NetFPGA Workshop
Day 2





Presented by:

Jad Naous (Stanford University)

Andrew W. Moore (Cambridge University)

Hosted by:
Manolis Katevenis
at
FORTH, Crete
September 16 - 17, 2010

http://NetFPGA.org

Purpose

Build a complete NetFPGA design

Learn:

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

Overview

- **Project: Cryptographic NIC**
- **Infrastructure**
- **Implementation**
- **Simulation and debug**
- **Registers**
- **Build and test hardware**
- **Software integration**

Overview

- **Project: Cryptographic NIC**
- **Infrastructure**
- **Implementation**
- **Simulation and debug**
- **Registers**
- **Build and test hardware**
- **Software integration**

Project: Cryptographic NIC

Implement a network interface card (NIC) that encrypts upon transmission and decrypts upon reception

Cryptography

XOR function

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

XORing a value with itself *always* yields 0

XOR written as: ^ ⊻ ⊕

XOR is *commutative:* (A ^ B) ^ C = A ^ (B ^ C)

Cryptography (cont.)

Simple cryptography:

- Generate a secret key
- Encrypt the message by XORing the message and key
- Decrypt the ciphertext by XORing with the key

Explanation:

(M ^ K) ^ K = M ^ (K ^ K) ← Commutativity

= M ^ 0 ← A ^ A = 0

= M

Cryptography (cont.)

Example:

Message: 00111011

Key: 10110001

Message ^ Key: 10001010

Key: 10110001

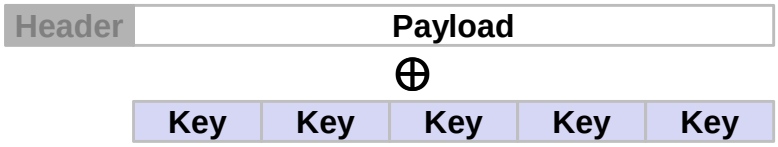
Message ^ Key ^ Key: 00111011

Cryptography (cont.)

Idea: Implement simple cryptography using XOR

– 32-bit key

– Encrypt every word in payload with key



Overview

- Project: Cryptographic NIC
- **Infrastructure**
- Implementation
- Simulation and debug
- Registers
- Build and test hardware
- Software integration

Infrastructure

- **NetFPGA package contents**
  - Reusable Verilog modules
  - Verification infrastructure
  - Build infrastructure
  - Utilities
  - Software libraries
- **Tree structure**

NetFPGA package contents

- **Projects:**
  - HW: router, switch, NIC, buffer sizing router
  - SW: router kit, SCONE
- **Reusable Verilog modules**
- **Verification infrastructure:**
  - simulate full board with PCI + physical interfaces
  - run tests against hardware
  - test data generation libraries (eg. packets)
- **Build infrastructure**
- **Utilities:**
  - register I/O, packaging, …
- **Software libraries**

Reusable Verilog modules

Category	Modules	
I/O interfaces	Ethernet MAC CPU DMA queues CPU register queues	MDIO PCI
Output queues	SRAM-based DRAM-based	BRAM-based
Output port lookup	Router (CAM-based) Learning switch (CAM-based)	NIC Hardwired
Memory interfaces	SRAM	DRAM
Miscellaneous	FIFOs Generic register module	Rate limiter

Verification infrastructure

- **Simulation: `nf_run_test.pl`**
  - allows testing *before* synthesis
  - catches many bugs
- **Hardware tests: `nf_regress_test.pl`**
  - test synthesized hardware
- **Test data generation libraries:**
  - easily create test data:
  - many standard packet formats supported out of the box
  - easily add support for custom formats

Build infrastructure

- **Register system:**
  - allocates memory to modules
  - generates “include” files for various languages
- **Build/synthesis:**
  - required shared modules documented XML (shared with register system)
  - shared modules pulled in during synthesis
  - resultant bitfile checked for timing errors

Utilities

- **Bitfile download:** `nf_download`
- **Register I/O:** `regread`, `regwrite`
- **Device querying:** `nf_info`
- **SRAM dumping:** `lib/scripts/sram_dump`

Software libraries

- **Libraries for interfacing with NetFPGA:**
  - C, Perl, Java, partial Python support

Tree Structure (1)

netfpga

- **bin** (scripts for running simulations and setting up the environment)
- **bitfiles** (contains the bitfiles for all projects that have been synthesized)
- **lib** (shared Verilog modules, libraries 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** (libraries to interact with reference designs, create test data, and manage simulations/regression tests)
- **python** (common libraries to aid in regression tests)
- **scripts** (utility scripts – less commonly used than those in the bin directory)
- **verilog** (modules that can be reused in designs)

Tree Structure (3)

projects/crypto_nic

- **doc** (project specific documentation)
- **include** (XML files defining project and any local modules, auto-generated Verilog register defines)
- **lib** (C/Perl defines for registers)
- **regress** (regression tests to test generated bitfiles)
- **src** (non-library Verilog code used for synthesis and simulation)
- **sw** (software elements of the project)
- **synth** (project-specific .xco files to generate cores, Makefile to implement the design)
- **verif** (simulation tests)

