

NTUEE DCLAB

---

## LAB 2: RSA256 解密機

---

*Graduate Institute of Electronics Engineering*  
National Taiwan University

# Outline

---

- Introduction
  - Lab requirements
- RSA256 cryptosystem
- System-on-Chip (SoC) and Qsys
- Implementation
  - RSA256 core
  - RSA256 wrapper
- Code template
- System setup and run testing program
- Report regulations

# Introduction

---

- RSA256解密機
  - PC 端透過 RS232 傳輸金鑰與密文給 FPGA
  - FPGA 接收資料並進行解密運算
  - 解密完成後 FPGA 透過 RS232 將答案傳回給 PC 端
- 實驗目的
  - 實作巨大整數運算，了解不同運算方式對硬體效率的影響，體會硬體加速的不可取代性
  - 實作大型的輸入輸出界面，理解模組溝通的基礎模式與系統間通訊的匯流排(bus)觀念

# Lab Requirements

---

- Key0 可以reset
- 通過測資，正確解密
  - Code template 助教有提供三筆
  - 另外一筆隱藏測資會在 demo 當天公佈
  - 設計可為單次使用(每次解密前要先按reset)
- Bonus (demo 時與 report 中皆應清楚詳細說明)
  - 不需 reset 即可連續解密多份密文(不同金鑰)
  - 其他能想到的變化

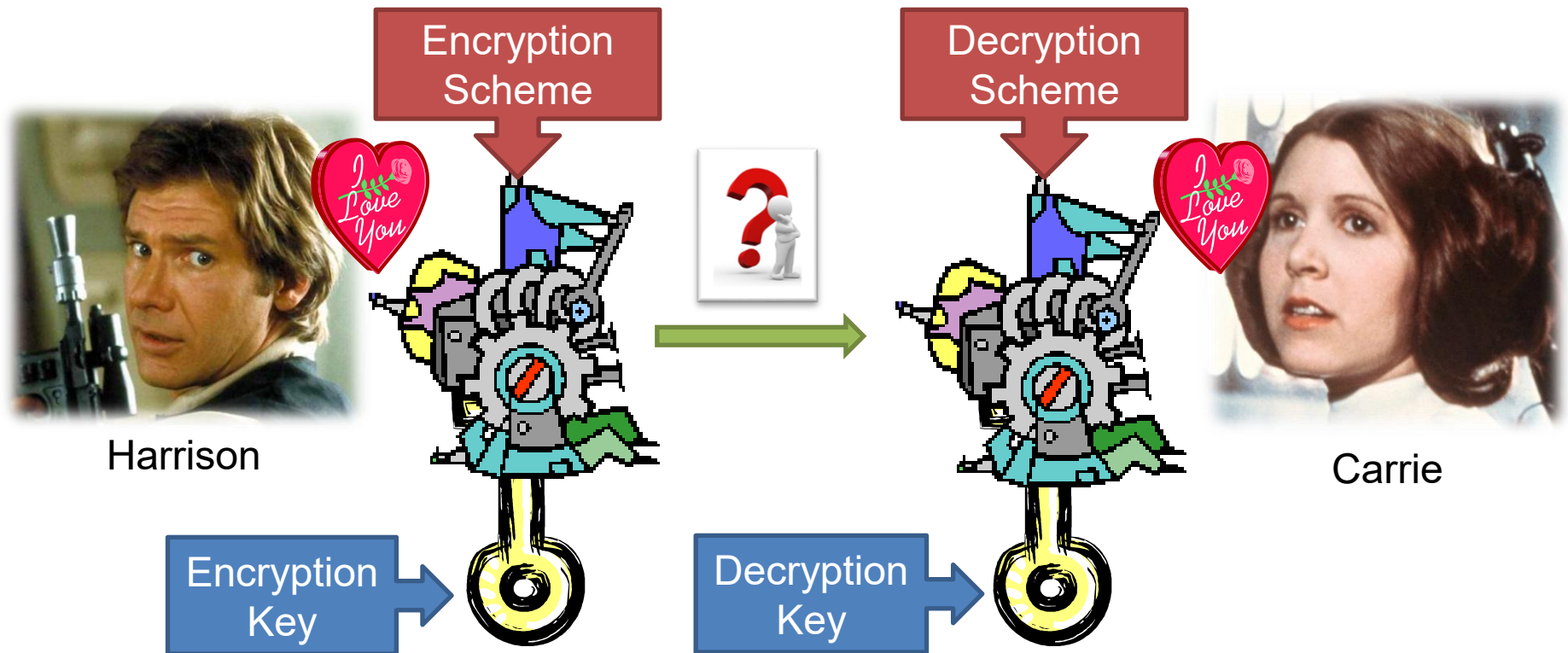
# Outline

---

- Introduction
  - Lab requirements
- **RSA256 cryptosystem**
- System-on-Chip (SoC) and Qsys
- Implementation
  - RSA256 core
  - RSA256 wrapper
- Code template
- System setup and run testing program
- Report regulations

