namely User-mode emulation, Full-system emulation, and KVM Hosting. The host machine used for this project has Intel x86_64 processor and Ubuntu 18.04.4 LTS operating system.

Background

In user-mode emulation, QEMU can run programs compiled for the target processor supporting different instruction set architecture on the host processor. In full-system emulation mode, QEMU emulates an entire system (virtual machine) using Dynamic Binary Translation, including processor, peripherals, interrupts, and more. Suppose the target and host machines have the same architecture. In that case, QEMU can use KVM (Kernel-based virtual machine), open-source software that is a full virtualization solution for Linux on x86 hardware and currently supports Intel-V and AMD-V KVM to have faster emulation speeds. ARM KVM is also possible. The kernel component and user space component of KVM are included in mainline Linux and mainline QEMU respectively.

For choosing target development boards, it was considered that the development board is present in Embedded System Lab, SVNIT and can be emulated using QEMU. FPGA boards were also considered but their emulation did not run successfully due to various reasons. Hence, Raspberry Pi 3 was chosen as the target machine because the emulation results can be compared with the working of actual hardware if needed. It has a quad-core ARM Cortex A-53 processor that is readily available on development boards and has well-developed models across multiple simulation tools.

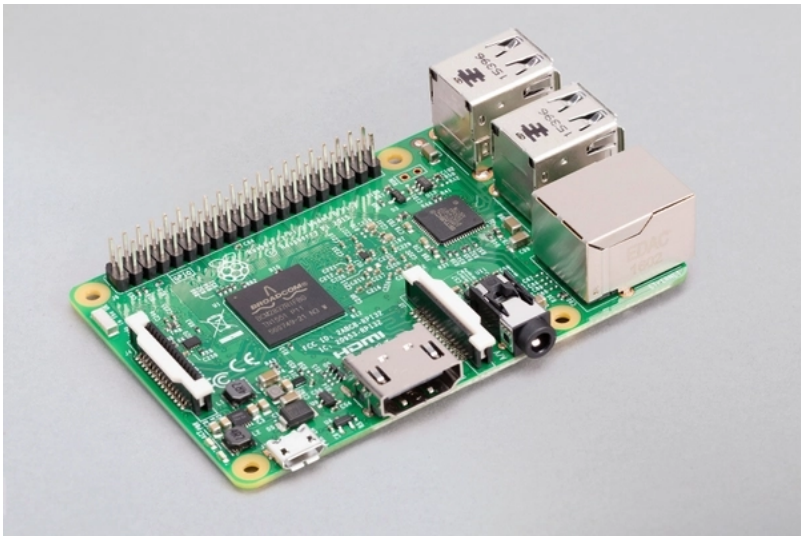


Figure 1.1 Raspberry Pi 3 Model B (Source)

Bare Metal Programming for ARM was also explored to some extent for emulation using QEMU. But because of its disadvantages such as difficult to code and debug, increase in complexity, etc., it was not used.

Listing 1.1: Sample Hello World code for Bare Metal ARM

```
volatile unsigned int * const UART0DR = (unsigned int *)0x101f1000;

void print_uart0(const char *s) {
 while(*s != '\0') { /* Loop until end of string */
 *UART0DR = (unsigned int)(*s); /* Transmit char */
 s++; /* Next char */
 }
}

void c_entry() {
 print_uart0("Hello world!\n");
}
```

Emulation Mode for Project

Among three modes of operation of QEMU, Full-system emulation was chosen for this project. It sets up a complete virtual machine and helps develop and test cross-platform software, which may require certain hardware interactions. KVM was not used due to the different guest and host architectures. User mode emulation can be used to run programs

compiled for other Instruction Set Architecture on any supported architecture. It requires certain cross-compiling tools.

