Secure Integration of
Cryptographic Primitives

Patrick Schaumont
schaum@vt.edu
ECE Department, Virginia Tech

A System contains Hardware & Software

- Systems are driven by major processor architectures
  - X86, ARM, 8051, PIC
  - But real products differentiate using hardware
- Secure hardware presents a **key value** proposition
Key Value = Protect the weakest link

![Death Star structure with a proton torpedo in a vent](http://www.obh.snafu.de/~madley/starwars/)

Objectives of this talk

- Provide an appreciation for the strengths and weaknesses of hardware and software in secure embedded systems
- Demonstrate different integration strategies for hardware and software
- Experiment with sample applications and platforms

<table>
<thead>
<tr>
<th>Ciphers</th>
<th>AES</th>
<th>Trivium</th>
<th>ECC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platforms</td>
<td>HW-extension for ARM (32-bit)</td>
<td>HW-extension for 8051 (8-bit)</td>
<td></td>
</tr>
</tbody>
</table>

Handson
Intended Audience

- Cryptographic Engineers
  - Improve crypto implementations with a system-level approach
- Hardware Engineers
  - Address the system integration issues of their components
- Secure Software Engineers
  - looking to leverage advantages of secure hardware
- Experimentalists

Organization of this talk

- Part 1: Hardware versus Software
  - Intro, HW/SW Codesign in Secure Applications
  - Hands-on - Design of a simple cipher

- Part 2: Hardware-Software Interfacing
  - HW/SW Interface Design and Modeling
  - Hands-on - Integration of a simple cipher

- Part 3: Alternative and Secure HW/SW Interfaces
  - Alternative HW/SW Interfaces
  - Hands-on - Optimization of a cipher system
Part 1
Hardware versus Software

Hardware and Software - Assumptions

- **Hardware**
  - Word-parallel synchronous single-clock
  - 'RTL' Register Transfer Level

- **Software**
  - Sequential single-thread
  - C program
  - Compile
  - Opcode/data
  - Standard RISC
  - MEM
  - CPU
  - Data
  - ADR
## HW/SW, strengths and weaknesses

### Hardware
- Parallel Execution of Ops
- Fixed in time (cycles), Variable in resources (area)
- Timing constraints are easy, Area constraints are hard.
- Flexibility is hard
- Complex Data Processing
- Modeling $\neq$ Implementation
- IP (Intellectual Property) is hard to find, hard to transfer

### Software
- Sequential Execution of Ops
- Fixed in resources, Variable in execution time
- Timing constraints are hard, Area constraints are easy(ier)
- Flexibility is easy
- Complex Control Processing
- Modeling $==$ Implementation
- IP is easy to find, hard to transfer

---

### In many respects, Hardware and Software use dual design philosophies that lead to dual design results
Concurrent Design of Hardware/Software

System Description (C, Matlab, ..)

System Partitioning

very difficult!

Hardware Description (Parallel, RTL)

Software Description (Sequential)

Hardware Mapping

Software Mapping

Hardware

CPU

MEM

Interface

adr

data

Variant 1 - Accelerate Software

System Description (C, Matlab, ..)

Incrementally move into Hardware

Map to Software

Hardware Description (Parallel, RTL)

Software Description (Sequential)

Hardware Mapping

Software Mapping

Hardware

CPU

MEM

Interface

adr

data
Variant 2 - Platform-Based Design

'Function'

System Description
(C, Matlab, ..)

↓↓

System Mapping

Platform Programs
App
SW
App
HW

'Architecture'

CPU

adr

data

Interface

MEM

e.g.

FPGA

Programmable/
Extendible
Platform

Mapping Algorithms - Generic Concerns

crypto
algorithm
1.
map into
HW or SW
2.
3.

