Development of Type-2 Hypervisor for MIPS64 Based Systems

October 15

2013

[2nd Deliverable]

This document is version 2 of first report and includes the implementation details of current deliverable of "Development of Type 2 Hypervisor for MIPS64 based Systems" project, funded by National ICT R & D Fund Pakistan. This report starts with brief description of project objectives, technical details of our approach, challenges and their solutions. Complete description of testing infrastructure, test cases and test results are discussed later on. The report concludes with the impact of current deliverable on the overall project progress.

Test Cases Result Report

High Performance Computing and Networking Laboratory HPCNL
Al-Khwarizmi Institute of Computer Science, University of Engineering and Technology Lahore Pakistan

Table of Contents

1.Project Description

The main objective of this project is to develop an open source Type 2 hypervisor, for Linux-based MIPS64 embedded devices. Type-2 means that it is a hosted hypervisor which runs on MIPS64 based Linux systems as a Linux process. It is intended that the hypervisor will (1) support installation and execution of un-modified MIPS64 Linux guest(s) on un-modified MIPS64 Linux host (2) take advantage of virtualization for improved hardware utilization and performance optimization, by using multiple MIPS cores. Our focus on MIPS is due to the fact that MIPS based systems are lagging behind in the use of virtualization. One of the reasons is that many MIPS based processors are used in low end consumer devices like TV set top box, GPS navigation system and printers. There isn't a clear cut use case for virtualization here. But few of the MIPS vendors target higher end embedded devices like network switches and routers, GSM/LTE base station equipment and MIPS based blade servers. There are clear-cut virtualization use cases for this higher-end MIPS segment.

The development started on April 1, 2013 and first deliverable is due after 3.5 months i.e. July 15, 2013. In first deliverable, we need to build the required infrastructure. The infrastructure should print guest kernel banner on console, at the end of 1st deliverable. Second deliverable is due after 6.5 months of commencement data i.e. October 15, 2013. The milestone in 2nd

deliverable is the dynamic code patching of one sensitive guest instruction with one safer instruction.

2.Development Strategy

We are following a hybrid approach to develop the hypervisor. Executable binary is loaded in the address space of hypervisor and mapped to a known memory address. Traditional trap-and-emulate technique is used to take control of each instruction. Hybrid approach works as following:

1. If the instruction is privileged, it is emulated.
2. If the instruction manipulates `sp`, `gp` and/or `k0` registers, it is dynamically patched before execution.
3. Otherwise, the instruction is executed directly on hardware as it is.

3.Challenges and Solutions

Development of a hypervisor is quite challenging. Runtime systems like hypervisor are typically sensitive to runtime overhead. Runtime overheads, like that of emulation, result in significant performance degradation if not taken care of. To reduce runtime overhead, our initial strategy was to emulate privileged instructions only and execute rest of the instructions on bare metal (hardware). On execution of privileged instruction in user mode, a trap is generated (i.e. `SIGILL` signal is raised). We implemented a signal handler that catches signal, fetch/decode the instruction and emulate its behavior.

Challenge 1
Standard C/C++ libraries (e.g. `glibc`) do not allow modification of `sp ($29)` and `gp ($28)` registers in user mode. Non-privileged instructions dealing with these registers can't be executed directly on hardware. Similarly, `K0 ($26)` and `K1 ($27)` registers produce unexpected results because they are interrupt handling registers used by kernel and potentially not used by user programs.

Solution 1
In addition to emulation of privileged instructions, we implemented the code for emulation of non-privileged instructions involving `gp` and `sp` register.

Challenge 2
The next challenge was that any instruction can potentially manipulate `gp` and `sp` registers and we may end up in emulating all instructions, resulting in poor performance.

Solution 2
We implemented code for dynamic code patching and patched all instructions involving `sp($29)`, `gp($28)` and `k1($27)` registers. Patched instructions were harmlessly executed on hardware and contents of corresponding registers were updated later (in a trap handler).

Challenge 3
To ensure correct execution of guest code, we need to use debugger extensively during development. With the increasing number of executed instructions, debugging information becomes complex and hard to read. In case of an error condition, we need to determine the instruction that produced error. Searching the error-causing instruction between two states of emulator is not a trivial task.

Solution 2
In this stage, we generate trap on every instruction so that debugging and testing could be made easier. Now, the guest code is executed using a hybrid approach: privileged instructions are emulated, instructions involving `sp`, `gp`, `k0` registers are patched and the rest are allowed to execute on hardware unchanged.

4.Testing Infrastructure

Testing infrastructure involves MIPS64 evaluation board with multicore Octeon processor, hardware debugger (JTAG), development system and testing routines. We need rigorous testing to make sure that guest kernels run in complete isolation from each other and from host kernel. Similarly, on each instruction execution in virtualized environment, changes to system state should imitate the changes made by executing the same in real environment.

4.1.Test Cases

Hypervisor manipulates (i.e. emulation/code patching) guest code to use privileged hardware resources controlled by host kernel. Hence, various test cases are needed to make sure the consistency and integrity of guest code. Up to current deliverable, our focus is on the test cases discussed in following subsections.

