NetFPGA Summer Course

Presented by:

Noa Zilberman

Yury Audzevich

Technion

August 2 – August 6, 2015

http://NetFPGA.org

Previously Covered

- The NetFPGA platform
- Network Review
- The Base Reference Router
- The Life of a Packet Through the NetFPGA
- Infrastructure
- Examples of Using NetFPGA
- Example Project: Crypto Switch
- Simulation and Debug

Tutorial Outline

- **Register Infrastructure**
  - Explain register system
  - Use AXI Lite registers modules to implement register
  - Add register access stimulus to define Crypto Switch encryption key
  - Interface with software
- **Build and Test Hardware**
  - Build
  - Explanation of Hardware Tests
  - Write Hardware Tests
  - Program the board
- **Group Discussion**
  - Project Ideas
  - Scope of work that can be accomplished
- **Team up for Projects**
  - Team leaders will describe projects

Section I: Register Infrastructure

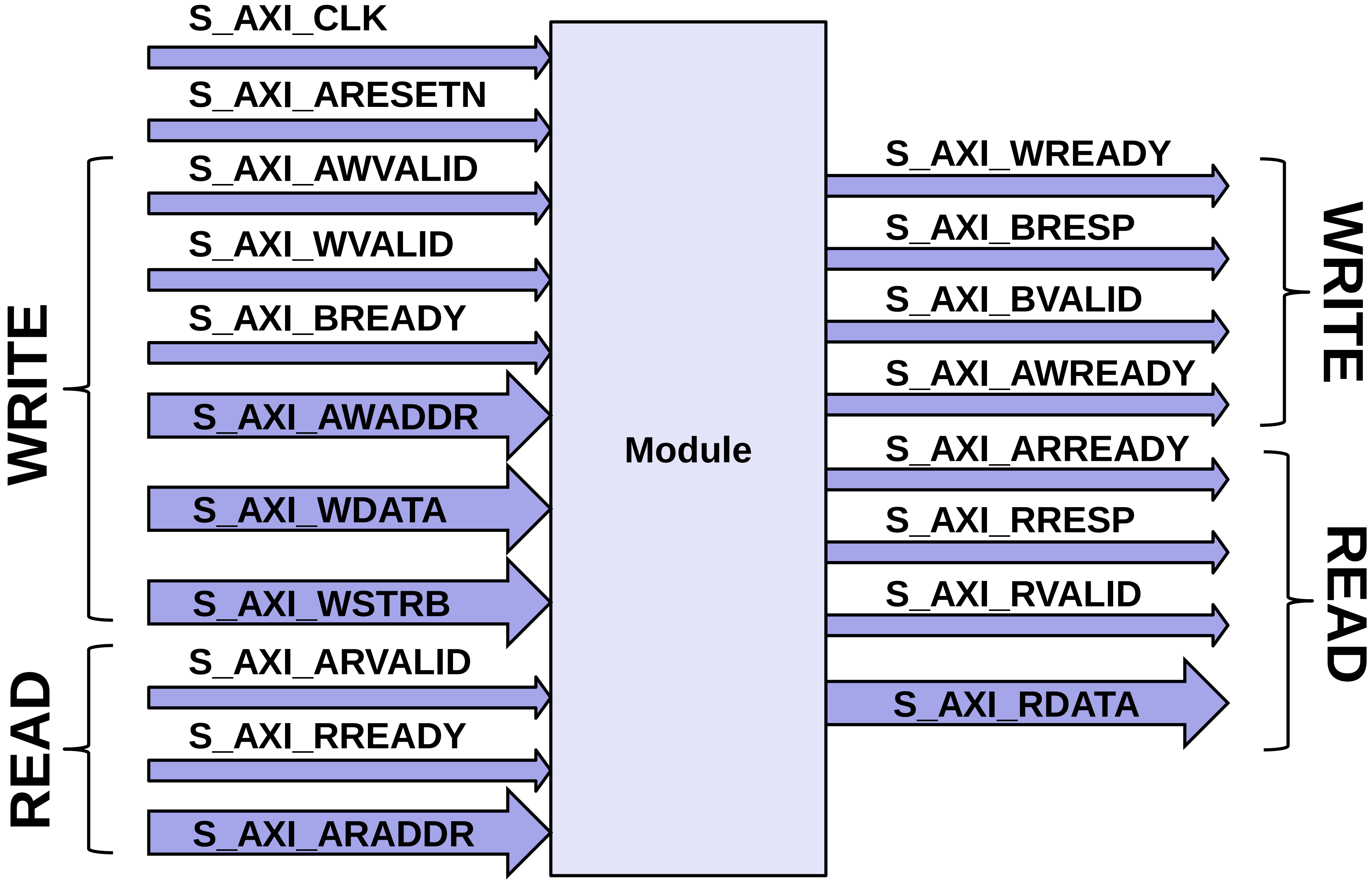
Specifying the Key via a Register

- **Set the key via a register**
  - Instead of a constant value
- **Requires understanding the registers system ☺**
- **Registers system:**
  - Automatically generated
  - Implementing registers in a module
    - Use automatically generated cpu_regs module
  - Need to implement the registers’ functional logic