# Introduction to Cryptography

- Communication is insecure
- Use cryptosystems to protect communications



# RSA Cryptosystem

---

- Carrie select a key pair and release the **encryption key/public key (公鑰)**
  - Everyone knows the encryption key
  - RSA is a public cryptosystem
- If Harrison wants to communicate with Carrie, he uses the encryption key released by Carrie to encrypt
- Carrie uses the **decryption key/private key (私鑰)** to decipher the encrypted message
  - Only Carrie knows the decryption key
  - Message sent from Harrison to Carrie is secured

# Key Pair Selection Scheme

---

- Randomly select 2 distinct prime numbers  $p$  and  $q$ 
  - For security reason,  $p = 2p' + 1$  and  $q = 2q' + 1$ , where  $p'$  and  $q'$  are also prime numbers.
- Evaluate  $N = pq$  and  $\Phi(N) = (p - 1)(q - 1)$
- Choose  $e$  such that  $\gcd(e, \Phi(N)) = 1$
- Find the integer  $d$  ( $0 < d < \Phi(N)$ ) such that  $ed - k\Phi(N) = 1$
- Finally, release the number pair  $(N, e)$  and keeps  $(d, p, q, \Phi(N))$  in secret
  - $(N, e)$  is the public key
  - $(N, d)$  is the private key



# RSA Encryption and Decryption

- $x$  is the message to be sent
- $y$  is the encrypted message actually being sent

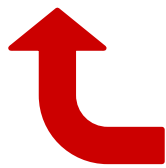
- Encryption

- $y = x^e \bmod N$

- Decryption

- $x = y^d \bmod N$

$(N, e)$  is the public key  
 $(N, d)$  is the private key



For RSA256:  $p, q$  are 128b, the rest are 256b

**Hard to calculate!**

**Need an efficient way to compute!**

**But how?**

# Exponentiation by Squaring

- **Reduce the amount of multiplication**
- Use the binary representation of the exponent
- Example: assume  $d = 12$

$$\begin{aligned} - y^d &= y^{12_{10}} = y^{1100_2} = (y^8)^1 \cdot (y^4)^1 \cdot (y^2)^0 \cdot (y^1)^0 \\ - y^d \pmod N &= [(y^8) \cdot (y^4)] \pmod N \end{aligned}$$

```
function EXPONENTIATION_SQUARING( $N, y, d$ )
```

```
   $t \leftarrow y$ 
```

```
   $m \leftarrow 1$ 
```

```
  for  $i \leftarrow 0$  to 255 do
```

```
    if  $i$ -th bit of  $d$  is 1 then
```

```
       $m \leftarrow m \cdot t \pmod N$ 
```

```
    end if
```

```
       $t \leftarrow t^2 \pmod N$ 
```

```
  end for
```

```
  return  $m$ 
```

```
end function
```

**How to calculate this?**

$\triangleright t \rightarrow t^2 \rightarrow t^4 \rightarrow \dots$

Calculates  $y^d \pmod N$

# Algorithm 1: Modulo of Products

---

- Now,  $y^d \pmod N$  becomes several  $ab \pmod N$  operations
- **Further replace multiplication with additions**
- Example: assume  $a = 12$ 
  - $ab \pmod N = 12_{10} \cdot b \pmod N = 1100_2 \cdot b \pmod N$   
 $= (8b + 4b) \pmod N$
  - $8b = 2 \cdot 4b = 2 \cdot 2 \cdot 2b$  (can be compute with similar way)
- Perform modulo operation every iteration to prevent overflow

# Algorithm 1: Modulo of Products

**function** MODULOPRODUCT( $N, a, b, k$ )

▷  $k$  is number of bits of  $a$

$t \leftarrow b$

$m \leftarrow 0$

**for**  $i \leftarrow 0$  to  $k$  **do**

**if**  $i$ -th bit of  $a$  is 1 **then**

**if**  $m + t \geq N$  **then**

$m \leftarrow m + t - N$

**else**

$m \leftarrow m + t$

**end if**

▷ perform modulo operation in each iteration

**end if**

**if**  $t + t \geq N$  **then**

$t \leftarrow t + t - N$

**else**

$t \leftarrow t + t$

**end if**

▷ perform modulo operation in each iteration

**Need additional hardware for comparison**

**end for**

**return**  $m$

**end function**

Calculates  $a \times b \pmod{N}$

# Algorithm 2: Montgomery Algorithm

---

- Alternative way to perform  $ab \pmod{N}$  and prevent overflow
  - Calculate  $ab \cdot 2^{-i} \pmod{N}$  instead
- Example: assume  $a = 12, i = 4$ 
  - $ab \cdot 2^{-4} = 4'b1100 \cdot b \cdot 2^{-4}$ 
$$= \left( ( (0 \cdot 2^{-1} + 0) \cdot 2^{-1} + b) \cdot 2^{-1} + b \right) \cdot 2^{-1}$$
  - $2^{-1}$  is multiplied in every iteration so it won't overflow