Overview

- Project: Cryptographic NIC
- Infrastructure
- **Implementation**
- Simulation and debug
- Registers
- Build and test hardware
- Software integration

Getting started with a new project (1)

- **Projects:**
  - Each design represented by a project
  - Location: netfpga/projects/<proj_name>
    - netfpga/projects/crypto_nic
  - Consists of:
    - Verilog source
    - Simulation tests
    - Hardware tests
    - Libraries
    - Optional software

Getting started with a new project (2)

– Normally:
  - copy an existing project as the starting point

– Today:
  - pre-created project

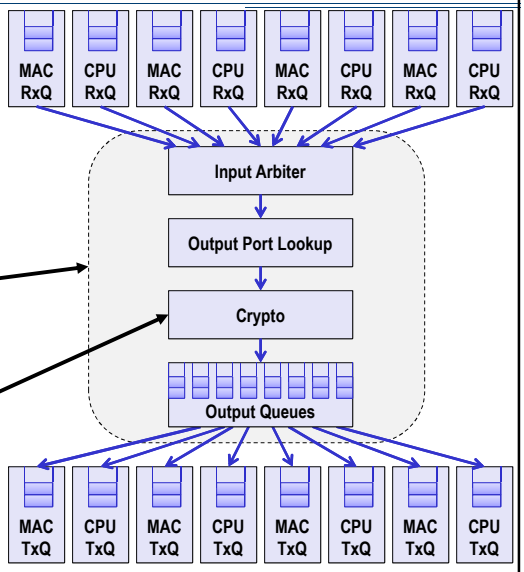
– Missing from pre-created project:
  - Verilog files (with crypto implementation)
  - Simulation tests
  - Hardware tests
  - Custom software

Getting started with a new project (3)

Typically implement functionality in one or more modules *inside* the user data path

User data path

Crypto module to encrypt and decrypt packets



Getting started with a new project (4)

– Shared modules included from netfpga/lib/verilog
  - Generic modules that are re-used in multiple projects
  - Specify shared modules in project's include/project.xml

– Local src modules override shared modules

– crypto_nic:

Local	Shared
user_data_path.v crypto.v	Everything else

Exploring project.xml (1)

- **Location: project/<proj_name>/include**

```
<?xml version="1.0" encoding="UTF-8"?>
<nf:project …>

  <nf:name>Crypto NIC</nf:name>

  <nf:description>NIC with basic crypto support</nf:description>

  <nf:version_major>0</nf:version_major>
  <nf:version_minor>1</nf:version_minor>
  <nf:version_revision>0</nf:version_revision>

  <nf:dev_id>0</nf:dev_id>
```

Short name

Description

Version information
- **indicate bitfile version**

Unique ID to identify project
See: http://netfpga.org/foswiki/bin/view/NetFPGA/OneGig/DeviceIDList

Exploring project.xml (2)

```
<nf:use_modules>
   core/io_queues/cpu_dma_queue
   core/io_queues/ethernet_mac
   core/input_arbiter/rr_input_arbiter
   core/nf2/generic_top
   core/nf2/reference_core
   core/output_port_lookup/nic
   core/output_queues/sram_rr_output_queues
   core/sram_arbiter/sram_weighted_rr
   core/user_data_path/reference_user_data_path
   core/io/mdio
   core/cpci_bus
   core/dma
   core/user_data_path/udp_reg_master
   core/io_queues/add_rm_hdr
   core/strip_headers/keep_length
   core/utils/generic_regs
   core/utils
 </nf:use_modules>
```

Shared modules to load from lib/verilog

Exploring project.xml (3)

```
<nf:memalloc layout="reference">
   <nf:group name="core1">
     <nf:instance name="device_id" />
     <nf:instance name="dma" base="0x0500000"/>
     <nf:instance name="mdio" />
     <nf:instance name="nf2_mac_grp" count="4" />
     <nf:instance name="cpu_dma_queue" count="4" />
   </nf:group>
   <nf:group name="udp">
     <nf:instance name="in_arb" />
     <nf:instance name="crypto" />
     <nf:instance name="strip_headers" />
     <nf:instance name="output_queues" />
   </nf:group>
 </nf:memalloc>
</nf:project>
```

Specify where to instantiate modules, the number of instances, and the memory addresses to use

Getting started with a new project (5)

Tasks:

Set the project that we'll be working with:

1. Add the following lines to the end of ~/.bashrc:

```
export NF_DESIGN_DIR=$NF_ROOT/projects/crypto_nic
export
    PERL5LIB=$NF_ROOT/lib/Perl5:$NF_DESIGN_DIR/lib/Per
    l5
```

2. Type: source ~/.bashrc

Copy reference files as starting points:

3. Copy the following files from netfpga/lib/verilog/core into netfpga/projects/crpyto_nic/src

```
user_data_path/reference_user_data_path/src/user_data_path.v
module_template/src/module_template.v
```


Getting started with a new project (6)

Create crypto.v from module_template.v:

1. Rename the *local* module_template.v to crypto.v
2. Change the module name *inside* crypto.v (first non-comment line of the file)

3. Add the crypto module to the user data path

user_data_path.v (1)

```
module user_data_path
   #(
      parameter DATA_WIDTH = 64,
      ...
   )
   (
      ...
   )

   //------------------ Internal parameters ----------------------
   ...

   //----------------- Input arbiter wires/regs ------------------
   ...
```

Module port declaration

user_data_path.v (2)

```
   //-------------- output port lut wires/regs --------------------
   wire [CTRL_WIDTH-1:0]            op_lut_in_ctrl;
   wire [DATA_WIDTH-1:0]            op_lut_in_data;
   wire                             op_lut_in_wr;
   wire                             op_lut_in_rdy;

   ...

   //------- output queues wires/regs ------
   ...
```

Wire declarations for the output port lookup module.
Duplicate this section, and replace op_lut with crypto

user_data_path.v (3)

```
//--------- Connect the data path -----------
input_arbiter #(
   ...
) input_arbiter (
   ...
)

output_port_lookup #(
   ...
) output_port_lookup (
   ...
)

...
```

Module instantiations.

1. Duplicate the output_port_lookup instantiation
2. Rename to crypto
3. Remove all parameters (inside the first set or parentheses)
4. In the output_port_lookup instantiation, replace oq_ with crypto_
5. In the crypto instantiation, replace op_lut_ with crypto_

We've inserted the new module into the pipeline

Getting started with a new project (7)

Run a simulation to verify changes:

1. nf_run_test.pl --major nic --minor short

Now we can implement the crypto functionality

More Verilog: Assignments 1

- **Continuous assignments**
  - appear *outside* processes (`always @` blocks):

```
assign foo = baz & bar;
```

  - targets must be declared as `wires`
  - always “happening” (*ie*, are concurrent)

More Verilog: Assignments 2

- **Non-blocking assignments**
  - appear *inside* processes (`always @` blocks)
  - use only in *sequential* (clocked) processes:

```
always @(posedge clk) begin
    a <= b;
    b <= a;
end
```

  - occur in next *delta* (‘moment’ in simulation time)
  - targets must be declared as `regs`
  - never clock any process other than with a clock!

More Verilog: Assignments 3

- **Blocking assignments**
  - appear *inside* processes (`always @` blocks)
  - use only in *combinatorial* processes:
    - (combinatorial processes are much like continuous assignments)

```
always @(*) begin
        a = b;
        b = a;
end
```



  - occur one after the other (as in sequential langs like C)
  - targets must be declared as `regs` – even though not a register
  - never use in sequential (clocked) processes!

More Verilog: Assignments 3

- Blocking assignments
  - appear *inside* processes (`always @` blocks)
  - use only in *combinatorial* processes:
    - (combinatorial processes are much like continuous assignments)

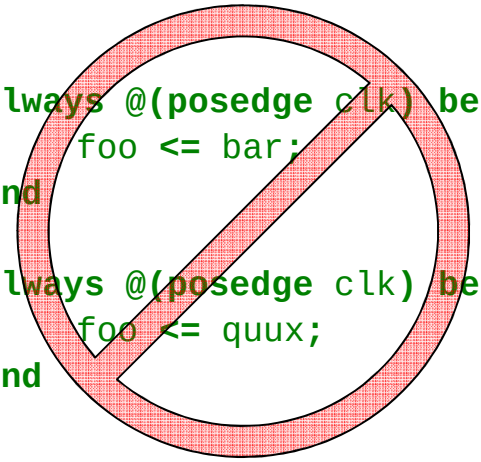
```
always @(*) begin
        tmp = a;
        a = b;
        b = tmp;
end
```

unlike non-blocking, have to use a temporary signal

  - occur one after the other (as in sequential langs like C)
  - targets must be declared as `regs` – even though not a register
  - never use in sequential (clocked) processes!

(hints)

- **Never assign one signal from two processes:**

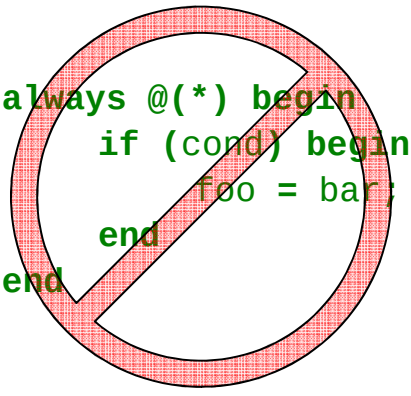


```
always @(posedge clk) begin
    foo <= bar;
end

always @(posedge clk) begin
    foo <= quux;
end
```


(hints)

- **In combinatorial processes:**
  - take great care to assign in all possible cases



```
always @(*) begin
    if (cond) begin
        foo = bar;
    end
end
```