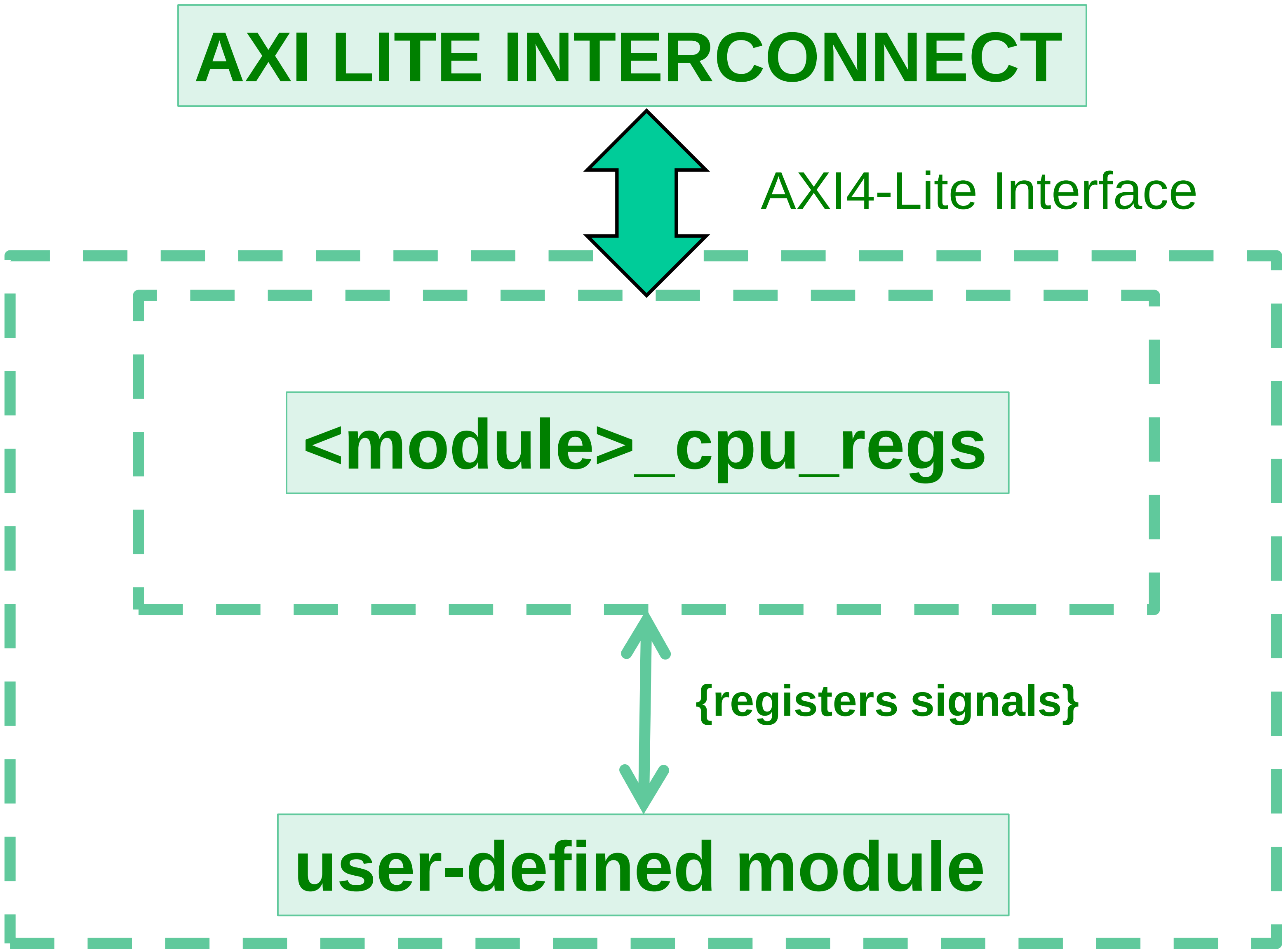
Registers bus

- Yesterday we learnt that packets stream follows the AXI4-Stream paradigm
- Register communication follows the AXI4-Lite paradigm
- The AXI4-Lite interface provides a point-to-point bidirectional interface between a user Intellectual Property (IP) core and the AXI Interconnect

Register bus (AXI4-Lite interface)