System
Integration

Hardware

1. Perfect match possible
2. Flexibility is hard
   - Multiple Crypto Algorithms
   - Encryption & Decryption
   - Modes of Operation
3. System Communication is hard

Software

1. Approximate match
   - Storage
   - Computations
   - Communication
2. Flexibility is easy
3. System Communication is easy

P. Schaumont, ECRYPT Summer School 2006
Mapping Algorithms - Security Concerns

- Server
- Client
  - root-of-trust
  - Architecture-level attacks
  - Protocol/Algorithm-level validation
  - Noncritical software
    - Architecture-level validation
  - Critical SW
    - Microarchitecture-level validation
    - Critical HW
      - Circuit-level attacks
      - DPA-resistant HW

Structured Side-Channel Defenses

- Architecture
  - uArchitecture
    - Circuit
      - Physical
  - MEM
    - Constant-Time SW
  - CPU
    - Bus Masking
    - Constant-Power HW
    - Secure HW
    - Shielded Circuit
HW/SW Modeling - Abstraction Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Communication</th>
<th>Computation</th>
<th>HW == SW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>Primitive Operations</td>
<td>Untimed Processes</td>
<td></td>
</tr>
<tr>
<td>TLM</td>
<td>PV Transactions</td>
<td>Untimed Processes</td>
<td>HW and SW are distinct</td>
</tr>
<tr>
<td>PV-T</td>
<td>Transactions</td>
<td>Timed Processes</td>
<td></td>
</tr>
<tr>
<td>RTL</td>
<td>Cycle Accurate</td>
<td>RTL</td>
<td></td>
</tr>
<tr>
<td>Gates</td>
<td>Sub-cycle Accurate</td>
<td>Gates</td>
<td></td>
</tr>
</tbody>
</table>

PV(-T) = programmers' view (with time)

Software Modeling and Simulation

- **Native SW simulation**: directly on host machine
  - + No Xcompile, No ISS
  - - No Timing, only functional simulation

- **Interpreted SW simulation**: uses Xcompile, ISS, simulate per instruction
  - + Most flexible, most accurate
  - - Slow (100x .. 1000X)

- **Compiled SW simulation**: uses Xcompile, translate SW binary to host binary
  - + Much faster then interpreted (10x .. 100x)
  - - Not universal, requires specialized translator

- **Instruction-accurate vs Cycle-accurate**
  - Simulators trade off speed, accuracy, visibility
**GEZEL**

- Cycle-based Hardware Description Language
  - Deterministic and Implementation-oriented
  - Based on split control/datapath modeling (FSMD)
  - Easy to learn and use - 11-page LRM
- Hardware Simulation Kernel
  - Open-source (C++) with co-simulation backend
  - Library block concept
  - Toggle/Operation Profiler
- VHDL/Testvector Backend

---

**Example FSMD Model**

```cpp
dp updown(out a : ns(4)) {
    reg c : ns(4);
    sfg inc { c = c + 1;
        a = c;
    }
    sfg dec { c = c - 1;
        a = c;
    }
}

csm ctl_updown(updown) {
    initial s0;
    state s1;
    @s0 if (c < 10) then (inc) -> s0;
    else (dec) -> s1;
    @s1 if (c > 0) then (dec) -> s1;
    else (inc) -> s0;
}
```

---

P. Schaumont, ECRYPT Summer School 2006

---

P. Schaumont, ECRYPT Summer School 2006
Summary

- Hardware and software
  - Strengths and weaknesses
  - Approaches for codesign
- Secure hardware and software
  - Tree of trust
- Modeling of hardware and software
  - Software modeling - cross-compilation, instruction-set simulation
  - Hardware modeling - GEZEL
A closer look

- Three possible tracks:
  - Advanced Encryption Standard
  - Trivium
  - Elliptic Curve Point Multiplier

- Step 1: Hardware development
Part 2
Hardware-Software Interfacing
Hardware-Software Interfacing

- Microprocessor Bus
- Memory-mapped Hardware Encapsulation
- Synchronization
- Interface Models in GEZEL

Microprocessor Bus - AMBA

- High-speed bus
- Unidirectional wires
- Multi-master
- Pipelined transfers
- Burst transfers
- Split transfer (decouple M-S)

- Single-master bus (bridge)
- Unidirectional wires
- Simple read/write transfers
Sample Transfer: Periph->Bridge->ARM

[AMBA Specification Rev. 2.0]

Example - AHB/APB in Atmel AT91SAM7
Data I/O from SW to HW does not benefit from CPU pipeline

Adapt Hardware I/O to bus
- Match wordlengths
- Multiplex multiple inputs/outputs

Define instruction-set
- HW I/O multiplexing
- HW mode selection
- Start/stop control

Key issue: Balance communication and computation
- Data I/O from SW to HW does not benefit from CPU cache
- Control handshaking of SW with HW does not benefit from CPU pipeline
Example: AES in AT91SAM7

- AMBA-PB
- Peripheral Control Unit
  - control
  - status
  - mask

- AES Kernel
  - Data
  - Key
  - IV
  - Mode
  - Enc/Decr
  - ECB/CBC/...

