

Day 2: NetFPGA Cambridge Spring School Module Development and Testing



Presented by:
Andrew W. Moore and David Miller
(University of Cambridge)
Martin Žádník
(Brno University of Technology)

Cambridge UK
March 15-19, 2010

<http://NetFPGA.org>

Outline

- **Tree Structure**
- **Develop a cryptography module**
 - Quick overview of XOR “cryptography”
 - Implement crypto module
 - Write software simulations
 - Synthesize
 - Write hardware tests

Tree Structure

NF2

- bin** (scripts for running simulations and setting up the environment)
- bitfiles** (contains the bitfiles for all projects that have been synthesized)
- lib** (stable common modules and common parts needed for simulation/synthesis/design)
- projects** (user projects, including reference designs)

Tree Structure (2)

lib

- C** (common software and code for reference designs)
- java** (contains software for the graphical user interface)
- Makefiles** (makefiles for simulation and synthesis)
- Perl5** (common libraries to interact with reference designs and aid in simulation)
- python** (common libraries to aid in regression tests)
- scripts** (scripts for common functions)
- verilog** (modules and files that can be reused for design)

Tree Structure (3)

projects

- doc** (project specific documentation)
- include** (contains file to include verilog modules from lib, and creates project specific register defines files)
- regress** (regression tests used to test generated bitfiles)
- src** (contains non-library verilog code used for synthesis and simulation)
- SW** (all software parts of the project)
- synth** (contains user .xco files to generate cores and Makefile to implement the design)
- verif** (simulation tests)

Cryptography

- Simple cryptography – XOR

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Cryptography (cont.)

- **Example:**

Message: 00111011

Key: 10110001

Message \wedge Key: 10001010

Message \wedge Key \wedge Key: 00111011

- **Explanation:**

- $A \wedge A = 0$
- So, $M \wedge K \wedge K = M \wedge 0 = M$

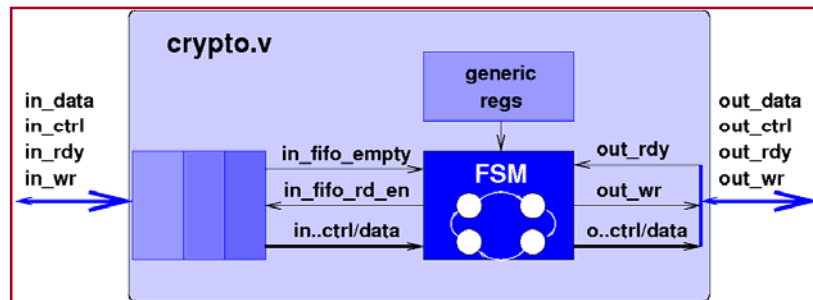
Implementing a Crypto Module (1)

- **What do we want to encrypt?**

- IP payload only
 - Plaintext IP header allows routing
 - Content is hidden
- Encrypt bytes 35 onward
 - Bytes 1-14 – Ethernet header
 - Bytes 15-34 – IPv4 header (assume no options)
- Assume all packets are IPv4 for simplicity

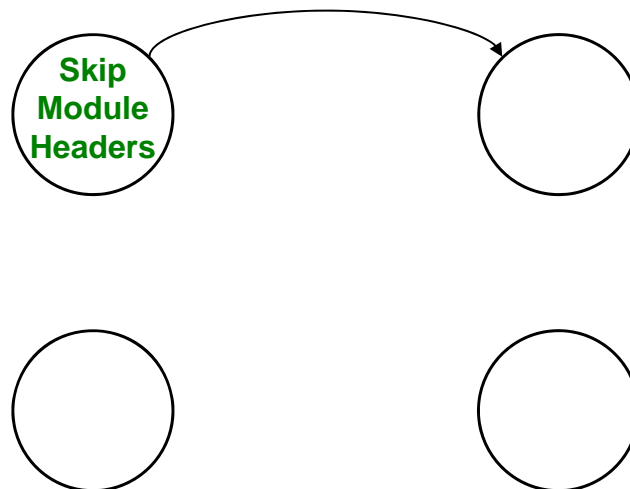
Implementing a Crypto Module (2)

- **State machine (draw on next page):**
 - Module headers on each packet
 - Datapath 64-bits wide
 - 34 / 8 is not an integer! ☹️
- **Inside the crypto module**