  - (latches ‹as opposed to flip-flops› are bad for timing closure)

(hints)

- **In combinatorial processes:**
  - take great care to assign in all possible cases

```
always @(*) begin
     if (cond) begin
          foo = bar;
     else
          foo = quux;
     end
end
```


(hints)

- **In combinatorial processes:**
  - (or assign a default)

```
always @(*) begin
     foo = quux;

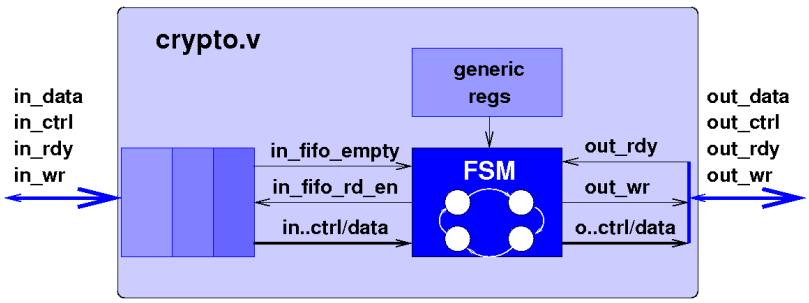
     if (cond) begin
          foo = bar;
     end
end
```

Implementing the Crypto Module (1)

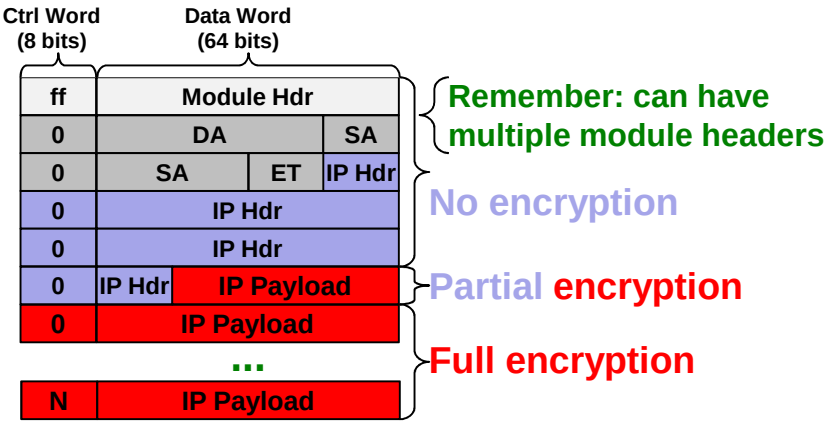
- **What do we want to encrypt?**
  - IP payload only
    - Plaintext IP header allows routing
    - Content is hidden
  - Encrypt bytes 35 onward
    - Bytes 1-14 – Ethernet header
    - Bytes 15-34 – IPv4 header (assume no options)
  - Assume all packets are IPv4 for simplicity

Implementing the 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**



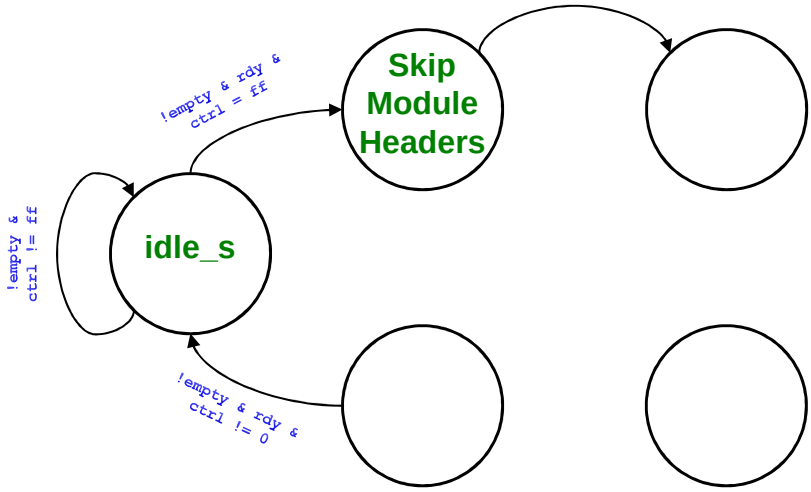
Example packet



N = 0x80 >> (vld_bytes - 1)

Crypto Module State Diagram

Hint: We suggest 5 operational states (4 if you're feeling adventurous) plus one initialization/startup state



Implementing the Crypto Module (3)

Implement your state machine inside crypto.v

- Use a static key initially

Suggested sequence of steps:

1. Create a static key value
   - Constants can be declared in the module with localparam:
     `localparam MY_EXAMPLE = 32'h01234567;`
2. Implement your state machine without modifying the packet
3. Update your state machine to modify the packet by XORing the key and the payload
   - Use two copies of the key to create a 64-bit value to XOR with data words

module_template.v (1)

```
module module_template
   #(
      parameter DATA_WIDTH = 64,
      parameter CTRL_WIDTH = DATA_WIDTH/8,
      parameter UDP_REG_SRC_WIDTH = 2
   )
   (
      ...
   )

   //------------------------ Signals-------------------------------
   ...

   //------------------ Local assignments -----------------------
   ...
```