- Peripheral Control Unit
  - control
  - status
  - mask

 AES Control in AT91SAM7

- Software-controlled operation
  - AES
    - start
    - done
    - control.bit
    - interrupt request
    - mask.bit
    - status.bit

- Direct-Memory Access Controlled Operation
  - AES
    - start
    - done
    - (PDC)
    - DMA control
    - mask.bit
    - status.bit
    - transmit pointer
    - receive pointer
    - memory
    - size
    - transmit pointer
    - receive pointer
    - memory
    - size
Encapsulation Overhead for AES

- Using optimizing compiler:
  - HW has 7X improvement over optimized SW OpenSSL AES
  - But HW is over 17X slower than hardware kernel by itself!

- Using non-optimizing compiler:
  - HW has only 4.7X improvement over SW, 36X slower than kernel
  - Data movement is bottleneck

..and there's additional latency within SW!

<table>
<thead>
<tr>
<th>ROM/RAM</th>
<th>Complexity</th>
<th>Typical Latency*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large OS systems</td>
<td>64M &gt; 100 processes</td>
<td>1000's</td>
</tr>
<tr>
<td>Medium OS systems</td>
<td>16M A few 10’s processes</td>
<td>1000's</td>
</tr>
<tr>
<td>Small OS systems</td>
<td>4M A few processes</td>
<td></td>
</tr>
<tr>
<td>Virtual Machines</td>
<td>1M A layered program</td>
<td>100's</td>
</tr>
<tr>
<td>Custom OS</td>
<td>256K A few C programs</td>
<td>10's</td>
</tr>
<tr>
<td>MicroKernels</td>
<td>16K C program</td>
<td></td>
</tr>
<tr>
<td>Microcontrollers</td>
<td>4K Micro/Asm programs</td>
<td>1's</td>
</tr>
<tr>
<td>MicroPrograms</td>
<td>1K</td>
<td></td>
</tr>
</tbody>
</table>

* Latency = Cycle Count Latency to go from a User Application to HW and back
Synchronization

- Hierarchy of activities to maintain data precedence
- Has profound impact on CPU/HW internal operation
  - Pipelining (HW/SW), wait states (HW), cache (SW)

Synchronization: Message Passing

```c
void send(int d) {
    *data = d;  // define data
    *req = !(*req);  // handshake
    while (*ack != *req) ;
}

dp rcp(in d: ns(32));
    in req : ns(1); out ack : ns(1)) {
    reg rack : ns(1);
    reg rd : ns(32);
    always {
        rack = req;
        ack = rack;
        rd = (ack != req) ? d : rd;
    }
}
```

16 cycles round-trip (with full optimization)
Synchronization: Shared Memory

```c
volatile unsigned int *shared = (volatile unsigned int*) 0x80001000;

int accumulate() {
    *req = !(*req);
    // hardware has access here
    while (*ack != *req) {
        for (i=0; i<..; i++)
            acc += shared[i];
        return acc;
    }
}
```

7.2 cycles per word (accumulating 20K words in SW from a 4K buffer written by hardware)

- Further enhancement using DMA

**GEZEL Modeling**

- IPBLOCK: User-defined Simulator Primitive to capture interface between HW model and core

```gezel
ipblock myarm {
    iptype "armsystem";
}
```

```gezel
ipblock src(out data : ns(8)) {
    iptype "armsystemsource";
    ipparm "core=myarm";
    ipparm "address=0x80000000";
}
```

P. Schaumont, ECRYPT Summer School 2006
Summary

- Microprocessor Bus
  - AMBA AHB/APB
  - Atmel AT91SAM7 microcontroller
- Hardware-microprocessor bus interfaces
  - AES in the AT91SAM7
  - Overhead of encapsulation
- Hardware-software communication
  - Synchronization - shared memory

A closer look

- at two target platforms:
  - ARM with Memory-mapped interface
  - 8051 with Port-mapped interface

- Step 2: Coprocessor Integration
  - AES
  - Trivium
  - ECC
Part 3
Alternative and Secure Hardware-Software Interfaces

Alternative and Secure HW/SW interfaces

- Alternative Integration Strategies
  - Custom Instructions
  - Loosely coupled processors
- Secure Partitioning
  - Oracle partitioning
  - Process Isolation
- Beyond Hardware-Software Codesign
  - Components and Platforms
Alternative Integration Strategies

1. Pull computation into the CPU
   - Special Function Units in processor (ASIP)
   - Coprocessor bus and Dedicated Ports

   ![Diagram showing ARM, On-chip Memory, Timer, Parallel I/O, Memory Interface, DMA Bus Master, Bridge, UART, AES, and AHB connections.]