Cross compiling tools for C Programs can be installed using:

- sudo apt install qemu-user qemu-user-static gcc-aarch64-linux-gnu binutils-aarch64-linux-gnu binutils-aarch64-linux-gnu-dbg build-essential
- sudo apt install gcc-arm-linux-gnueabihf binutils-arm-linux-gnueabihf binutils-arm-linux-gnueabihf-dbg

User-mode emulation Example

```
#include <stdio.h>

int main(void) {
    return printf("Hello, World!\n");
}
```

Sample Code

Execute ARM64 binaries:

- aarch64-linux-gnu-gcc -o hello hello.c
- qemu-aarch64 -L /usr/aarch64-linux-gnu ./hello
- Hello, World!

Parallel Programming

Working of multi-core architecture is demonstrated using multiprocessing in Python programming language and QEMU-MTTCG. MTTCG (Multi-Threaded Tiny Code Generator) enables to run one guest thread per guest CPU or vCPU (virtual CPU) in the full-system emulation. It increases performance by parallelizing the computational load of the emulation. Python was used as it is well documented and easy to learn.

Multiprocessing Implementation

For examining the functionality of a multi-core processor, four types of problems were implemented. These problems deal with the same code running on different cores, different codes running on different cores, producer-consumer problem, and data synchronization. The time required for execution of sample codes may change.

1. Same Code running on Different Cores

   Code:

```
import multiprocessing
import time

def factorial(num):
    start  = time.perf_counter()
    fact = 1
    for i in range(1,num+1):
        fact = fact * i
    print("factorial time consumed",time.perf_counter()-start)



if __name__ == "__main__":
    p1 = multiprocessing.Process(target=factorial, args=(50000, ))
    p2 = multiprocessing.Process(target=factorial, args=(50000, ))
    p3 = multiprocessing.Process(target=factorial, args=(50000, ))
    start  = time.perf_counter()
    p1.start()
    p2.start()
    p3.start()

    p1.join()
    p2.join()
    p3.join()
    print("total time consumed: ",time.perf_counter()-start)
```

   Result:

```
factorial time consumed 10.279674720000003
factorial time consumed 10.414373984000122
factorial time consumed 10.493155152000327
total time consumed:  10.556066031999762
```

2. Different Codes running on Different Cores

Code:

```
import multiprocessing
import time

def factorial(num):
    start  = time.perf_counter()
    fact = 1
    for i in range(1,num+1):
        fact = fact * i

    print("factorial time consumed",time.perf_counter()-start)

def sum_of_power1(n):
    start  = time.perf_counter()
    ans = sum(x**1000 for x in range(n))
    print("sum of power 1000 time consumed",time.perf_counter()-start)

def sum_of_power(n):
    start  = time.perf_counter()
    var = sum(x**1200 for x in range(n))
    print("sum of power 1200 time consumed",time.perf_counter()-start)

if __name__ == "__main__":
    p1 = multiprocessing.Process(target=factorial, args=(100000, ))
    p2 = multiprocessing.Process(target=sum_of_power1, args=(100000, ))
    p3 = multiprocessing.Process(target=sum_of_power, args=(100000, ))
    start  = time.perf_counter()
    p1.start()
    p2.start()
    p3.start()

    p1.join()
    p2.join()
    p3.join()
    print("total time consumed: ",time.perf_counter()-start)
```

Result:

```
factorial time consumed 60.86587870400001
sum of power 1000 time consumed 64.18488619199991
sum of power 1200 time consumed 77.24359712
total time consumed:  77.36640960000011
```

