Debug R3000 With GXemul

Launch GXemul In DEBUG Mode

```
root@os-lab: ~/os-lab$ /OSLAB/gxemul -E testmips -C R3000 -M 64 ./gxemul/vmlinux -V
GXemul> help
Available commands:
  allsettings                   show all settings
  breakpoint ...                manipulate breakpoints
  continue                      continue execution
  device ...                    show info about (or manipulate) devices
  dump [addr [endaddr]]         dump memory contents in hex and ASCII
  emuls                         print a summary of all current emuls
  focus x[,y[,z]]               changes focus to cpu x, machine x, emul z
  help                          print this help message
  itrace                        toggle instruction_trace on or off
  lookup name|addr              lookup a symbol by name or address
  machine                       print a summary of the current machine
  ninstrs [on|off]              toggle (set or unset) show_nr_of_instructions
  pause cpuid                   pause (or unpause) a CPU
  print expr                    evaluate an expression without side-effects
  put [b|h|w|d|q] addr, data    modify emulated memory contents
  quiet [on|off]                toggle quiet_mode on or off
  quit                          quit the emulator
  reg [cpuid][,c]               show GPRs (or coprocessor c's registers)
  step [n]                      single-step one (or n) instruction(s)
  tlbdump [cpuid][,r]           dump TLB contents (add ',r' for raw data)
  trace [on|off]                toggle show_trace_tree on or off
  unassemble [addr [endaddr]]   dump memory contents as instructions
  version                       print version information
  x = expr                      generic assignment

In generic assignments, x must be a register or other writable settings
variable, and expr can contain registers/settings, numeric values, or symbol
names, in combination with parenthesis and + - * / & % ^ | operators.
In case there are multiple matches (i.e. a symbol that has the same name as a
register), you may add a prefix character as a hint: '#' for registers, '@'
for symbols, and '$' for numeric values. Use 0x for hexadecimal values.
```

下面将介绍几个常用的功能。

Breakpoint, Step & Unassemble, Dump

GXemul 提供了断点、单步执行（指令级别）的调试功能；除此之外，GXemul 还提供了反汇编、内存导出等功能。

Add Breakpoint

如下，启动模拟器后，在内部控制台中输入 `breakpoint add page_insert` ，即可为 `page_insert` 函数增加断点。随后使用 `c` （continue）命令让模拟器继续运行，模拟器将运行至下一个断点处。

以 `page_insert` 为例：

```
GXemul> breakpoint add page_insert
  0: 0x80014240 (page_insert)
GXemul> c
main.c: main is start ...
init.c: mips_init() is called
Physical memory: 65536K available, base = 65536K, extended = 0K
to memory 80401000 for struct page directory.
to memory 80431000 for struct Pages.
pmap.c:  mips vm init success
<page_insert>
80014240: 27bdffd0      addiu   sp,sp,-48
BREAKPOINT: pc = 0x80014240
(The instruction has not yet executed.)
```