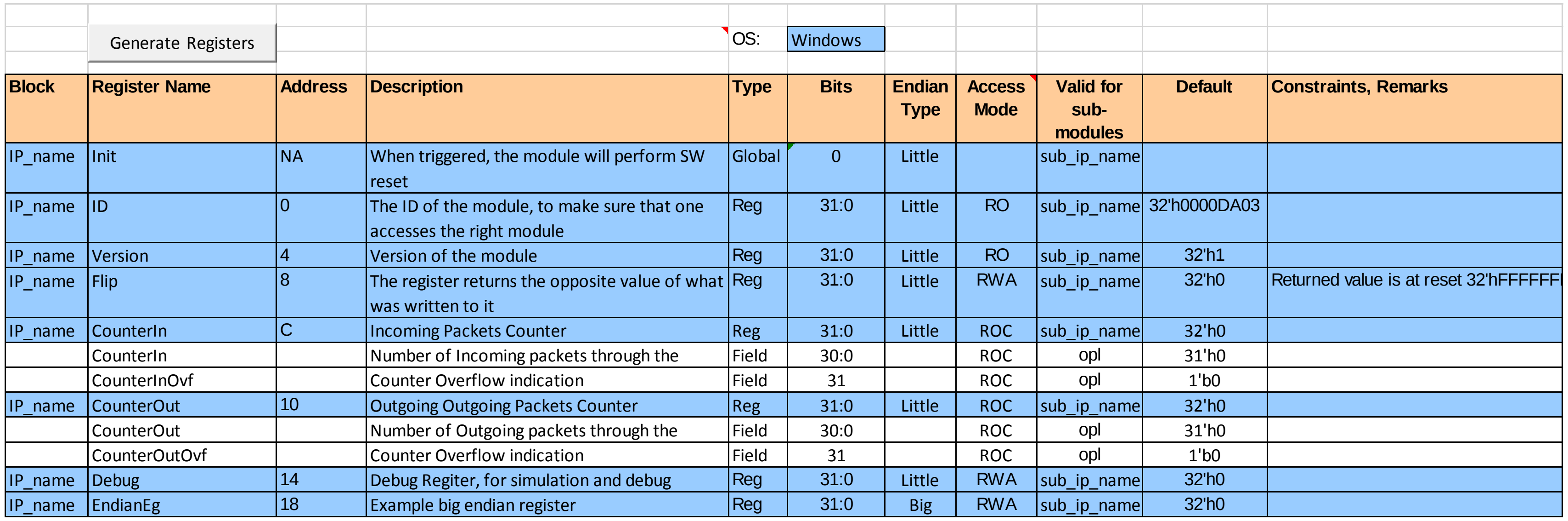


Register bus



Registers – Module generation

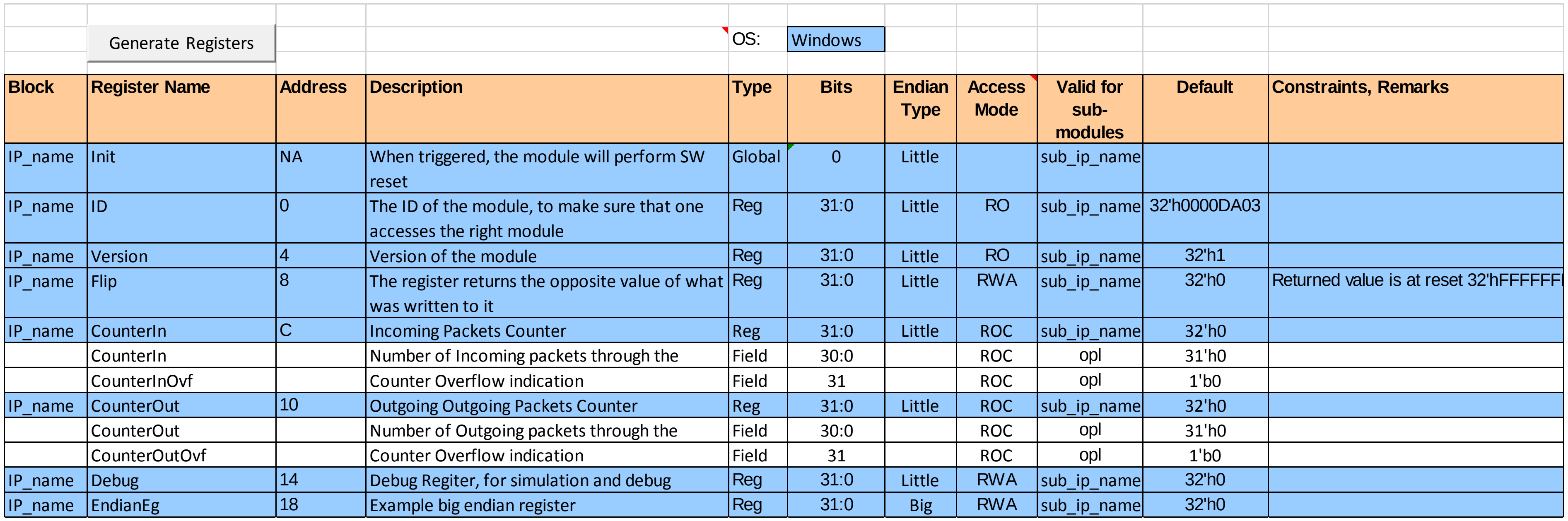
- Spreadsheet based
- Defines all the registers you intend to support and their properties
- Generates a python script (regs_gen.py), which generates the outputs



Generate Registers   OS: Windows

Block	Register Name	Address	Description	Type	Bits	Endian Type	Access Mode	Valid for sub-modules	Default	Constraints, Remarks
IP_name	Init	NA	When triggered, the module will perform SW reset	Global	0	Little		sub_ip_name		
IP_name	ID	0	The ID of the module, to make sure that one accesses the right module	Reg	31:0	Little	RO	sub_ip_name	32'h0000DA03	
IP_name	Version	4	Version of the module	Reg	31:0	Little	RO	sub_ip_name	32'h1	
IP_name	Flip	8	The register returns the opposite value of what was written to it	Reg	31:0	Little	RWA	sub_ip_name	32'h0	Returned value is at reset 32'hFFFFFF
IP_name	CounterIn	C	Incoming Packets Counter	Reg	31:0	Little	ROC	sub_ip_name	32'h0	
	CounterIn		Number of Incoming packets through the	Field	30:0		ROC	opl	31'h0	
	CounterInOvf		Counter Overflow indication	Field	31		ROC	opl	1'b0	
IP_name	CounterOut	10	Outgoing Outgoing Packets Counter	Reg	31:0	Little	ROC	sub_ip_name	32'h0	
	CounterOut		Number of Outgoing packets through the	Field	30:0		ROC	opl	31'h0	
	CounterOutOvf		Counter Overflow indication	Field	31		ROC	opl	1'b0	
IP_name	Debug	14	Debug Regiter, for simulation and debug	Reg	31:0	Little	RWA	sub_ip_name	32'h0	
IP_name	EndianEg	18	Example big endian register	Reg	31:0	Big	RWA	sub_ip_name	32'h0	

Registers – Module generation



Generate Registers

OS: Windows

Block	Register Name	Address	Description	Type	Bits	Endian Type	Access Mode	Valid for sub-modules	Default	Constraints, Remarks
IP_name	Init	NA	When triggered, the module will perform SW reset	Global	0	Little		sub_ip_name		
IP_name	ID	0	The ID of the module, to make sure that one accesses the right module	Reg	31:0	Little	RO	sub_ip_name	32'h0000DA03	
IP_name	Version	4	Version of the module	Reg	31:0	Little	RO	sub_ip_name	32'h1	
IP_name	Flip	8	The register returns the opposite value of what was written to it	Reg	31:0	Little	RWA	sub_ip_name	32'h0	Returned value is at reset 32'hFFFFFFF
IP_name	CounterIn	C	Incoming Packets Counter	Reg	31:0	Little	ROC	sub_ip_name	32'h0	
	CounterIn		Number of Incoming packets through the	Field	30:0		ROC	opl	31'h0	
	CounterInOvf		Counter Overflow indication	Field	31		ROC	opl	1'b0	
IP_name	CounterOut	10	Outgoing Outgoing Packets Counter	Reg	31:0	Little	ROC	sub_ip_name	32'h0	
	CounterOut		Number of Outgoing packets through the	Field	30:0		ROC	opl	31'h0	
	CounterOutOvf		Counter Overflow indication	Field	31		ROC	opl	1'b0	
IP_name	Debug	14	Debug Regiter, for simulation and debug	Reg	31:0	Little	RWA	sub_ip_name	32'h0	
IP_name	EndianEg	18	Example big endian register	Reg	31:0	Big	RWA	sub_ip_name	32'h0	

Registers – Module generation

Access Modes:

- RO - Read Only (by SW)
- ROC - Read Only Clear (by SW)
- WO - Write Only (by SW)
- WOE - Write Only Event (by SW)
- RWS - Read/Write by SW
- RWA - Read/Write by HW and SW
- RWCR - Read/Write clear on read (by SW)
- RWCW - Read/Write clear on write (by SW)

Registers – Module generation

Endian Mode:

- **Little Endian – Most significant byte is stored at the highest address**
  - **- Mostly used by CPUs**
- **Big Endian - Most significant byte is stored at the lowest address**
  - **- Mostly used in networking**
  - **- e.g. IPv4 address**

Registers – Generated Modules

- **<module>_cpu_regs.v – Interfaces AXI-Lite to dedicated registers signals**
  To be placed under under <core name>/hdl
- **<module>_cpu_regs_defines.v – Defines per register: width, address offset, default value**
  To be placed under under <core name>/hdl
- **<module>_cpu_template.v – Includes template code to be included in the top core Verilog.**
  This file can be discarded after updating the top core verilog file.

