Principles of HW/SW communication

Lecture 09 on Dedicated systems

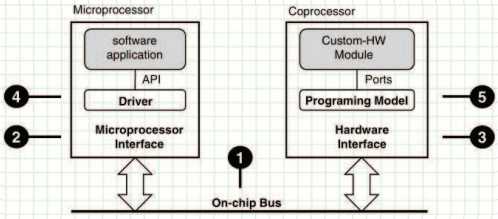
Teacher: Giuseppe Scollo

University of Catania
Department of Mathematics and Computer Science
Graduate Course in Computer Science, 2016-17

Table of Contents

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



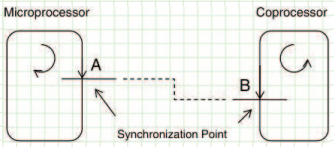
Schaumont, Figure 9.1 - The hardware/software interface

Figure 9.1 presents a synopsis of the elements in a hardware/software interface

the function of the hardware/software 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



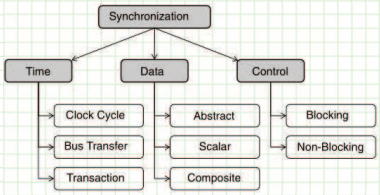
Schaumont, Figure 9.2 - Synchronization point

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 $S \leftarrow 0$

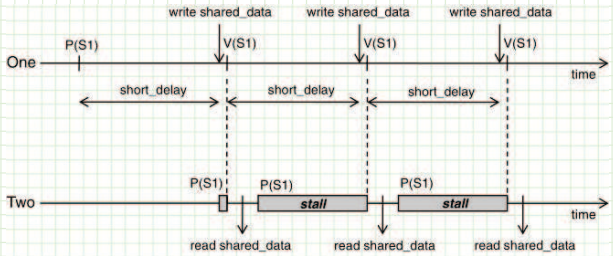
$V(S)$: release resource, $S \leftarrow 1$

```
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

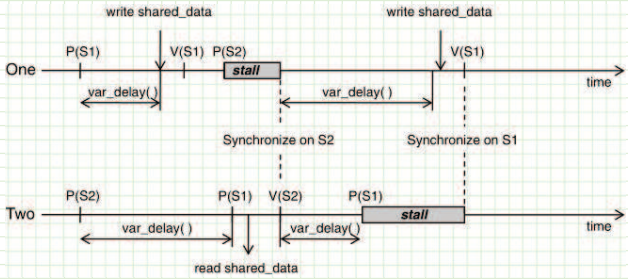
S1 is used to synchronize entity two, S2 is used to synchronize entity one

```
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

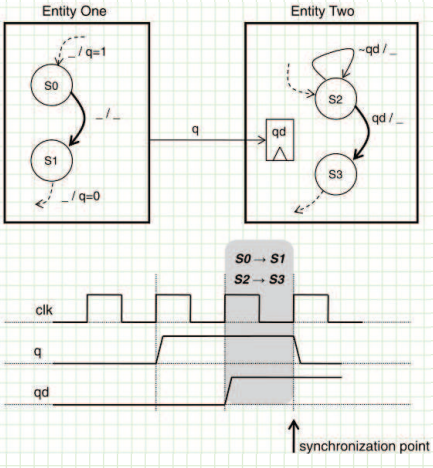
figure 9.5 illustrates the case where:

- 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, and in this case the synchronization is done using semaphore S1

synchronization with handshake

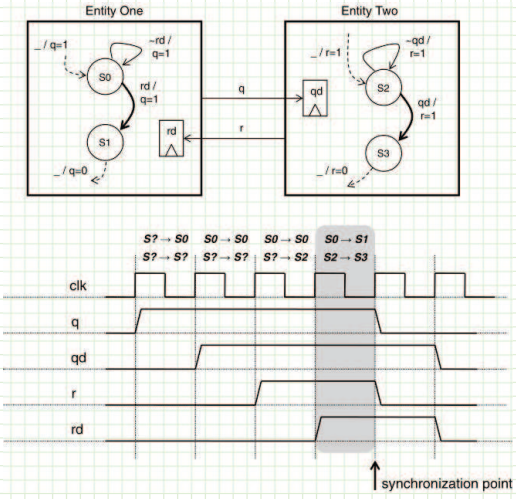
in parallel systems, a centralized semaphore may not be feasible; a common alternative is

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

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

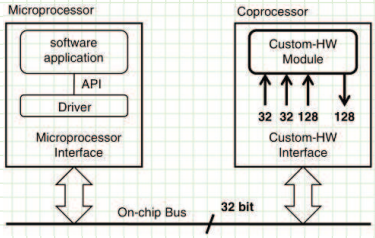
performance constraint factors

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

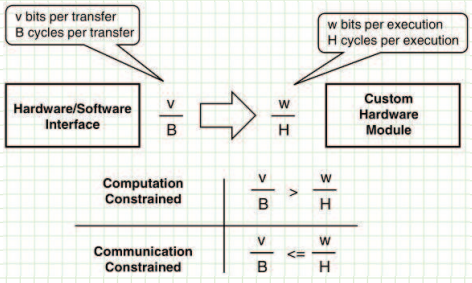
- however, the hardware/software interface is also relevant to the resulting *system* performance

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

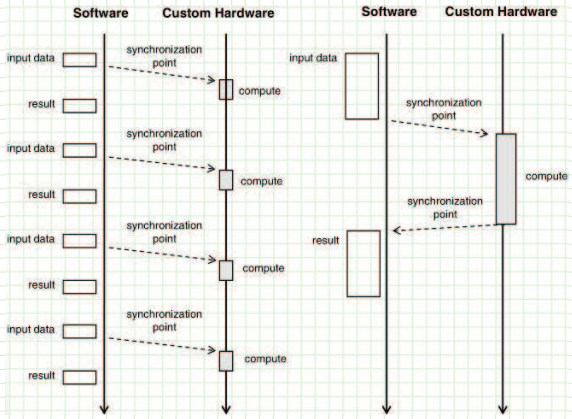
tight or loose coupling

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.: achieving 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

references

recommended readings:

Schaumont (2012) Ch. 9, Sect. 9.1-9.4