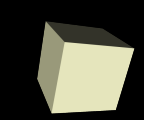
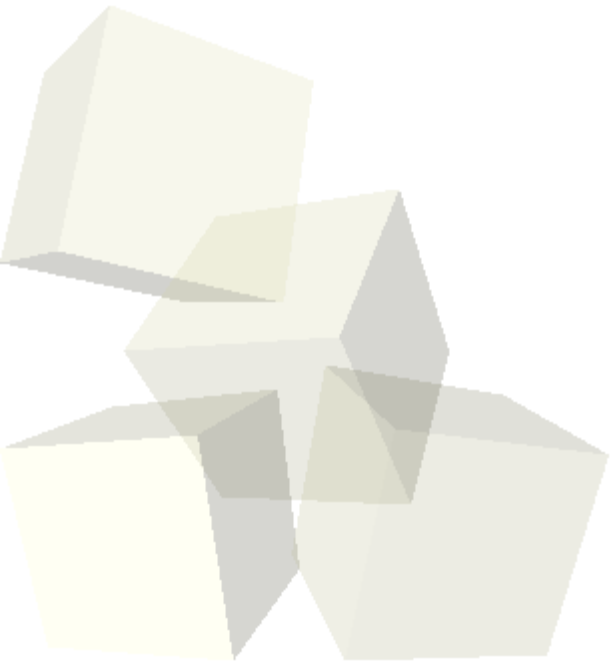


