

Development of Type-2 Hypervisor for MIPS64 Based Systems

December 15

2013

[3rd Deliverable]

This document is version 3 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. The 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



Table of Contents

Table of Contents	2
Table of Figures	3
1. Project Description	4
2. High Level Design	4
3. Development Strategy	5
4. Challenges and Solutions	5
5. Testing Infrastructure	7
5.1. Test Cases	7
5.1.1. Matching system states	7
5.1.2. Execution path	7
5.1.3. Comparing Console Output	8
5.1.4. Progress	8
5.2. Memory Management Unit (MMU)	8
5.2.1. GVA to GPA Translation	9
5.2.2. GPA to HVA Translation	9
5.2.3. Page Table	9
5.2.4. Translation Lookaside Buffer (TLB)	10
5.3. Central Interrupt Unit (CIU)	10
5.4. Test Results	12
5.4.1. Output of System State Matching Test	13
5.4.2. Output of Execution Path Test	13
5.4.3. Output of TLB Testing	13
5.4.4. Output of CIU Testing	15
5.4.5. Output on Hypervisor Console	18
6. Impact on Project Progress	18

Table of Figures

Figure 1: Multithreaded Design of Type-2 hypervisor.	5
Figure 2: Central Interrupt Unit. (a) Interrupt distribution from external devices to core. (b) Internal working of CIU. Inward arrow comes from external devices and outward arrow goes to all cores.	11
Figure 3: Memory mapping between Core and external devices.	12
Figure 4: Output of system state matching test.	13
Figure 5: Output of Execution Path Test.	14
Figure 6: Searching for random TLB entry.	15
Figure 7: TLB entries in TLB table.	16
Figure 8: 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.	17
Figure 9: Output of CIU. No pending interrupt on core 1.	18
Figure 10: Output of CIU. No pending interrupt on core 0.	19
Figure 11: Output on hypervisor console.	19

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. In 3rd deliverable, dynamic code patching is augmented by implementing cases where one sensitive instruction is replaced by more than one instructions.

2. High Level Design

Type-2 hypervisor behaves like an ordinary Linux process that could be scheduled by host operating system. However, this process has to present a holistic view of virtual hardware for guest operating system(s) to run on it. Virtual hardware consists of software representations of CPU cores, memory and peripheral devices. In real hardware, CPU cores and devices work concurrently and could be considered as processes or threads in software representation. Multiprocessing requires inter-process communication (IPC) but multithreading could be implemented using the shared address space. Each one has its own pros and cons. We selected multithreaded design for our hypervisor, as shown in Figure 1. It shows that each core and device is a

separate thread. Central interrupt unit (CIU) is another thread that dispatches pending interrupts to the cores using mapped memory.