Crypto Module State Diagram

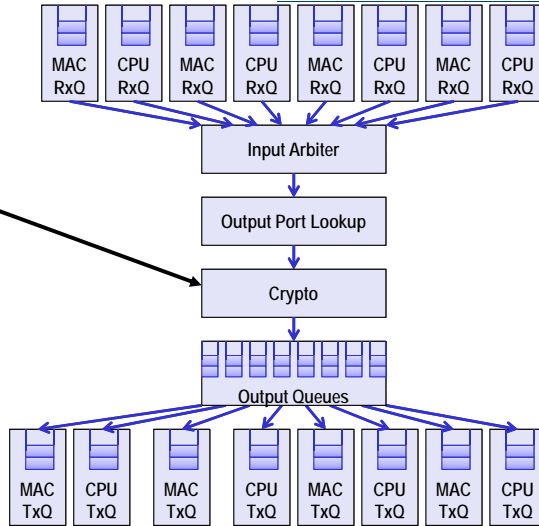
Hint: We suggest 4 operational states(3 if you're feeling adventurous) and one initialization/startup state



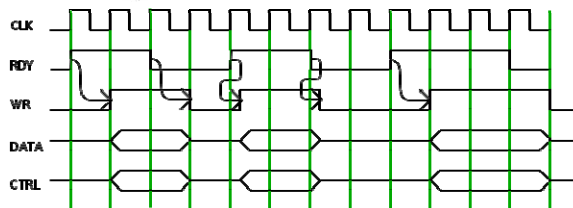
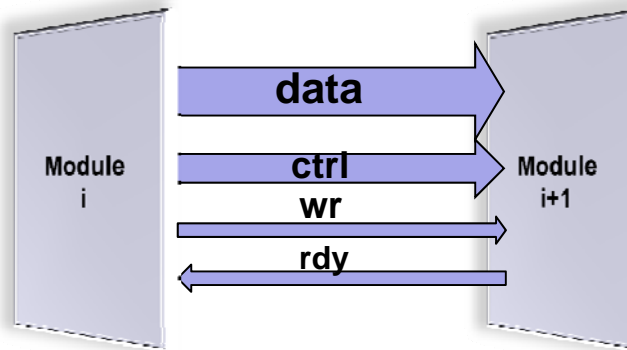
State Diagram to Verilog (1)

Module location

1. **Crypto module** to encrypt and decrypt packets



Inter-module Communication



State Diagram to Verilog (2)

- **Projects:**
 - Each design represented by a project
Format: NF2/projects/<proj_name>
 - NF2/projects/crypto_nic
 - Consists of:
 - Verilog source
 - Libraries
 - Simulation tests
 - Optional software
 - Hardware tests
 - Missing:
 - State diagram implementation
 - Simulation tests
 - Regression tests

State Diagram to Verilog (3)

- **Projects (cont):**
 - Pull in modules from NF2/lib/verilog
 - Generic modules that are re-used in multiple projects
 - Specify shared modules in project's include/lib_modules.txt
 - Local src modules override shared modules
 - crypto_nic:
 - Local user_data_path.v, crypto.v
 - Everything else: shared modules

State Diagram to Verilog (4)

- **Your task:**
 1. Copy
NF2/lib/verilog/module_template/src/module_template.v to
NF2/projects/crypto_nic/src/crypto.v
 2. Implement your state diagram in src/crypto.v
 - Small fallthrough FIFO
 - Generic register interface

Register definitions

- **Register symbols defined using XML** (new in NF2.0)
 1. All register symbols (both Verilog and PERL) controlled by XML definitions
See outputs `include/registers.v` and `lib/Perl5/reg_defines_crypto_nic.pm`
 2. Module specific registers defined in `include/crypto.xml`
 3. Module XML must be explicitly included in project XML definition `include/project.xml`
 4. More at:
http://netfpga.org/foswiki/bin/view/NetFPGA/OneGig/DevelopersGuide#Register_System

Generic Registers Module

```
generic_regs # (  
    .UDP_REG_SRC_WIDTH  (UDP_REG_SRC_WIDTH),  
    .TAG                 (`CRYPTO_BLOCK_ADDR),  
    .REG_ADDR_WIDTH     (`CRYPTO_REG_ADDR_WIDTH),  
    .NUM_COUNTERS       (0),  
    .NUM_SOFTWARE_REGS  (1),  
    .NUM_HARDWARE_REGS  (0))  
crypto_regs (  
    .reg_req_in          (reg_req_in),  
    ...  
    .reg_src_out         (reg_src_out),  
    ...  
    .software_regs       (key),  
    .hardware_regs       (),  
    ...
```

Testing: Simulation (1)

- **Simulation allows testing without requiring lengthy synthesis process**
- **NetFPGA provides Perl simulation infrastructure to:**
 - Send/receive packets
 - Physical ports and CPU
 - Read/write registers
 - Verify results
- **Simulations run in ModelSim/VCS**

