Handshake signaling for data transfer

Oprițoiu Flavius
flavius.opritoiu@cs.upt.ro

November 18, 2024

Introduction

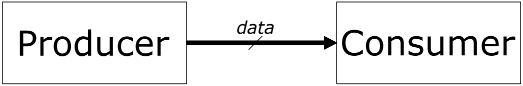
Objectives:

- Use handshake signaling for reliable data exchange between digital components

For reading:

- ❗ Chris Fletcher: "EECS150: Interfaces: "FIFO" (a.k.a. Ready/Valid)", [Flet09c]

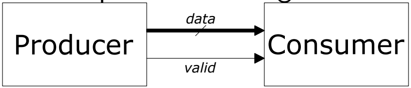
Handshake signaling permits rate adaption for the case of data transfers. For brevity, consider the case of two sequentially, synchronous components, exchanging data. One component is said to produce data (the producer) and the other is said to consume data (the consumer). The two are exchanging data in packets, using a common data bus.



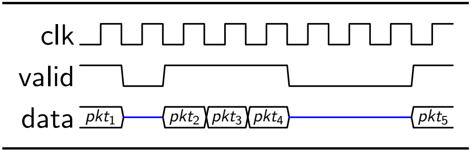
Adjusting transfer rate

If the producer can generate one data packet each clock cycle and the consumer is capable of processing it in the same clock cycle no handshake signaling is needed.

If the producer cannot generate packets at the rate the consumer can process them, a signal *valid* is added to producer, as an output, asserted when a new packet was put on the common data lines. The situation is depicted in the figure bellow:

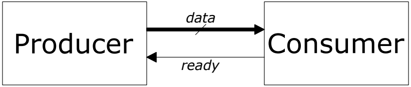


Consumer polls, each clock cycle, producer's *valid* output and, if asserted, retrieves the packet for further processing. If *valid* is not active consumer waits its assertion.

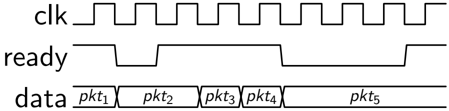


Adjusting transfer rate (contd.)

If the consumer cannot process packets at the rate the producer generates them, an output signal, *ready*, is added to consumer, asserted when new data can be received. The situation is depicted in the figure bellow:

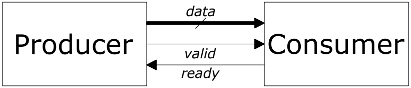


Producer puts the current packet on the data lines. It then polls the *ready* line and, if asserted, continues with generating the next packet, otherwise keeps the current packet on the data lines. The timing diagram bellow exemplifies the transfer:

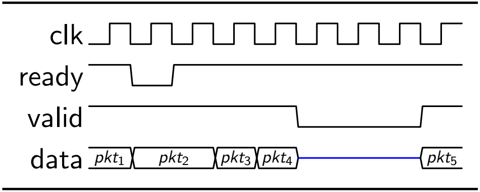


Handshake signaling

If both producer's generating new data and consumer's processing current packet require additional time, a valid/ready handshake protocol can be used. The *valid* and *ready* signals operates as described in the previous 2 slides. This interface is depicted bellow:



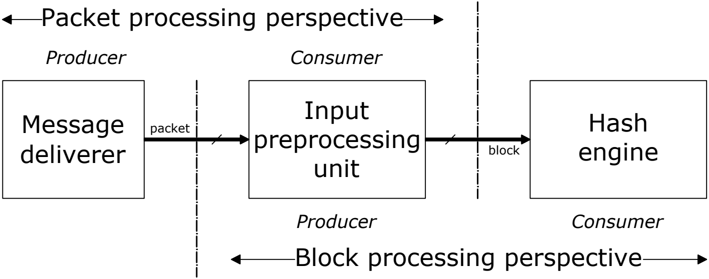
The interface is known also as FIFO interface, or "Ready/Valid" interface. Data transfer takes place on the rising edge of the clock cycle, when both *valid* and *ready* signals are asserted.



Solved problem

Control unit for the input preprocessing module of a cryptographic application

The input preprocessing unit (IPU, or simply *the unit*) of a Secure Hash Algorithm 2 (SHA-2) unit receives 64-bit packets of the message and assembles and delivers them, in 512-bit blocks, to the hash engine [FIPS15]. After receiving the complete message, the unit pads it and append message's size. The unit's data flow is depicted in the figure below. In either perspective, one component produces data and another one consumes it.



Solved problem (contd.)

The packet processing perspective

For brevity, no handshake signaling is used in communication with the *message deliverer*. After activation of the reset signal, *rst_b*, in each clock cycle, a new packet is delivered to the unit. The last packet of the message is marked by the deliverer by activating its *lst_pkt* output.

The unit reads and stores a new packet in each clock cycle, delivering to the *hash engine*, a new block once every 16 cycles. After receiving the last packet (asserted *lst_pkt*) the unit adds one padding packet and, depending on the length of the message, it appends 0, 1 or more zero packets. In the last block deliver at its output, the unit append in the least significant 64 bits the length in bits of the message.

Solved problem (contd.)

The block processing perspective

For communicating with the hash engine, a *valid* signal is used, *blk_val* asserted by the unit when a new block is available at its output. When delivering, the last block, along activation of *blk_val*, the unit asserts another output, *msg_end* marking completion of message transmission.

Control unit of the input preprocessing module:

- implements SHA-2's preprocessing phase
- manages unit's datapath
- evaluate signals from the *message deliverer*
- signal specific conditions to the *hash engine*

Solved problem (contd.)