# Algorithm 2: Montgomery Algorithm

**function** MONTGOMERYALGORITHM( $N, a, b$ )

$m \leftarrow 0$

**for**  $i \leftarrow 0$  to 255 **do**

**if**  $i$ -th bit of  $a$  is 1 **then**

$m \leftarrow m + b$

**end if**

**if**  $m$  is odd **then**

$m \leftarrow m + N$

**end if**

$m \leftarrow \frac{m}{2}$

**end for**

**if**  $m \geq N$  **then**

$m \leftarrow m - N$

**end if**

**return**  $m$

**end function**

▷ 4~6: replace multiplication with successive addition

▷ 7~10: calculate the modulo of  $a \cdot 2^{-1}$   
→ Montgomery reduction

Calculates  $a \times b \times 2^{-256} \pmod{N}$

# Overall Algorithm for RSA256

- **Recall our original goal: calculate  $y^d \pmod{N}$** 
  - Replace exponentiation  $\rightarrow$  multiplication  $\rightarrow$  addition
  - Use Montgomery to further avoid overflow

```
function RSA256MONT( $N, y, d$ )  
   $t \leftarrow \text{ModuloProduct}(N, 2^{256}, y, 256)$   $\longrightarrow t = y * 2^{256} \pmod{N}$   
   $m \leftarrow 1$   
  for  $i \leftarrow 0$  to 255 do  
    if  $i$ -th bit of  $d$  is 1 then  
       $m \leftarrow \text{MontgomeryAlgorithm}(N, m, t)$   $\longrightarrow m * t * 2^{-256} \pmod{N}$   
    end if  
     $t \leftarrow \text{MontgomeryAlgorithm}(N, t, t)$   $\longrightarrow t * t * 2^{-256} \pmod{N}$   
  end for  
  return  $m$   
end function
```

Calculates  $y^d \pmod{N}$

# Overall Algorithm for RSA256

---

- The “exponentiation by squaring” heuristic is used
  - Multiplication (and squaring) are substituted by consecutive additions
- Applying the Montgomery algorithm can avoid overflow
  - Need to pre-multiply  $y$  by  $2^{256}$ , as shown in line 2
- Perform modulo in the intermediate stages to keep the operands below  $N$



# Outline

---

- Introduction
  - Lab requirements
- RSA256 cryptosystem
- **System-on-Chip (SoC) and Qsys**
- Implementation
  - RSA256 core
  - RSA256 wrapper
- Code template
- System setup and run testing program
- Report regulations

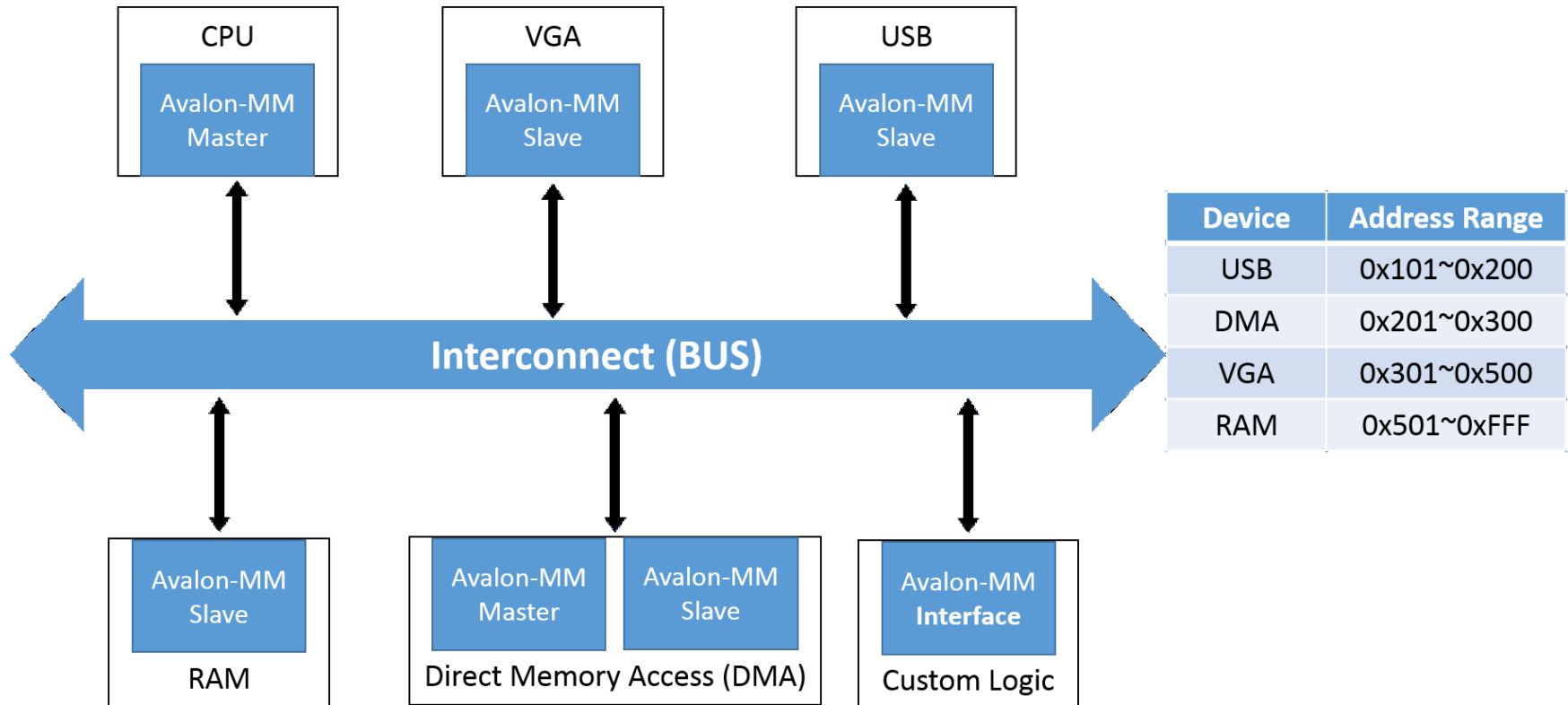
# System-on-Chip (SoC)

