

Spezifikation zum Chipentwurf
„Verschlüsselungsadapter“

Moritz Beller,
8.8.2005

Prolog

Der Chip soll mit einer seriellen Schnittstelle eines Computers verbunden werden können. Mittels eines wohldefinierten Protokolls sollen Byte-Folgen an ihn gesendet werden, aus denen der Chip erkennt, ob sie den Key darstellen, oder einen Datenblock, der ver- oder entschlüsselt werden kann. Vor dem Ver- und Entschlüsseln muss einmalig für jedes Anschalten (nach Unterbrechung der Stromversorgung des Chips) ein 256 Bit langer Key übertragen werden. Danach können die Daten, ebenfalls Blöcke von 256 Bit Länge, folgen. Sollte ein Key oder Block < 256 Bit seien, so wird er mit NULL aufgefüllt.

Nach erfolgter Arbeit sendet der Chip den Block wieder zurück – chiffriert, oder dechiffriert, je nach dem, welche Aufforderung über das Protokoll gesendet wurde.

Die Module – Übersicht

chip.v ist das Top-Modul. Es instanziiert `protocol.v` mit dem Namen *mp* und erzeugt eine Instanz des RS232-Topmoduls namens *rs232*.

protocol.v, welches durch `chip.v` instanziiert wird, ruft seinerseits eine Instanz des AES-Moduls auf, nämlich *aes*.

aes.v erzeugt keine neuen Module.

Das chip-Modul

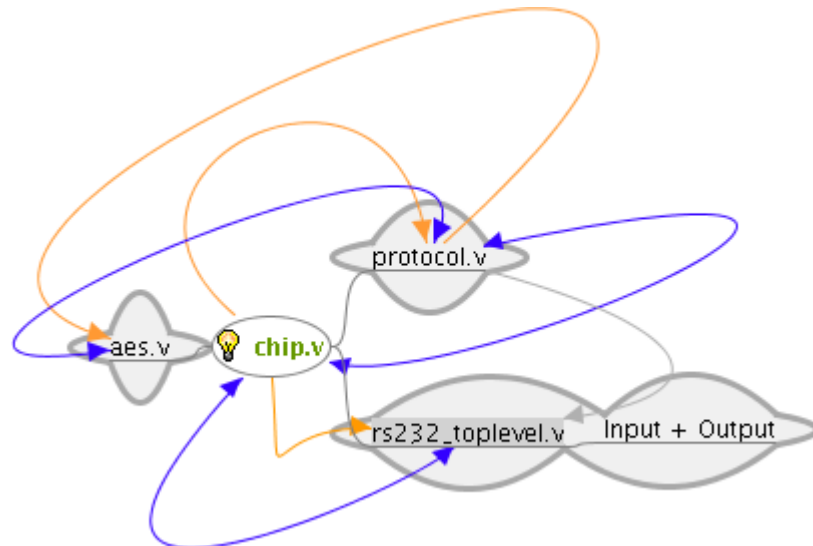
```
module chip (reset, clock);  
    input clock;  
    input reset;
```

Dieses ruft *mp* auf, mit dem Inputstream von RS232.

Es stellt Register bereit, die als Output von *mp* kommen. Diese beinhalten das Ver- bzw. Entschlüsselte. Diese Nutzdaten gibt *chip* wieder über RS232 aus. Die FSM ist sehr rudimentär und kennt nur drei Zustände:

- Es liegen keine Daten von *mp* vor: Dann wird auch nichts über RS232 gesendet (FSM_VOID).
- Muss noch mindestens ein Byte gesendet werden, wechselt die FSM nach FSM_SEND, worin dieses mit Belegung der nets für rs232 erledigt wird.
- War das Senden erfolgreich, wird vom Zustand FSM_SENT wieder nach FSM_SEND

gewechselt, sofern noch zu sendende Bytes existieren.



Orangene Pfeile symbolisieren Instanziierungen, blaue Pfeile zeigen Kommunikation an: **chip** instanziiert **rs232** und **mp**, **mp** instanziiert **aes**. Den Output von **rs232** leitet **chip** an **mp** weiter, das diesen Stream nach den im Protokoll definierten Merkmalen untersucht und bei erfolgreicher Überprüfung an **aes** weiterleitet.

