ReSP: A Non-Intrusive Transaction-Level Reflective MPSoC Simulation Platform for Design Space Exploration

G. Beltrame[†] C. Bolchini[§] L. Fossati[§] A. Miele[§] D. Sciuto[§]

[†]European Space Agency
Microelectronics Section

[§]Politecnico di Milano
Dipartimento di Elettronica e Informazione

January 17, 2008

Outline

Outline

Overview

ReSP: *Re*flective *S*imulation *P*latform

- A Virtual Platform for Hardware/Software codesign

Overview

ReSP: *Re*flective *S*imulation *P*latform

- A Virtual Platform for Hardware/Software codesign
- Provides the speed of C++, the modeling facilities of SystemC and the reflective and scripting capabilities of Python
  - Reflection allows a non-intrusive visibility on all the platform elements

Overview

ReSP: *Re*flective *S*imulation *P*latform

- A Virtual Platform for Hardware/Software codesign
- Provides the speed of C++, the modeling facilities of SystemC and the reflective and scripting capabilities of Python
  - Reflection allows a non-intrusive visibility on all the platform elements

Advantages of the Approach

1. easy integration of external IPs and models
2. fine grain control of system-level simulation
3. effortless development of tools for system analysis and design space exploration

Virtual Platforms

Definition

Virtual Platform: a system level model that represents the real system behavior

Virtual Platforms

Definition

Virtual Platform: a system level model that represents the real system behavior

- operates at the level of processor instructions, memory accesses, and data packet transfers, as opposed to RTL
- used in system-level design (ESL) for application functional and performance analysis

Virtual Platforms

Definition

Virtual Platform: a system level model that represents the real system behavior

- operates at the level of processor instructions, memory accesses, and data packet transfers, as opposed to RTL
- used in system-level design (ESL) for application functional and performance analysis

- Enables Hardware/Software codesign
- Improves reuse of models during ESL design

Related Work

- StepNP: A System–Level Exploration Platform for Network Processors (Paulin at al., 2000):
  - First widespread SystemC platform
  - Require *special wrappers* around the component models
  - Access to simulation via SIDL, a custom interface definition language
- Exploiting TLM and object introspection for system-level simulation (Beltrame at al., 2005): introduces the concept of introspection
  - Intrusive approach, requiring manual component modifications
  - Based on StepNP
- Embed scripting inside SystemC (Vennin at al., 2006): proposes integration among SystemC and Python
  - Python is used to embed scripting inside SystemC modules, but requires modifications to the SystemC kernel
  - reduces code size, at the expense of speed reduction (10x reduction)

Related Work

- StepNP: A System–Level Exploration Platform for Network Processors (Paulin at al., 2000):
  - First widespread SystemC platform
  - Require *special wrappers* around the component models
  - Access to simulation via SIDL, a custom interface definition language
- Exploiting TLM and object introspection for system-level simulation (Beltrame at al., 2005): introduces the concept of introspection
  - Intrusive approach, requiring manual component modifications
  - Based on StepNP
- Embed scripting inside SystemC (Vennin at al., 2006): proposes integration among SystemC and Python
  - Python is used to embed scripting inside SystemC modules, but requires modifications to the SystemC kernel
  - reduces code size, at the expense of speed reduction (10x reduction)

Related Work

- StepNP: A System–Level Exploration Platform for Network Processors (Paulin at al., 2000):
  - First widespread SystemC platform
  - Require *special wrappers* around the component models
  - Access to simulation via SIDL, a custom interface definition language
- Exploiting TLM and object introspection for system-level simulation (Beltrame at al., 2005): introduces the concept of introspection
  - Intrusive approach, requiring manual component modifications
  - Based on StepNP
- Embed scripting inside SystemC (Vennin at al., 2006): proposes integration among SystemC and Python
  - Python is used to embed scripting inside SystemC modules, but requires modifications to the SystemC kernel
  - reduces code size, at the expense of speed reduction (10x reduction)

Outline

ReSP: Simulation Core

- Based on the OSCI SystemC TLM library
  - Used for keeping simulation time
  - Used for low level communication
  - Python wrappers have been automaticaly created