4.1.1. Matching system states

In our case, system state consists of the values of general purpose registers and some of coprocessor 0 (CP0) registers at a particular instance. In order to verify the correct working of hypervisor, we run (same) executable binary

directly on Cavium MIPS64 board and through hypervisor. We get real system state on each privileged instruction by using JTAG and compare both outputs (hypervisor and JTAG) for verification. JTAG provides the facility of setting hardware breakpoints at each privileged instruction to stop and take log of system state. Without setting breakpoints, it logs the state at every instruction execution.

4.1.2. Execution path

Due to emulation and code patching, guest code execution path may differ from that of the same binary running directly on board. Taking Log at breakpoints may fail due to unavailability of a priori information about execution path of guest code. For example, if guest code sway from the path containing some breakpoint, we would not be able to take system state at that breakpoint and state matching test result will be misleading.

Logging system state after each instruction execution could help in avoiding the situation of taking wrong execution path. This allows us to debug the potential causes of error (if any) by looking at system state before and after the execution of malfunctioning instruction. However, there is inherent overhead of logging state at each instruction execution. There were about 339351 instructions executed by u-boot. JTAG created a file of about 6MB in approximately 7 hours. Generated file contains data (i.e. general purpose registers + CP0 registers content) of about 2600 states. To reduce state logging time, we decided to use a small binary (i.e. code for irrelevant external devices is commented out) and take log on Quick Emulator (QEMU). To take log on QEMU, we used the expertise of another HPCNL team working on a different project titled “System Mode Emulation in QEMU”.

4.1.3. Comparing Console Output

On reaching the stage where console is get attached with our hypervisor, the binaries, executing within hypervisor, starts emitting messages on console. It serves as another way of validation, whereby output of our hypervisor is compared with that of real MIPS system.

4.1.4. Progress

The progress is tracked by identifying labeled blocks, in binary code. The blocks are identified by following the control flow of binary. When the instructions in one block are executed, its label is noted and control is conditionally/unconditionally transferred to the next block in control flow.

This way we measure the progress that how many blocks have been executed and how many left.
Emulation and code patching may lead to infinite loops in the code. For example, if emulation/patching changes system state in such a way that control is transferred to one of prior blocks of the current block, the hypervisor will enter into an infinite loop. We need to avoid the situations like this in order to make progress.

4.1.5. Translation Lookaside Buffer (TLB)

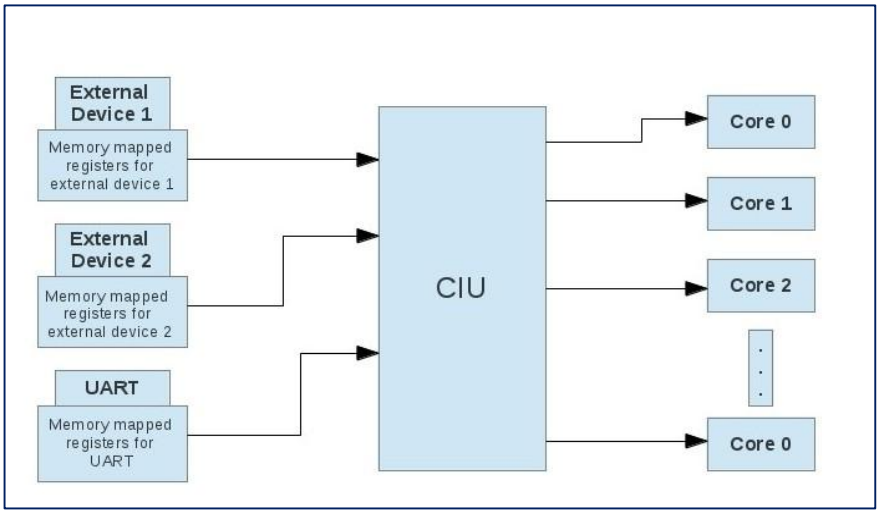
TLB is a cache used to speedup virtual address to physical address translation. In case of type 2 hypervisor, TLB translates guest virtual address to host virtual address. There are four basic TLB functions: `probe`, `read`, `write-random` and `write-index`. TLB `probe` searches for a TLB entry using the value of `EntryHi` register of co-processor 0 (CP0). If valid entry is found, it places index of TLB entry in CP0 `index` register, otherwise it sets probe bit of `index` register and consult page table. TLB `read` gets value from CP0 `index` register and checks the validity of data at this index. If data is valid, the components of entry (i.e. `entryHi`, `entryLo0`, `entryLo1` and `page-mask`) are moved to corresponding CP0 registers. Otherwise TLB `read` raises invalid data exception. TLB `write-random` gets index of TLB entry from CP0 `random` register and checks the validity of data at the index. If entry is dirty, it raises dirty data exception, otherwise it writes corresponding values of CP0 registers (i.e. `entryHi`, `entryLo0`, `entryLo1` and `page-mask`) to the TLB entry at that index. TLB `write-index` works same as TLB `write-random` except that it gets index value from CP0 `Index` register.