Registers – Generated Modules

Same contents as module>_cpu_regs_defines.v, but in different formats, used by software, build and test harness:

- **<module>_regs_defines.h**
  To be placed under under <core name>/data
- **<module>_regs_defines.tcl**
- To be placed under under <core name>/data
- **<module>_regs_defines.txt – used by test harness**
- To be placed under under <core name>/data

Adding Registers Logic - Example

- **Usage examples:**

```
always @(posedge axi_aclk)
      if (~resetn_sync) begin
              id_reg <= #1    `REG_ID_DEFAULT;
              ip2cpu_flip_reg <= #1    `REG_FLIP_DEFAULT;
              pktin_reg <= #1    `REG_PKTIN_DEFAULT;
              end
      else begin
              id_reg <= #1    `REG_ID_DEFAULT;
              ip2cpu_flip_reg <= #1    ~cpu2ip_flip_reg;
              pktin_reg <= #1  pktin_reg_clear ? 'h0  :  pkt_in ? pktin_reg + 1: pktin_reg ;
           end
```

NetFPGA-Host Interaction

–Register reads/writes via ioctl system call

–Useful command line utilities

```
cd ~/NetFPGA-SUME-
  alpha/lib/sw/std/apps/sume_riffa_v1_0_0/
./rwaxi -a 0x44010000
./rwaxi -a 0x44010000 -w 0x1234
```

You must program the FPGA and load the driver before using these commands!

Can I collect the registers addresses in a unique .h file?

NetFPGA-Host Interaction

– Need to create the sume_register_defines.h file
  - `cd $NF_DESIGN_DIR/hw`
  - `make reg`

– The sume_register_defines.h file will be placed under `$NF_DESIGN_DIR/sw/embedded/src`

NetFPGA-Host Interaction

Required steps:

- Generate .h file per core
  - Automatically generated by the python script
- Edit $NF_DESIGN_DIR/hw/tcl/$NF_PROJECT_NAME_defines.tcl
  - Indicate the address mapping you use
- Edit $NF_DESIGN_DIR/hw/tcl/export_regiters.tcl
  - Indicate the location of all IP cores used
    - Default path assumed is under \lib\hw\cores

NetFPGA-Host Interaction

– sume_register_defines.h is automatically generated when creating a project

- Using NetFPGA TCL scripts, the .h file will match the hardware
- Note that changes in the GUI will not be reflected!

– Post implementation, for the SDK, use $NF_DESIGN_DIR/hw/tcl/export_hardware.tcl

- Uses vivado's export
- Does not include the registers list, only memory map

Testing Registers with Simulation

Testing Registers with Simulation

- **nftest_regwrite(address, value)**
  - nftest_regwrite(0x44010008, 0xABCD)
- **nftest_regread(address)**
  - nftest_regread(0x44010000)
- **nftest_regread_expect(address, expected_value)**
  - nftest_regread_expect(0x44010000, 0xDA01)
- **Can use registers names**
  - nftest_regread(SUME_INPUT_ARBITER_0_ID)
- **Use within run.py**
- **You don’t need to edit any other file**