---

- Integrate the entire system onto a single chip
  - May contain digital, analog, mixed-signal, RF functions
  - Allows large systems to be built with existing modules
- Master
  - initialize requests
- Slave
  - respond to requests
- Bus
  - interconnection between master and slave IPs
  - The protocols are similar to memory read/write
  - Ex: AMBA/AHB/AXI (ARM), Avalon (Altera)

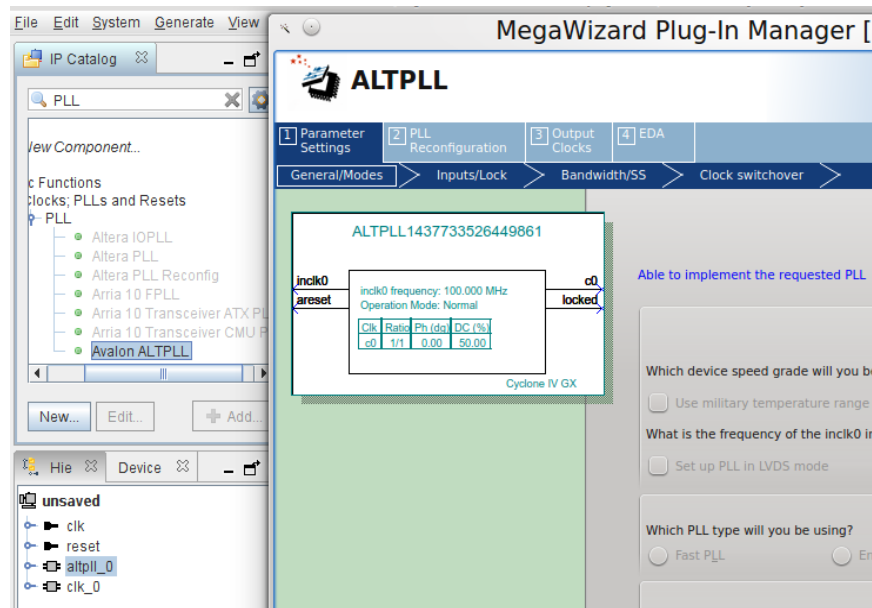
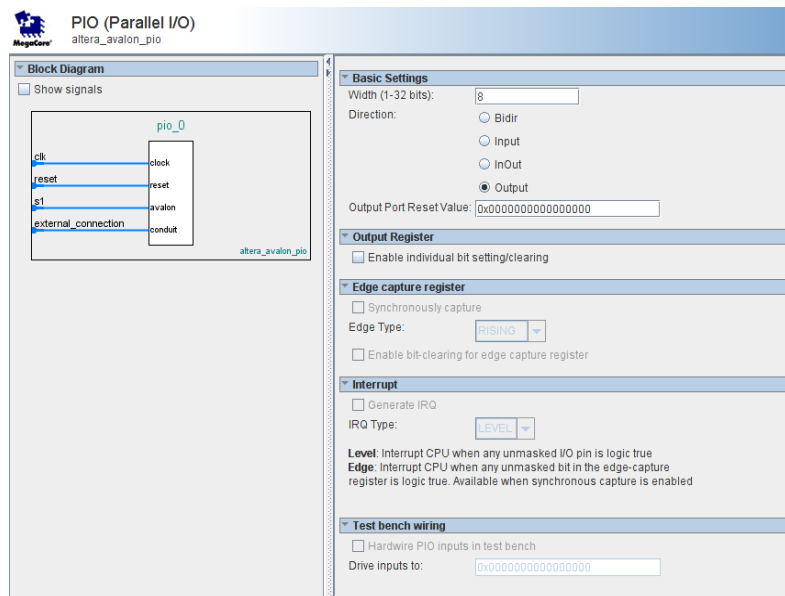
# Memory Mapped I/O (MMIO)

- CPU uses address to access the RAM
- Some addresses in SoC are mapped to I/O of IP
  - Access them just like accessing the RAM