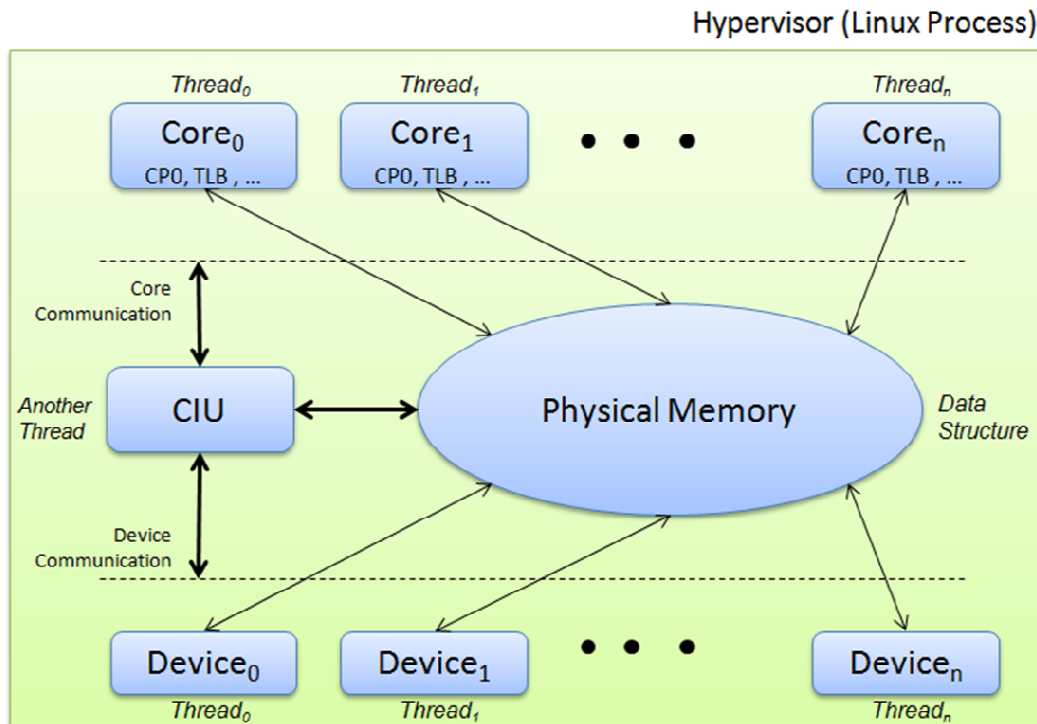


Figure 1: Multithreaded Design of Type-2 hypervisor.

3. 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.

4. 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 3

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.

5. 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.

5.1. 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.

5.1.1. 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.

5.1.2. 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".

5.1.3. 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.

5.1.4. 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.

5.2. Memory Management Unit (MMU)

The purpose of memory management unit is to translate virtual addresses to physical addresses. For virtual address translation, some rules are already defined by physical hardware and we implemented these rules in software to provide the virtualization of MMU used by guest operating system(s). In case of hypervisor, it is used to translate guest virtual address (GVA) to host virtual address (HVA). To translate GVA to guest physical address (GPA), we use same method as used by the hardware. For translation of GPA to HVA, we use hashmap to store information of all regions mapped in host virtual address space.

5.2.1. GVA to GPA Translation

MIPS64 architecture supports both 32-bit and 64-bit Addressing modes. In 32-bit addressing mode, address segment is defined by upper 3 bits (i.e. bits 32-29) of virtual address. If these bits are 100 then it is `kseg0` region. It is directly mapped to physical memory. If these bits are 101, address is from `kseg1` region and this is also directly mapped to physical memory. In both previous cases, lower 20 bits represent physical address. For 110, region is `ksseg`. This is not directly mapped and we have to search for it in TLB for address translation. For 111, region is `kseg3` which is not directly mapped and we have to search TLB for valid entry to translate the address. If these bits are 0xx then it is `useg`. Translation for `useg` is slightly different. If `ERL` bit of `status` register of `CP0` is set then `useg` is directly mapped to physical memory. If `ERL` bit is not set then we have to check TLB to get physical address.

In 64-bit addressing mode, address segment is defined by upper 2 bits (i.e. bits 63-62) of virtual address. If these bits are 10 then this is `xkphys` region which is directly mapped to physical memory or I/O devices. If 49th bit of virtual address is 0 then it is memory access and lower 29 bits represent physical address of memory. If 49th bit is 1 then it is I/O address and data is load/store from respective device. If these bits are 11 then it is `xkseg` region which isn't directly mapped and we have to search TLB for valid address translation. For 01, region is `xsseg` which is also to be searched in TLB for translation. For 00, region is `xuseg`. If `ERL` bit of `status` register of `CP0` is set then it is directly mapped otherwise TLB translation would be required.

5.2.2. GPA to HVA Translation

All physical memory regions of a machine are mapped in virtual address space of hypervisor. Once we get the valid translation for GVA, we have to translate that physical address to HVA in order to access valid data. After getting valid physical address, we found the memory region or I/O device to which it belongs. We simply find HVA for required memory region or I/O device using hashmap. Once we get a valid GVA-to-HVA translation, we can simply execute the respective instruction.

5.2.3. Page Table

In MIPS no physical page table is provided by hardware and page table is solely managed by operating system. Hence, there is no need to implement page table.

5.2.4. 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 GVA to HVA. 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.