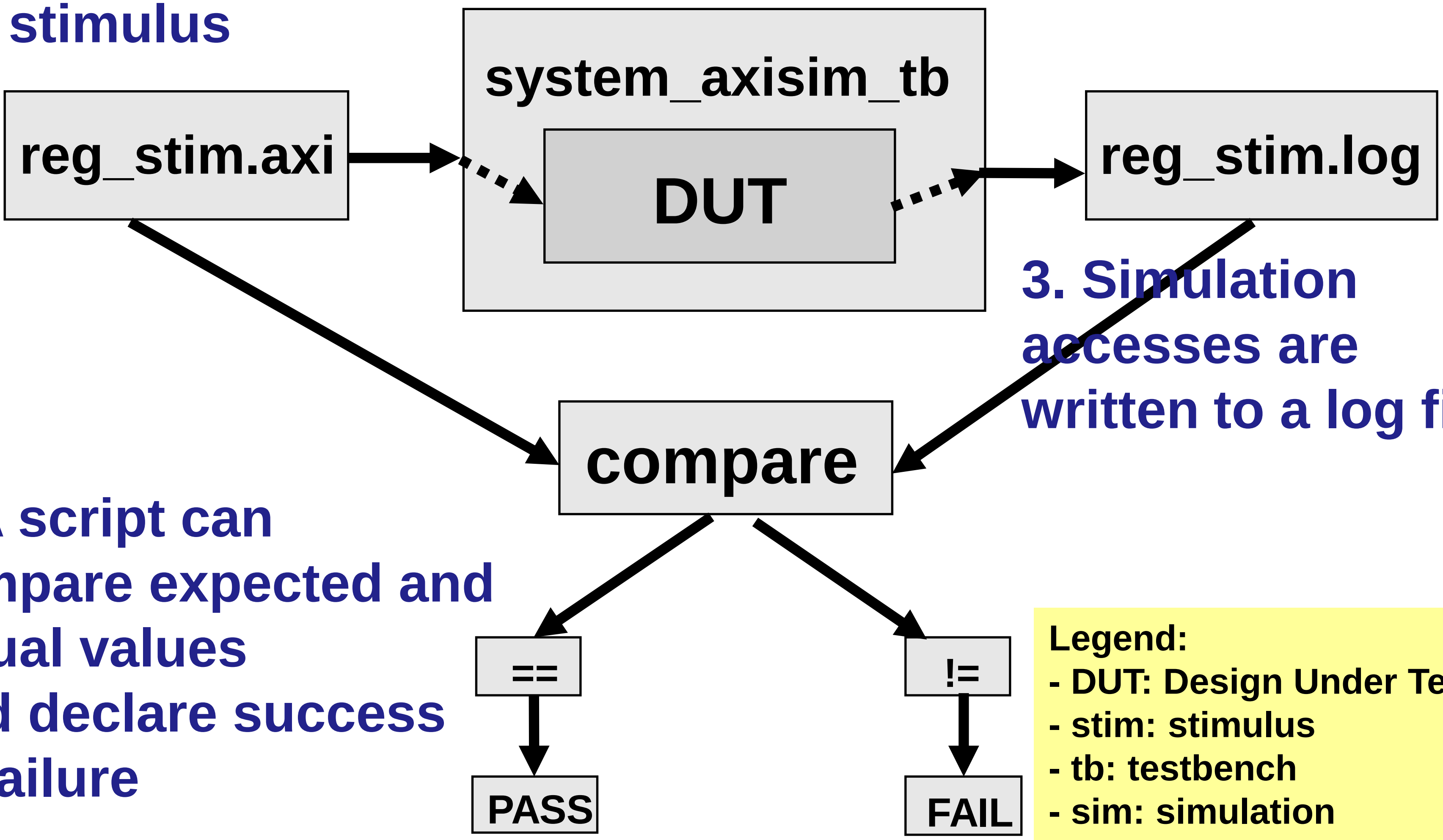
Simulating Register Access

1. Define register stimulus

2. The testbench executes the stimulus

3. Simulation accesses are written to a log file

4. A script can compare expected and actual values And declare success or failure

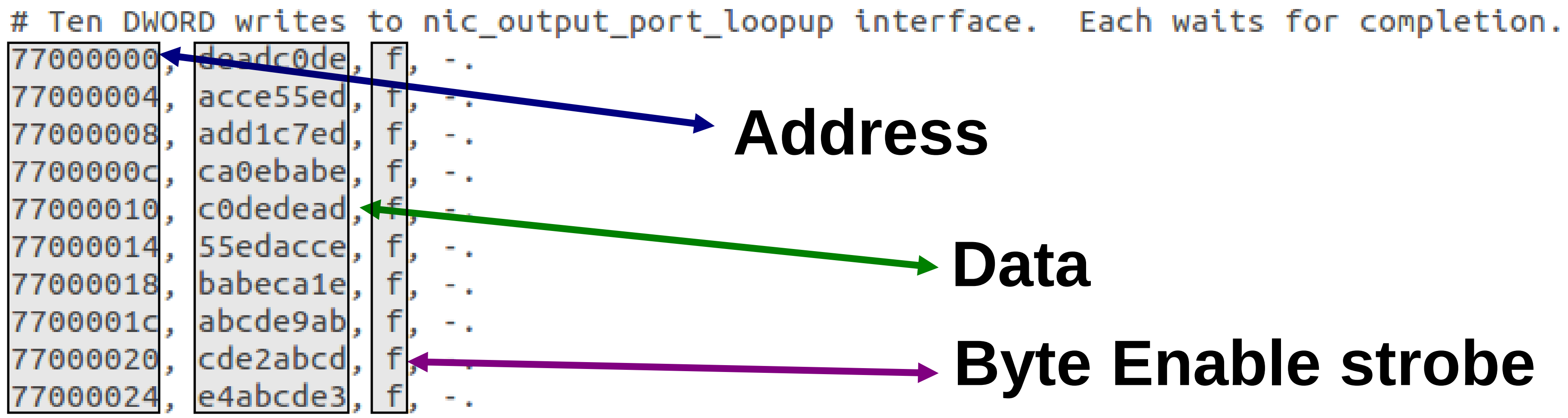


Legend:
- DUT: Design Under Test
- stim: stimulus
- tb: testbench
- sim: simulation

Registers Stimulus (1)

```
cd $NF_DESIGN_DIR/test/
less reg_stim.axi
```

- **An example of write format :**



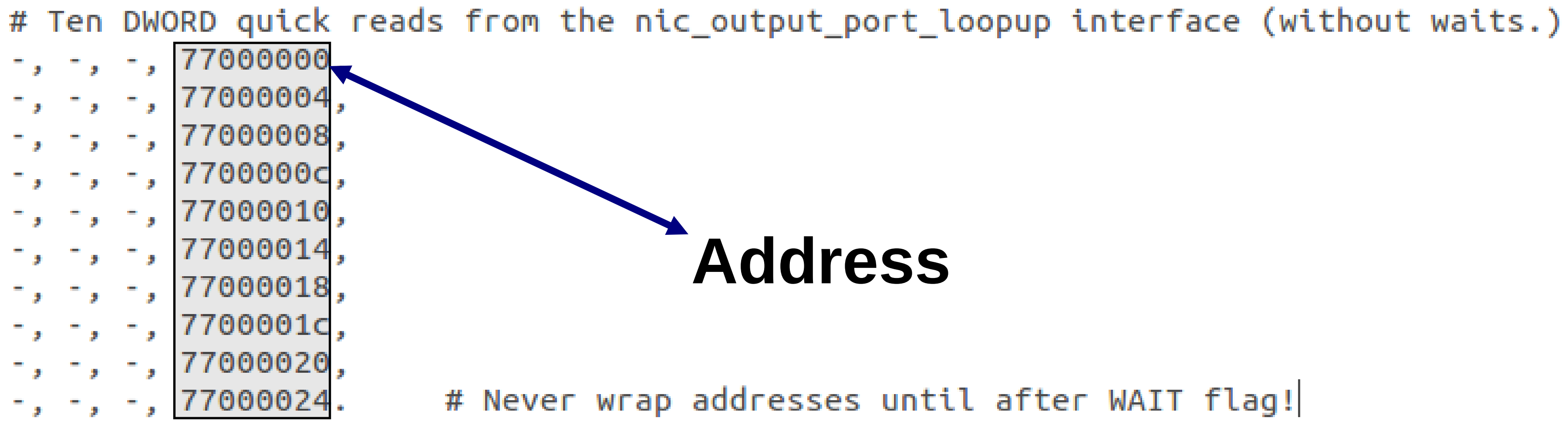
```
# Ten DWORD writes to nic_output_port_loopup interface.  Each waits for completion.
77000000, deadc0de, f, -.
77000004, acce55ed, f, -.
77000008, add1c7ed, f, -.
7700000c, ca0ebabe, f, -.
77000010, c0dedead, f, -.
77000014, 55edacce, f, -.
77000018, babeca1e, f, -.
7700001c, abcde9ab, f, -.
77000020, cde2abcd, f, -.
77000024, e4abcde3, f, -.
```

with other useful information like, time, barriers etc..