On TLB miss, page table functions are called and guest virtual address is searched in the page table. If found, corresponding host virtual address is returned, otherwise a new memory region is allocated using `mmap()` and its address is returned. Current implementation does not impose any restriction on memory allocation (i.e. it will be implemented in future deliverables). To reclaim guest memory, one possible solution is to use OOM killer of guest kernel.

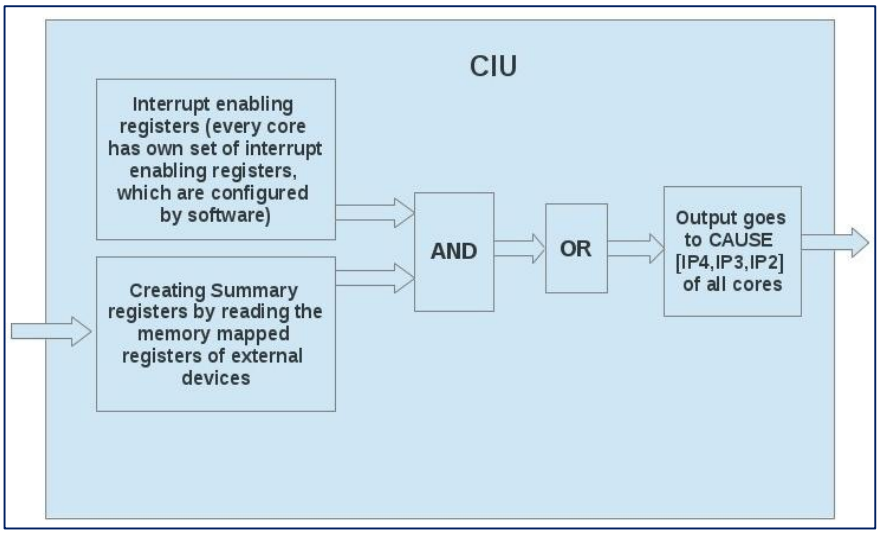
4.1.6. Central Interrupt Unit (CIU)

CIU is responsible for dispatching interrupt requests (coming) from external devices to a particular core. CIU is discussed here in context of our test bed i.e. Cavium Networks OCTEON Plus CN57XX evaluation board [1]. CIU distributes a total of 37 interrupts i.e. 3 per core plus 1 for PCIe. Three interrupts for each core set/unset bit 10, 11, 12 of `Cause` register of the core. Using these `cause` register bits, interrupt handler of a core could

prioritize different interrupts. Interrupt requests from external devices are accumulated in a 72-bit summary vectors with naming convention CIU_INT<core#>_SUM<0|1|4>. Summarized interrupts reach to their ultimate destination by using corresponding 72 bits interrupt enable vector with naming convention CIU_INT<core#>_EN<0|1> and CIU_INT<core#>_EN4_<0|1>.



(a)



(b)

Figure 1: Central Interrupt Unit. (a) Interrupt distribution from external devices to core. (b) Internal working of CIU. Inward arrow is coming from external devices and outward arrow is going to the all cores.

Interaction of CIU, external devices and cores is shown in Figure 1 (a). CIU reads memory mapped registers of the external devices to know about pending interrupt requests and sets corresponding bits of cause register of target

core. Figure 1 (b) shows a simplest description of the internal working of CIU, where interrupt identification/handling is done in software.
At this stage, we implemented a simplest abstraction of CIU. Currently, CIU is a separate unit and is only reading UART's memory mapped registers. UART writing and other devices would be implemented in future. CIU itself has set of `summary` and `enable` registers for every core. An interrupt request goes to only those cores that had enabled the interrupt by configuring its `enable` register. In current code, CIU reads UART's Interrupt Identification Register (IIR), extracts identity bits and set/clear the corresponding `summary` registers bits. These `summary` registers for every core are than "AND " with their `enable` registers to set or clear `cause` register's bit 10, 11 and 12.

4.2.Test Results

The sample output of system state test, execution path test, TLB, page table, CIU and hypervisor console is elaborated in this section.

4.2.1. Output of System State Matching Test

We trap at every instruction and create a state-file. We match this state file with qemu log state-file to see if any register contains different contents. Mismatches are written in other file as shown in figure 2.

4.2.2. Output of Execution Path Test

We face difficulties in debugging if QEMU log is missing instruction log at different points. To ensure that the hypervisor is on the right track we match the Program Counter (PC) values taken by hypervisor and all the PC values taken in qemu log, as shown in figure 3.

4.2.3. Output of TLB Testing

To test TLB mechanism, random TLB entries are generated and searched in TLB. A TLB miss is obvious because the entry is newly generated. Hence, probe bit is set and TLB `write-random` function is called to places this entry at the index present in CP0 `random` register. Random register is incremented and entry is searched again. On TLB hit, we call TLB `read` to fetch the entry from the index set by TLB probe, as shown in Figure 4.
Then TLB `write-index` function is called that writes TLB entry at the index present in `index` register. As `index` register was set by TLB probe, it writes the entry at same index that was previously written by TLB `write-random`. TLB probe and TLB read are called again and then a new random entry is generated. This process is repeated 640 times.