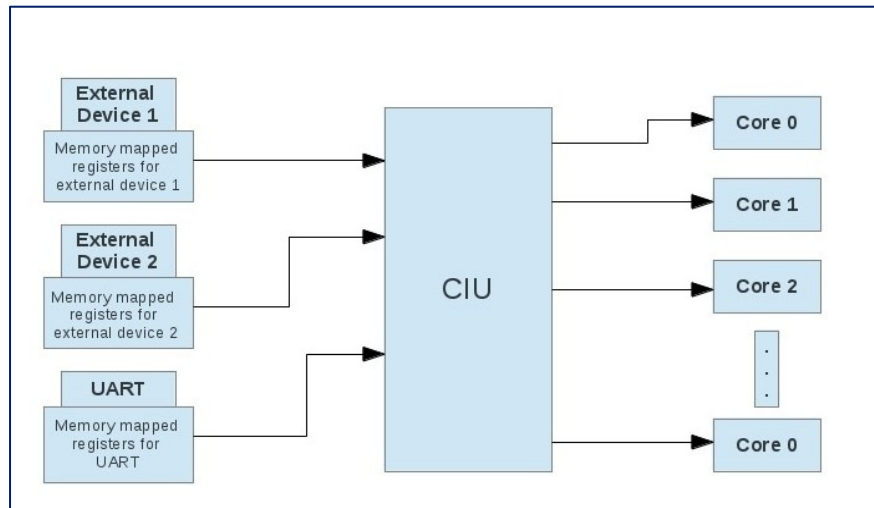
On TLB miss, page table functions are called and GVA is searched in the page table. If found, corresponding HVA is returned, otherwise a new memory region is allocated using `mmap()` and its address is returned. Current implementation does not impose any restriction on memory allocation (i.e. it will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

5.3. Central Interrupt Unit (CIU)

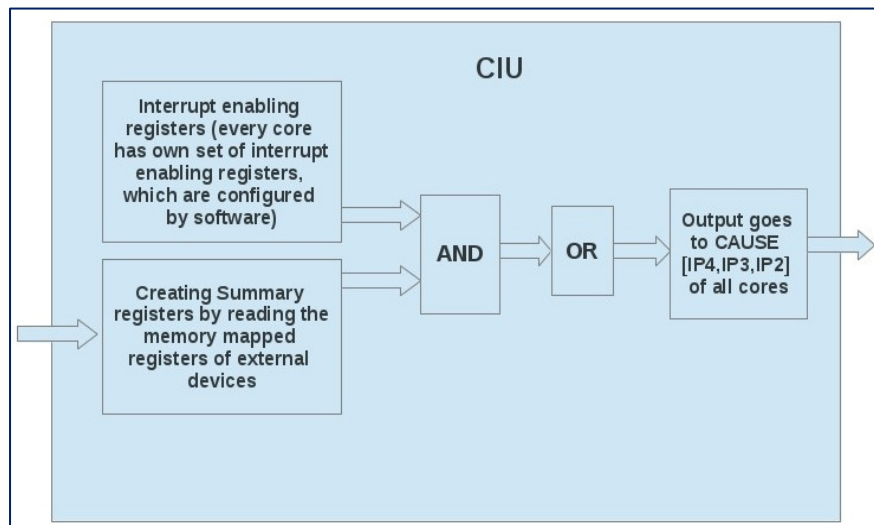
CIU is responsible for dispatching interrupt requests (coming) from external devices to a particular core. CIU is discussed here in context of our test bed i.e. Cavium Networks OCTEON Plus CN57XX evaluation board [1]. CIU distributes a total of 37 interrupts i.e. 3 per core plus 1 for PCIe. Three interrupts 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>`.

Interaction of CIU, external devices and cores is shown in Figure 2 (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 2 (b) shows a simplest description of the internal working of CIU, where interrupt identification/handling is done in software.

We have implemented a simplest abstraction of CIU. It has been integrated in a copy of main hypervisor code and works as a separate thread (see Figure 1). CIU is only reading CP0's `cause` register. As UART is not fully developed yet, UART's memory mapped registers are artificial (for the time being). UART



(a)



(b)

Figure 2: Central Interrupt Unit. (a) Interrupt distribution from external devices to core. (b) Internal working of CIU. Inward arrow comes from external devices and outward arrow goes to all cores.

