HW/SW communication, on-chip bus systems

Lecture 09 on Dedicated systems

Teacher: Giuseppe Scollo

University of Catania
Department of Mathematics and Computer Science
Graduate Course in Computer Science, 2017-18

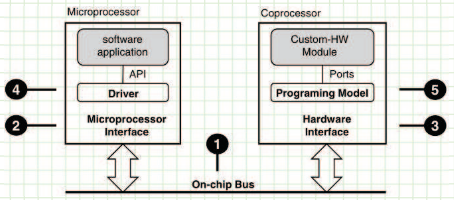
Table of Contents

lecture topics

outline:

- components of the hardware/software interface
- the synchronization problem: concepts and dimensions
- synchronization schemes
  - synchronization with semaphores
  - synchronization with handshakes
  - blocking and nonblocking data transfer
- performance constraint factors: computation vs. communication
- tight or loose coupling
- a few on-chip bus standards
- components and physical implementation of an on-chip bus
- bus timing diagrams
- abstraction of a few standard busses in a generic bus definition

the hardware/software interface



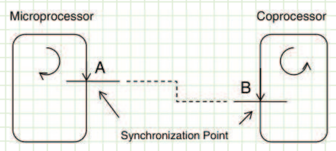
Schaumont, Figure 9.1 - The hardware/software interface

Figure 9.1 presents a synopsis of the elements in a HW/SW interface

the function of the HW/SW interface is to connect the software application to the custom-hardware module; this objective involves five elements:

1. *on-chip bus*: either *shared* or *point-to-point*, it transports data between the microprocessor module and the custom-hardware module
2. *microprocessor interface*: hardware and low-level firmware to allow a software program to 'get out' of the microprocessor, e.g. by coprocessor instructions or memory access instructions
3. *hardware interface*: handles the on-chip bus protocol, and makes the data available to the custom-hardware module through registers or dedicated memory
4. *software driver*: wraps transactions between hardware and software into software function calls, while mapping software data structures into structures that fit hardware communication
5. *programming model*: presents an abstraction of the hardware to the software application; to implement this mapping, the hardware interface may require additional storage and controls

the synchronization problem



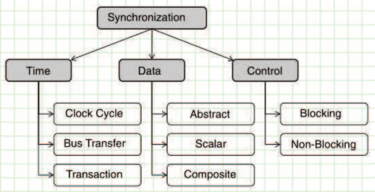
Schaumont, Figure 9.2 – Synchronization point

synchronization: the structured interaction of two otherwise independent and parallel entities

in figure 9.2, synchronization guarantees that point *A* in the execution thread of the microprocessor is tied to point *B* in the control flow of the coprocessor

synchronization is needed to support communication between parallel subsystems: every *talker* needs to have a *listener* to be heard

- e.g., in a dataflow system, hardware and software actors need to synchronize on their token transfers
- even if the dataflow edge is implemented as a FIFO memory, the requirement to synchronize does not go away, for the FIFO has finite capacity, hence the sender needs to wait when the FIFO is full, while the receiver needs to wait when the FIFO is empty



Schaumont, Figure 9.3 – Dimensions of the synchronization problem

three *orthogonal* dimensions of the synchronization problem:

time: time granularity of interactions

data: structural complexity of transferred data

control: relationship between local control flows

synchronization with a semaphore

semaphore: a synchronization primitive *S* to control access over an abstract, shared resource, by operations:

P(S): (try to) get access, wait if *S*=0, else *S*←0

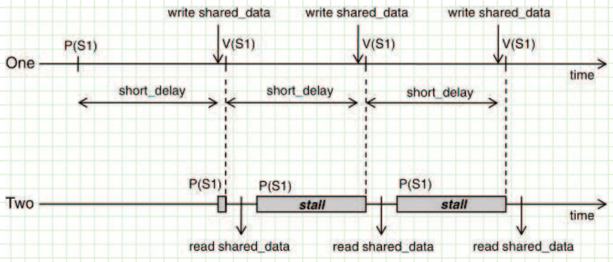
V(S): release resource, *S*←1

```
int shared_data;
semaphore S1;

entity one {
  P(S1);
  while (1) {
    short_delay();
    shared_data = ...;
    V(S1);              // synchronization point
  }
}

entity two {
  short_delay();
  while (1) {
    P(S1);              // synchronization point
    received_data = shared_data;
  }
}
```