Step

如下，使用 `step` 即可让模拟器执行 `1` 条汇编指令（基本指令）。

```
GXemul> step
```

进一步地，使用 `step 100` 即可让模拟器执行 `100` 条汇编指令（基本指令）。

```
GXemul> step 100
```

仍以上面的 `page_insert` 为例：

```
GXemul> breakpoint add page_insert
  0: 0x80014240 (page_insert)
GXemul> c
...
<page_insert>
80014240: 27bdffd0      addiu   sp,sp,-48
BREAKPOINT: pc = 0x80014240
(The instruction has not yet executed.)
GXemul> step 20
<page_insert>
80014240: 27bdffd0      addiu   sp,sp,-48
80014244: afbf0028      sw      ra,40(sp)       [0x803fff58]
80014248: afb30024      sw      s3,36(sp)       [0x803fff54]
8001424c: afb20020      sw      s2,32(sp)       [0x803fff50]
80014250: afb1001c      sw      s1,28(sp)       [0x803fff4c]
80014254: afb00018      sw      s0,24(sp)       [0x803fff48]
80014258: 00808821      move    s1,a0
8001425c: 00a09021      move    s2,a1
80014260: 00c08021      move    s0,a2
80014264: 34f30200      ori     s3,a3,0x0200
80014268: 00c02821      move    a1,a2
8001426c: 00003021      move    a2,zr
80014270: 0c005045      jal     0x80014114      <pgdir_walk>
80014274: 27a70010 (d)  addiu   a3,sp,16
<pgdir_walk(0x83fff000,0x7f3fd000,0,0x803fff40,..)>
<pgdir_walk>
80014114: 27bdffd8      addiu   sp,sp,-40
80014118: afbf0024      sw      ra,36(sp)       [0x803fff2c]
8001411c: afb20020      sw      s2,32(sp)       [0x803fff28]
80014120: afb1001c      sw      s1,28(sp)       [0x803fff24]
80014124: afb00018      sw      s0,24(sp)       [0x803fff20]
80014128: 00a08821      move    s1,a1
8001412c: 00051582      srl     v0,a1,22
```

Unassemble

如下，使用 `unassemble` 命令，导出某一个地址后续（或附近）的汇编指令序列。

```
GXemul> unassemble
80014130: 00021080      sll     v0,v0,2
80014134: 00448021      addu    s0,v0,a0
80014138: 8e020000      lw      v0,0(s0)
8001413c: 00000000      nop
80014140: 30420200      andi    v0,v0,0x0200
80014144: 14400022      bne     zr,v0,0x800141d0        <pgdir_walk+0xbc>
80014148: 00e09021      move    s2,a3
8001414c: 10c0001d      beq     zr,a2,0x800141c4        <pgdir_walk+0xb0>
80014150: 00000000      nop
80014154: 0c004ff8      jal     0x80013fe0      <page_alloc>
80014158: 27a40010      addiu   a0,sp,16
8001415c: 2403fffc      addiu   v1,zr,-4
80014160: 10430031      beq     v1,v0,0x80014228        <pgdir_walk+0x114>
80014164: 2402fffc      addiu   v0,zr,-4
80014168: 8fa40010      lw      a0,16(sp)
8001416c: 3c028002      lui     v0,0x8002
80014170: 8c43928c      lw      v1,-28020(v0)
80014174: 00000000      nop
80014178: 00831823      subu    v1,a0,v1
8001417c: 00031883      sra     v1,v1,2
```

Dump Data

如下，使用 `dump` 命令，导出某一个地址后续（或附近）的内存信息。下面以查看 `curenv` 的值以及其指向的进程控制块为例：

```
GXemul> dump curenv
0x800167a0           804320e8 00000003 00000000      .C .........
0x800167b0  00000002 804321d0 00000000 00000000  .....C!.........
0x800167c0  80432000 00000001 00000000 00000000  .C .............
0x800167d0  00000000 00000000 00000000 00000000  ................
0x800167e0  00000000 00000000 00000000 00000000  ................
..........
```

根据 `curenv` 的定义，其类型为 `struct Env *`，因此 `0x800167a4` 中存储的是一个地址（指向一个 `struct Env`）。根据上述 `dump` 结果，可以知道这个全局指针指向了 `0x804320e8` 这一地址。再通过 `dump 0x804320e8` 即可找到当前 `curenv` 指向的 `struct Env`：

```
GXemul> dump 0x804320e8
0x804320e0                      00000000 00000000          ........
0x804320f0  00000001 00000025 00400920 0040d2c4  .......%.@. .@..
0x80432100  0040d2c4 7f3fdfcc 00000000 00000000  .@...?..........
0x80432110  00000000 00000000 00000000 00000000  ................
0x80432120  00000000 00000000 0040d2cc 00000007  .........@......
0x80432130  00000000 7f3fdfcc 00000000 00400920  .....?.......@.
0x80432140  00000000 00000000 00000000 00000000  ................
0x80432150  7f3fdb80 82000000 00000000 7f3fdb80  .?...........?..
0x80432160  00000000 004009d0 10081004 00000000  .....@..........
0x80432170  00000000 00408000 00001000 00400aa0  .....@.......@..
0x80432180  00400aa0 804321d0 800167b4 00000c01  .@...C!...g.....
0x80432190  00000000 00000001 83ff3000 03ff3000  ..........0...0.
0x804321a0  00000000 80019270 00000001 00000000  .......p........
0x804321b0  00000000 00000000 00000000 00000000  ................
0x804321c0  00000000 00000000 00000002 00000000  ................
0x804321d0  00000000 00000000 00000000 00000000  ................
0x804321e0  00000000 00000000                    ........
```