3. Producer-Consumer Problem

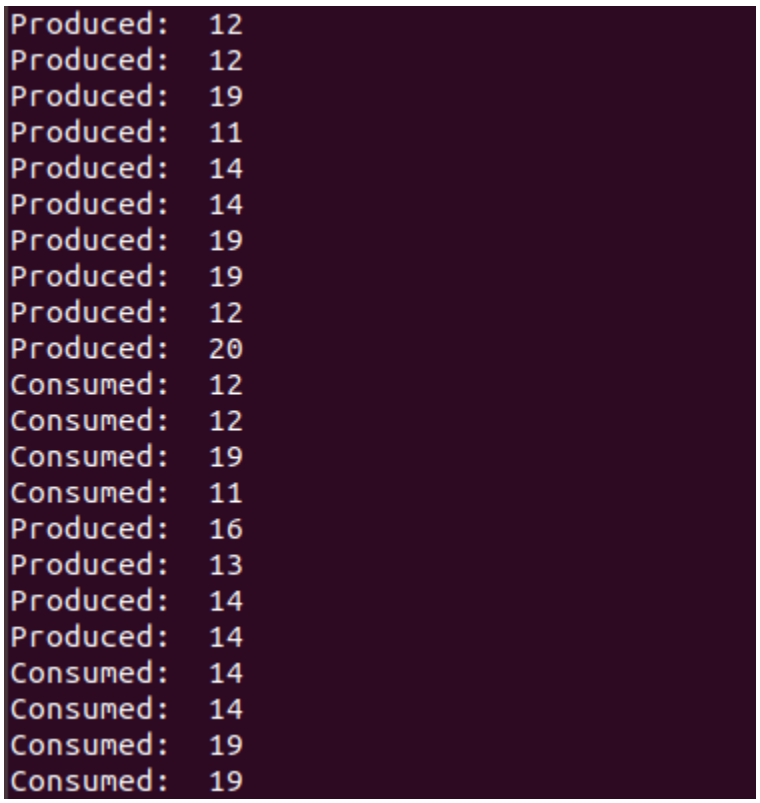
Code:

```
import multiprocessing
import time
import random

size = 10
q = multiprocessing.Queue(size)

#produces numbers and puts them in q
def producer(q, lock):
    while True:
        lock.acquire()
        if not q.full():
            x = random.randint(11,20)
            q.put(x)
            print("Produced: ",x)
            time.sleep(random.random())
        lock.release()

#accesses and removes numbers from q
def consumer(q, lock):
    while True:
        lock.acquire()
        if not q.empty():
            x = q.get()
            print("Consumed: ", x)
            time.sleep(random.random())
        lock.release()

if __name__ == '__main__':

    lock = multiprocessing.Lock()

    p1 = multiprocessing.Process(target=producer, args=(q, lock))
    p2 = multiprocessing.Process(target=consumer, args=(q, lock))
```

```
p1.start()
p2.start()

p1.join()
p2.join()
```

In this code, the Lock class prevents concurrent access to a shared resource by multiple processes in parallel programming. The lock is implemented using a Semaphore object provided by the Operating System.

Result:



4. Data Synchronization

Code:

```
import multiprocessing

# function to withdraw from account
def withdraw(balance, lock):
```

```
    for _ in range(10000):
        lock.acquire()
        balance.value = balance.value - 1
        lock.release()

# function to deposit to account
def deposit(balance, lock):
    for _ in range(10000):
        lock.acquire()
        balance.value = balance.value + 1
        lock.release()

def perform_transactions():

    # initial balance (in shared memory)
    balance = multiprocessing.Value('i', 1000)

    lock = multiprocessing.Lock()

    p1 = multiprocessing.Process(target=withdraw, args=(balance,lock))
    p2 = multiprocessing.Process(target=deposit, args=(balance,lock))

    p1.start()
    p2.start()

    p1.join()
    p2.join()

    # print final balance
    print("Final balance = {}".format(balance.value))

if __name__ == "__main__":
    for _ in range(10):

        # perform same transaction process 10 times
        perform_transactions()
```

Result:

```
Final balance = 1000
Final balance = 1000
Final balance = 1000
Final balance = 1000
Final balance = 1000
Final balance = 1000
Final balance = 1000
Final balance = 1000
Final balance = 1000
Final balance = 1000
```

If Lock is not used, the value of the shared variable may become unpredictable.