Registers Stimulus (2)

```
cd $NF_DESIGN_DIR/test/both_testreg_crypto
less reg_stim.axi
```

- **An example read format :**

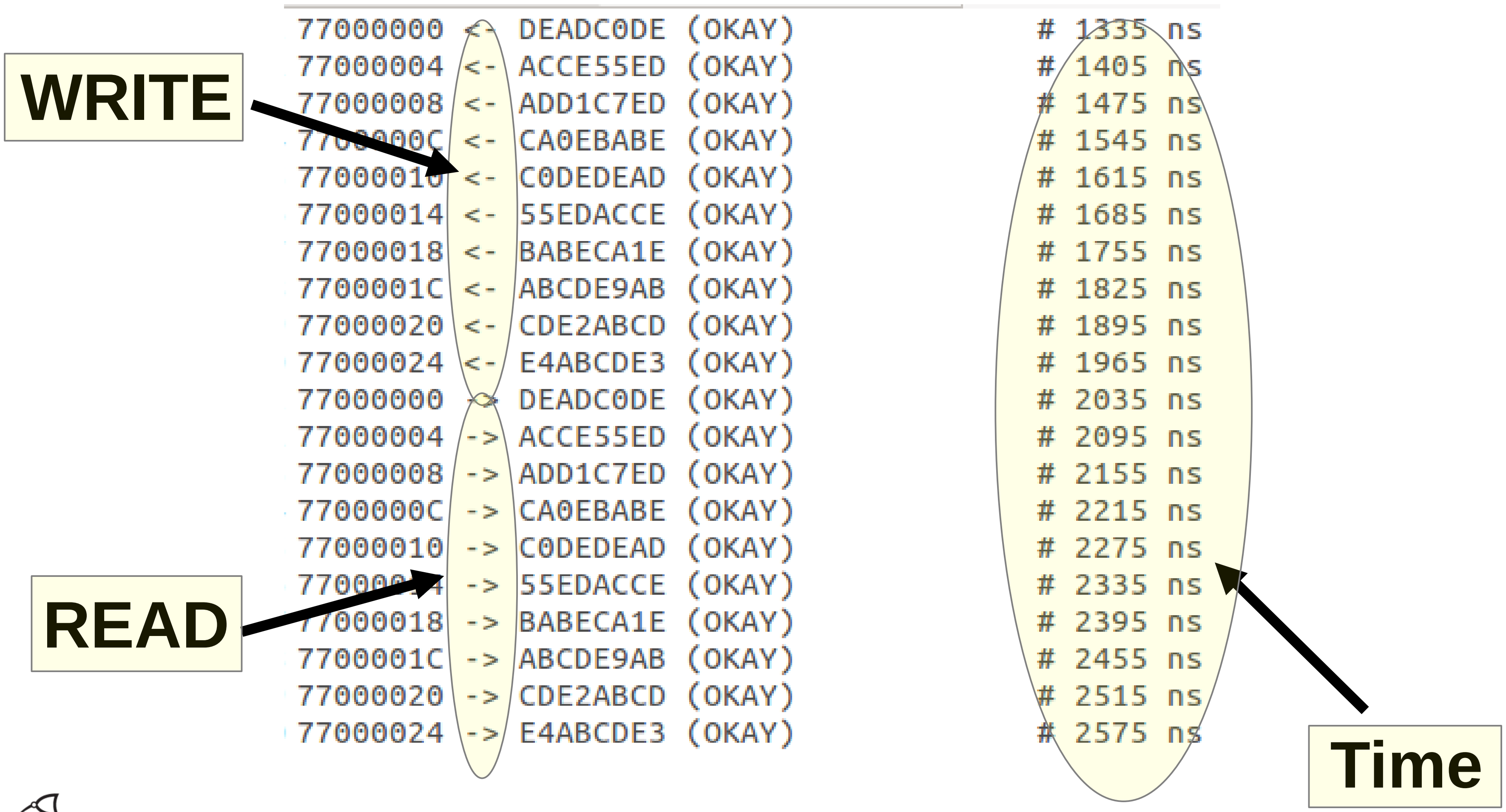


```
# Ten DWORD quick reads from the nic_output_port_loopup interface (without waits.)
-, -, -, 77000000,
-, -, -, 77000004,
-, -, -, 77000008,
-, -, -, 7700000c,
-, -, -, 77000010,
-, -, -, 77000014,
-, -, -, 77000018,
-, -, -, 7700001c,
-, -, -, 77000020,
-, -, -, 77000024.      # Never wrap addresses until after WAIT flag!
```

with other useful information like, time, barriers etc..