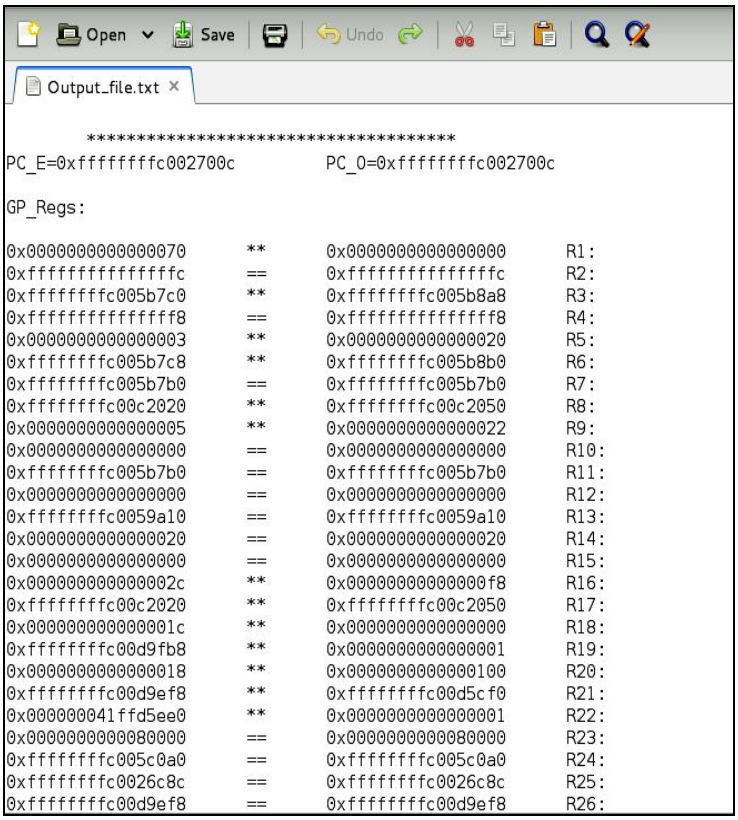


Figure 2. Output of system state matching test.

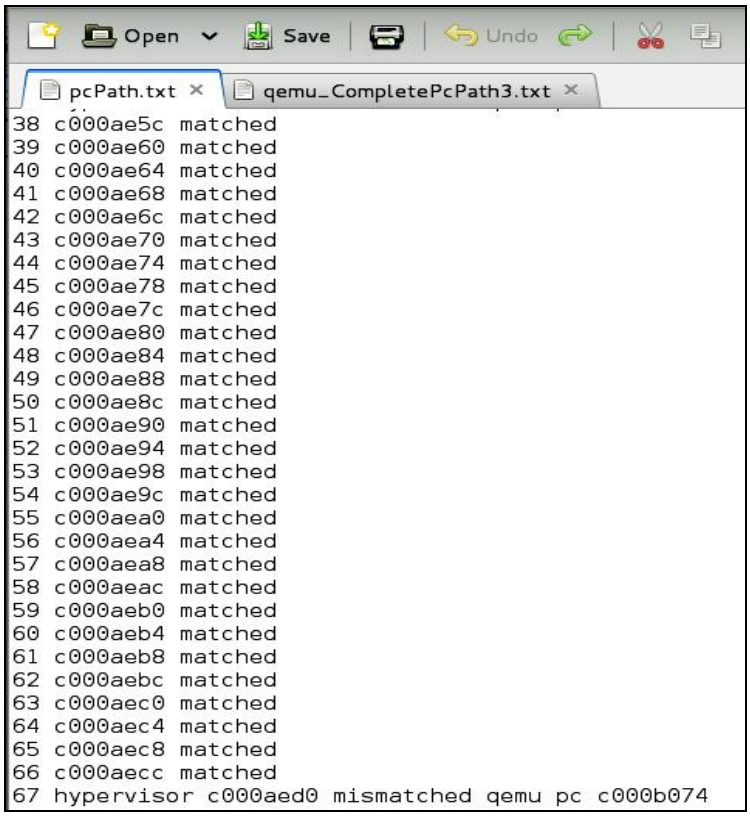


Figure 3. Output of Execution Path Test.

