Hardware interfaces

Lecture 11 on Dedicated systems

Teacher: Giuseppe Scollo

University of Catania
Department of Mathematics and Computer Science
Graduate Course in Computer Science, 2019-20

Table of Contents

lecture topics

outline:

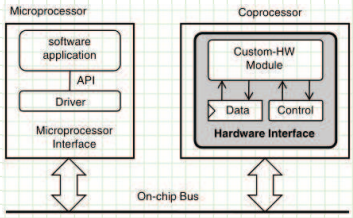
- the coprocessor hardware interface: functions and layout
- hardware data design
  - data addressing
  - multiplexing and masking
- control design
  - hierarchical control
- address map
- instruction set
- example: a hardware acceleration case
  - design decisions
  - Avalon interface and programming model

functions of the coprocessor hardware interface

a hardware interface connects a custom hardware module to a coprocessor bus or an on-chip bus

the hardware interface steers the I/O ports of the custom hardware module

hardware interface design should match the flexibility of custom hardware design to the realities of the hardware/software interface

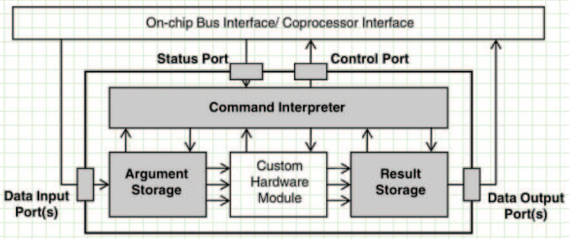


Schaumont, Figure 12.1 - The hardware interface maps a custom-hardware module to a hardware-software interface

typical functions of the hardware interface:

- *data transfer*: read/write operations on the on-chip bus, or handshakes on a coprocessor bus, or even optimized high-throughput or bursty data transfers, e.g. using a DMA controller
- *wordlength conversion*: data operands of the custom-hardware module can be arbitrary in size and number, whereas data suitable for on-chip bus communication are limited in both respects
- *operand storage*: local storage for arguments and parameters of the custom hardware: arguments are updated every time the module executes; parameter updates may be infrequent
- *instruction set*: the custom instruction set is a key feature of the hardware interface, its design defines the software view of the custom hardware component
- *local control*: implementation of local control interactions with the custom hardware module, such as sequencing a series of microoperations in response to a single software command

layout of the coprocessor hardware interface



Schaumont, Figure 12.2 - Layout of a coprocessor hardware interface

components commonly found in a hardware interface:

- a data input buffer for *argument storage*
- a data output buffer for *result storage*
- a *command interpreter* for local control based on software commands

from the perspective of the custom hardware module, it is common to partition the collection of ports into data input/output ports and control/status ports

> note that, in fig. 12.2, control signals and data signals are orthogonal: control flows vertically, and data flows horizontally

the separation of control and data is an important design aspect, for, in a coprocessor design, the granularity of interaction between data and control is chosen by the designer

> both *data-design* and *control-design* are going to be discussed next

data addressing

features of a coprocessor data port: *wordlength*, *direction* and *update rate*

> extremes for the update rate are: *parameter*, only set upon module reset, and *function argument*, which changes value upon each execution of the hardware module

to make a good mapping of actual hardware ports to custom interface ports, it is convenient to start from the features of the actual hardware ports

> e.g., a coprocessor for the GCD function, specified as `int gcd(int m, int n)`, would have two input ports and one output port, all 32-bit wide, with frequent update rate

when this module is implemented as a memory-mapped coprocessor, the ports of the hardware interface will be implemented as memory-mapped registers