Schaumont, Listing 9.1 – One-way synchronization with a semaphore



Schaumont, Figure 9.4 – Synchronization with a single semaphore

synchronization points: when entity one calls V(S1), so unlocking the stalled entity two

this scheme only works under the assumption that entity two is faster in reading the shared data than entity one is in writing it

just assume the opposite, viz. move the `short_delay()` function call from the while-loop in entity one to the while-loop in entity two ...

generally, in the *producer/consumer* scenario, both entities may need to wait for each other

synchronization with two semaphores

the situation of unknown delays can be addressed with a two-semaphore scheme

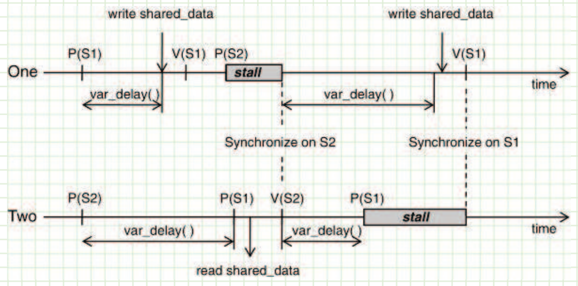
S1 is used to synchronize entity two,
S2 is used to synchronize entity one

```
int shared_data;
semaphore S1, S2;

entity one {
  P(S1);
  while (1) {
    variable_delay();
    shared_data = ...;
    V(S1);   // synchronization point 1
    P(S2);   // synchronization point 2
  }
}

entity two {
P(S2);
  while (1) {
    variable_delay();
    P(S1);   // synchronization point 1
    received_data = shared_data;
    V(S2);   // synchronization point 2
  }
}
```

Schaumont, Listing 9.2 - Two-way synchronization with two semaphores



Schaumont, Figure 9.5 - Synchronization with two semaphores

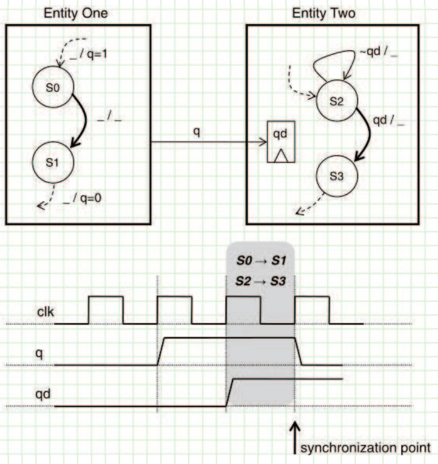
figure 9.5 illustrates the case where:

- on the first synchronization, entity one is quicker than entity two, and the synchronization is done using semaphore S2, whereas
- on the second synchronization, entity two is faster, hence synchronization is done using semaphore S1

synchronization with handshake

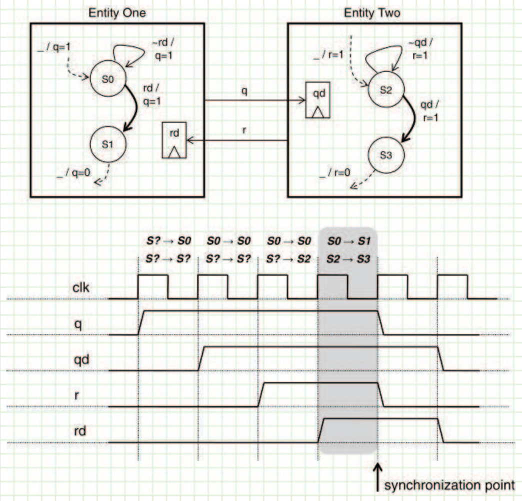
in parallel systems, a centralized semaphore may not be feasible; a common alternative is

a *handshake*: a signaling protocol based on signal levels; the most simple one is:



Schaumont, Figure 9.6 - One-way handshake

one-way handshake has a similar limitation as one-semaphore synchronization, the solution is:



Schaumont, Figure 9.7 - Two-way handshake

blocking and nonblocking data transfer

if a sender or receiver arrives too early at a synchronization point, should it wait idle until the proper condition comes along, or should it go off and do something else?

