RSA Box

Emily Pakulski (enp2111)
Jaykar Nayeck (jan2150)
Adam Incera (aji2112)
Noah Stebbins (nes2137)
“A fast and secure hardware accelerator for RSA encryption with a clear, simple interface for programmer use.”
Initial goals

Provide a simple, well-defined interface for a host machine to carry out RSA cryptography operations on a dedicated piece of hardware.

1. Implement RSA algorithms using SystemVerilog.
2. Provide a software interface (Linux device driver, wrapper (in C), and example interface) to use the RSA Box.
High-level design

RSA_Box

- Data in[0:31]
- Address[0:2]
- Data out[0:31]
- Reset
- Write

Key registers:
- [0:63] private key register: p
- [0:63] private key register: q
- [0:31] public key register: e
- [0:127] public key register: n

Encryption block

Decryption block

Symbol: *
Original vs. final design

Observation: parts of the RSA algorithm are “fixed costs”, others are “marginal costs”. Final design prioritizes lowering the overhead for repeated operations, rather than all operations -- highly costly
Extended Euclid’s algorithm moved to software.

Observation: implementing operations for large-bit values is time-consuming and not always possible. We changed our algorithms to use fewer operations and focused on speeding up encryption/decryption.
Contributions

- **Jaykar**: primary hardware framework writer, device driver, hardware/software interface (first version)
- **Emily**: C wrapper, hardware/software interface, C interface
- **Adam**: multiplier block and exponentiation
- **Noah**: private key generation and primality testing
Software/Hardware Interface

- Created 14 operation “ISA” that C wrapper sends to device driver to communicate with hardware.
- Lesson learned: standardize this earlier.
- OS was really helpful -- we struggled with the device driver lab3 code.
Private Key Generation (Software)

- **Private Key: Extended Euclid’s in Python**
  - computes modular multiplicative inverse ➔ private key, piped into C

- **Public Key:**
  - initial approach: Miller-Rabin + Linear Backoff
  - final approach: hard-coded list of 64 bit primes
Hardware implementation

- Optimized modular multiplication from 6 cycles to 2 cycles per bit
- Set up a parallel block for modular exponentiation so encryption and decryption can run simultaneously
Encrypting & Decrypting (Hardware)

● Modular multiplication block
  ○ Multiplies two 128-bit numbers and reduces on a 128-bit modulus in 257 clock cycles

● Modular exponentiation block
  ○ Performs exponentiation in $O(n)$ time where $n$ is the bit length of the exponent
Modular Exponentiation Algorithm

```
function modular_pow(base, exponent, modulus)
    Assert :: (modulus - 1) * (modulus - 1) does not overflow base
    result := 1
    base := base mod modulus
    while exponent > 0
        if (exponent mod 2 == 1):
            result := (result * base) mod modulus
            exponent := exponent >> 1
        base := (base * base) mod modulus
    return result
```

Source: http://en.wikipedia.org/wiki/Modular_exponentiation
Where we struggled (Git history)

**TL;DR:** Should have taken the pre-reqs. Advanced Logic Design would have been nice.