Ist **aes** fertig mit der ihm aufgetragenen Operation, geht der Weg genau umgekehrt: Die Daten finden über **mp** nach **chip** ihren Weg, wovon sie via **rs232** nach außen verschickt werden.

Das mp-Modul

```
module miniprotocol (istream, valid, gotstate, readytosend, reset, clock);  
    input [7:0] istream;  
    input      clock;  
    input      reset;  
    input      valid;  
    output     readytosend;  
    output [255:0] gotstate;
```

istream ist der RS232-Inputstream.

valid ist ein Handshakesignal von RS232: Ist *istream* tatsächlich korrekt?

gotstate speichert das Verschlüsselte/Entschlüsselte und kommt von *aes*

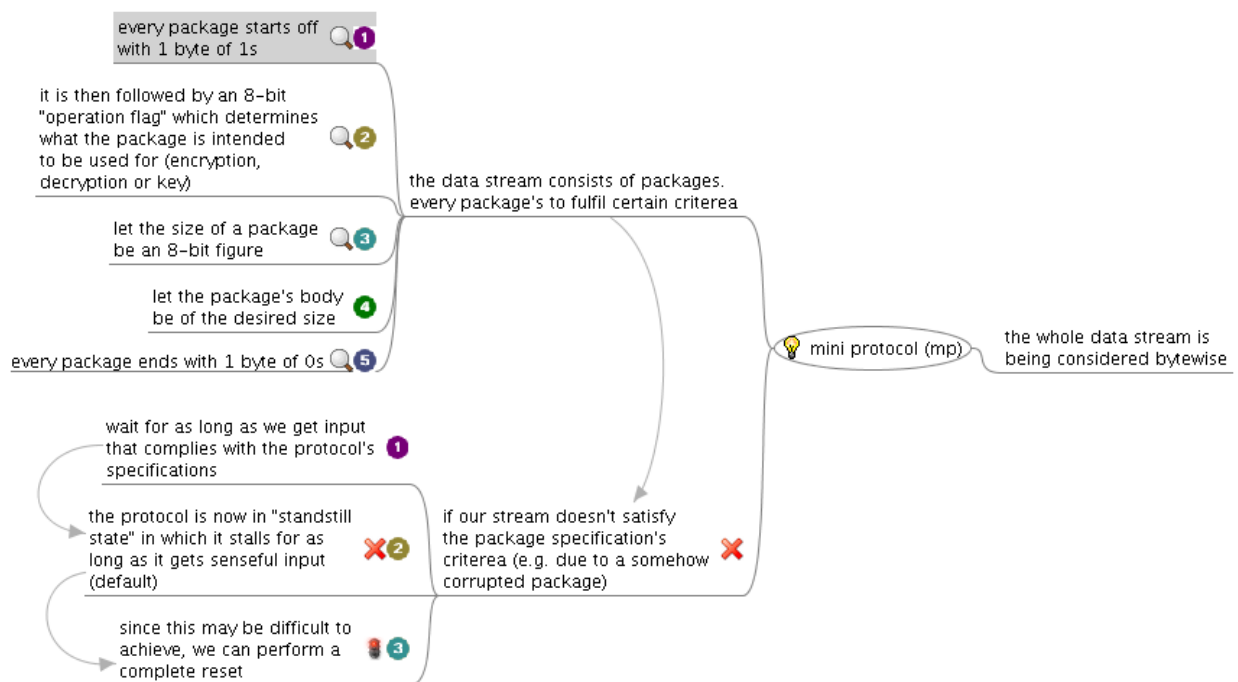
readytosend ist ein weiteres Handshakesignal und zeigt an, ob *gotstate* auch wirklich die fertige

Version ist.

Miniprotocol hat die FSM-Zustände

PFS_VOID = 3'b000,
PFS_HEADER = 3'b001,
PFS_OF = 3'b010,
PFS_PSIZE = 3'b011 und
PFS_BODY = 3'b100.