2. Push communication out of the CPU
   - Network on Chip
   - Shared Memory & Direct Memory Access
   - Streaming processing

Pull into CPU - LEON3 Example

- Make use of dedicated processor instructions
  - Floating-point
  - General Coprocessor
- Tightly integrates to processor pipeline: pipeline stall, pipeline flush
- Compared to standard bus (APB), data bandwidth typically x8
  - Larger wordlength
  - One transfer/cycle
**Pull into CPU - Xtensa Example**

Inline Assembly

```c
#define OP2x2_1(D1, D2, S1, S2) \
    asm volatile("smullnv %0, %1, %2, %3": \
        "=&r"(D1),"=&r"(D2): \
        "r"(S1),"r"(S2));
```

Use in C, e.g.

```c
OP2x2_1(a, b, c, d);
```

**Pull into CPU – SimIt-ARM Example**

Inline Assembly

```c
#define OP3x1(a, b, c, d) \
    asm volatile("smullnv %0, %1, %2, %3": \
        "=&r"(a),"=&r"(b): \
        "r"(c),"r"(d));
```

Use in C, e.g.

```c
OP3x1(a, b, c, d);
```
Push out of CPU: Checksum Example

- Compared to LEON2/50MHz (using Virtex-2)
  - 66X energy savings for checksum function in HW over SW
  - 33x performance improvement
  - at 32% area overhead due to additional HW

Push out of CPU – Ping Pong Buffer Example
Secure Partitioning

- Partition a design on the basis of side channel leakage of the root-of-trust
  - Oracle Partitioning
  - Process Isolation

Oracle Partitioning

- Hide control-flow side channels by deferring decisions to an oracle
Process Isolation

See also A. Tanenbaum’s Article in IEEE Computer, May 2006.

http://www.trusted-logic.com/ (now ARM TrustZone)

Need to extend the concept of isolation into new processor hardware

Example: Context-switched AES

Agents manage coprocessor state (registers) locally

Agents establish a secure channel to application SW
  • Access Control generates a unique random number nonce
  • Subsequent SW accesses are authenticated by nonce

Context switch much faster than using software
  • About 16X for typical AES including mode-state
Components and Platforms

Component Design

Platform Design

Platform-based Design

Programming Interface

Simulation & Refinement Kernel

Integration Interface

FSMD

ISA

GEZEL Kernel

ARM ISS

IPBLOCK

Memory Bus

Scheduling & Interconnect

GEZEL

ARM-C

Platform

Application Independent

Application-Domain Specific

Application Specific

Building platforms using ipblock

ipblock → FSMD → ipblock

GEZEL Kernel

ipblock

ipblock

ipblock

ipblock

C++ Functional Model

C++ Socket

C++ Testbench

C++ Simulator ISS Matlab ...

network

Application

Platform
Building platforms using ipblock

GEZEL Spec

```cpp
ipblock ip1(out data : ns(8)) {
    ipType "myblock";
    ipParm "parm1=myparm";
}
```

C++ Implementation of ipblock

```cpp
class myblock {
    public:
        myblock();
        void run(); // called once per cycle
        void setparm(char *p); // called with p = "parm1.."
    };
```

libmyblock.so  Dynamically Compiled Library

IP Reuse and Exchange

<table>
<thead>
<tr>
<th>Hardware Developer</th>
<th>System Integrator</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEZEL</td>
<td>GEZEL</td>
</tr>
<tr>
<td>IP creation &amp; evaluation</td>
<td>IP creation &amp; evaluation</td>
</tr>
<tr>
<td>FSMD</td>
<td>ARM-C</td>
</tr>
<tr>
<td>VHDL code generation</td>
<td>VHDL code generation</td>
</tr>
<tr>
<td>generate simulation view</td>
<td>generate simulation view</td>
</tr>
<tr>
<td>(black-box view)</td>
<td>(black-box view)</td>
</tr>
<tr>
<td>IP transfer</td>
<td>IP transfer</td>
</tr>
<tr>
<td>VHDL transfer</td>
<td>VHDL transfer</td>
</tr>
<tr>
<td>Platform Implementation</td>
<td>Platform Implementation</td>
</tr>
</tbody>
</table>
Summary