# Lab 3: lowRISC with a trace debugger

Manuel J. Bellido Díaz

Abril de 2017



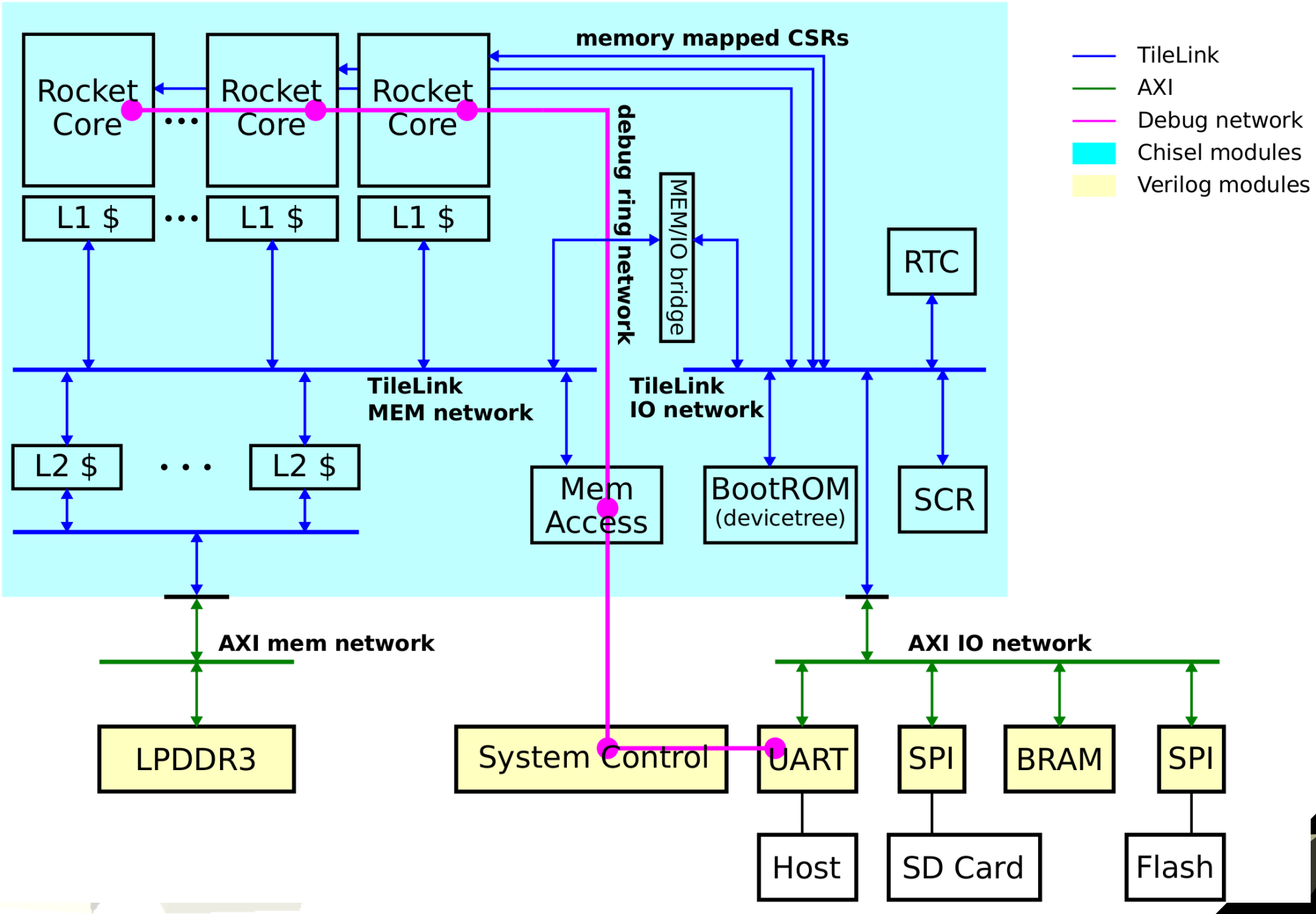
# lowRISC with a trace debugger

- Uno de las cosas importantes para poder desarrollar un SoC tanto a nivel hardware como software es que tenga un buen sistema de depuración on-chip
- En este tutorial se presenta el desarrollo realizado para dotar de infraestructura de debug a lowRISC
- El trabajo esta desarrollado a partir de la versión Untether a la que se añade la infraestructura de debug. Esta preparada una implementación hardware del sistema sobre la plataforma NEXYS4 DDR.
- **Localización del tutorial:**

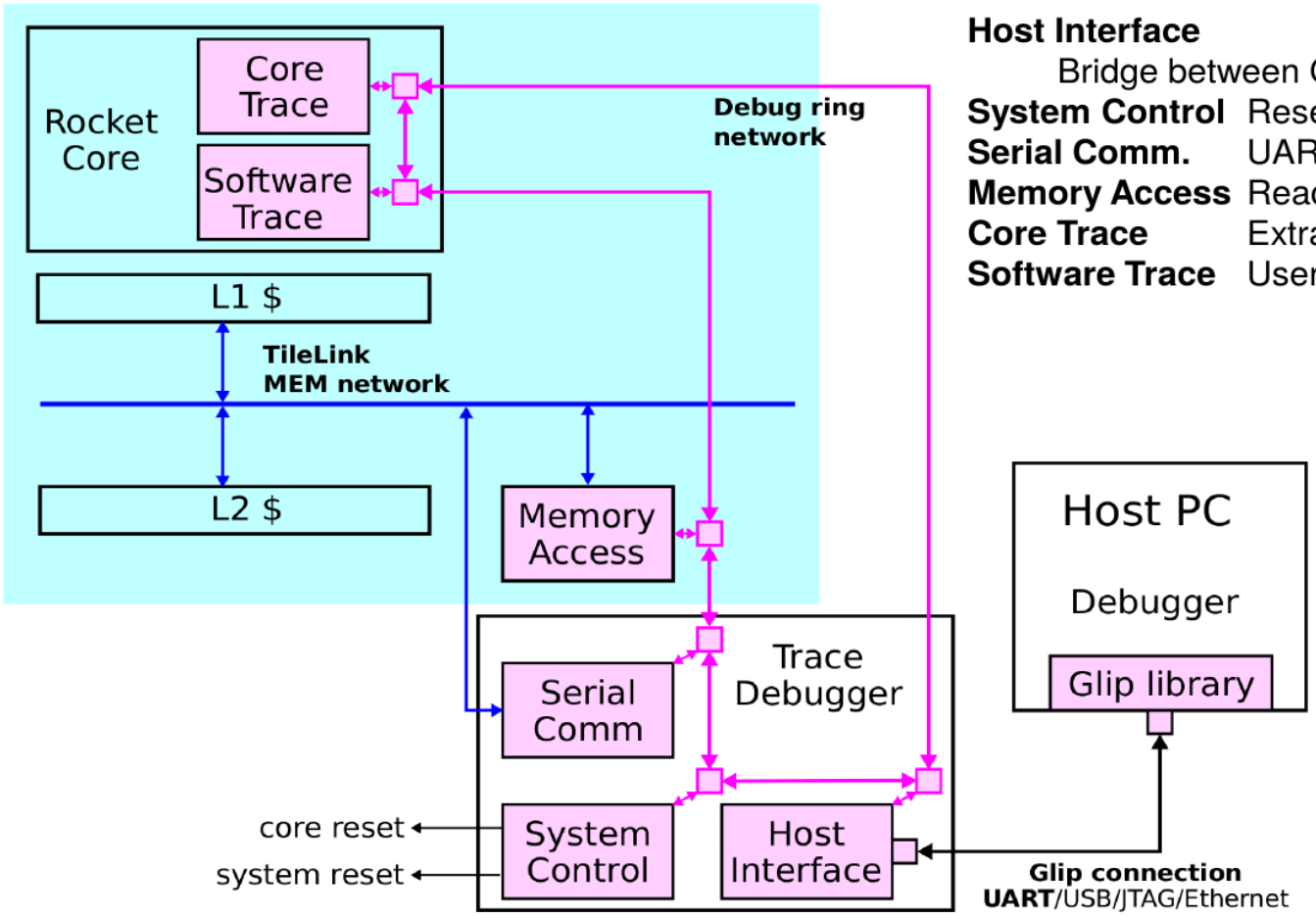
  http://www.lowrisc.org/docs/debug-v0.3/