SHA-2 input preprocessing algorithm

```
procedure DeliverMessage
    $MessageLength \leftarrow 0$    ▷ set message size to 0
    $index \leftarrow 0$    ▷ set current register file index to 0
    loop
        $RegisterFile[index] = packet$    ▷ store received packet in register file
        $index \leftarrow (index + 1) \bmod 2^3$    ▷ increment register file index
        $MessageLength \leftarrow MessageLength + 64$    ▷ increment message's size
        if $index == 0$ then signal new block
        if $lst\_pkt == 1$ then    ▷ start message padding
            $RegisterFile[index] =$ 64'h8000000000000000)    ▷ 1 followed by 63 of 0s
            $index \leftarrow (index + 1) \bmod 2^3$
            while $index \neq 7$ do    ▷ when index is 7, append message size packet
                if $index == 0$ then signal new block
                $RegisterFile[index] =$ 64'h0000000000000000    ▷ all-zero packet
                $index \leftarrow (index + 1) \bmod 2^3$
            end while
            $RegisterFile[index] = MessageLength$    ▷ message size packet
            signal new block
            signal message end    ▷ message was transmitted completely
            break    ▷ leave transmission loop
    end loop
end procedure
```

Solved problem (contd.)

Control path of the unit

The signals used by the control path to manage the datapath:

- *c_up*: increments the counter storing register file's index
- *clr*: clears the register storing the message size and the counter storing register file's index
- *mgln_pkt*: appends the message length packet
- *pad_pkt*: appends a padding packet
- *st_pkt*: stores the current packet in the register file
- *zero_pkt*: appends a zero packet

The control path takes the register file's current index, *idx*, as input.

Solved problem (contd.)

Unit's interface with message deliverer and hash engine

Unit's interface signals with the message deliverer:

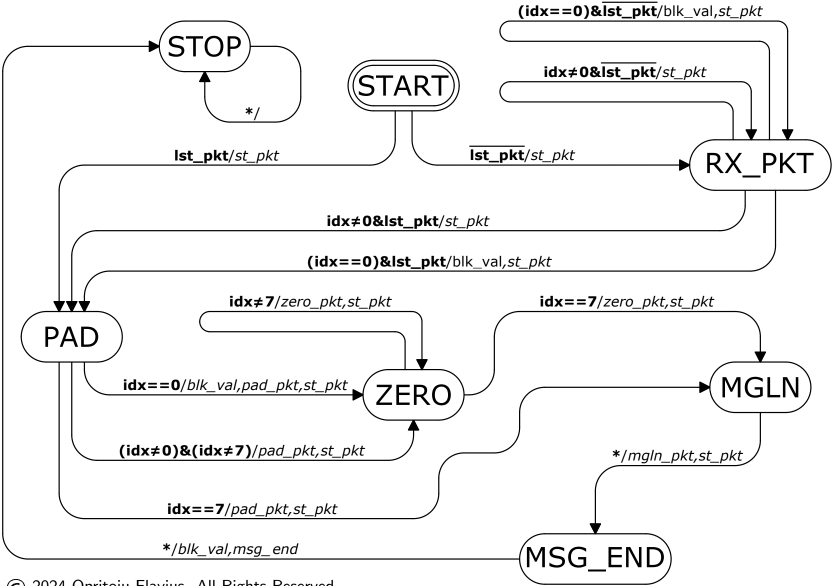
- *lst_pkt* (input): asserted when receiving the final packet
- *pkt* (64-bit input): current message packet

Unit's interface signals with the hash engine:

- *blk* (512-bit output): the current block
- *blk_val* (output): asserted when a new block is available
- *msg_end* (output): asserted when delivering the final block

Solved problem (contd.)

Control unit's transition diagram


STOP
START
RX_PKT
PAD
ZERO
MGLN
MSG_END
*/
(idx==0)&$\overline{\textbf{lst\_pkt}}$/blk_val,*st_pkt*
idx≠0&$\overline{\textbf{lst\_pkt}}$/*st_pkt*
lst_pkt/*st_pkt*
$\overline{\textbf{lst\_pkt}}$/*st_pkt*
idx≠0&lst_pkt/*st_pkt*
(idx==0)&lst_pkt/blk_val,*st_pkt*
idx≠7/*zero_pkt,st_pkt*
idx==7/*zero_pkt,st_pkt*
idx==0/*blk_val,pad_pkt,st_pkt*
(idx≠0)&(idx≠7)/*pad_pkt,st_pkt*
idx==7/*pad_pkt,st_pkt*
*****/*mgln_pkt,st_pkt*
*****/*blk_val,msg_end*

Solved problem (contd.)

Control unit's transition diagram

Preprocessing module's control unit is defined as a Mealy Finite State Machine (FSM).

Internal states:

- **START**: initial state, reached after activation of *rst_b*
- **RX_PKT**: receive a new packet until *lst_pkt* is asserted
- **PAD**: append a padding packet
- **ZERO**: append a zero packet
- **MGLN**: append message length packet
- **MSG_END**: announce the end of message transmission
- **STOP**: final state; no output is activated

References

[Flet09c] C. Fletcher. EECS150: Interfaces: "FIFO" (a.k.a. Ready/Valid). [Online]. Available: https://inst.eecs.berkeley.edu/~cs150/Documents/Interfaces.pdf (Last accessed 25/10/2017).

[FIPS15] National Institute of Standards and Technology, "FIPS PUB 180-4: Secure Hash Standard," Gaithersburg, MD 20899-8900, USA, Tech. Rep., Aug. 2015. [Online]. Available: http://dx.doi.org/10.6028/NIST.FIPS.180-4 (Last accessed 06/04/2016).