As TLB could have 64 entries at max, additional entries require a replacement policy. After setting all entries, TLB entries are printed, as shown in Figure 5.

```
File  Edit  View  Search  Terminal  Help
octeon:/home/kics/Usama_Data/VExecutor# ./dist/Debug/MIPS64-Linux-x86/vexecutor
Wired value is 0.
tlbr=>Entry not found
tlbwr=>TLB Written: At index 63, Random: 63
tlbp:TLB found entry:63
tlbr=>TLB read: At index 63, Random: 63
Valid:TLB_Invalid exception handling
tlbwi=>TLB Written: At index 63, Random: 63
tlbp:TLB found entry:63
tlbr=>TLB read: At index 63, Random: 63
Valid:TLB_Invalid exception handling
############################################################
tlbr=>Entry not found
tlbwr=>TLB Written: At index 63, Random: 0
tlbp:TLB found entry:0
tlbr=>TLB read: At index 0, Random: 0
Valid:TLB_Invalid exception handling
tlbwi=>TLB Written: At index 0, Random: 0
tlbp:TLB found entry:0
tlbr=>TLB read: At index 0, Random: 0
Valid:TLB_Invalid exception handling
############################################################
tlbr=>Entry not found
tlbwr=>TLB Written: At index 63, Random: 1
tlbp:TLB found entry:1
tlbr=>TLB read: At index 1, Random: 1
Valid:TLB_Invalid exception handling
tlbwi=>TLB Written: At index 1, Random: 1
tlbp:TLB found entry:1
tlbr=>TLB read: At index 1, Random: 1
Valid:TLB_Invalid exception handling
############################################################
tlbr=>Entry not found
tlbwr=>TLB Written: At index 63, Random: 2
tlbp:TLB found entry:2
tlbr=>TLB read: At index 2, Random: 2
Valid:TLB_Invalid exception handling
```

Figure 4. Searching for random TLB entry.

To test page table, a random guest virtual address is generated and searched in the page table. Obviously, there is no matching entry in page table because this is the newly generated address. Hence, it maps a new memory region and returns its address. This process is repeated several times. Each time it maps a new region, places translation in page table and returns translated address. The output is shown in Figure 6 (a). After creating appropriate entries in page table, same process is repeated again for all the generated addresses and we get valid translation now, as shown in Figure 6 (b). Then whole page table is printed in Figure 6 (c) and reverse page table, shown in Figure 6 (d), is also managed to use for future testing of hypervisor.

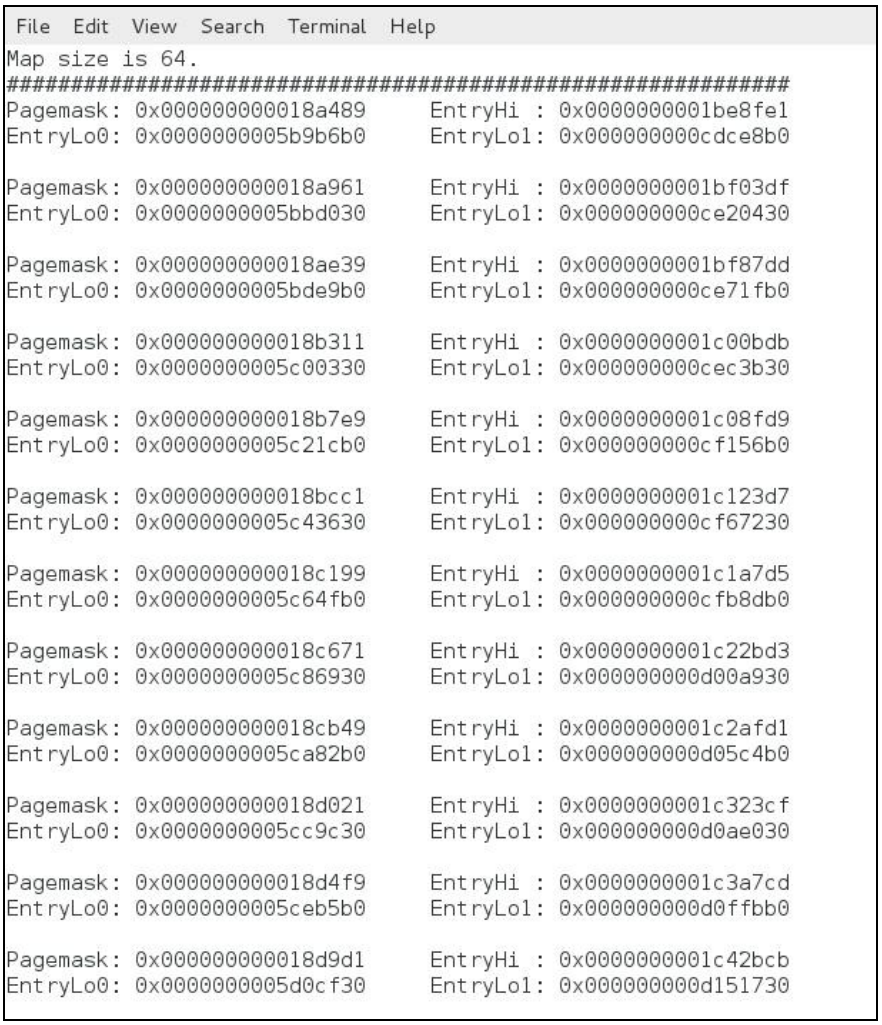
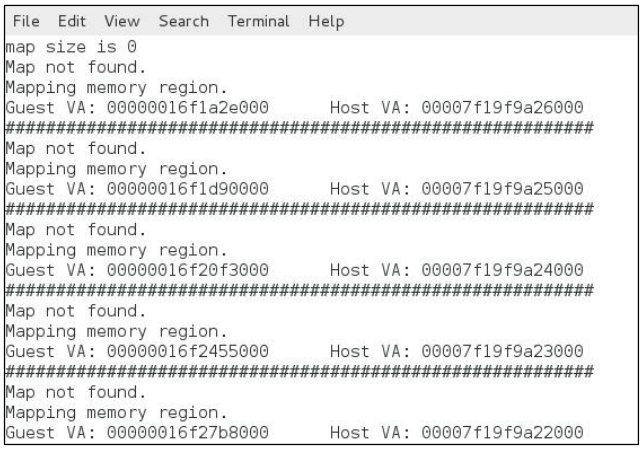