- Organización del tutorial:
  - Apartado 1: describe la infraestructura de debug. Consta de varios subapartados.
  - Apartado 2: Dedicado a contar como preparar el entorno de desarrollo. **NO** se pueden utilizar los toolchains de la versión untether. Habrá que instalar el software de debug.
  - Apartado 3: Demostración de debug sobre lowRISC: Simulaciones y demostración on-chip sobre FPGA.
  - Apartado 4: Otras cosas relativas a lowRISC. Este apartado no lo trabajaremos.
- La infraestructura de debug esta basada en OPENSOC Debug:
  - http://opensocdebug.org/
  - https://www.youtube.com/watch?v=cQeL3NHrB8w

# 1.- Overview of the debug infrastructure



# Trace Debugger Internals




**Host Interface**
Bridge between Glip and debug ring

| | |
|---|---|
| **System Control** | Reset and pause control |
| **Serial Comm.** | UART emulator |
| **Memory Access** | Read/write L2 |
| **Core Trace** | Extract instruction trace |
| **Software Trace** | User defined trace |

# Enumeration & System Control

- System Enumeration
  - Each debug module has a unique ID used as destination for debug packets
  - Fixed ID for **Host Interface** (0) and **System Control** (1)
  - **System Control** has the total number of modules and communication parameters of the on-chip debug network
  - Each debug module has a set of compulsory registers: type, vendor, version
  - Host side debug software is then able to discover all modules by enumeration

- System Control
  - Total number of modules and parameters for debug network
  - Set/Reset system and processor cores

# Memory Access & Serial Comm.

- Memory Access
  - Provide a coherent access to L2
  - Allow debugger to read/write memory/cache
  - Allow load elf (program) or binary data

- Serial Comm.
  - Emulate a UART16550 IP.
  - Allow UART communication through debugger (share debugger & UART cable)
  - Can be instantiated multiple times if needed

# Core Trace

- Function: collect information from the core execution
  - Reconstruct program flow
  - Verify register values
  - Performance analysis
- Trace collection
  - JAL (function call), jump and branch, change of privilege modes
  - ToDo: more traces and run-time configurable filters
- Trace event generation
  - Packetized with timestamp, send to host over debug network
  - Current: Simple overflow handling (drop but record #drops)
  - Future:
    - Better network flow control / QoS
    - Circular buffering and trace recording to DRAM

# Example Core Trace

```
# time   event
06570d02 enter init_tls
06570d22 enter memcpy
06570d67 leave memcpy
06570d76 enter memset
06570dae leave memset
06570dcd leave memset
06570dd5 leave init_tls
06570ddb enter thread_entry
06570e22 leave thread_entry
06570e28 enter main
06570e60 enter trace_event0
06570e91 leave trace_event0
06570e96 enter trace_event1
06570ea9 leave trace_event1
06570eb3 enter trace_event2
06570eca leave trace_event2
06570ee3 leave main
06571085 enter exit
065710b3 enter syscall
06571131 change mode to 3
065711ba enter handle_trap
0657127e enter tohost_exit
Overflow, missed 12 events
Overflow, missed 25 events
Overflow, missed 28 events
Overflow, missed 28 events
Overflow, missed 28 events
```

# Software Trace

- Function: minimally-invasive code instrumentation
  - Light-weighted alternative to printf()
  - Performance measurement between code points, etc.
  - Can be release unchanged (safety) with minimal performance impact
- Thread-safe trace procedure
  - A trace event: (id, value)
  - Write to $a0 (value), tracked by **Software Trace**
  - Write to a dedicated CSR with (id), which triggers an event
- Trace event generation (same with **Core Trace**)
  - Trace event generation
  - Packetized with timestamp, send to host over debug network
  - Future: Better network flow control / QoS