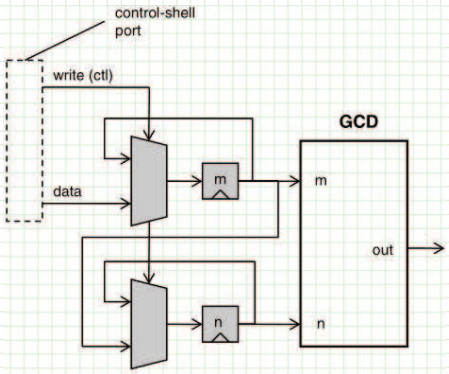
> a straightforward approach is to map each actual hardware port to a distinct memory-mapped register, as this makes each port of the hardware module independently addressable from software

however, it may not always be possible to allocate an arbitrary number of memory-mapped ports in the hardware interface—in that case, one needs to multiplex the custom-hardware module ports over the hardware interface ports

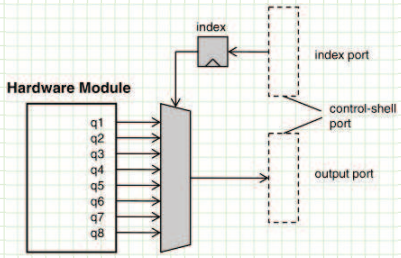
> besides shortage of interface ports, another reason for multiplexing may be the infrequent update rate of some ports of the custom hardware module, so that it is inefficient to allocate a separate interface port for each of them

multiplexing and masking

multiplexing can be implemented in different ways: the first is *time-multiplexing* of the hardware module ports; the second is to use an *index register* in the hardware interface



Schaumont, Figure 12.3 - Time-multiplexing of two hardware-module ports over a single control-shell port



Schaumont, Figure 12.4 - Index-register to select one of eight output ports

multiplexing is also useful to handle long operands piecewise, whereby the operand can be provided one piece at a time by means of time-multiplexing

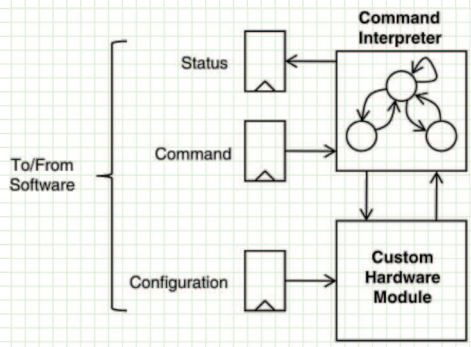
masking is a technique to work with very short operands, e.g. to group several single-bit ports of the hardware module in a hardware interface port: a *mask register* is used to this purpose, to bit-mask the module ports involved in an update, e.g.: `new_hw_port = (old_hw_port & ~mask) | (upd_value & mask)`

control design

control design in a coprocessor is the collection of activities to generate control signals and to capture status signals

> the result is a set of custom-tailored *commands* or instructions that can be executed by the coprocessor

figure 12.5 shows a generic architecture to control a custom hardware module

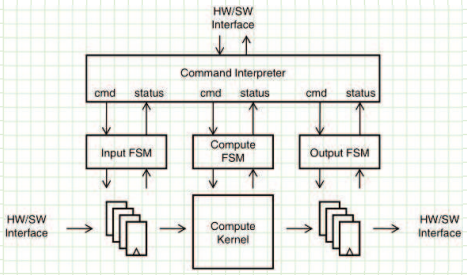


Schaumont, Figure 12.5 - Command design of a hardware interface

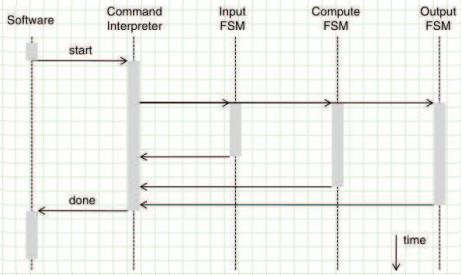
- a *command interpreter*, the top-level controller in the coprocessor, accepts commands from the software and returns status information
- while a command is a one-time control operation, a *configuration* is a value which will affect the execution of the coprocessor over an extended period of time, possibly over multiple commands

hierarchical control

figure 12.6 shows the architecture of a coprocessor that can achieve communication/computation overlap, as illustrated in figure 12.7



Schaumont, Figure 12.6 - Hierarchical control in a coprocessor



Schaumont, Figure 12.7 - Execution overlap using hierarchical control

the command interpreter analyzes each command from software and splits it up into a combination of commands for the lower-level FSMs