Registers Access Log

```
cd $NF_DESIGN_DIR/test/both_testreg_crypto
less reg_stim.log
```



```
77000000 <- DEADC0DE (OKAY)                    # 1335 ns
77000004 <- ACCE55ED (OKAY)                    # 1405 ns
77000008 <- ADD1C7ED (OKAY)                    # 1475 ns
7700000C <- CA0EBABE (OKAY)                    # 1545 ns
77000010 <- C0DEDEAD (OKAY)                    # 1615 ns
77000014 <- 55EDACCE (OKAY)                    # 1685 ns
77000018 <- BABECA1E (OKAY)                    # 1755 ns
7700001C <- ABCDE9AB (OKAY)                    # 1825 ns
77000020 <- CDE2ABCD (OKAY)                    # 1895 ns
77000024 <- E4ABCDE3 (OKAY)                    # 1965 ns
77000000 -> DEADC0DE (OKAY)                    # 2035 ns
77000004 -> ACCE55ED (OKAY)                    # 2095 ns
77000008 -> ADD1C7ED (OKAY)                    # 2155 ns
7700000C -> CA0EBABE (OKAY)                    # 2215 ns
77000010 -> C0DEDEAD (OKAY)                    # 2275 ns
77000014 -> 55EDACCE (OKAY)                    # 2335 ns
77000018 -> BABECA1E (OKAY)                    # 2395 ns
7700001C -> ABCDE9AB (OKAY)                    # 2455 ns
77000020 -> CDE2ABCD (OKAY)                    # 2515 ns
77000024 -> E4ABCDE3 (OKAY)                    # 2575 ns
```

Replacing Static Key

- **In the crypto project, replace the static key with the key from the registers**
  - Use a RWA or a RWS register for your key
  - Assign a default value through the RW default register

- **Hint: Is this a Big or a Little Endian register?**

- **Check in simulation if the system still works correctly**
  - Use both_testreg_crypto

Section II: Build and Test Hardware

Synthesis

- **To synthesize your project:**

```
cd $NF_DESIGN_DIR
make
```

Hardware Tests

- **Test compiled hardware**
- **Test infrastructure provided to**
  - Read/Write registers
  - Read/Write tables
  - Send Packets
  - Check Counters

Python Libraries

- **Start packet capture on interfaces**
- **Clear all tables in hardware**
- **Create packets**
  - MAC header
  - IP header
  - PDU
- **Read/Write registers**
- **Read/Write reference router tables**
  - Longest Prefix Match
  - ARP
  - Destination IP Filter
- **The same libraries used in the simulation infrastructure…**

Creating a Hardware Test

Useful functions:

- Register access:
  - libsume.regwrite(addr, value)
  - libsume.regread_expect(addr, expect)
- Packet generation:
  - make_IP_pkt(…) – see wiki
  - encrypt_pkt(key, pkt)
  - decrypt_pkt(key, pkt)
- Packet transmission/reception:
  - nftest_send_phy(interface, pkt)
  - nftest_expect_phy(interface, pkt)
  - nftest_send_dma(interface, pkt)
  - nftest_expect_dma(interface, pkt)

Understanding Hardware Test

- `cd $NF_DESIGN_DIR/test/both_crypto_test`
- `vim run.py`
- "isHW" indicates HW test
- "connections/conn" file declares the physical connections

```
nf0:eth1
nf1:eth2
nf2:
nf3:
```

- "global/setup" file defines the interfaces

```
proc = Popen(["ifconfig","eth2","192.168.101.1"],
stdout=PIPE)
```

Your task:

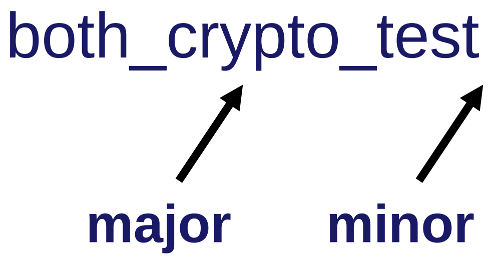
– Remember to source the settings.sh file

– Edit run.py to create your test

– Edit setup and conn files

Running Hardware Tests

- **Use command nf_test.py**
  - Required Parameter
    - sim hw or both (right now only use hw)
  - Optional parameters
    - --major <major_name>
    - --minor <minor_name>



- **Run the command**

```
nf_test.py hw --major regtest --minor crypto
```

Running Hardware Tests

- **Having problems?**

- **Take advantage of the wiki!**

  https://github.com/NetFPGA/NetFPGA-SUME-public/wiki/Hardware-Tests

  – Detailed explanations
  – Tips for debug

Recap

Build a complete NetFPGA design

Learn:

- **Module creation (Verilog)**
- **Reference pipeline integration**
- **Verification via simulation**
- **Verification via hardware tests**
- **Interaction with software**

…and now let’s program the board!!!

Program the NetFPGA

Several options:

- **Program the bit file using Vivado's Hardware Manager**
- **Load a bit file for FPGA programming using impact script**

```
~/NetFPGA-SUME-alpha/tools/scripts/load_bitfile.py \
  -i $DESIGN_DIR/bitfiles/crypto_switch.bit
```

- **Use Xilinx Microprocessor Debugger (XMD)**

```
xmd
fpga –f <filename.bit>
```

WHILE YOU WAIT…. Here is one we built earlier:

```
~/NetFPGA-SUME-alpha/tools/scripts/load_bitfile.py -i \
~/NetFPGA-SUME-alpha/projects/crypto_switch_solution/bitfiles/
crypto_switch.bit
```

Loading the driver

- **Compile SUME driver:**
  - cd ~/NetFPGA-SUME-alpha/lib/sw/std/driver/sume_riffa_v1_0_0
  - make
  - make install
  - modprobe sume

- **Must reset the computer after programming the FPGA**
  - For proper detection and enumeration of PCIe
- **If you already had a running board**
  - cd $SUME_FOLDER/tools/scripts/reconfigure
  - source pci_rescan_run.sh
  - rescans the pcie bus (does not always succeed)

Section III: Projects

Thoughts for Developers

- **Build Modular components**
  - Describe shared registers
  - Consider how modules would be used in larger systems
- **Define functionality clearly**
  - Through regression tests
  - With repeatable results
- **Disseminate projects**
  - Post open-source code
  - Document projects on Web, Wiki
- **Expand the community of developers**
  - Answer questions on the Email list
  - Collaborate with your peers to build new applications