# Example Software Trace

- Trace DMA durations

```
#define TRACE(id,v) \
  asm volatile ("mv   a0,%0": :"r" ((uint64_t)v) : "a0"); \
  asm volatile ("csrw 0x8f0, %0" :: "r"(id));

#define TRACE_DMA_BUFFER(b) TRACE(0x1001,b)
#define TRACE_DMA_START(i,s,b) TRACE(0x1002,i) \
                               TRACE(0x1002,s) \
                               TRACE(0x1002,b)
#define TRACE_DMA_FINISH(i) TRACE(0x1003,i)
```

Header

```
uint8_t *buffer = malloc(42);
TRACE_DMA_BUFFER(buffer);

TRACE_DMA_START(slotid,src,buffer);
dma_transfer(slotid,incoming,buffer);
TRACE_DMA_FINISH(slotid);
```

Source

```
# time    id      value
00002590  0x1001  0xe20c000ac20fc588
00002593  0x1002  0x0000000000000001
00002595  0x1002  0xffff0800000c0000
00002597  0x1002  0xe20c000ac20fc588
00002985  0x1003  0x0000000000000001
```

Trace Log


Visualization

# Debug Procedure

## Command Line Interface

```
# reset and pause cores
reset -halt
# load a test program
mem loadelf test.elf 3
# enable core trace
ctm log ctm.log 4
# enable software trace
stm log stm.log 5
# open a terminal (xterm)
terminal 2
# run the test
start
```

| *ID* | *Module* |
|---|---|
| 0 | Host interface |
| 1 | System Control |
| 2 | Ser. Comm. |
| 3 | Mem. Access |
| 4 | Core Trace |
| 5 | Software Trace |

## Python Script

```
import opensocdebug
import sys

if len(sys.argv) < 2:
    print "Usage: runelf.py <filename>"
    exit(1)

elffile = sys.argv[1]

osd = opensocdebug.Session()

osd.reset(halt=True)

for m in osd.get_modules("STM"):
    m.log("stm{:03x}.log".format(m.get_id()))

for m in osd.get_modules("CTM"):
    m.log("ctm{:03x}.log".format(m.get_id()),
elffile)

for m in osd.get_modules("MAM"):
    m.loadelf(elffile)

osd.start()
```

# Extra Features of the Debugger

- Uniform debug environment for both Sim/FPGA
  - DPI based Glip interface for simulation.
  - Support UART and trace debugging in both RTL and FPGA simulation.

- Python frontend
  - Allow further tool integration (deliver as a python library).
  - Off-line trace analysis.
  - Easy command extension.

# Future Work for Debugger

- Improve trace collection:
  - Trace compression: Reduce event number and size
  - Trace filtering: Run-time filter configuration
  - Trace triggering: (Cross-) trigger events
  - GUI tools for better trace analysis
- Integrate run-control solution(s):
  - Traditional GDB-like debugger
  - SiFive, Roa Logic & PULP
  - Hopefully support both through a common interface
- On-chip trace processing (research):
  - Analyse/process traces on-chip possibly on minion cores
  - Get from basic information to knowledge!

# 2. Prepare the environment

- En este apartado se prepara el entorno para trabajar con la versión de lowRISC que incorpora el interfaz de debug.
- Son tres tareas, por orden:
  - Descargar el repositorio y activar las variables de entorno a través del script “set_env.sh”
  - Configurar las herramientas para trabajar con lowRISC genéricas:
    - Vivado: ya esta instalado
    - Verilator: Debe estar instalado del tutorial anterior
    - Toolchains para lowRISC: Habrá que volver a crear los toolchains (no sirven los de la versión anterior)
  - Preparar el software de Debug (Open SoC Debug Software)
    - Faltan algunos paquetes para que todo funcione correctamente:

      sudo apt-get install libtool libelf-dev python-dev

# 3. Debug walkthrough

- Este apartado esta dedicado a conocer los pasos básicos para realizar un debug en el sistema. Consta de dos partes: debug a través de simulación y debug on-chip con implementación del sistema en FPGA.
- SubApartado: Connecting to RTL simulation and enumeration
  - Este subapartado compila el sistema para ser simulado y establece los pasos básicos para conectar el sistema simulado con el software de debug
- SubApartado: A debug session
  - En este subapartado se muestra el procedimiento básico para realizar una sesión de debug.
  - Se realiza una sesión de debug sobre un sistema simulado.

- SubApartado: Running on the FPGA
  - Este subapartado muestra el proceso de debug de un sistema implementado sobre la FPGA
    - Realizar sección Run the pre-built FPGA demo with a trace debugger
    - Realizar sección Build your own bitstream and images (solo la parte que incluye el debug)
    - Cargar los diferentes programas compilados a traves del sistema de debug:
      - hello, sdram, sdcard, .....