Figure 5: TLB entries in TLB table.

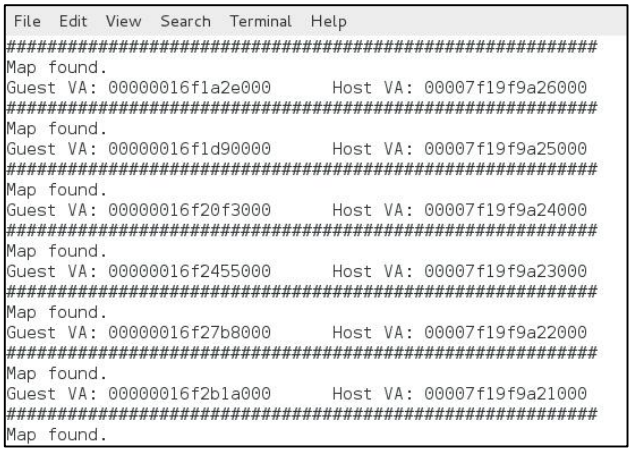
4.2.4. Output of CIU Testing

Artificial UART registers are read to test the code. UART registers were set to see the effect on the 10,11 and 12 bits of `cause` register. If Interrupt ID (IID) field of IIR is 1 than there is no pending interrupt request. Otherwise, it represent the ID of pending interrupt. In actual system `enable` register is set by the system but here we are setting it explicitly. The `cause` register is initialized with garbage value every time because CIU will only change the 9, 10 and 11 bits of `cause` register.

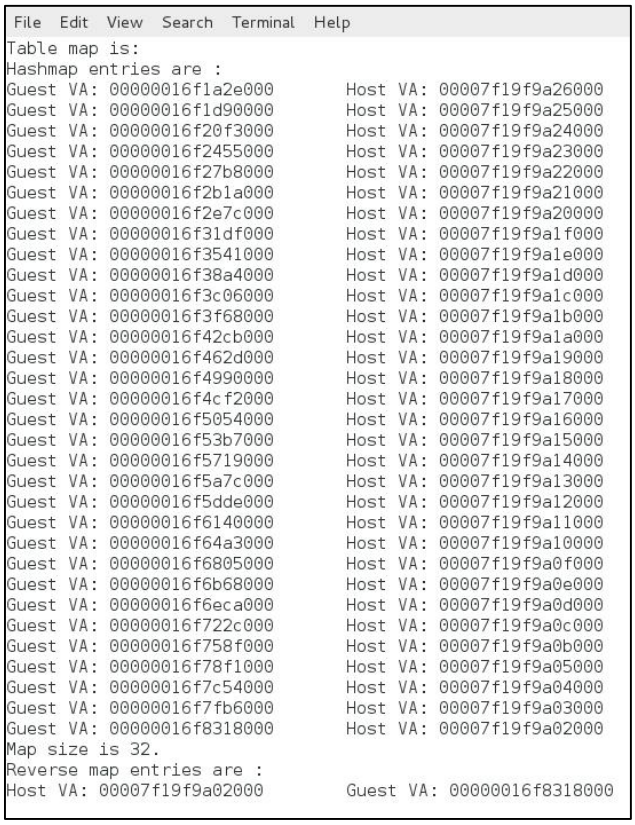
In source code of figure 7, `mio_uart0` IIR register is set to 6 to show that "Receiver line status" interrupt is present. Similarly, `mio_uart1` IIR register is set to 1 to represent no interrupt. Only core0's `enable` register is set. And all the other cores have disabled the hardware interrupts. Output for core 0 in Figure 7 shows that initially `cause` register is initialized by a garbage value.