Module port declaration

module_template.v (2)

```
//------------------------- Modules-------

  fallthrough_small_fifo #(
     .WIDTH(CTRL_WIDTH+DATA_WIDTH),
     .MAX_DEPTH_BITS(2)
  ) input_fifo (
     .din           ({in_ctrl, in_data}),   // Data in
     .wr_en         (in_wr),                // Write enable
     .rd_en         (in_fifo_rd_en),        // Read the next word
     .dout          ({in_fifo_ctrl, in_fifo_data}),
     .full          (),
     .nearly_full   (in_fifo_nearly_full),
     .prog_full     (),
     .empty         (in_fifo_empty),
     .reset         (reset),
     .clk           (clk)
  );
```

Packet data dumped in a FIFO. Allows some "decoupling" between input and output.

module_template.v (3)

```
generic_regs
  #(
     .UDP_REG_SRC_WIDTH   (UDP_REG_SRC_WIDTH),
     .TAG                 (0),
     .REG_ADDR_WIDTH      (1),
     .NUM_COUNTERS        (0),
     .NUM_SOFTWARE_REGS   (0),
     .NUM_HARDWARE_REGS   (0)
  ) module_regs (
     ...
  );
```

Generic registers.

Ignore for now – we'll explore this later

module_template.v (4)

```
//------------------------ Logic-------------------------------

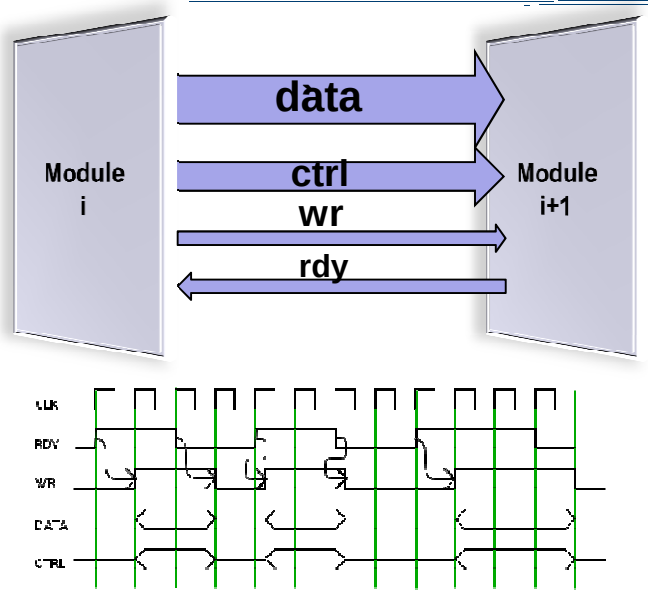
  always @(*) begin
     // Default values
     out_wr_int = 0;
     in_fifo_rd_en = 0;

     if (!in_fifo_empty && out_rdy) begin
        out_wr_int = 1;
        in_fifo_rd_en = 1;
     end
  end
```

Combinational logic to read data from the FIFO. (Data is output to output ports.)

You'll want to add your state in this section.

Inter-module Communication



Implementing the Crypto Module (3)

Implement your state machine inside crypto.v

– Use a static key initially

Suggested sequence of steps:

1. Create a static key value
   - Constants can be declared in the module with localparam:
     `localparam MY_EXAMPLE = 32'h01234567;`
2. Implement your state machine without modifying the packet
3. Update your state machine to modify the packet by XORing the key and the payload
   - Use two copies of the key to create a 64-bit value to XOR with data words

First Break

(Write your Verilog module)

Overview

- Project: Cryptographic NIC
- Infrastructure
- Implementation
- **Simulation and debug**
- Registers
- Build and test hardware
- Software integration

Testing: Simulation (1)

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

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

  Note: This test will ***not*** work once your crypto module is implemented!

Testing: Simulation (3)

Useful functions:

Always set batch to 0

Register access:

nf_PCI_read32(delay, batch, addr, expect)
nf_PCI_write32(delay, batch, addr, value)

Packet generation:

make_IP_pkt(length, da, sa, ttl, dst_ip, src_ip)
encrypt_pkt(key, pkt)
decrypt_pkt(key, pkt)

Packet transmission/reception:

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)

Testing: Simulation (4)

Task:

Implement tests for encryption and decryption

Modify the following tests:

netfpga/projects/crypto_nic/verif/test_crypto_encrypt/make_pkts.pl
netfpga/projects/crypto_nic/verif/test_crypto_decrypt/make_pkts.pl

Look at test_nic_short as an example of creating IP packets and sending/receiving them