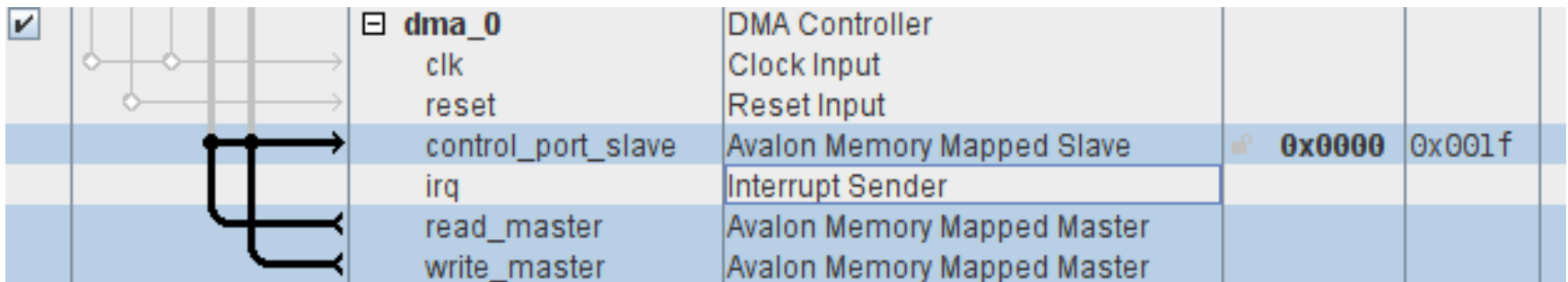
# Qsys: Altera SoC Tool

- Upon opening Qsys, there will be a **clock source module**
  - Converts raw signals (conduits) to clock and reset (negedge)
- **Parallel I/O modules** can create read/write slave interface
  - For key, switch, LED, ...
- **PLL** converts 50MHz (default clock) to almost any frequency



# More on Qsys

- You can add more modules to the design
  - Like the core and wrapper you implemented (as master)
  - Possible connection will be displayed, click to enable
- Connected signals are colored black
  - Slaves are associated with address ranges
  - Masters uses this address to access the slaves

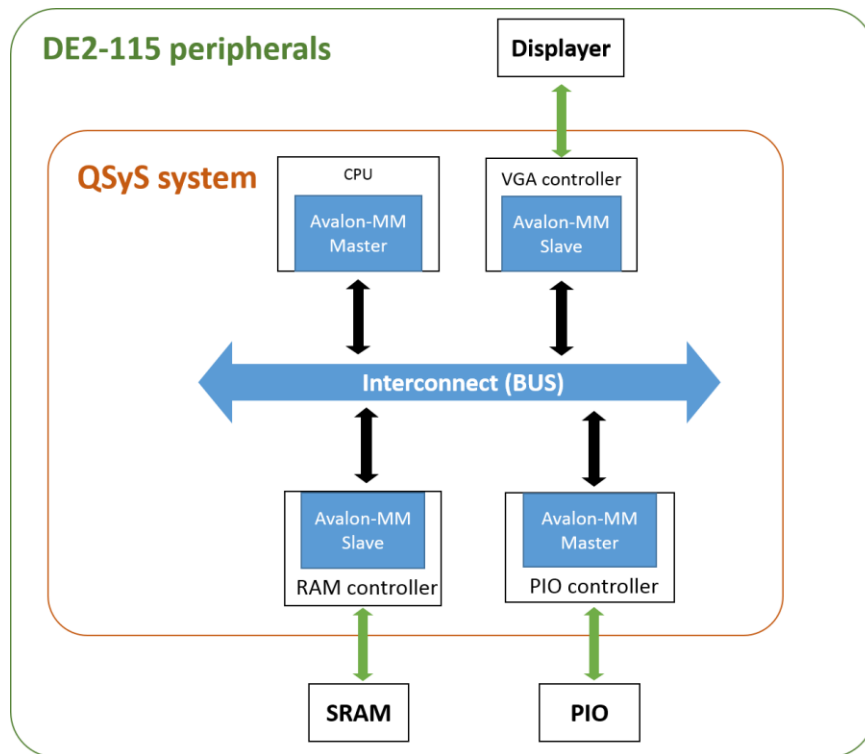


The screenshot shows the Qsys component browser with a search filter 'dma\_0'. The component 'dma\_0' is expanded, showing its internal signals and their connections. The 'control\_port\_slave' signal is connected to the 'dma\_0' component, and the 'read\_master' and 'write\_master' signals are connected to the 'dma\_0' component. The 'dma\_0' component is an Avalon Memory Mapped Slave. The 'read\_master' and 'write\_master' signals are Avalon Memory Mapped Masters. The 'dma\_0' component is associated with the address range 0x0000 to 0x001f.