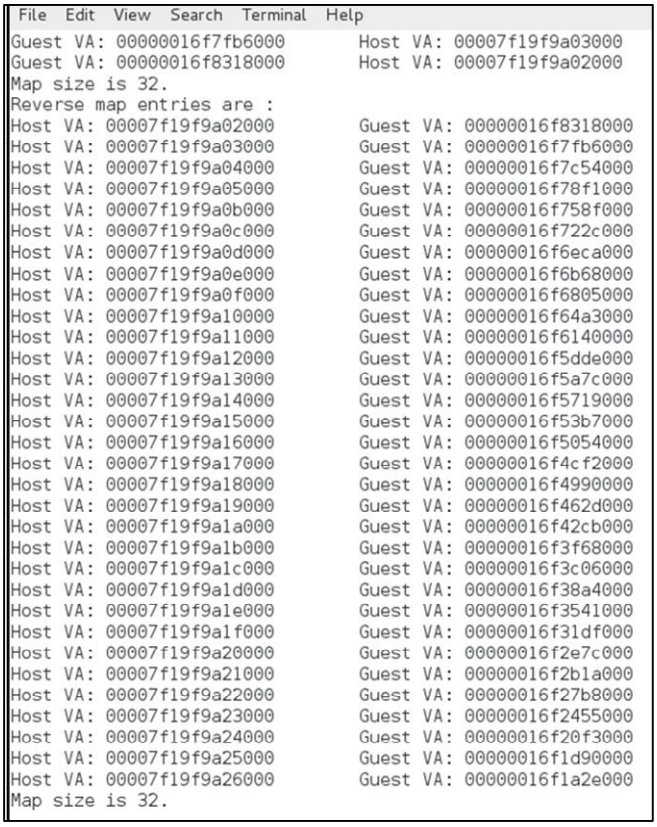
(a)



(b)



(c)



(d)

Figure 6: Output of TLB and Page Table testing. Searching entries in (a) empty page table, and (b) page table having valid entries. (c) Whole Page table with valid translation. (d) : Reverse mapping of page table.

The `summary` register's 34th bit (`uart 0`) is set, making it `400000000`. Corresponding 34th bit in `enable` register is also set, which means that the 9th bit of cause will be set. The enable register for 10th and 11th bit are zero, so `cause` bits would be cleared.

Initially the `xxxxxxxx6ac8` is changed to `xxxxxxxx66c8` by setting 9th bit and clearing 10th and 11th bit. For core 1, as none of the `enable` registers

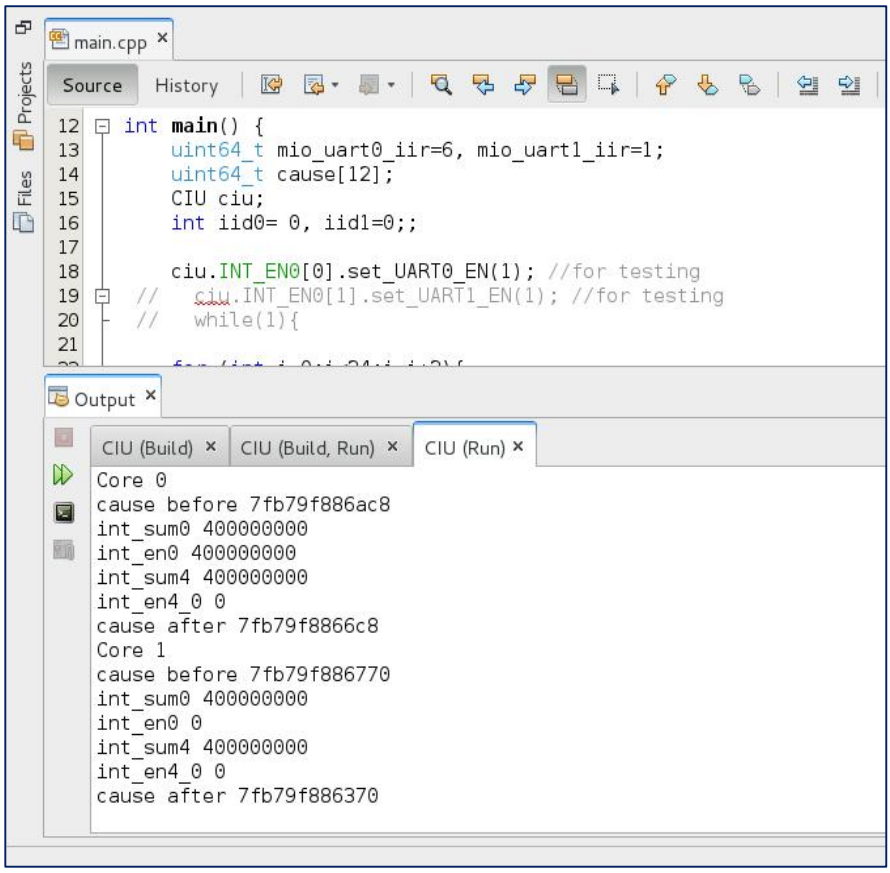


Figure 7. Output of CIU. No pending interrupt on core 1.

are configured so three bits would be cleared i.e. xxxxxxxx6770 changes to xxxxxxxx6370.