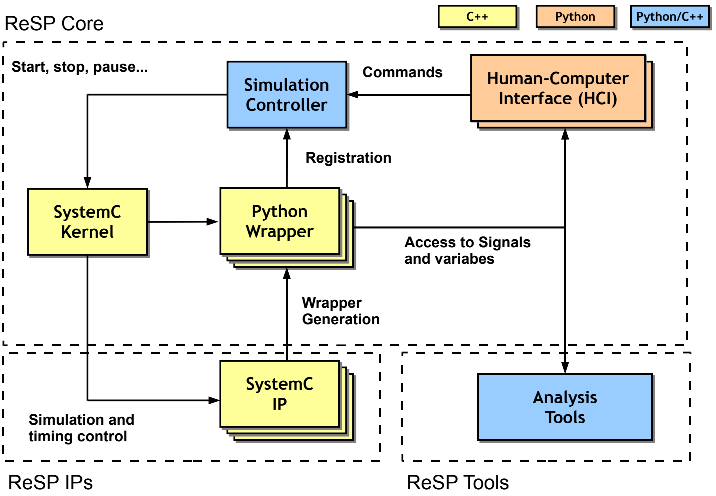
ReSP: Simulation Core

- Based on the OSCI SystemC TLM library
  - Used for keeping simulation time
  - Used for low level communication
  - Python wrappers have been automaticaly created
- Provides a simulation controller
  - Extends SystemC with asynchronous control
  - Keeps basic statistics about the simulation
  - Instantiates and connects the available architectural components

ReSP: Simulation Core

- Based on the OSCI SystemC TLM library
  - Used for keeping simulation time
  - Used for low level communication
  - Python wrappers have been automaticaly created
- Provides a simulation controller
  - Extends SystemC with asynchronous control
  - Keeps basic statistics about the simulation
  - Instantiates and connects the available architectural components
- Interacts with the user through one or more *H*uman *C*omputer *I*nterfaces
  - E.g. An extended Python console and a socket interface

ReSP: Overall Structure



ReSP: Tools

- debugger: Remote Stub for GNU/GDB
  - uses GDB's Serial Remote Interface
  - it defines custom commands to control the flow of time
- profiler: produces statistics on the executed software
  - no instrumentation in the software
- fault injector: simulates the system behaviour in presence of faults
  - follows the SoftWare-Implemented Hardware Fault Injection approach
- More tools are in the making, e.g. power analyzer

ReSP: Component Models

Easy Intergration of Any SystemC Module

- Thanks to the automatic wrapper generation
- Favors IP reusability

ReSP: Component Models

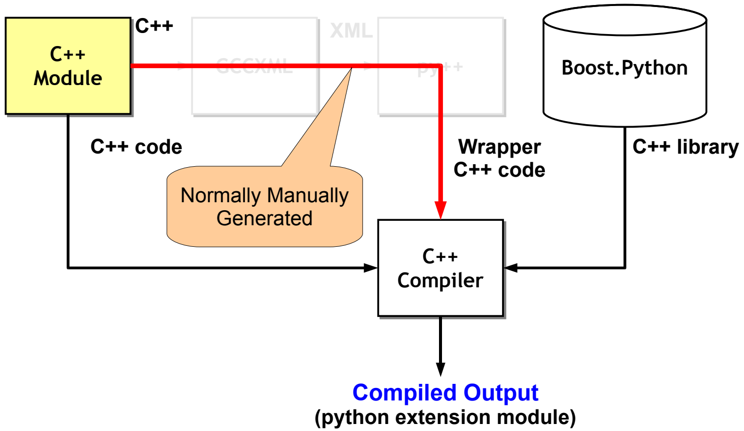
Easy Intergration of Any SystemC Module

- Thanks to the automatic wrapper generation
- Favors IP reusability

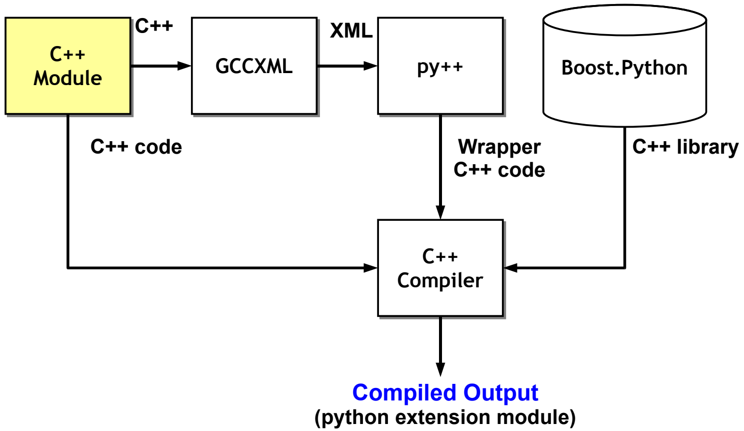
The following hardware component models are part of ReSP:

1. processors cores: ARM7TDMI, PowerPC 405, Leon2, MIPS and Nios2
   - written using the ArchC *Architectural Description Language*
2. interconnections: arbitrated bus
3. memories: simple memories, Leon3 L1 cache, coherent directory based caches
4. miscellaneous: timer and interrupt controller of the ARM PID board, PC16x5x UART model

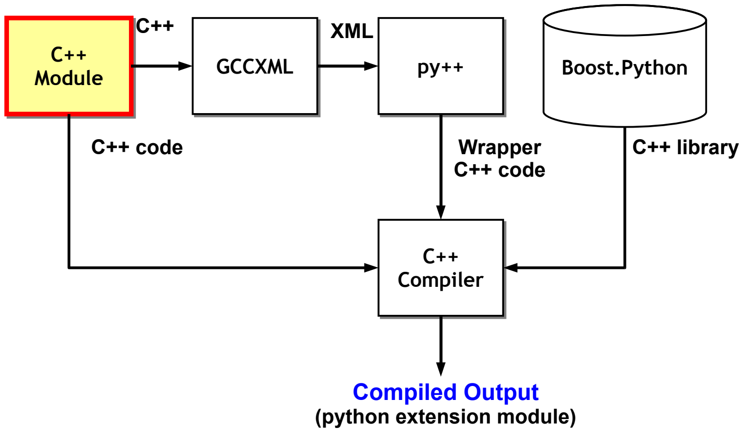
ReSP: Wrapper Generation



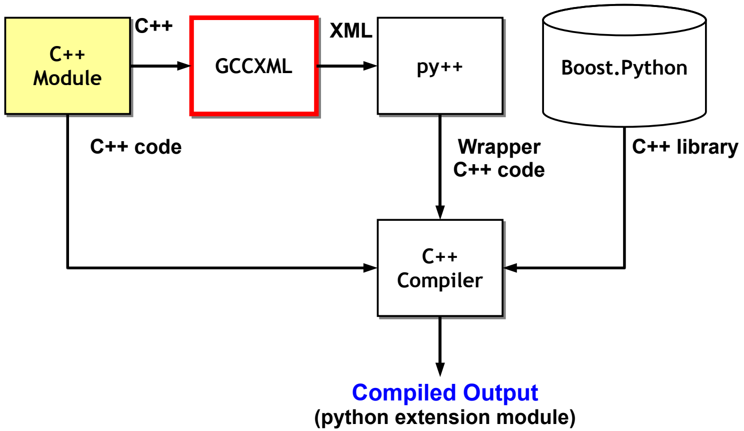
ReSP: Wrapper Generation



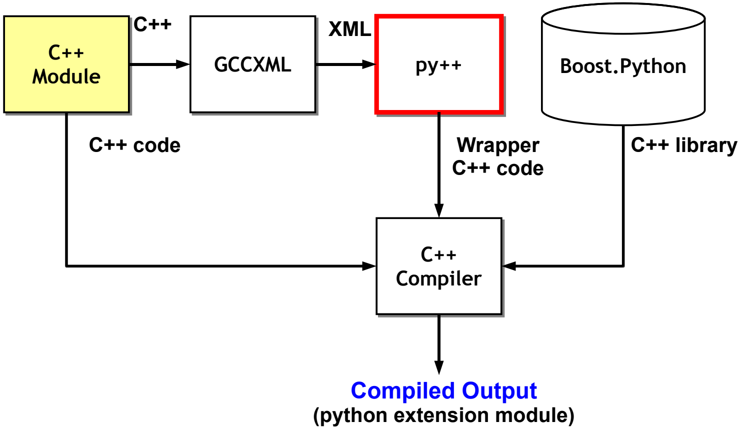
ReSP: Wrapper Generation



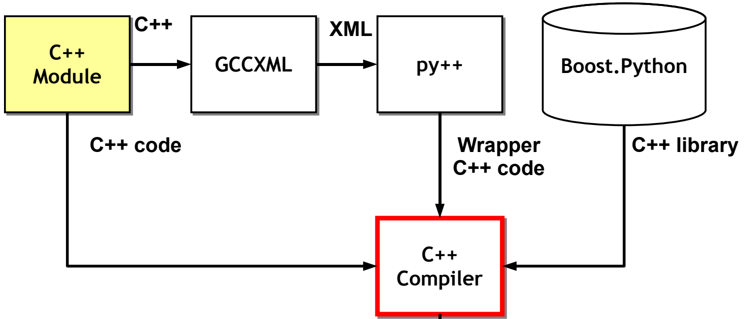
ReSP: Wrapper Generation



ReSP: Wrapper Generation



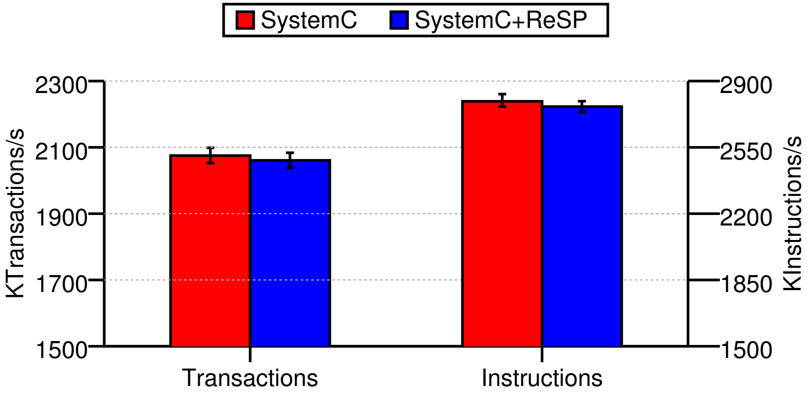
ReSP: Wrapper Generation



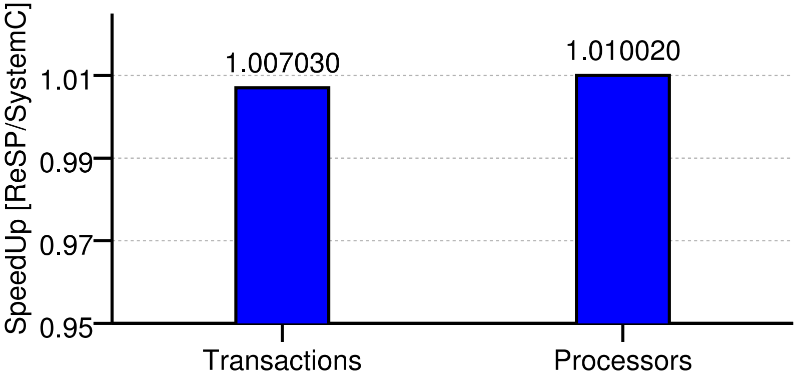
Compiled Output
(python extension module)

Outline

Analysis of Reflection Overhead I



Analysis of Reflection Overhead II



Outline

Using ReSP for Fault Analysis
Background

- Reflective capabilities are used for implementing fault injection facilities
    - Single Event Upsets are simulated by modifying the models' internal state
    - Follows the SoftWare-Implemented Hardware Fault Injection approach (SWIHFI)
    - No code instrumentation is required

Using ReSP for Fault Analysis
Background

- Reflective capabilities are used for implementing fault injection facilities
  - Single Event Upsets are simulated by modifying the models' internal state
  - Follows the SoftWare-Implemented Hardware Fault Injection approach (SWIHFI)
  - No code instrumentation is required
- Related Work:
  - Classical approaches pursue fault injection by means of code instrumentation
  - Only a few works have exploited reflective programming but they do not consider SystemC hardware models

Using ReSP for Fault Analysis
Hardened Code Example

Original code	Modified Code

`a = 3;`	`a0 = 3;`
	`a1 = 3;`
	`if (a0 != a1)`
	`  error();`

Using ReSP for Fault Analysis
Results

- Fault injection campaign carried out by injecting more than 10000 faults in a Leon2 processor

Using ReSP for Fault Analysis
Results

- Fault injection campaign carried out by injecting more than 10000 faults in a Leon2 processor
- Experimental results:

Application	Register	Faults	No Error	Error HW Detected	Error SW Detected	Error Not detected
ELPF	Reg. Bank	2000	1787	51	152	10
	PC Reg.	1000	775	12	207	6
	Other Regs	600	591	0	9	0
FIR	Reg. Bank	2000	1742	85	154	19
	PC Reg.	1000	663	93	235	9
	Other Regs	600	571	0	27	2
Kalman	Reg. Bank	2000	1540	185	271	4
	PC Reg.	1000	591	62	346	1
	Other Regs	600	593	0	7	0
TOTAL		10800	8853	488	1408	51

The results are coherent to what presented in **Combined software and hardware techniques for the design of reliable IP processors**, *Rebaudengo et Al.*

Outline

Conclusions: Future Work

- Support for the TLM 2.0 Draft 2 standard
    - Currently Draft 1 is used
    - We expect significant improvements in simulation speed

Conclusions: Future Work

- Support for the TLM 2.0 Draft 2 standard
  - Currently Draft 1 is used
  - We expect significant improvements in simulation speed
- Callback facilities
  - The status of the models is monitored
  - Actions are taken in correspondence of particular events

Conclusions: Future Work

- Support for the TLM 2.0 Draft 2 standard
    - Currently Draft 1 is used
    - We expect significant improvements in simulation speed
- Callback facilities
    - The status of the models is monitored
    - Actions are taken in correspondence of particular events
- Design Space Exploration algorithms
    - Necessary for tuning complex MP-SoC

Conclusions: Wrap-Up

1. Virtual Platform targeted to Multi-Processor Systems-On-Chip
2. Based on Python and SystemC with automatic wrapper generation
    - Python augments ReSP with Reflective Capabilities
    - Reflection allows a non-intrusive visibility on all the simulated elements
3. No significant overhead due to Python
4. Fine grain control of the simulation
5. Fault Injection case study demonstates the usefulness of the technology

Conclusions: Wrap-Up

1. Virtual Platform targeted to Multi-Processor Systems-On-Chip
2. Based on Python and SystemC with automatic wrapper generation
   - Python augments ReSP with Reflective Capabilities
   - Reflection allows a non-intrusive visibility on all the simulated elements
3. No significant overhead due to Python
4. Fine grain control of the simulation
5. Fault Injection case study demonstates the usefulness of the technology

Conclusions: Wrap-Up

1. Virtual Platform targeted to Multi-Processor Systems-On-Chip
2. Based on Python and SystemC with automatic wrapper generation
   - Python augments ReSP with Reflective Capabilities
   - Reflection allows a non-intrusive visibility on all the simulated elements
3. No significant overhead due to Python
4. Fine grain control of the simulation
5. Fault Injection case study demonstates the usefulness of the technology

Conclusions: Wrap-Up

1. Virtual Platform targeted to Multi-Processor Systems-On-Chip
2. Based on Python and SystemC with automatic wrapper generation
   - Python augments ReSP with Reflective Capabilities
   - Reflection allows a non-intrusive visibility on all the simulated elements
3. No significant overhead due to Python
4. Fine grain control of the simulation
5. Fault Injection case study demonstates the usefulness of the technology

Conclusions: Wrap-Up

1. Virtual Platform targeted to Multi-Processor Systems-On-Chip
2. Based on Python and SystemC with automatic wrapper generation
   - Python augments ReSP with Reflective Capabilities
   - Reflection allows a non-intrusive visibility on all the simulated elements
3. No significant overhead due to Python
4. Fine grain control of the simulation
5. Fault Injection case study demonstates the usefulness of the technology

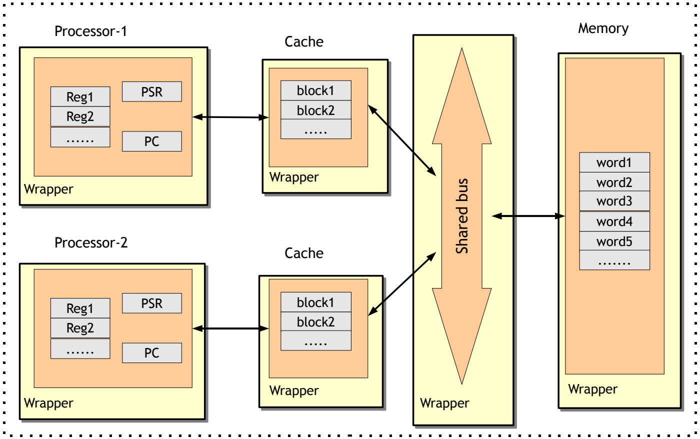
Thank You

Any Questions?