- Alternative Hardware-Software Interfaces
  - Custom Instructions
  - Loosely coupled coprocessors
- Secure Hardware-Software Interfacing
  - Partitioning - Oracle example
  - Isolation
- Ideas in Platform design
  - Extensibility through ipblock
  - Reusability through ipblock

A closer look

- at two alternative target platforms:
  - ARM with special-function units
  - 8051 with shared-memory interface

- Step 3: Coprocessor optimization
  - AES
  - Trivium
  - ECC
Design Examples

Example 1: Trivium in GEZEL

Pseudocode: [De Canniere et al, eSTREAM]

```
for i = 1 to N do
    t1 = s66 ^ s93
    t2 = s162 ^ s177
    t3 = s243 ^ s288
    zi = t1 ^ t2 ^ t3
    t1 = t1 ^ s91 & s92 ^ s171
    t2 = t2 ^ s175 & s176 ^ s264
    t3 = t3 ^ s286 & s287 ^ s69
    (s1, s2, ..., s93) = (t3, s1, ..., s92)
    (s94, s95, ..., s177) = (t1, s94, ..., s176)
    (s178, s179, ..., s288) = (t2, s178, ..., s287)
end for
```

Diagram:

- 93 bit
- 84 bit
- 111 bit
- Logic
- z
Ex1: Trivium Kernel in GEZEL

```plaintext
dp trivium(in si: ns(288)); // state input
  out so : ns(288); // state output
  out z : ns(1)); // crypto bit out
sig t1, t2, t3 : ns(  1);
sig t11, t22, t33 : ns(  1);
sig saa : ns( 93);
sig sbb : ns( 84);
sig scg : ns(111);
always {
  t1 = si[65] ^ si[92];
t2 = si[161] ^ si[176];
t3 = si[242] ^ si[287];
z = t1 ^ t2 ^ t3;
t11 = t1 ^ (si[90] & si[91]) ^ si[170];
t22 = t2 ^ (si[174] & si[175]) ^ si[263];
t33 = t3 ^ (si[285] & si[286]) ^ si[ 68];
saa = si[0:92] # t33;
sbb = si[93:176] # t11;
scc = si[177:287] # t22;
so = scg # sbb # saa;
}
```

Ex1: Trivium Key Schedule in GEZEL

```plaintext
dp keyschedule(in ld : ns(1)); // reload key & iv
  in iv : ns(80); // initialization vector
  in key : ns(80); // key
  out e : ns(1)); // output valid
  in si : ns(288); // state input
  out so : ns(288)); // state output
reg s : ns(288); // state register
reg cnt : ns(11)); // initialization counter
sig saa : ns( 93);
sig sbb : ns( 84);
sig scg : ns(111);
sig cte : ns(111);
always {
  saa = ld ? key : si[0:92];
sbb = ld ? iv : si[93:176];
  cte = 7;
  scg = ld ? (cte << 108) : si[177:287];
s = scg # sbb # saa;
s = s;
cnt = ld ? 1152 : (cnt ? cnt-1 : cnt);
e = (cnt ? 0 : 1);
}
```
Ex1: 1-bit per cycle Trivium

dp triviumtop(in ld : ns(1); // reload key & iv
in iv : ns(80); // initialization vector
in key : ns(80); // key
out z : ns(1); // encrypted output
out e : ns(1)) { // output valid

sig si, so0 : ns(288);
sig z0 : ns(1);
use keyschedule(ld, iv, key, e, si, so0);
use trivium(so0, si, z0);
always {
  z = z0;
}
}