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**

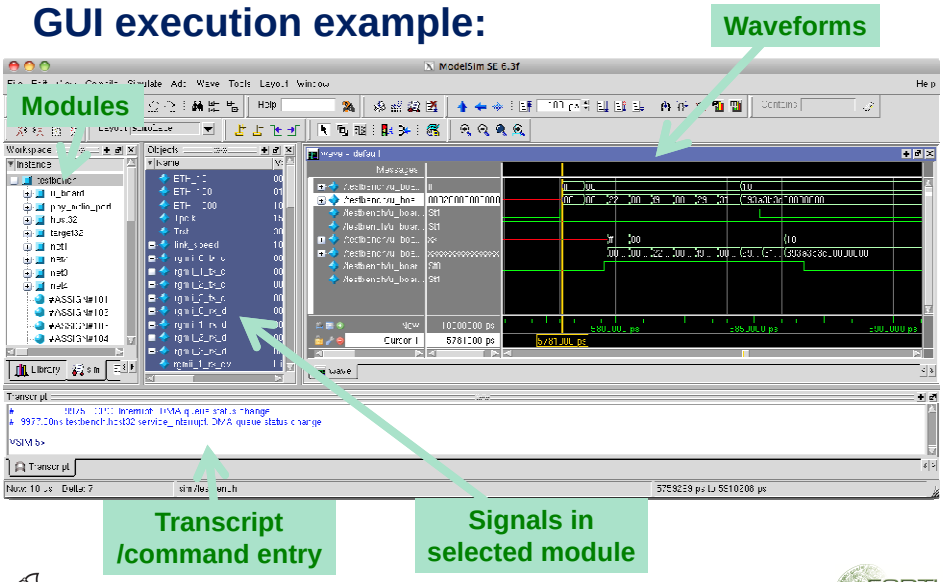
Running Simulations

Non-GUI execution example:

```
# 10756.00ns testbench.host32.service_interrupt: Info: Interrupt signaled
#               10935   Host read  0x00000044 with cmd 0x6:  Disconnect with Data,
#               10995   CPCI Interrupt:  DMA ingress xfer complete
#               11175   Host read  0x00000148 with cmd 0x6:  Disconnect with Data,
#               11415   Host read  0x00000150 with cmd 0x6:  Disconnect with Data,
# 11475.00ns testbench.host32.service_interrupt: Info: DMA ingress transfer complete.
#               11655   Host read  0x00000040 with cmd 0x6:  Disconnect with Data,
# Timecheck: 13645.00ns
#               20100 Simulation has reached finish time - ending.
# ** Note: $finish    : /home/summercamp/netfpga/lib/verilog/core/testbench/target32.v
#    Time: 20100 ns  Iteration: 0  Instance: /testbench/target32
--- Simulation is complete. Validating the output.
    Comparing simulation output for port 1 ...
    Port 1 matches [1 packets]
    Comparing simulation output for port 2 ...
    Port 2 matches [0 packets]
--- Test PASSED (test_nic_short)
Test test_nic_short passed!
------------SUMMARY--------------
PASSING TESTS:
          test_nic_short
FAILING TESTS:
TOTAL: 1 PASS: 1  FAIL: 0
```


Running Simulations

GUI execution example:



Running Simulations

GUI execution example (cont)

Try the following:

nf_run_test.pl --major crypto --minor encrypt –gui

In the transcript window of the GUI:

do wave.do

run 10us

You should see waveforms of packets going in and coming out of the crypto module

Running Simulations

- **When running modelsim interactively:**
  - Click "no" when simulator prompts to finish
  - Changes to Verilog can be recompiled without quitting ModelSim (unless `make_pkts.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

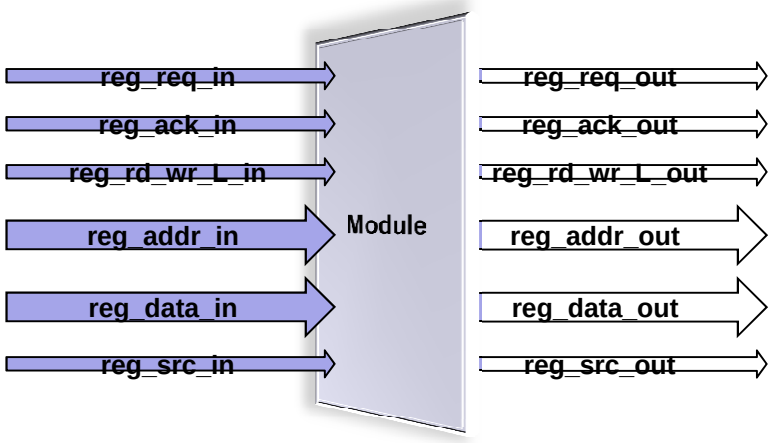
Overview

- Project: Cryptographic NIC
- Infrastructure
- Implementation
- Simulation and debug
- **Registers**
- Build and test hardware
- Software integration

Specifying the key via a register

- **Can set the key via a register instead**
- **Need to understand the register system ☺**
- **Register system:**
  - Specify registers provided by module in the module XML file
  - Implement registers in module
    - Can usually use generic_regs