Signal	Component	Address Range
clk	DMA Controller	
reset	Reset Input	
control_port_slave	Avalon Memory Mapped Slave	0x0000 - 0x001f
irq	Interrupt Sender	
read_master	Avalon Memory Mapped Master	
write_master	Avalon Memory Mapped Master	

# Add Qsys Module to Your Design

- Generate Qsys qip module and Verilog file
- Add the qip to your project and connect the corresponding wires under the top module (DE2\_115.sv)



Follow the step-by-step tutorial to generate your Qsys module for Lab2!

# Outline

---

- Introduction
  - Lab requirements
- RSA256 cryptosystem
- System-on-Chip (SoC) and Qsys
- **Implementation**
  - RSA256 core
  - RSA256 wrapper
- Code template
- System setup and run testing program
- Report regulations

# General Roadmap

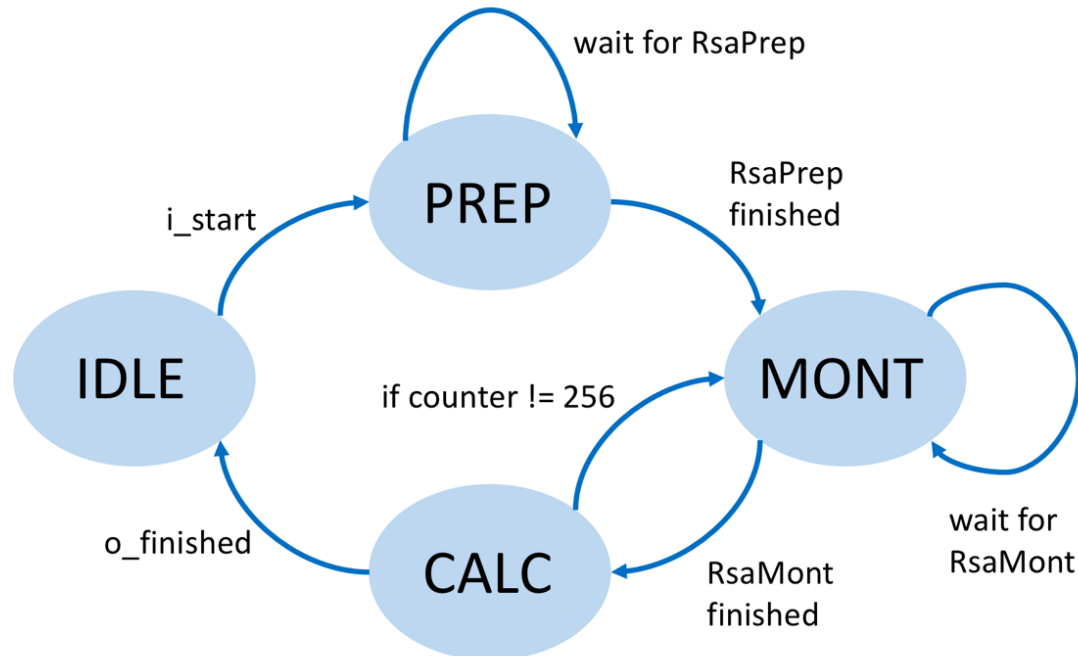
---

- Create a project
- Implement the RSA256 core
- Implement a wrapper to control RS232 and your core
- Build Qsys system
- Compile and program