Result (Lock not used):

```
Final balance = -223
Final balance = 794
Final balance = 37
Final balance = -361
Final balance = -2091
Final balance = -869
Final balance = 516
Final balance = -1412
Final balance = 532
Final balance = 919
```

Conclusion

In this project, a multicore processor was emulated using QEMU. System Emulation provided a virtual model of an entire machine to run guest OS. Various areas related to the working of multicore processors were examined using four simple codes in the Python programming language. Parallel processing reduces the total time taken to complete multiple processes by distributing them to various cores of a processor and increasing overall efficiency. Locking deals with race conditions so that multiple operations can access a shared resource concurrently and give accurate results.

Appendix A:QEMU Installation

- Installing Dependencies
  - sudo apt-get install git libglib2.0-dev libfdt-dev libpixman-1-dev zlib1g-dev
  - sudo apt-get install git-email
  - sudo apt-get install libaio-dev libbluetooth-dev libbrlapi-dev libbz2-dev
  - sudo apt-get install libcap-dev libcap-ng-dev libcurl4-gnutls-dev libgtk-3-dev
  - sudo apt-get install libibverbs-dev libjpeg8-dev libncurses5-dev libnuma-dev
  - sudo apt-get install librbd-dev librdmacm-dev
  - sudo apt-get install libsasl2-dev libsdl1.2-dev libseccomp-dev libsnappy-dev libssh2-1-dev
  - sudo apt-get install libvde-dev libvdeplug-dev libvte-2.90-dev libxen-dev liblzo2-dev
  - sudo apt-get install valgrind xfslibs-dev
  - sudo apt-get install libnfs-dev libiscsi-dev
  - sudo apt-get install -y ninja-build

- Download and Build
  - wget https://download.qemu.org/qemu-6.0.0.tar.xz
  - tar xvJf qemu-6.0.0.tar.xz
  - cd qemu-6.0.0
  - ./configure
  - make

Launch QEMU

Required kernel and dtb files must be extracted from the Raspberry Pi OS image or downloaded from this Github repository.
After having the required files and ISO image, resize the image and launch QEMU using the command given below:

```
 sudo qemu-system-aarch64 -M raspi3 -append "rw earlyprintk loglevel=8
console=ttyAMA0,115200 dwc_otg.lpm_enable=0 root=/dev/mmcblk0p2 rootdelay=1"
-dtb ./native-emulation/dtbs/bcm2710-rpi-3-b-plus.dtb -sd
2021-05-07-raspios-buster-armhf.img -kernel ./native-emulation/5.4.51\
kernels/kernel8.img -m 1G -smp 4 -monitor null -serial stdio -nographic --accel
tcg,thread=multi
```

In the above command, the system is modeled with the following specifications:

- 1 GB RAM
- 4 ARMv8 cores
- Raspbian OS Operating system
- Simple command-line interface

Resize the Raspbian Image

- qemu-img resize 2021-05-07-raspios-buster-armhf.img 16G

Resizing the image is necessary for more local disk space.

References

- Enabling Parallelized-QEMU for Hardware/Software Co-Simulation Virtual Platforms. (Source)
- Evaluating Gem5 and QEMU Virtual Platforms for ARM Multicore Architectures. (Source)
- Qemu Documentation. (Source)
- QEMU-MTTCG. (Source)
- Native emulation of Rpi2/3 using Qemu's Raspi2/3 machine. (Source)
- Raspberry Pi 3. (Source)
- QEMU CPU Emulator User Documentation. (Source)
- Multiprocessing — Process-based parallelism. (Source)
- Multiprocessing with Python. (Source)
- Running Arm Binaries on x86. (Source)
- Hello world for bare metal ARM using QEMU. (Source)
- Kernel Virtual Machine. (Source)
- Synchronization and Pooling of processes in Python. (Source)
- Using QEMU for cross-platform development. (Source)
- Xilinx QEMU User Guide. (Source)
- Multithreading VS Multiprocessing in Python. (Source)