Register bus



Module XML file (1)

- **Each module (with registers) has an XML file**

```
<?xml version="1.0" encoding="UTF-8"?>
<nf:module ...>
   <nf:name>crypto</nf:name>
   <nf:description>Registers for Crypto Module</nf:description>

   <nf:prefix>crypto</nf:prefix>

   <nf:location>udp</nf:location>

   <nf:blocksize>64</nf:blocksize>
```

Name/description

Prefix appears before register names in source code

Location: where in the design should this module be instantiated?
udp = user data path

Amount of memory to allocate to the block (in bytes, use k/m to indicate kilo/megabytes)

Module XML file (2)

```
   <nf:registers>

      <nf:register>
         <nf:name>key</nf:name>
         <nf:description>The Key value used by the Crypto
    Module</nf:description>
         <nf:type>generic_software32</nf:type>
      </nf:register>

   </nf:registers>

   <nf:constants>
   </nf:constants>

   <nf:types>
   </nf:types>
</nf:module>
```

Register declaration: need name, description, and width or type

Can also declare constants and data types

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    (),
      ...
```

Make sure you declare key as a 32-bit wire

Replacing static key with register

- Replace the static key with the key from the registers
- Update your simulations to set the key

Overview

- Project: Cryptographic NIC
- Infrastructure
- Implementation
- Simulation and debug
- Registers
- **Build and test hardware**
- Software integration

Synthesis

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

Second Break

(while hardware compiles)

Hardware Tests

- **Test compiled hardware**
- **Test 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 = NF::IP_pkt->new(len => $length,
    DA => $DA, SA => $SA,
    ttl => $TTL, dst_ip => $dst_ip,
    src_ip => $src_ip);

Creating a Regression Test (2)

- **Your task:**

  1. Template files
     netfpga/projects/crypto_nic/regress/test_crypto_encrypt/run

  2. Implement your hardware tests

Running Regression Test

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

Third Break

(while testing hardware)

Overview

- Project: Cryptographic NIC
- Infrastructure
- Implementation
- Simulation and debug
- Registers
- Build and test hardware
- **Software integration**

Interface with software

- **Write two simple utilities:**
  - getkey – get the current key and print it
  - setkey – set the key to a value specified on the command line
  - skeleton C files in the sw directory
  - build with ‘make’
- **Two functions:**