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 then "AND" with their enable registers to set or clear cause register's bit 10, 11 and 12.

In integrated code, shared memory regions are defined for CIU to work with other components of virtual board (see Figure 1). Figure 3 shows these shared memory regions for core0, CIU and a single device i.e. UART. Region overlapping and dotted lines represent the accessibility and access mode of registers, respectively. For example, CP0 Cause register belongs to core0, CIU can access it but UART cannot. As Cause register belongs to core0, it can be read-written by core0 but it is read-only for CIU. IIR register of UART is read-only for CIU and Core0, hence it is at the intersection of three regions and has dotted boundary. CIU's summary registers are read-only for core0, hence dotted and at the intersection of two regions. As CIU's enable register is readable and writable for core0 and CIU, it has solid boundary and lies in overlapped region.

5.4. Test Results

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

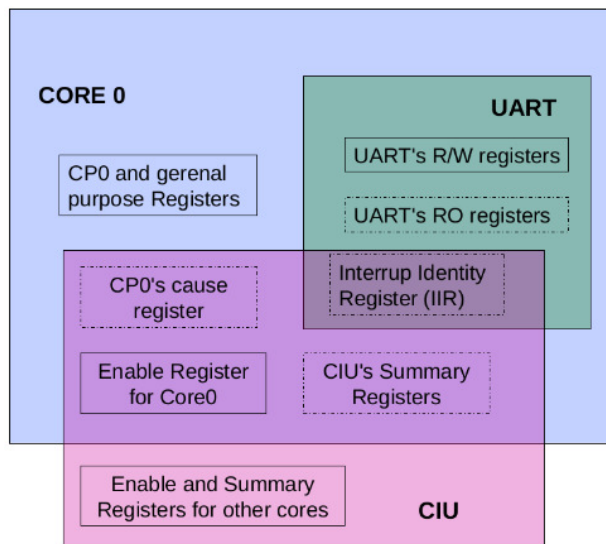


Figure 3: Memory mapping between Core and external devices.

5.4.1. Output of System State Matching Test

We trap at every instruction to create a state-file. This state-file is matched with QEMU log state-file to see if any register contains different contents. Mismatches are written in other file as shown in Figure 4.

5.4.2. 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 5.

5.4.3. 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 place this entry at the