Ex1: 2-bit per cycle Trivium

dp trivium1 : trivium

dp triviumtop(in ld : ns(1); // reload key & iv
in iv : ns(80); // initialization vector
in key : ns(80); // key
out z : ns(2); // encrypted output
out e : ns(1)) { // output valid

sig si, so0, sol : ns(288);
sig z0, z1 : ns(1);
use keyschedule(ld, iv, key, e, si, sol);
use trivium (so0, sol, z0);
use trivium1 (sol, si, z1);
always {
  z = z0 # z1;
}
}
Ex1: GEZEL -> implementation

GEZEL -> (fdlvhd) -> VHDL -> (synplify, ise) -> FPGA

<table>
<thead>
<tr>
<th></th>
<th>Spartan3 XS400-4</th>
<th>Virtex4 XC4VLX15-12</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Trivium1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flip-Flop</td>
<td>288 + 11</td>
<td>288 + 11</td>
</tr>
<tr>
<td>Slices (4-LUT)</td>
<td>160 (317)</td>
<td>161 (318)</td>
</tr>
<tr>
<td>Post-PAR clock</td>
<td>4.226 ns</td>
<td>2.019 ns</td>
</tr>
<tr>
<td><strong>Trivium8</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flip-Flop</td>
<td>288 + 8</td>
<td>288 + 8</td>
</tr>
<tr>
<td>Slices (4-LUT)</td>
<td>181 (353)</td>
<td>181 (351)</td>
</tr>
<tr>
<td>Post-PAR clock</td>
<td>3.987 ns</td>
<td>1.94 ns</td>
</tr>
<tr>
<td><strong>Trivium32</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flip-Flop</td>
<td>288 + 6</td>
<td>288 + 6</td>
</tr>
<tr>
<td>Slices (4-LUT)</td>
<td>267 (516)</td>
<td>267 (516)</td>
</tr>
<tr>
<td>Post-PAR clock</td>
<td>4.637 ns</td>
<td>2.075 ns</td>
</tr>
</tbody>
</table>

Example 2: Trivium Coprocessor for ARM

Memory-Mapped Interface

Encapsulation Hardware

P. Schaumont, ECRYPT Summer School 2006
Ex2: Encapsulation Interface (1)

dp  triviumtop(in data : na[32]);
    out douting : na[32];
in  in : na[32];
    out status : na[32]);
sig 1d, go, lv, key, z, s0;
    ivr0, ivr1, ivr2 : na[32];
    ivr0, ivr1, ivr2 : na[32];
    triviumtop(id, go, lv, key, z, s0);
    key0, key1, key2 : na[32];
    triviumtop(id, go, lv, key, z, s0);
    triviumtop(id, go, lv, key, z, s0);
    if (z) { // program new IV
        ivr0 = (c2[24:26] == 0x2) & (c2[25:12] == 0x0);  // 1 dim : ivr13: 0];
        ivr1 = (c2[24:26] == 0x2) & (c2[25:12] == 0x0);  // 1 dim : ivr13: 0];
        ivr2 = (c2[24:26] == 0x2) & (c2[25:12] == 0x0);  // 1 dim : ivr13: 0];
        ivr = ivr2 & ivr1; ivr = ivr;
    }
    // program new KEY
    key0 = (c2[24:26] == 0x2) & (c2[25:12] == 0x0);  // 1 dim : keyy[31: 0];
    key1 = (c2[24:26] == 0x2) & (c2[25:12] == 0x0);  // 1 dim : keyy[31: 0];
    key2 = (c2[24:26] == 0x2) & (c2[25:12] == 0x0);  // 1 dim : keyy[31: 0];
    key = key2 & key1 & key0; key = keyy;
    id = (c2[24:26] == 0x3) ? 1 : 0; // control start
    status = e; // read status
    douting = z; // read output data
// trivium kernel control
    triviumtop(c2[24:26]);
    go = (c2[24:26] == 0x5) & (olddata == 0x5));
    (c2[24:26] == 0x5) & (olddata == 0x5));
    (c2[24:26] == 0x5) & (olddata == 0x5));