- a *blocking* data transfer will stall the execution flow of the software or hardware until the data-transfer completes

  e.g., if software has implemented the data transfer using function calls, then these functions do not return until the data transfer has completed

- a *nonblocking* data transfer will not stall the execution flow, but the data transfer may be unsuccessful

  a software function that implements a nonblocking data transfer will need to introduce an additional status flag that can be tested

both of the semaphore and handshake schemes discussed earlier implement a blocking data-transfer

in order to use these primitives for a non-blocking data transfer, the outcome of the synchronization operation should be testable without actually engaging in it

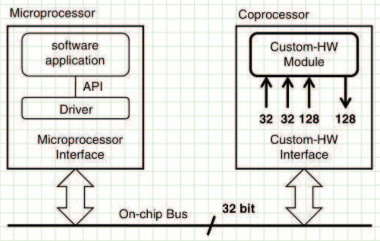
performance constraint factors

computational speedup is often the motivation for the design of custom hardware

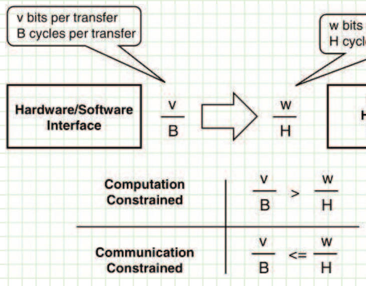
however, the hardware/software interface is also relevant to the resulting *system* performance

communication constraints need to be evaluated as well!

e.g., assume the custom-HW module in fig. 9.8 takes 5 clock cycles to compute the result, with a 320-bit total data transfer size per execution: can the system actually perform at a rate of 320/5 = 64 bits per cycle?



Schaumont, Figure 9.8 - Communication constraints of a coprocessor



Schaumont, Figure 9.9 - Communication-constrained system vs. computation-constrained system

the number of clock cycles needed per execution of the custom hardware module is related to its *hardware sharing factor* (HSF) $=_{def}$ number of available clock cycles in between each I/O event

Architecture	HSF
Systolic array processor	1
Bit-parallel processor	1–10
Bit-serial processor	10–100
Micro-coded processor	>100

Schaumont, Table 9.1 - Hardware sharing factor

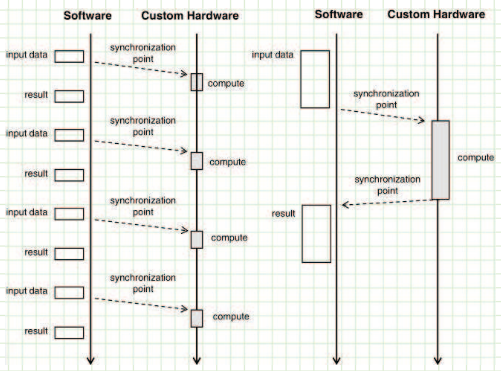
tight or loose coupling

coupling indicates the level of interaction between execution flows in software and custom hardware

tight = frequent synchronization | data transfer

loose = the opposite

coupling relates synchronization with performance



Schaumont, Figure 9.10 - Tight coupling versus loose coupling

Factor	Coprocessor interface	Memory-mapped interface
Addressing	Processor-specific	On-chip bus address
Connection	Point-to-point	Shared
Latency	Fixed	Variable
Throughput	Higher	Lower

Schaumont, Table 9.2 - Comparing a coprocessor interface with a memory-mapped interface

example: difference between

coprocessor interface: attached to a dedicated port on the processor

memory-mapped interface: attached to the memory bus of the processor

N.B.: a high degree of parallelism in the overall design may be easier to achieve with a loosely-coupled scheme than with a tightly-coupled scheme

on-chip bus standards

four families of on-chip bus standards, among the most widely used ones:

- *AMBA* (Advanced Microcontroller Bus Architecture): family of bus systems used by ARM processors
- *CoreConnect*: bus system for the PowerPC line of IBM processors
- *Wishbone*: open-source bus system proposed by SiliCore Corporation, used by many open-source hardware components, e.g. those in the OpenCores project
- *Avalon*: bus system for SoC applications of Altera's Nios processors

two main classes of bus configurations: *shared* and *point-to-point*

further variants depending on speed, interface, topology, etc,, see table 10.1

a generic shared bus and a point-to-point one are considered next, abstracting common features of all of them

Bus	High-performance shared bus	Periferal shared bus	Point-to-point bus
AMBA v3	AHB	APB	
AMBA v4	AXI4	AXI4-lite	AXI4-stream
CoreConnect	PLB	OPB	
Wishbone	Crossbar topology	Shared topology	Point to point topology
Avalon	Avalon-MM	Avalon-MM	Avalon-ST

Schaumont, Table 10.1 - Bus configurations for existing bus standards

Legenda

AHB	AMBA highspeed bus
APB	AMBA peripheral bus
AXI	advanced extensible interface
PLB	processor local bus
OPB	onchip peripheral bus
MM	memory-mapped
ST	streaming

bus components

a shared bus on-chip typically consists of a few *segments*, connected by *bridges*; every transaction is initiated by a bus *master*, to which a *slave* responds; if they are on different segments, then the bridge acts as a slave on one side and as a master on the other side, while performing *address translation*

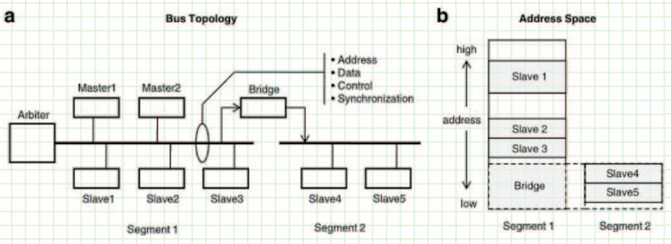
four classes of bus signals:

data: separate data lines for read and write

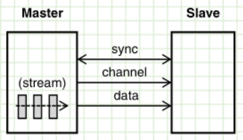
address: decoding may be centralized or local by the slaves

command: to distinguish read from write, often qualified by more signals

synchronization: clocks, distinct per bus segment, and possibly others, such as: handshake signals, time-out, etc.



Schaumont, Figure 10.1 - (a) Example of a multi-master segmented bus system. (b) Address space for the same bus



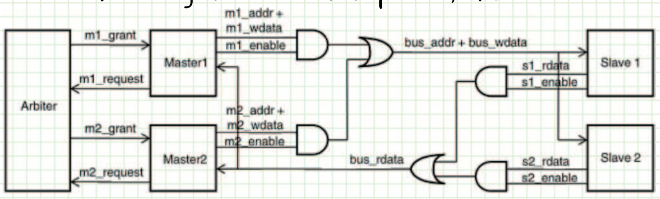
Schaumont, Figure 10.2 - Point-to-point bus

a point-to-point bus is a dedicated physical connection between a master and a slave, for unlimited stream data transfer

- no address lines, but there may be for logical channel, in case of multiplexing of several streams over the same physical bus
- synchronization similar to the handshake protocol seen before

physical implementation of on-chip busses

figure 10.3 shows the physical layout of a typical on-chip bus segment with two masters and two slaves, where AND and OR gates in the center of the diagram serve as multiplexers, of both address and data lines



Schaumont, Figure 10.3 - Physical interconnection of a bus. The *_addr, *_wdata, *_sdata signals are signal vectors. The *_enable, *_grant, *_request signals are single-bit signals

signal naming convention about read/write data:

writing data means sending it from master to slave

reading data means sending it from slave to master

bus arbitration ensures that only one component may drive any given bus line at any time

naming conventions help one to infer functionality and connectivity of wires based on their names