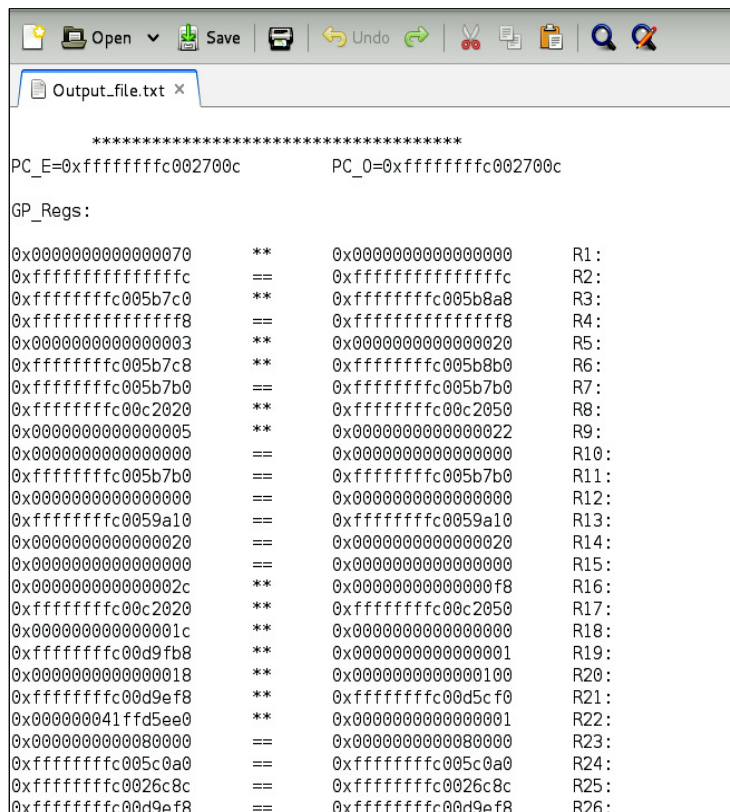


Figure 4: Output of system state matching test.

```
38 c000ae5c matched
39 c000ae60 matched
40 c000ae64 matched
41 c000ae68 matched
42 c000ae6c matched
43 c000ae70 matched
44 c000ae74 matched
45 c000ae78 matched
46 c000ae7c matched
47 c000ae80 matched
48 c000ae84 matched
49 c000ae88 matched
50 c000ae8c matched
51 c000ae90 matched
52 c000ae94 matched
53 c000ae98 matched
54 c000ae9c matched
55 c000aea0 matched
56 c000aea4 matched
57 c000aea8 matched
58 c000aeac matched
59 c000aeb0 matched
60 c000aeb4 matched
61 c000aeb8 matched
62 c000aebc matched
63 c000aec0 matched
64 c000aec4 matched
65 c000aec8 matched
66 c000aecc matched
67 hypervisor c000aed0 mismatched qemu pc c000b074
```

Figure 5: Output of Execution Path Test.

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 6.

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.

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 7. To test page table, a random GVA 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 8 (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 8 (b). Then whole page table is printed in Figure 8 (c) and reverse page table, shown in Figure 8 (d), is also managed to use for future testing of hypervisor.

```

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 6: Searching for random TLB entry.

5.4.4. 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 9, `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

```

File Edit View Search Terminal Help
Map size is 64.
#####
Pagemask: 0x000000000018a489      EntryHi : 0x0000000001be8fe1
EntryLo0: 0x0000000005b9b6b0      EntryLo1: 0x000000000cdce8b0

Pagemask: 0x000000000018a961      EntryHi : 0x0000000001bf03df
EntryLo0: 0x0000000005bbd030      EntryLo1: 0x000000000ce20430

Pagemask: 0x000000000018ae39      EntryHi : 0x0000000001bf87dd
EntryLo0: 0x0000000005bde9b0      EntryLo1: 0x000000000ce71fb0

Pagemask: 0x000000000018b311      EntryHi : 0x0000000001c00bdb
EntryLo0: 0x0000000005c00330      EntryLo1: 0x000000000cec3b30

Pagemask: 0x000000000018b7e9      EntryHi : 0x0000000001c08fd9
EntryLo0: 0x0000000005c21cb0      EntryLo1: 0x000000000cf156b0

Pagemask: 0x000000000018bcc1      EntryHi : 0x0000000001c123d7
EntryLo0: 0x0000000005c43630      EntryLo1: 0x000000000cf67230

Pagemask: 0x000000000018c199      EntryHi : 0x0000000001c1a7d5
EntryLo0: 0x0000000005c64fb0      EntryLo1: 0x000000000cfb8db0

Pagemask: 0x000000000018c671      EntryHi : 0x0000000001c22bd3
EntryLo0: 0x0000000005c86930      EntryLo1: 0x000000000d00a930

Pagemask: 0x000000000018cb49      EntryHi : 0x0000000001c2afd1
EntryLo0: 0x0000000005ca82b0      EntryLo1: 0x000000000d05c4b0

Pagemask: 0x000000000018d021      EntryHi : 0x0000000001c323cf
EntryLo0: 0x0000000005cc9c30      EntryLo1: 0x000000000d0ae030

Pagemask: 0x000000000018d4f9      EntryHi : 0x0000000001c3a7cd
EntryLo0: 0x0000000005ceb5b0      EntryLo1: 0x000000000d0ffbb0

Pagemask: 0x000000000018d9d1      EntryHi : 0x0000000001c42bcb
EntryLo0: 0x0000000005d0cf30      EntryLo1: 0x000000000d151730

```

Figure 7: TLB entries in TLB table.

for core 0 in Figure 9 shows that initially cause register is initialized by a garbage value.

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

In Figure 10, 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 and 11 will be cleared and bit 10 will be set for core 0 i.e. xxxxxxxxdac8 changes to xxxxxxxxcac8 in the output.