In figure 8, uart0 has no interrupt and uart1 is receiving an interrupt with id 6. For core0, bit 9 and 10 of enable register is set and cause register is get initialized with garbage value. As summary register shows the presence of an interrupt and bit 35 is set, it means that uart1 interrupt is present. Its enable register should also be set for uart1, in order to pass on the pending interrupt. Hence, bit 9 11 will be cleared and bit 10 will be set for core 0 i.e. xxxxxxxxdac8 changes to xxxxxxxxcac8 in the output. For core1, nothing is enabled so all three bits would be cleared i.e. xxxxxxxxd770 changes to xxxxxxxxc370.

4.2.5. Output on Hypervisor Console

During execution, hypervisor makes a call to the code written for console I/O. On console attachment, the binaries, executing within hypervisor, can start printing on the hypervisor console. To validate virtual execution of binaries, hypervisor console output (e.g. shown in Figure 9) was compared with that of real host system console.

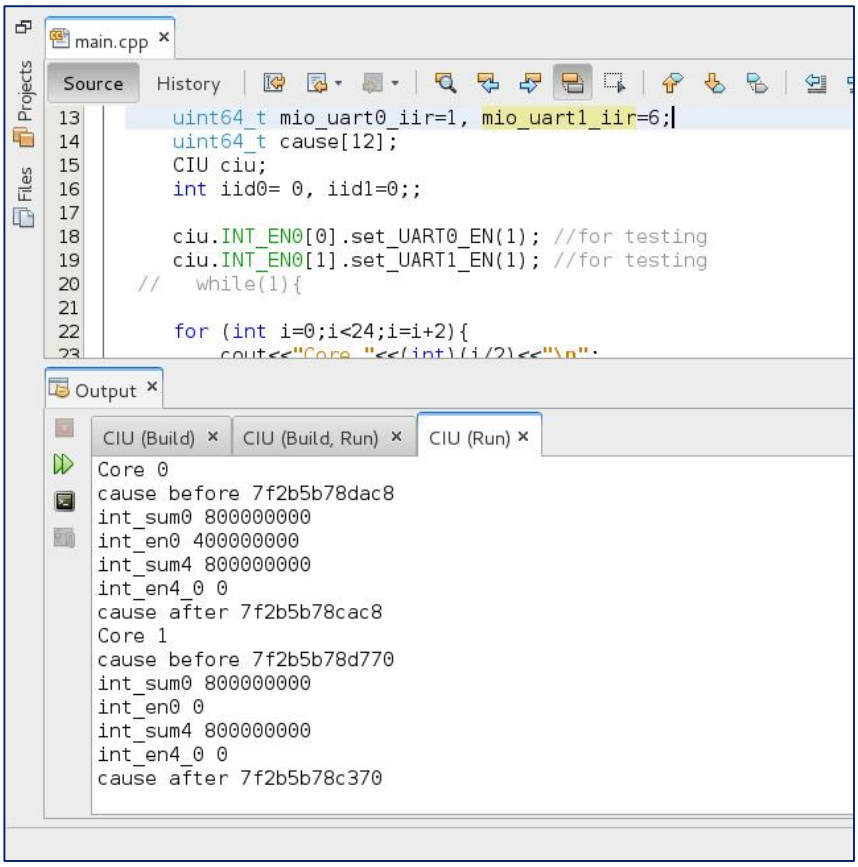


Figure 8. Output of CIU. No pending interrupt on core 0.

```
root@octeon:/home/Asad_data/hypervisor2-clone# ./dist/Debug/GNU-Linux-x86/hypervisor2-clone

        **********************   Main():Start Here  *************************
        Loaded binary address...: 0x000000004b883000
REGION_ADDR...: 0x000000002ba4c000 REGION_SIZE = 0x80
REGION_ADDR...: 0x000000002ba4d000 REGION_SIZE = 0x200
REGION_ADDR...: 0x000000004bc83000 REGION_SIZE = 0x10000000
REGION_ADDR...: 0x000000005bc83000 REGION_SIZE = 0x80000
REGION_ADDR...: 0x000000005bd03000 REGION_SIZE = 0x80000
                    :
                    :
U-Boot 1.1.1 (Development build, svnversion: u-boot:exported, exec:exported) (Bu

BIST check passed.
Warning: Board descriptor tuple not found in eeprom, using defaults
EBH5610 board revision major:1, minor:0, serial #: unknown
OCTEON CN56XX-NSP pass 2.0, Core clock: 0 MHz, DDR clock: 0 MHz (0 Mhz data rate
DRAM:  1024 MB
Clearing DRAM........ done
Flash boot bus region not enabled, skipping NOR flash config
                    :
                    :
```

Figure 9: Output on hypervisor console.

5.Impact on Project Progress

The project is on track and making good progress. During development, we occasionally find dependences between the milestones that leads to the partial development of some future milestones, in addition to the current milestones. This thing has no adverse impact on project progress but advantageous in true sense.

References

[1] Cavium Networks OCTEON Plus CN54/5/6/7XX, Hardware Reference Manual CN54/5/6/7XX-HM-2.4E, January 2009, chapter 14 (CIU).

~*~*~*~