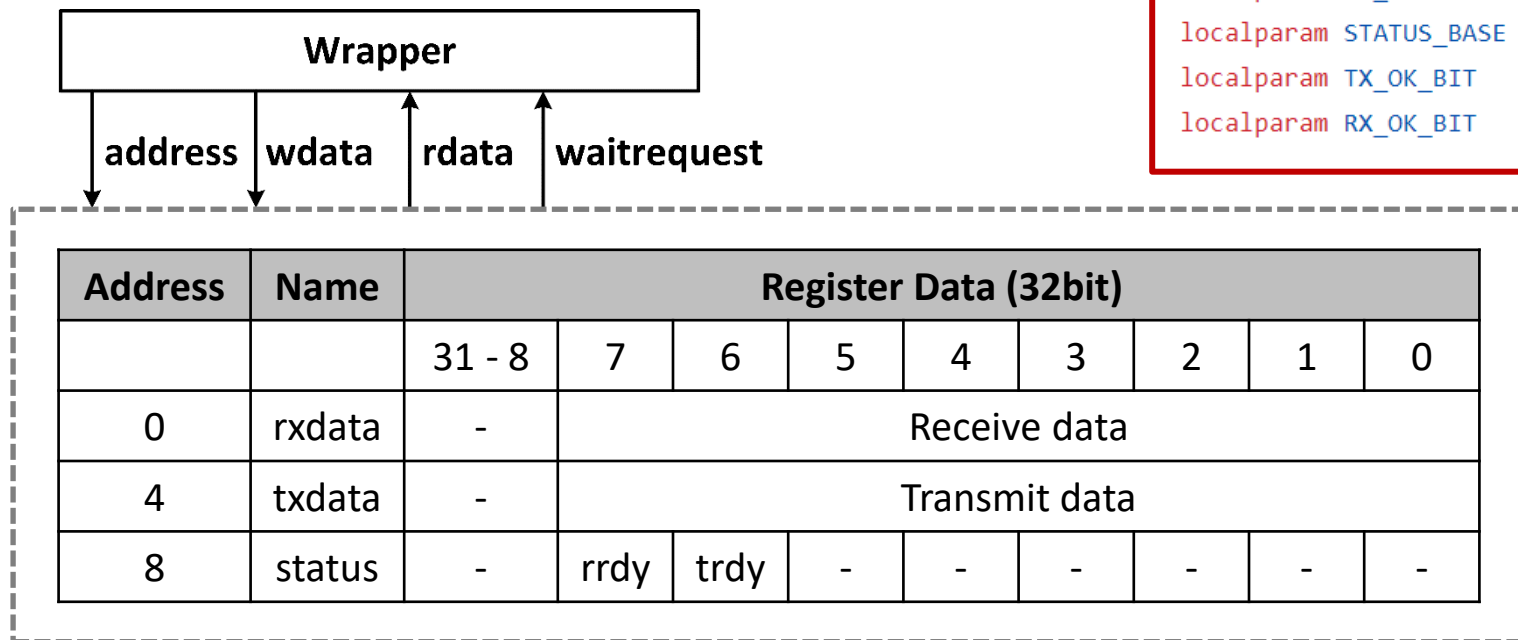
# RSA256 Core

- Divide the functions based on the Algorithm into submodules
  - During PREP, execute submodule **RsaPrep** (ModuloProduct)
    - Calculate  $y \cdot 2^{256} \bmod(N)$
  - During MONT, execute submodule **RsaMont**
    - Calculate  $t \leftarrow t^2 \cdot 2^{-256} \bmod(N)$  and  $m \leftarrow mt \cdot 2^{-256} \bmod(N)$
    - Dedicate one instant to each calculation for parallel processing



# RSA256 Wrapper: RS232 Protocol

- Very old (1969) and very simple protocol
  - Only has two signal lines receiver/transmitter (RX/TX)
  - Very slow (~10KB/s)
- Here, we use Qsys IP with register mapping
  - Access data by address BASE+0, 4, 8, ...



```
localparam RX_BASE      = 0*4;  
localparam TX_BASE      = 1*4;  
localparam STATUS_BASE = 2*4;  
localparam TX_OK_BIT    = 6;  
localparam RX_OK_BIT    = 7;
```

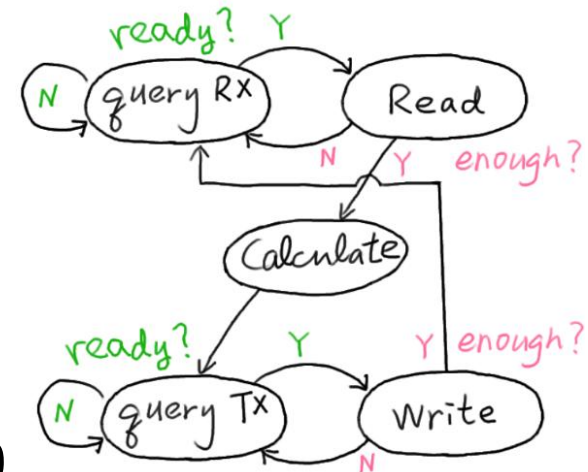
# RSA256 Wrapper: RS232 Explained

---

- rxdata: set avm\_address = 0 to access
  - Read data from avm\_readdata[7:0]
  - only valid when avm\_waitrequest == 0 and rrdy == 1
- txdata: set avm\_address = 4 to access
  - Write data using avm\_writedata[7:0]
  - only valid when avm\_waitrequest == 0 and trdy == 1
- status: set avm\_address = 8 to access
  - Read the 7<sup>th</sup> and 6<sup>th</sup> bit to check the ready signals
  - only valid when avm\_waitrequest == 0
- See “<https://www.intel.com/programmable/technical-pdfs/683130.pdf>” page 152 to learn more

# RSA256 Wrapper: Guideline

- 操作Qsys生成的RS232 IP
  - 先讀入資料(key & cipher text)
  - 讀取完後交給core進行解密
  - 將解密完資料(plain text)寫出
- 每一次讀寫register要確認waitrequest為0
- 在讀寫前要先確定IP準備好了
  - 讀取BASE+8的[7]和[6]（分別代表read & write ready）
  - Ex: 當addr給BASE+8，readdata[7]代表RX準備情況
- 讀寫時每次只有8 bits
  - 所以每一筆256b資料要分32次讀
  - Ex: 當addr給BASE+0，readdata[7:0]是RX送來的8bit資料
- 每一次讀寫完都要切回status確認是否可以讀寫下筆資料



# Outline

---

- Introduction
  - Lab requirements
- RSA256 cryptosystem
- System-on-Chip (SoC) and Qsys
- Implementation
  - RSA256 core
  - RSA256 wrapper
- **Code template**
- System setup and run testing program
- Report regulations

# Code Template

---

- DE2\_115/
  - Design setup files
- pc\_python/
  - Python executable test program for PC
- tb\_verilog/
  - Verilog testbench for core and wrapper
- [Rsa256Core.sv](#)
  - Implement RSA256 decryption algorithm here.
- [Rsa256Wrapper.sv](#)
  - Implement controller for RS232 protocol
  - Including reading check bits and read/write data.