```
File Edit View Search Terminal Help
map size is 0
Map not found.
Mapping memory region.
Guest VA: 00000016f1a2e000      Host VA: 00007f19f9a26000
#####
Map not found.
Mapping memory region.
Guest VA: 00000016f1d90000      Host VA: 00007f19f9a25000
#####
Map not found.
Mapping memory region.
Guest VA: 00000016f20f3000      Host VA: 00007f19f9a24000
#####
Map not found.
Mapping memory region.
Guest VA: 00000016f2455000      Host VA: 00007f19f9a23000
#####
Map not found.
Mapping memory region.
Guest VA: 00000016f27b8000      Host VA: 00007f19f9a22000
#####
Map not found.
Mapping memory region.
Guest VA: 00000016f2b1a000      Host VA: 00007f19f9a21000
#####
Map not found.
Mapping memory region.
Guest VA: 00000016f27b8000      Host VA: 00007f19f9a22000
```

(a)

```
File Edit View Search Terminal Help
#####
Map found.
Guest VA: 00000016f1a2e000      Host VA: 00007f19f9a26000
#####
Map found.
Guest VA: 00000016f1d90000      Host VA: 00007f19f9a25000
#####
Map found.
Guest VA: 00000016f20f3000      Host VA: 00007f19f9a24000
#####
Map found.
Guest VA: 00000016f2455000      Host VA: 00007f19f9a23000
#####
Map found.
Guest VA: 00000016f27b8000      Host VA: 00007f19f9a22000
#####
Map found.
Guest VA: 00000016f2b1a000      Host VA: 00007f19f9a21000
#####
Map found.
```

(b)

```
File Edit View Search Terminal Help
Table map is:
Hashmap entries are :
Guest VA: 00000016f1a2e000      Host VA: 00007f19f9a26000
Guest VA: 00000016f1d90000      Host VA: 00007f19f9a25000
Guest VA: 00000016f20f3000      Host VA: 00007f19f9a24000
Guest VA: 00000016f2455000      Host VA: 00007f19f9a23000
Guest VA: 00000016f27b8000      Host VA: 00007f19f9a22000
Guest VA: 00000016f2b1a000      Host VA: 00007f19f9a21000
Guest VA: 00000016f2e7c000      Host VA: 00007f19f9a20000
Guest VA: 00000016f31df000      Host VA: 00007f19f9a1f000
Guest VA: 00000016f3541000      Host VA: 00007f19f9a1e000
Guest VA: 00000016f38a4000      Host VA: 00007f19f9a1d000
Guest VA: 00000016f3c06000      Host VA: 00007f19f9a1c000
Guest VA: 00000016f3f68000      Host VA: 00007f19f9a1b000
Guest VA: 00000016f42cb000      Host VA: 00007f19f9a1a000
Guest VA: 00000016f462d000      Host VA: 00007f19f9a19000
Guest VA: 00000016f4990000      Host VA: 00007f19f9a18000
Guest VA: 00000016f4cf2000      Host VA: 00007f19f9a17000
Guest VA: 00000016f5054000      Host VA: 00007f19f9a16000
Guest VA: 00000016f53b7000      Host VA: 00007f19f9a15000
Guest VA: 00000016f5719000      Host VA: 00007f19f9a14000
Guest VA: 00000016f5a7c000      Host VA: 00007f19f9a13000
Guest VA: 00000016f5dde000      Host VA: 00007f19f9a12000
Guest VA: 00000016f6140000      Host VA: 00007f19f9a11000
Guest VA: 00000016f64a3000      Host VA: 00007f19f9a10000
Guest VA: 00000016f6805000      Host VA: 00007f19f9a0f000
Guest VA: 00000016f6b68000      Host VA: 00007f19f9a0e000
Guest VA: 00000016f6eca000      Host VA: 00007f19f9a0d000
Guest VA: 00000016f722c000      Host VA: 00007f19f9a0c000
Guest VA: 00000016f758f000      Host VA: 00007f19f9a0b000
Guest VA: 00000016f78f1000      Host VA: 00007f19f9a05000
Guest VA: 00000016f7c54000      Host VA: 00007f19f9a04000
Guest VA: 00000016f7fb6000      Host VA: 00007f19f9a03000
Guest VA: 00000016f8318000      Host VA: 00007f19f9a02000
Map size is 32.
Reverse map entries are :
Host VA: 00007f19f9a02000      Guest VA: 00000016f8318000
```

(c)