> e.g., for a coprocessor which has an addressable register set in the input or output buffer, the register address may be embedded into the command coming from software

to effectively achieve execution overlap, a *pipelining* of the FSM actions is to be organized, where the command interpreter should adapt to the individual schedules of the lower-level FSMs

address map

programmer's model = control design + data design

> the *programmer's model*, that is the software view of a hardware module, includes a collection of the memory areas used by the custom hardware module, and a definition of the commands (or instructions) understood by the module

the *address map* reflects the organization of software-readable and software-writable storage elements of the hardware module; its design should consider the viewpoint of the software designer rather than the hardware designer, thus:

- a given memory-mapped address should always affect the same hardware register, regardless of whether the operation on it is a read or a write
- by default all memory-mapped registers should be read/write; in some cases, read-only registers are justified, such as for example to implement registers that reflect hardware status information or sampled-data signals; however, there are very few cases that justify a write-only register
- the address map should respect the alignment of the processor; e.g., extracting bits number 5–12 out of a 32-bit word is more complicated than extracting the second byte of the same word

instruction set

the design of a good instruction set is a hard problem, that requires the codesigner to make a proper trade-off between flexibility and efficiency

it strongly depends on the function of the custom-hardware module

here are a few generic design guidelines:

- one can distinguish three classes of instructions: one-time commands, on-off commands, and configurations; their mix affects the general behavior of the hardware module, it should be aimed at minimizing the amount of control interaction between the software driver and the hardware module
- design the synchronization between software and hardware at multiple levels of abstraction, that is, not just at the data transfer level but also at the algorithmic level
- another synchronization problem occurs when multiple software users share a single hardware module; this may be solved either by serializing coprocessor usage or by implementing a context switch in the hardware module
- finally, reset design must be carefully considered; an example of flawed reset design is when a hardware module can only be initialized by means of full system reset—it makes sense to define one or several instructions for the hardware module to handle module initialization and reset

example: design decisions for a hardware acceleration case

a recent lab tutorial presented a software implementation of the delay computation of a Collatz trajectory with given start point

hardware implementations of the same function were the subject of previous lab experiences

e.g. the third lab experience produces a VHDL description of it

the performance measurements carried out on the software implementation show that it consumes almost all of the program execution time

problem: accelerate the program execution by using the hardware implementation of the aforementioned function

a first alternative to evaluate: to integrate the hardware function as a custom instruction or as a memory-mapped coprocessor?

the second option seems better, for at least two reasons:

- the first option is *blocking*
- the data transfer size in each interaction is very small

other design decisions depend on this first decision, as follows

Avalon interface and programming model for the sample case

the VHDL description of the circuit which computes the function is to be embedded into a component equipped with Avalon interfaces for the Clock, Reset, and Avalon MM Slave signals, so as to receive the initial data by a write operation and to return the result by a reply to a read operation

- *multicycle* data transfers are possible thanks to the Avalon signal `waitrequest`, set by the slave to defer the response to a read or write request by an arbitrary number of cycles

addressing of the coprocessor: since the (initial data) write and (final result) read operations take place at different times and have the same data size, a single address suffices

- for the sake of simplicity, it is convenient to use the 32-bit Avalon signals `writedata`, `readdata` in the hardware interface for this address, with internal conversion to 16-bit for the corresponding internal I/O ports of the circuit which computes the function

software driver: two macros and a function may be defined for the bus access software interface: `DC_RESET(d)`, `DC_START(d,x0)`, `unsigned int delay(d)`, where `d` is the address assigned to the coprocessor

- these project ideas will be developped in the next lab tutorial

references

recommended readings:

- Schaumont (2012) Ch. 12, Sect. 12.1-12.3.1, 12.4

for further consultation:

- Schaumont (2012) Ch. 12, Sect. 12.3.2
- Avalon® Interface Specifications, Ch. 1-3, MNL-AVABUSREF, Intel Corp., 2019.10.08