Sie entsprechen den Bezeichnungen im Bild. OF steht für operation flag.



Beispiel: Um einen Key von 1 Bit Länge und dem Inhalt „1“ zu übertragen, ist Folgendes nötig:

```
/* Starting a test stream */  
  
#10 _input = 8'b11111111; // every package starts off with 1 byte of 1s  
  
#10 _input = 8'b10000000;  
  
#10 _input = 8'b00000000; // there is no package body for the start of  
streams!  
  
#10 _input = 8'b00000000; // end of package
```

Hier wird zuerst ein valider Stream gestartet, dh. ab jetzt dürfen key und block folgen (setzt

stream in **mp** auf 1). Dann kommt der Key:

```
/* Sending a key */
    #10 _input = 8'b11111111; // every package starts off with 1 byte of
1s
    #10 _input = 8'b00000100; // operation flag: 10 = encryption, 11 = de
..., 100 = keytransmitting
    #10 _input = 8'b00000001; // length of package (usually = 256/8)
    // transmit key:
    #10 _input = 8'd1;          // value of ey
    #10 _input = 8'b00000000;
```

key in **mp** hat nun den Wert 1. Zum Schluss sollte man den Stream noch beenden. Davor können jedoch beliebig viele Datenblöcke übertragen werden, die gemäß ihres „operation flag“ entweder ver- oder entschlüsselt werden, und zwar mit Zuhilfenahme des gerade übermittelten Keys. Sofern nicht noch einmal ein neuer Key übertragen wird, verwenden alle den gerade übertragenen von „1“.

```
/* Exiting the test stream: We are polite and finish it properly */
    #10 _input = 8'b11111111; // every package starts off with 1 byte of 1s
    #10 _input = 8'b11000000; // end - operation - flag
    #10 _input = 8'b00000000; // there is no package body for the end of
streams!
    #10 _input = 8'b00000000; // end of package
```

Das aes-Modul

```
module aesmain (ikey, block, direction, valid, cipherstate, makevalid,
reset, clock);
    input [255:0] ikey;
    input [255:0] block;
    input          direction;
    input          valid; // handshake signal. comes from protocol.v: is ikey
/ block valid?

    output [255:0] cipherstate;
    output          makevalid; // third handshake signal. have we finished
en/decrypting?
```

```
input      reset;
input      clock;
```

`ikey` ist der Inputkey, wie **mp** ihn liefert.

`block` ist der zu ver/entschlüsselnde Block, wie **mp** ihn liefert

`direction` ist die Richtung: Soll entschlüsselt (= 0) oder verschlüsselt (= 1) werden? Wird von **mp** übermittelt.

`valid` ist ein weiteres Handshakesignal, das von **mp** kommt: Sind `block` und `ikey` wirklich fertig gesammelt?

`cipherstate` trägt den Wert der aktuellen Ver/Entschlüsselungsrunde.

Für `makevalid = 1` wurden alle Runden durchlaufen und die Operation ist fertig. `Makevalid` ist ferner ein weiteres Handshake-Signal, das **mp** anzeigt, wann es den Output an chip weiterleiten kann (über die Variable `gotcipherstate`), damit nicht versehentlich ein noch in der Chiffrierphase befindlicher `cipherstate` über RS232 ausgegeben wird.

Die FSM von AES kennt nur 4 Zustände für AES, ist aber trotzdem nicht trivial!

```
FSM_SubBytes = 3'b000,
FSM_ShiftRows = 3'b001,
FSM_MixColumns = 3'b010,
FSM_AddRoundKey = 3'b011,
FSM_GETKEY = 3'b110,
FSM_GETBLOCK = 3'b111;
```

In `GETKEY` wird `ikey` gelesen. Zum Einlesen in das Array `keyring` wird nach dem Muster vorgegangen, das in FIPS 197 (siehe docu-Verzeichnis) Pflicht ist: `prepareblock`. Eine einfache Übernahme des Blocks führt zu völlig anderen Ergebnissen, die zwar auch reversibel sind, dann aber keine korrekte AES-Implementierung gemäß FIPS 197 darstellen.