```
File Edit View Search Terminal Help
Guest VA: 00000016f7fb6000      Host VA: 00007f19f9a03000
Guest VA: 00000016f8318000      Host VA: 00007f19f9a02000
Map size is 32.
Reverse map entries are :
Host VA: 00007f19f9a02000      Guest VA: 00000016f8318000
Host VA: 00007f19f9a03000      Guest VA: 00000016f7fb6000
Host VA: 00007f19f9a04000      Guest VA: 00000016f7c54000
Host VA: 00007f19f9a05000      Guest VA: 00000016f78f1000
Host VA: 00007f19f9a0b000      Guest VA: 00000016f758f000
Host VA: 00007f19f9a0c000      Guest VA: 00000016f722c000
Host VA: 00007f19f9a0d000      Guest VA: 00000016f6eca000
Host VA: 00007f19f9a0e000      Guest VA: 00000016f6b68000
Host VA: 00007f19f9a0f000      Guest VA: 00000016f6805000
Host VA: 00007f19f9a10000      Guest VA: 00000016f64a3000
Host VA: 00007f19f9a11000      Guest VA: 00000016f6140000
Host VA: 00007f19f9a12000      Guest VA: 00000016f5dde000
Host VA: 00007f19f9a13000      Guest VA: 00000016f5a7c000
Host VA: 00007f19f9a14000      Guest VA: 00000016f5719000
Host VA: 00007f19f9a15000      Guest VA: 00000016f53b7000
Host VA: 00007f19f9a16000      Guest VA: 00000016f5054000
Host VA: 00007f19f9a17000      Guest VA: 00000016f4cf2000
Host VA: 00007f19f9a18000      Guest VA: 00000016f4990000
Host VA: 00007f19f9a19000      Guest VA: 00000016f462d000
Host VA: 00007f19f9a1a000      Guest VA: 00000016f42cb000
Host VA: 00007f19f9a1b000      Guest VA: 00000016f3f68000