# RSA256Wrapper Interface

---

Signal Name	I/O	Width	Description
avm_rst	Input	1	active high, asynchronous reset
avm_clk	Input	1	25MHz generated from 50MHz with ALTPLL
avm_address	Output	5	designates which registers it is accessing
avm_read	Output	1	indicates that the module is reading from PC
avm_readdata	Input	32	Input data according to the address
avm_write	Output	1	indicates that the module is writing to PC
avm_writedata	Output	32	output data according to the address
avm_waitrequest	Input	1	should only access the registers when this is low

# RSA256Core Interface

---

Signal Name	I/O	Width	Description
i_clk	Input	1	Connected from avm_clk
i_rst	Input	1	Active high, asynchronous reset
i_start	Input	1	Pull high after all data is prepared to start calculation
i_a	Input	256	Cipher text, to be decoded
i_d	Input	256	Encryption key, only sent from PC once
i_n	Input	256	Encryption key, only sent from PC once
o_a_pow_d	Output	256	Decoded text
o_finished	Output	1	Indicating message has been decoded



# Debug Core and Wrapper

---

- Testbench for core and wrapper are provided in tb\_verilog/
- To run simulation for core
  - `vcs tb.sv Rsa256Core.sv ...`
- To run simulation for wrapper
  - `vcs test_wrapper.sv PipelineCtrl.v PipelineTb.v \`  
`Rsa256Wrapper.sv Rsa256Core.sv ...`

(Please refer to *Intro\_debugging\_tools.pdf* for more details)

- Use [nWave](#) to check the waveform and happy debugging!
  - It is advised to test individual modules first

# Tips and FAQ

---

- Remember to import the qsf file we provided
- Be aware of register overflow
  - Two 256-bit numbers added has a range of  $[0, 2^{257}-2]$ , which takes up 257 bits
  - How about 3 such numbers added consecutively? When might this occur?
- Try to test each submodule separately (with makeshift testbenches) before combining them into a whole system.
  - Easier to pinpoint the problem during debugging
- Write testbenches and use nWave to debug
- It is recommended to use ***ready***, ***valid*** signals to communicate between submodules (Refer to AXI protocol to learn more)

# Outline

---

- Introduction
  - Lab requirements
- RSA256 cryptosystem
- System-on-Chip (SoC) and Qsys
- Implementation
  - RSA256 core
  - RSA256 wrapper
- Code template
- **System setup and run testing program**
- Report regulations

# System Setup



# Testing Program

---

- Environment setup
  - Install Python3
  - `sudo apt-get install python3-pip`
  - `(sudo) pip3 install pySerial`
- Usage
  - Copy key and encrypted data (enc.bin and key.bin) next to the executable
  - `./rs232.py [COM? | /dev/ttyS0 | /dev/ttyUSB0]`
- Several test data are already provided

# Decryption Flow

---

- The executable will
  - Send 32-byte divisor N
  - Send 32-byte exponent d
  - Loop
    - Send 32B cipher text y
    - (Your module calculates the result)
    - Receive 31-Byte plain text x
- Note: a zero byte is padded to the front of each 32-byte plain text to prevent overflow
  - The size of plain text is 31 bytes
  - The size of cipher text in enc.bin is 32 bytes

## FAQ

- 在pc\_python/golden/rsa.py當中有列出加密和解密所使用的n、e、d，使用./rs232.py來傳輸key.bin的時候是把n跟d接在一起

[illegible]

# Outline

---

- Introduction
  - Lab requirements
- RSA256 cryptosystem
- System-on-Chip (SoC) and Qsys
- Implementation
  - RSA256 core
  - RSA256 wrapper
- Code template
- System setup and run testing program
- Report regulations



# Report Regulations

---

- 內容應包含
  - File Structure
  - System Architecture (必須包含Data Path)
  - Hardware Scheduling (FSM or Algorithm Workflow)
  - Fitter Summary 截圖
  - Timing Analyzer 截圖
  - 遇到的問題與解決辦法，心得與建議
- 一組交一份，以pdf檔繳交
- 命名方式：teamXX\_lab2\_report.pdf
  - Ex: team01\_lab2\_report.pdf
- 繳交期限：demo當天午夜
  - 遲交每三天\*0.7

# Submission Rules

---

- 繳交檔案架構

```
team01_lab2
- team01_lab2_report.pdf
- src
  - < all of your verilog code >.v
```

- 將 teamXX\_labX 資料夾包成一個 zip 後，上傳到NTU Cool，一組繳交一份
- src 資料夾內的 Verilog 可自行命名，只要在 report 中有說明層級架構即可
- 繳交期限：**demo 日當天 23:59 前**
- 若未遵守繳交格式會酌情扣分**

# Questions?