`GET_BLOCK` tut Selbiges für den Block.

Für die übrigen Zustände siehe das Diagramm.

In der letzten Runde wird der Zustand `FSM_MixColumns` definitionsgemäß ausgelassen.

Nach 14 Runden ist AES fertig. `cipherstate` enthält dann das Ergebnis der gewünschten Operation, also entweder Ver- oder Entschlüsselung.

Die Entschlüsselung ist genau das Gegenstück zur Verschlüsselung. Ob entschlüsselt (1) oder verschlüsselt (0) wird, zeigt `direction` an. Was bedeutet das nun aber? Zum Entschlüsseln wird mit `round = 14`, statt mit `round = 1` wie beim Verschlüsseln, gestartet. Die FSM-Zustände werden genau umgekehrt durchlaufen: Von `FSM_AddRoundKey` geht es nach `FSM_ShiftRows` und dann nach `FSM_SubBytes`. Erst ab `round = 13` wird `FSM_MixColumns` durchgeführt.

Das Modul `aes` implementiert Rijndael in seiner „maximalen“ Stufe: Mit 256 Bit Keylänge und 256 Bit Schlüssellänge. Dies hat Auswirkungen auf die Funktionen `ShiftRows`, `MixColumns` und `KeySchedule`.

Warum also wurden 256 Bit gewählt? Um 128 Bit zu encoden, wären 10 Runden, dh. 10 nacheinander ausgeführte Durchläufe von `FSM_SubBytes`, `FSM_ShiftRows`, `FSM_MixColumns`, `FSM_AddRoundKey` nötig gewesen. Mit 256 Bit, also der doppelten Datenmenge, sind es aber nur 14, dh. effektiv werden 6 volle Rundengänge mit je 4 Operationen, also insgesamt 24 Operationen gespart. Dieser Vorteil sollte genutzt werden.

Die Keylänge von 256 bringt größere Sicherheit, sind schon erste theoretische Attacken auf AES denkbar („8000 quadratische Gleichungen mit 16000 Unbekannten“).