```
readReg(nf2device *dev, int address, unsigned *rd_data);
writeReg(nf2device *dev, int address, unsigned *wr_data);
```

Recap

Build a complete NetFPGA design

Learn:

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

WRAP-UP

NetFPGA.org



Project Ideas for the NetFPGA

- IPv6 Router (in high demand)
- TCP Traffic Generator
- Valiant Load Balancing
- Graphical User Interface (like CLACK)
- MAC-in-MAC Encapsulation
- Encryption / Decryption modules
- RCP Transport Protocol
- Packet Filtering (Firewall, IDS, IDP)
- TCP Offload Engine
- DRAM Packet Queues
- 8-Port Switch using SATA Bridge
- Build our own MAC (from source, rather than core)
- Use XML for Register Definitions

http://www.netfpga.org/foswiki/NetFPGA/OneGig/ModuleWishlist

Group Discussion

- **Your plans for using the NetFPGA**
  - Teaching
  - Research
  - Other
- **Resources needed for your class**
  - Source code
  - Courseware
  - Examples
- **Your plans to contribute**
  - Expertise
  - Capabilities
  - Collaboration Opportunities

Sample of NetFPGA Designs

Project (Title & Summary)	Base	Status	Organization	Docs.
IPv4 Reference Router	2.0	Functional	Stanford University	Guide
Quad-Port Gigabit NIC	2.0	Functional	Stanford University	Guide
Ethernet Switch	2.0	Functional	Stanford University	Guide
Hardware-Accelerated Linux Router	2.0	Functional	Stanford University	Guide
Packet Generator	2.0	Functional	Stanford University	Wiki
OpenFlow Switch	2.0	Functional	Stanford University	Wiki
DRAM-Router	2.0	Functional	Stanford University	Wiki
NetFlow Probe	1.2	Functional	Brno University	Wiki
AirFPGA	2.0	Functional	Stanford University	Wiki
Fast Reroute & Multipath Router	2.0	Functional	Stanford University	Wiki
NetThreads	1.2.5	Functional	University of Toronto	Wiki
URL Extraction	2.0	Functional	Univ. of New South Wales	Wiki
zFilter Sprouter (Pub/Sub)	1.2	Functional	Ericsson	Wiki
Windows Driver	2.0	Functional	Microsoft Research	Wiki
IP Lookup w/Blooming Tree	1.2.5	In Progress	University of Pisa	Wiki
DFA	2.0	In Progress	UMass Lowell	Wiki
G/PaX	?.?	In Progress	Xilinx	Wiki
Precise Traffic Generator	1.2.5	In Progress	University of Toronto	Wiki
Open Network Lab	2.0	In Progress	Washington University	Wiki
KOREN Testbed	?.?	In Progress	Chungnam-Korea	Wiki
RED	2.0	In Progress	Stanford University	Wiki
Virtual Data Plane	1.2	In Progress	Georgia Tech	Wiki
Precise Time Protocol (PTP)	2.0	In Progress	Stanford University	Wiki
Deficit Round Robin (DRR)	1.2	Repackage	Stanford University	Wiki

.. And more on http://netfpga.org/foswiki/NetFPGA/OneGig/ProjectTable

Thoughts for Developers

- **Build Modular components**
  - Describe shared registers (as per 2.0 release)
  - 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 in the Discussion Forum
  - Collaborate with your peers to build new applications

NetFPGA Developments

- **Next generation NetFPGA**
- **Open Source Network Hardware Community**

NetFPGA 10G

- Virtex-5 TXT 240
- 4 x 10Gbps SFP+
- x8 PCI-Express gen 1 and 2
- 2.3Gbit RLDRAM
- 864Mbit QDR II



- Officially signed off by Xilinx and available for pre-order from http://hitechglobal.com/Boards/PCIExpress_SFP+.htm

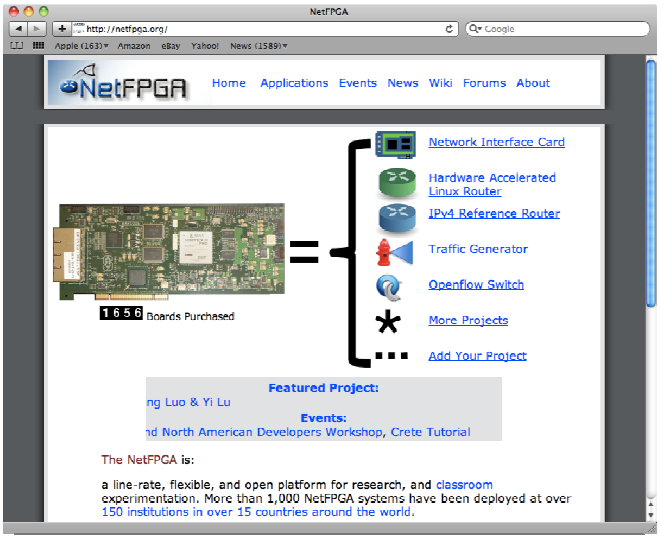
Implication

- NetFPGA 1G:
  - Usually produce single-function reference designs
  - Typically based on IP router example
- NetFPGA 10G
  - Space to build more complex systems
  - More motivation for repository of reusable blocks
    - Get some components from the repository
    - Contribute new components to the repository

Open Source Network Hardware Community

- Enable people to build interesting networking systems with FPGA-based implementations
- Allow people to focus on their particular areas of networking expertise and interest
- Provide lightweight coordination to share expertise in a systematic way

Visit http://NetFPGA.org



Survey

- **How did you like this this tutorial?**
  - What did you find useful?
  - What should be improved?
  - What should be removed?
  - What should be added?
- **Can we post the video from this event?**
  - If not, please let us know.
- **Complete On-line survey**
  - http://netfpga.org/tutorial_survey.html

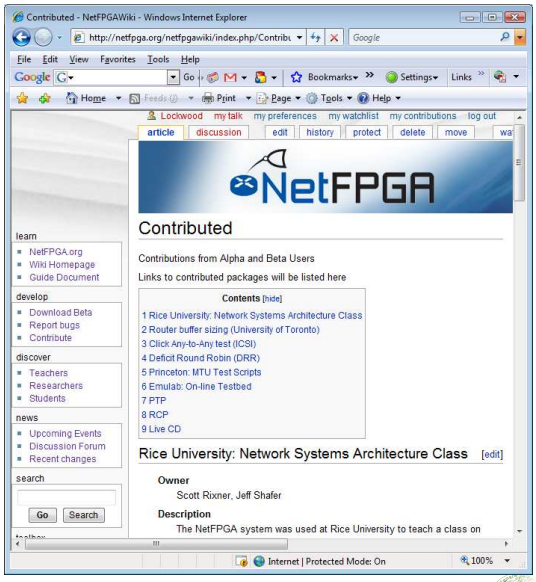
Join the NetFPGA.org Community

- **Log into the Wiki**
- **Access the Beta code**
- **Join the netfpga-beta mailing list**
- **Join the discussion forum**



Contribute to the Project

- Search for related work
- List your project on the Wiki
- Link your project homepage



Acknowledgments

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**

Special thanks to our Partners:

Patrick Lysaght, Veena Kumar, Paul Hartke, Anna Acevedo
Xilinx University Program (XUP)

Other NetFPGA Tutorial Presented At:

See: http://NetFPGA.org/tutorials/

Acknowledgments

- **Support for the NetFPGA project has been provided by the following companies and institutions**

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.*