即可查看当前正运行的进程控制块的信息。类似地，可以查看任何全局变量的值，以及大部分内核数据结构的信息。

Dump Registers

通过 `reg` 命令导出通用寄存器的值：

```
GXemul> reg
cpu0:  pc = 80012620
cpu0:  hi = 00000000  lo = 00000000
cpu0:                 at = 00000000  v0 = 00000000  v1 = 82000000
cpu0:  a0 = 803fffb8  a1 = 82000000  a2 = 82000000  a3 = 00000000
cpu0:  t0 = 7f3fdf48  t1 = 80012a60  t2 = 80012b20  t3 = 00000000
cpu0:  t4 = 00000000  t5 = 00000000  t6 = 00000000  t7 = 00000000
cpu0:  s0 = 7f4002b8  s1 = 00001c03  s2 = 00410000  s3 = 00000003
cpu0:  s4 = 00000000  s5 = 00000000  s6 = 00000000  s7 = 00000000
cpu0:  t8 = 00000000  t9 = 00000000  k0 = 7f3fdf48  k1 = 803fffb8
cpu0:  gp = 00000000  sp = 803ffeec  fp = 00000000  ra = 80012b50
```

通过 `reg, 0` 命令导出协处理器 0（CP0）中寄存器的值：

```
GXemul> reg, 0
cpu0:    index=00000f00   random=00003700 entrylo0=03fbb600 entrylo1=00000000
cpu0:  context=00001008 pagemask=00001fff    wired=00000000  reserv7=00000000
cpu0: badvaddr=00402b2c    count=00000000  entryhi=00402080  compare=00000000
cpu0:   status=10081004    cause=00001020      epc=004000c0     prid=00000220
```