For more details: http://www.resp-sim.org

Appendix

Experimental Session - I



We will use this architecture for software debugging

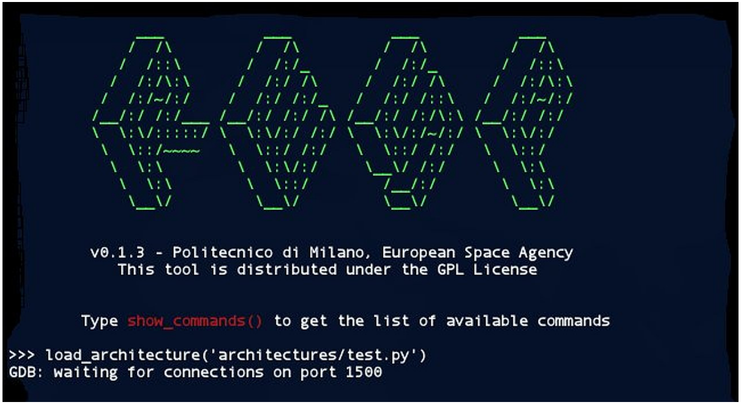
Experimental Session - II

```
proc1 = arm7.arm7('proc1')
mem = SimpleMemory32.SimpleMemory32('mem', 0x800000)
bus = pv_router32.pv_router32('SimpleBus',  2) #2 masters

manager.connectPortsForce(proc1,
  proc1.DATA_MEM_port.memory_port, bus, bus.target_port[0])
manager.connectPortsForce(proc1,
  proc1.PROG_MEM_port.memory_port, bus, bus.target_port[0])
manager.connectPortsForce(proc2,
  proc2.DATA_MEM_port.memory_port, bus, bus.target_port[1])
manager.connectPortsForce(proc2,
  proc2.PROG_MEM_port.memory_port, bus, bus.target_port[1])

manager.connectPortsForce(bus, bus.initiator_port, mem, mem.memPort)

bus.addBinding("mem.mem_SimpleMemPort", 0x0, 0x800000)
```

Experimental Session - II

```
parser = Parser.Parser('exampleApp.elf')
proc1.init(0, parser.getProgStart(),
     parser.getDataStart(), parser.getProgDim())
proc2.init(1, parser.getProgStart(),
     parser.getDataStart(), parser.getProgDim())
mem.loadApplication(parser.getProgData(),
     parser.getDataStart(), parser.getProgDim())

inter1 = GDBProcStub32.arm7tdmiStub(proc1)
stub1 = GDBStub32.GDBStub32(inter1, 1500)
proc1.setGDBStub(stub1)
inter2 = GDBProcStub32.arm7tdmiStub(proc2)
stub2 = GDBStub32.GDBStub32(inter2, 1501)
proc2.setGDBStub(stub2)
```

Experimental Session - III



```
      ___            ___            ___            ___
     /  /\          /  /\          /  /\          /  /\
    /  /::\        /  /:/_        /  /:/_        /  /::\
   /  /:/\:\      /  /:/ /\      /  /:/ /\      /  /:/\:\
  /  /:/~/:/     /  /:/ /:/_    /  /:/ /::\    /  /:/~/:/
 /__/:/ /:/___  /__/:/ /:/ /\  /__/:/ /:/\:\  /__/:/ /:/
 \  \:\/:::::/  \  \:\/:/ /:/  \  \:\/:/~/:/  \  \:\/:/
  \  \::/~~~~    \  \::/ /:/    \  \::/ /:/    \  \::/
   \  \:\         \  \:\/:/      \__\/ /:/      \  \:\
    \  \:\         \  \::/         /__/:/        \  \:\
     \__\/          \__\/          \__\/          \__\/


      v0.1.3 - Politecnico di Milano, European Space Agency
         This tool is distributed under the GPL License


     Type show_commands() to get the list of available commands

>>> load_architecture('architectures/test.py')
GDB: waiting for connections on port 1500
```

Experimental Session - IV

Connecting the debugger:

```
GNU gdb 6.7.1
..........
(gdb) target remote localhost:1500
```

Examining and modifying the components' status:

```
>>> hex(proc1.RB.read(14))
0xf200
>>> proc1.acp_pc.write(0x200)
>>> proc1.totalCycles
1500
```