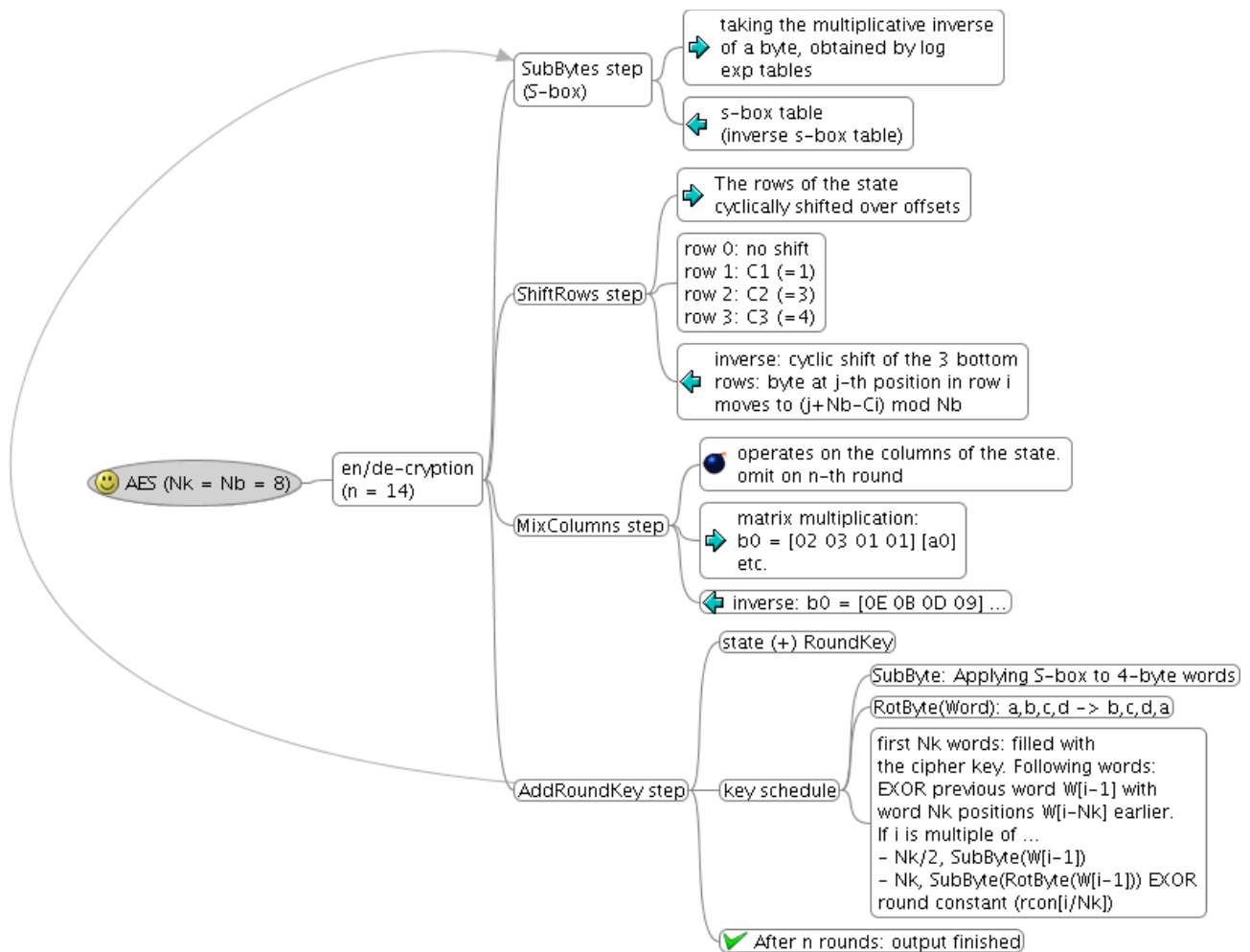
Zum Verständnis von AES ist es nötig zu wissen, dass sämtliche Operationen auf den Klartextblock beim Verschlüsseln, beim Entschlüsseln natürlich auf den chiffrierten Block, `cipherstate` angewendet werden.

Jeder der Tasks `SubBytes`, `ShiftRows`, `MixColumns` und `AddRoundKey` nimmt an `cipherstate` Veränderungen vor.

Zu Beginn wird `cipherstate` der Wert des einkommenden Blockes zugewiesen. Nach Beendigung von AES trägt folglich `cipherstate` auch wieder den fertigen Wert.

Erwähnenswert ist der `KeySchedule`-Task, der die Roundkeys berechnet. Es gibt für jede Runde einen neuen Roundkey aus dem Array `keyring`. Im Zustand `FSM_AddRoundKey`, das den Task `AddRoundKey` aufruft, wird `cipherstate` mit dem aktuellen Roundkey EXORED. Im Gegensatz zu den übrigen Tasks wie `SubBytes`, `ShiftRows` und `MixColumns` wird in `AddRoundKey` keine Unterscheidung durchgeführt, ob Ver- oder Entschlüsselung auf dem Plan steht: `AddRoundKey` ist sein eigenes Invers.

Das Array `keyring` wird sowohl für Ent- als auch Verschlüsselung einmal zu Beginn, direkt nach Erhalten des Keys gefüllt.



Die Zusammensetzung der Funktionen sbox, isbox, logt und expt beruht auf mathematischen Zusammenhängen. Sie könnten auch on-the-fly berechnet werden, würden dann aber einen wesentlich größeren Rechenaufwand verursachen, als dies bei fester Einspeicherung der Fall ist. Im Diagramm oben ist noch vom „multiplicative inverse“ die Rede. Dieses entfällt dadurch. Nk steht für die Keylänge * 32 und Nb für die Blocklänge * 32.

Epilog

Sämtliche Verilogdateien wurden ausführlich mit Kommentaren versehen. Synthesefähigkeit ist gegeben, tatsächliche Chipfähigkeit konnte leider in der Gesamtkombination nicht getestet werden (leider nur einzelne Teilfunktionen in aes, so z.B. sbox und mul).