Host VA: 00007f19f9a1c000      Guest VA: 00000016f3c06000
Host VA: 00007f19f9a1d000      Guest VA: 00000016f38a4000
Host VA: 00007f19f9a1e000      Guest VA: 00000016f3541000
Host VA: 00007f19f9a1f000      Guest VA: 00000016f31df000
Host VA: 00007f19f9a20000      Guest VA: 00000016f2e7c000
Host VA: 00007f19f9a21000      Guest VA: 00000016f2b1a000
Host VA: 00007f19f9a22000      Guest VA: 00000016f27b8000
Host VA: 00007f19f9a23000      Guest VA: 00000016f2455000
Host VA: 00007f19f9a24000      Guest VA: 00000016f20f3000
Host VA: 00007f19f9a25000      Guest VA: 00000016f1d90000
Host VA: 00007f19f9a26000      Guest VA: 00000016f1a2e000
Map size is 32.
```

(d)

Figure 8: 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.

For core1, nothing is enabled so all three bits would be cleared i.e. xxxxxxxxd770 changes to xxxxxxxxc370.

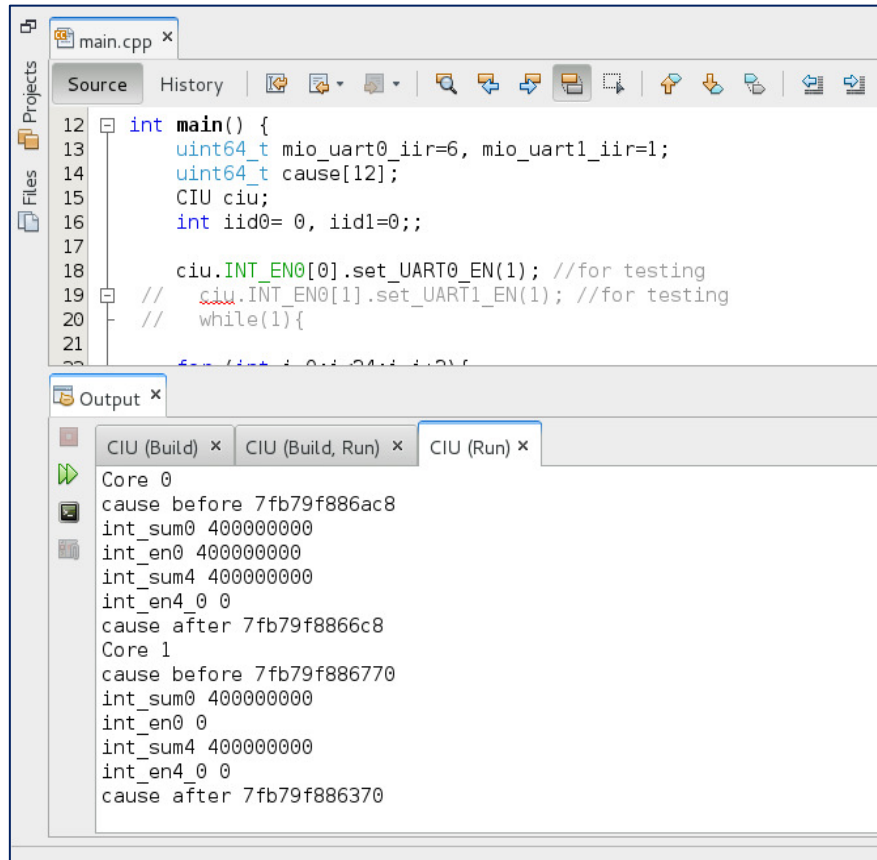


Figure 9: Output of CIU. No pending interrupt on core 1.

5.4.5. 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 11) was compared with that of real host system console.

6. 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.

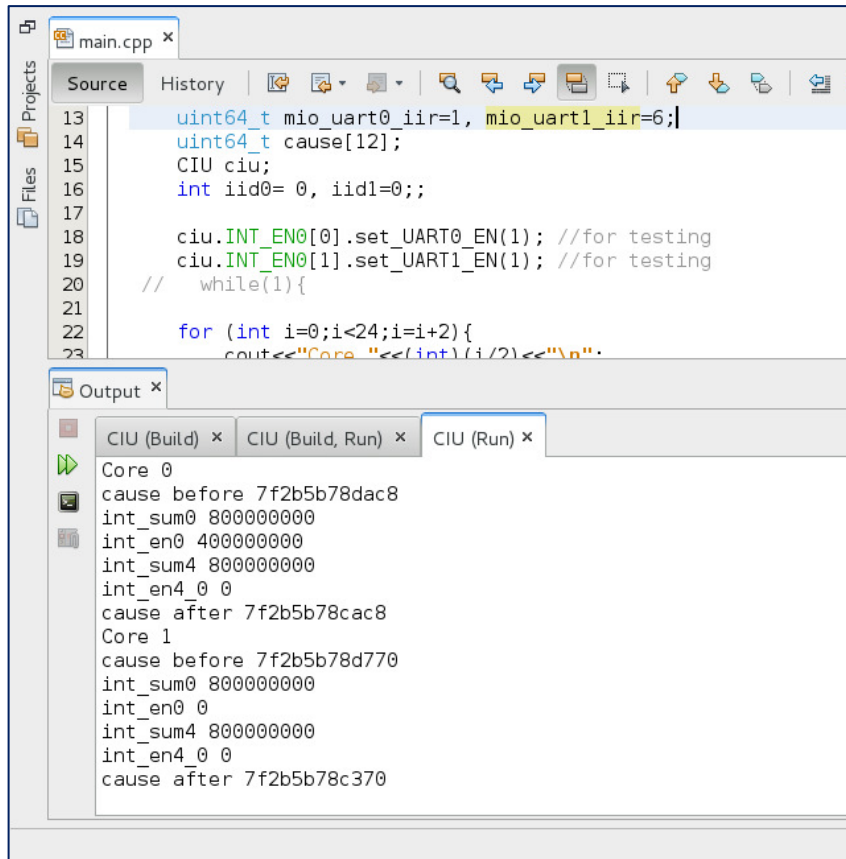


Figure 10: Output of CIU. No pending interrupt on core 0.

```

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 11: Output on hypervisor console.

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).

~*~*~*~