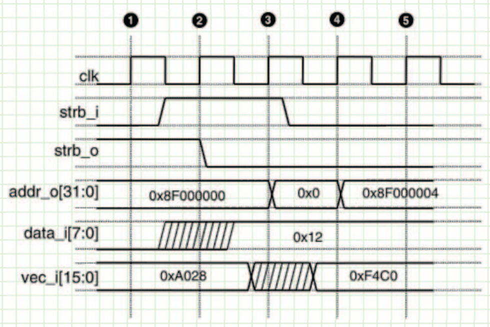
for example, a naming convention is very helpful to read a timing diagram, or to visualize the connectivity in a (textual) netlist of a circuit

a component pin name will reflect the functionality of that pin; bus signals, which are created by interconnecting component pins, follow a convention, too, in order to avoid confusion between similar signals

e.g., in figure 10.3, each of two master components has a wdata signal; to distinguish these signals, the component instance name is the prefix in the bus signal name (e.g. m2_wdata)

bus timing diagrams

because of the inherently parallel nature of a bus system, timing diagrams are extensively used to describe the timing relationships of bus signals



Schaumont, Figure 10.4 - Bus timing diagram notation

the diagram in figure 10.4 shows the notation to describe the activities in a generic bus over five clock cycles

- all signals are referenced to the upgoing edge of the clock signal, shown on top
- input signals in a clock cycle take their value *before* its starting clock edge
- output signals established in a clock cycle take their value *after* its ending clock edge

bus timing diagrams are very useful to describe the activities on a bus as a function of time

they are also a central piece of documentation for the design of a HW/SW interface

a generic bus definition

table 10.2 lists the signals that make up a generic bus, abstracting from any specific system

Signal name	Meaning
clk	Clock signal. All other bus signals are references to the upgoing clock edge
m_addr	Master address bus
m_data	Data bus from master to slave (write operation)
s_data	Data bus from slave to master (read operation)
m_rnw	Read-not-Write. Control line to distinguish read from write operations
m_sel	Master select signal, indicates that this master takes control of the bus
s_ack	Slave acknowledge signal, indicates transfer completion
m_addr_valid	Used in place of m_sel in split-transfers
s_addr_ack	Used for the address in place of s_ack in split-transfers
s_wr_ack	Used for the write-data in place of s_ack in split-transfers
s_rd_ack	Used for the read-data in place of s_ack in split-transfers
m_burst	Indicates the burst type of the current transfer
m_lock	Indicates that the bus is locked for the current transfer
m_req	Requests bus access to the bus arbiter
m_grant	Indicates bus access is granted

Schaumont, Table 10.2 - Signals on the generic bus

correspondence of standard busses to the generic bus

table 10.3 shows the correspondence of some of the generic bus signals to equivalent signals of the CoreConnect/OPB, AMBA/APB, Avalon-MM, and Wishbone busses

generic	CoreConnect/OPB	AMBA/APB	Avalon-MM	Wishbone
clk	OPB_CLK	PCLK	clk	CLK_I (master/slave)
m_addr	Mn_ABUS	PADDR	Mn_address	ADDR_O (master)
				ADDR_I (slave)
m_rnw	Mn_RNW	PWRITE	Mn_write_n	WE_O (master)
m_sel	Mn_Select	PSEL		STB_O (master)
m_data	OPB_DBUS	PWDATA	Mn_writedata	DAT_O (master)
				DAT_I (slave)
s_data	OPB_DBUS	PRDATA	Mb_readdata	DAT_I (master)
				DAT_O (slave)
s_ack	Sl_XferAck	PREADY	Sl_waitrequest	ACK_O (slave)

Schaumont, Table 10.3 - Bus signals for simple read/write on Coreconnect/OPB, ARM/APB, Avalon-MM and Wishbone busses

references

recommended readings:

Schaumont, Ch. 9, Sect. 9.1-9.4

Schaumont, Ch. 10, Sect. 10.1

readings for further consultation:

Schaumont, Ch. 10, Sect. 10.2-10.4