通过 `tlbdump` 命令导出 TLB 中的信息：

```
GXemul> tlbdump
cpu0: (index=0xf random=0x37)
  0: (invalid)
  1: (invalid)
  2: (invalid)
  3: (invalid)
  4: (invalid)
  5: (invalid)
  6: (invalid)
  7: (invalid)
  8: vaddr=0x00404000 (asid 01), paddr=0x03feb000 D
  9: vaddr=0x7f82f000 (asid 01), paddr=0x00430000 D
 10: vaddr=0x7fd80000 (asid 01), paddr=0x03fd9000 D
 11: vaddr=0x7fdff000 (asid 01), paddr=0x03ff3000 D
 12: vaddr=0x00408000 (asid 01), paddr=0x03fe7000 D
 13: vaddr=0x00402000 (asid 01), paddr=0x03fed000 D
 14: vaddr=0x00409000 (asid 01), paddr=0x03fe6000 D
 15: (invalid)
 16: vaddr=0x7f400000 (asid 01), paddr=0x00432000 D
 17: vaddr=0x0040d000 (asid 01), paddr=0x03fe2000 D
 18: vaddr=0x7f3fd000 (asid 01), paddr=0x03ff2000 D
 19: vaddr=0x00401000 (asid 01), paddr=0x03fee000 D
 20: vaddr=0x00400000 (asid 01), paddr=0x03ff0000 D
 21: vaddr=0x7f400000 (asid 03), paddr=0x00432000 D
 22: vaddr=0x00403000 (asid 04), paddr=0x03fa0000
 23: vaddr=0x00403000 (asid 05), paddr=0x03b69000 D
 24: vaddr=0x10041000 (asid 01), paddr=0x03b79000 D
 25: (invalid)
 26: vaddr=0x10040000 (asid 01), paddr=0x03b7a000 D
 27: (invalid)
 28: vaddr=0x7f400000 (asid 02), paddr=0x00432000 D
 29: vaddr=0x1003f000 (asid 01), paddr=0x03b7b000 D
 30: (invalid)
 31: vaddr=0x00403000 (asid 03), paddr=0x03fa0000
 32: vaddr=0x7f3fd000 (asid 03), paddr=0x03b88000 D
 33: vaddr=0x61000000 (asid 05), paddr=0x03fd8000 D
 34: vaddr=0x5fc04000 (asid 05), paddr=0x03b5f000 D
 35: vaddr=0x7fd7f000 (asid 05), paddr=0x03b62000 D
 36: vaddr=0x00408000 (asid 05), paddr=0x03b64000 D
 37: vaddr=0x1003e000 (asid 01), paddr=0x03b7c000 D
 38: (invalid)
 39: vaddr=0x00402000 (asid 03), paddr=0x03fa1000
 40: vaddr=0x00400000 (asid 03), paddr=0x03fa4000
 41: vaddr=0x00409000 (asid 04), paddr=0x03f9a000
 42: vaddr=0x00407000 (asid 04), paddr=0x03b82000 D
 43: vaddr=0x7f400000 (asid 04), paddr=0x00432000 D
 44: vaddr=0x0080d000 (asid 04), paddr=0x03fa7000 D
 45: vaddr=0x7f3fd000 (asid 04), paddr=0x03b86000 D
 46: vaddr=0x00402000 (asid 04), paddr=0x03fa1000
 47: vaddr=0x00400000 (asid 04), paddr=0x03fa4000
 48: vaddr=0x1003d000 (asid 01), paddr=0x03b7d000 D
 49: (invalid)
 50: vaddr=0x5fc01000 (asid 05), paddr=0x03fff000 D
 51: vaddr=0x00400000 (asid 05), paddr=0x03b6d000 D
 52: vaddr=0x00402000 (asid 05), paddr=0x03b6a000 D
 53: vaddr=0x00409000 (asid 05), paddr=0x03b63000 D
 54: vaddr=0x7f3fd000 (asid 05), paddr=0x03b70000 D
 55: vaddr=0x00402000 (asid 02), paddr=0x03fbb000 D
 56: vaddr=0x7f3fd000 (asid 02), paddr=0x03fc1000 D
 57: vaddr=0x00401000 (asid 02), paddr=0x03fbc000 D
 58: vaddr=0x00400000 (asid 02), paddr=0x03fbe000 D
 59: vaddr=0x10042000 (asid 01), paddr=0x03b78000 D
 60: vaddr=0x7fc40000 (asid 01), paddr=0x03fde000 D
 61: vaddr=0x10002000 (asid 01), paddr=0x03fdc000 D
 62: vaddr=0x10001000 (asid 01), paddr=0x03fdf000 D
 63: vaddr=0x10003000 (asid 01), paddr=0x03fd8000 D
```

Trace

使用 trace 可以帮助同学们了解程序的运行轨迹。

```
GXemul> trace
show_trace_tree = ON  (was: OFF)
GXemul> c
<env_run(0x80432488,0x804321d0,0,&env_sched_list,..)>
  <bcopy(0x81ffff64,0x804321d0,156)>
  <lcontext(0x83b81000,0x8043226c,0x8043226c,&env_sched_list,..)>
  <env_pop_tf(0x80432488,0x140,0x8043226c,&env_sched_list,..)>
  ........
```

Reference

- GXemul Debug