Ex2: Encapsulation Interface (2)

dp  triviumtop(in data : na[32]);
    out douting : na[32];
in  in : na[32];
    out status : na[32]);
sig 1d, go, lv, key, z, s0;
    ivr0, ivr1, ivr2 : na[32];
    key0, key1, key2 : na[32];
    triviumtop(id, go, lv, key, z, s0);
    triviumtop(id, go, lv, key, z, s0);
    triviumtop(id, go, lv, key, z, s0);
    if (z) { // program new IV
        ivr0 = (c2[24:26] == 0x2) & (c2[25:12] == 0x0);  // 1 dim : ivr13: 0];
        ivr1 = (c2[24:26] == 0x2) & (c2[25:12] == 0x0);  // 1 dim : ivr13: 0];
        ivr2 = (c2[24:26] == 0x2) & (c2[25:12] == 0x0);  // 1 dim : ivr13: 0];
        ivr = ivr2 & ivr1; ivr = ivr;
    }
    // program new KEY
    key0 = (c2[24:26] == 0x2) & (c2[25:12] == 0x0);  // 1 dim : keyy[31: 0];
    key1 = (c2[24:26] == 0x2) & (c2[25:12] == 0x0);  // 1 dim : keyy[31: 0];
    key2 = (c2[24:26] == 0x2) & (c2[25:12] == 0x0);  // 1 dim : keyy[31: 0];
    key = key2 & key1 & key0; key = keyy;
    id = (c2[24:26] == 0x3) ? 1 : 0; // control start
    status = e; // read status
    douting = z; // read output data
// trivium kernel control
    triviumtop(c2[24:26]);
    go = (c2[24:26] == 0x5) & (olddata == 0x5));
    (c2[24:26] == 0x5) & (olddata == 0x5));
    (c2[24:26] == 0x5) & (olddata == 0x5));

Ex2: Driver program in C

```c
int main() {
    volatile unsigned int *data = (unsigned int *) 0x80000000;
    volatile unsigned int *ctl = (unsigned int *) 0x80000004;
    volatile unsigned int *output = (unsigned int *) 0x80000000;
    volatile unsigned int *status = (unsigned int *) 0x80000008;

    // program iv
    *ctl = (1 << 24); // word 0
    *data = 0;
    *ctl = (1 << 24) | 0x1; // word 1
    *ctl = (1 << 24) | 0x2; // word 2

    // program key
    *ctl = (2 << 24); // word 0
    *data = 0x80;
    *ctl = (2 << 24) | 0x1; // word 1
    *ctl = (2 << 24) | 0x2; // word 2

    // run the key schedule
    while (! *status) {
        *ctl = (4 << 24); // start pulse
        if (*status) break;
        *ctl = (5 << 24);
    }

    // key stream is ready here
}
```

Example 3: Trivium Function Unit for ARM

![Diagram of Trivium Function Unit for ARM](image)
Ex3: Encapsulation Interface (1)

```plaintext
ipblock myarm {
    iptype  "armsystem";
    ipparm  "exec=trivium";
}

ipblock armsfu1(out d1, d2 : ns(32);
    in q1, q2 : ns(32)) {
    iptype  "armsfu2x2";
    ipparm  "core = myarm";
    ipparm  "device = 0";
}

ipblock armsfu2(out d1, d2, d3 : ns(32);
    in q1        : ns(32)) {
    iptype  "armsfu3x1";
    ipparm  "core = myarm";
    ipparm  "device = 0";
}

dp trivium320(in ni : ns(288);
    out so : ns(288);
    out z  : ns(32)) {
...
}

dp trivium321 : trivium320
```

Ex3: Encapsulation Interface (2)
Ex3: Driver program in C

```c
#include "armsfu.h"
int main() {
    int z1, z2, i;
    unsigned int stream[512];

    int key1 = 0x80;
    int key2 = 0x00000000;
    OP3x1_1(z1, key1, 0, 0);
    OP3x1_1(z1, f0, 0, 0);
    OP3x1_1(z1, key2, 0, 0);
    OP3x1_1(z1, f0, 0, 0);
    for (i=0; i<128; i++) {
        OP2x2_1(z1, z2, 0, 0);
        OP2x2_1(z1, z2, 1, 0);
    }
    return 0;
}
```

Further Reading and References

- **Methodology**
Further Reading and References

- **Methodology**

- **Modeling**

- **Coprocessor Architecture**

- **Coprocessor-software Interfacing Issues**

---

P. Schaumont, ECRYPT Summer School 2006

---

Further Reading and References

- **Modeling**

- **Coprocessor Architecture**

- **Coprocessor-software Interfacing Issues**

---

P. Schaumont, ECRYPT Summer School 2006
Further Reading and References

- Coprocessor-software Interfacing Issues

- Components