Testing: Simulation (2)

- Simulations located in project/verif
- Multiple simulations per project
 - Test different features
- **Example:**
 - crypto_nic/verif/test_nic_short
 - Send one packet from CPU, expect packet out physical port
 - Send one packet in physical port, expect packet to CPU

Testing: Simulation (3)

- **Useful functions:**
 - nf_PCI_read32(delay, batch, addr, expect)
 - nf_PCI_write32(delay, batch, addr, value)
 - nf_packet_in(port, length, delay, batch, pkt)
 - nf_expected_packet(port, length, pkt)
 - nf_dma_data_in(length, delay, port, pkt)
 - nf_expected_dma_data(port, length, pkt)
 - make_IP_pkt(length, da, sa, ttl, dst_ip, src_ip)
 - encrypt_pkt(key, pkt)
 - decrypt_pkt(key, pkt)

Testing: Simulation (4)

- **Your task:**
 1. Template files
 - NF2/projects/crypto_nic/verif/test_crypto_encrypt/make_pkts.pl
 - NF2/projects/crypto_nic/verif/test_crypto_decrypt/make_pkts.pl
 2. Implement your Perl verif tests
 - Use the example verif test (test_nic_short)

Running Simulations

- **Set env. variables to reference your project**
 - NF2_DESIGN_DIR=/root/NF2/projects/<project>
 - PERL5LIB=/root/NF2/projects/<project>/lib/Perl5:/root/NF2/lib/Perl5:
- **Use command nf2_run_test.pl**
 - Optional parameters
 - --major <major_name>
 - --minor <minor_name>
 - --gui (starts the default viewing environment)

test_crypto_encrypt

major

minor

Running Simulations

- **When running modelsim interactively:**
 - Click "no" when simulator prompts to finish
 - Changes to Verilog can be recompiled without quitting ModelSim (unless `make_ppts.pl` has changed):
 - `bash# cd /tmp/$(whoami)/verif/<projname>;
make model_sim`
 - `VSIM 5> restart -f; run -a`
 - Do ensure `$NF2_DESIGN_DIR` is correct
 - `bash# echo $NF2_DESIGN_DIR`

Running Simulations (optional)

- **When running modelsim interactively:**
 - Create new waveform windows:
 - `VSIM 4> view -new wave -title <my_title>`
 - Save waveform windows by clicking 'disc' icon
 - Restore waveform window:
 - `VSIM 5> view -new wave -title <my_title>`
 - `VSIM 6> do <my_wave>.do`

Synthesis

- **To synthesize your project**
 - Run make in the synth directory (NF2/projects/crypto_nic/synth)

Regression Tests

- **Test hardware module**
- **Perl Infrastructure provided to**
 - Read/Write registers
 - Read/Write tables
 - Send Packets
 - Check Counters

Example Regression Tests

- **Reference Router**
 - Send Packets from CPU
 - Longest Prefix Matching
 - Longest Prefix Matching Misses
 - Packets dropped when queues overflow
 - Receiving Packets with IP TTL ≤ 1
 - Receiving Packets with IP options or non IPv4
 - Packet Forwarding
 - Dropping packets with bad IP Checksum

Perl Libraries

- **Specify the Interfaces**
 - eth1, eth2, nf2c0 ... nf2c3
- **Start packet capture on Interfaces**
- **Create Packets**
 - MAC header
 - IP header
 - PDU
- **Read/Write Registers**
- **Read/Write Reference Router tables**
 - Longest Prefix Match
 - ARP
 - Destination IP Filter

Regression Test Examples

- **Reference Router**
 - Packet Forwarding
 - regress/test_packet_forwarding
 - Longest Prefix Match
 - regress/test_lpm
 - Send and Receive
 - regress/test_send_rec

Creating a Regression Test

- **Useful functions:**
 - nftest_regwrite(interface, addr, value)
 - nftest_regread(interface, addr)
 - nftest_send(interface, frame)
 - nftest_expect(interface, frame)
 - encrypt_pkt(key, pkt)
 - decrypt_pkt(key, pkt)

 - \$pkt = NF2::IP_pkt->new(len => \$length,
DA => \$DA, SA => \$SA,
ttl => \$TTL, dst_ip => \$dst_ip,
src_ip => \$src_ip);
 - \$pkt->packed (with nftest_{send,expect})

Creating a Regression Test (2)

- **Your task:**
 1. Template files
NF2/projects/crypto_nic/regress/test_crypto_encrypt/run
 2. Implement your Perl verif tests

Running Regression Test

- **Run the command**
nf2_regress_test.pl --project crypto_nic