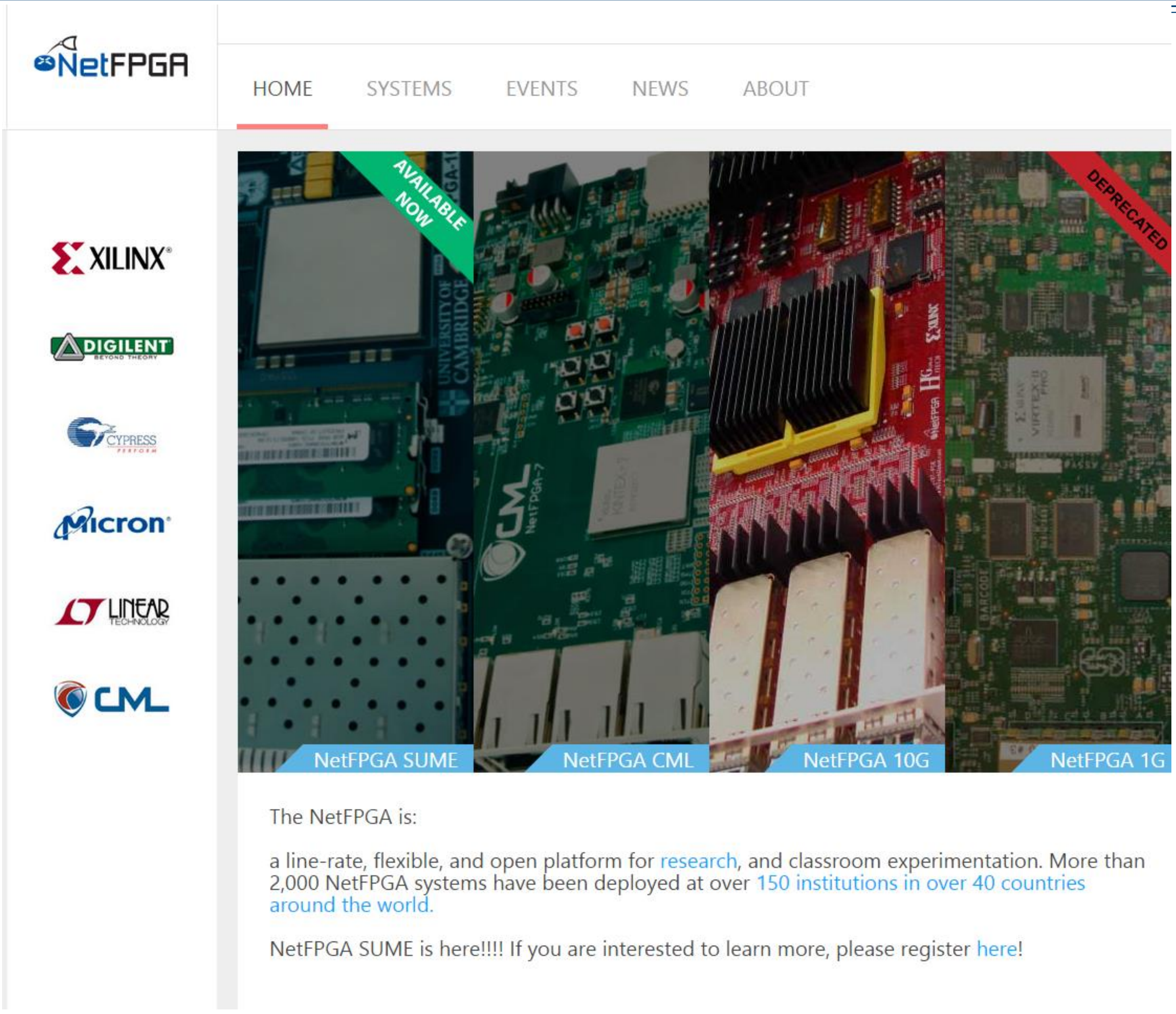
Project Ideas for the NetFPGA

- Drop 1-in-*N* packet module
- SRAM based output queues
- DRAM based output queues
- Streaming data transformations
- Tagging and Memory-mapping packets
- Rate-limiting module
- Input / Output scheduler
- TCAM based reference switch design
- A simple OpenFlow switch
- 40G Port
- ….

Previous Camp Projects

- Payload censoring block
- Simple cool packet generator
- Histogram of size of packets
- Improving the latency of OpenFlow switch
- ACL Firewall
- Stateless NAT
- Dynamic port-based firewall (SW project)
- Circuit breaker (HFT transactions monitor)

Visit http://NetFPGA.org



The NetFPGA is:

a line-rate, flexible, and open platform for research, and classroom experimentation. More than 2,000 NetFPGA systems have been deployed at over 150 institutions in over 40 countries around the world.

NetFPGA SUME is here!!!! If you are interested to learn more, please register here!

- This afternoon is time to try hw testing & synthesis crypto_switch design if you didn’t manage that

- Select your group, leader & project

- Specific, Realistic
- Time-bounded

- Modest is good…..

Acknowledgments (I)

NetFPGA Team at University of Cambridge (Past and Present):

Andrew Moore, David Miller, Muhammad Shahbaz, Martin Zadnik
Matthew Grosvenor, Yury Audzevich, Neelakandan Manihatty-Bojan,
Georgina Kalogeridou, Jong Hun Han, Noa Zilberman, Gianni Antichi,
Charalampos Rotsos, Marco Forconesi, Jinyun Zhang, Bjoern Zeeb

NetFPGA Team at Stanford University (Past and Present):

Nick McKeown, Glen Gibb, Jad Naous, David Erickson,
G. Adam Covington, John W. Lockwood, Jianying Luo, Brandon Heller, Paul Hartke, Neda Beheshti, Sara Bolouki, James Zeng,
Jonathan Ellithorpe, Sachidanandan Sambandan, Eric Lo

All Community members (including but not limited to):

Paul Rodman, Kumar Sanghvi, Wojciech A. Koszek,
Yahsar Ganjali, Martin Labrecque, Jeff Shafer, Eric Keller ,
Tatsuya Yabe, Bilal Anwer, Yashar Ganjali, Martin Labrecque,
Lisa Donatini, Sergio Lopez-Buedo

Kees Vissers, Michaela Blott, Shep Siegel, Cathal McCabe

Acknowledgements (II)

UNIVERSITY OF CAMBRIDGE

XILINX

EPSRC Pioneering research and skills

BROADCOM

Micron

CYPRESS PERFORM

ALGO-LOGIC

DIGILENT

Google

Atomic Rules

Global HiTech

__Disclaimer:__ Any opinions, findings, conclusions, or recommendations expressed in these materials do not necessarily reflect the views of the National Science Foundation or of any other sponsors supporting this project.

This effort is also sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-11-C-0249. This material is approved for public release, distribution unlimited. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.