

# Microprocessor interfaces

## Lecture 10 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

1. Microprocessor interfaces
2. lecture topics
3. memory-mapped register
4. integration in the memory hierarchy
5. mailbox
6. FIFO queue
7. shared memory
8. coprocessor interfaces
9. custom instruction interfaces
10. ASIP design flow
11. example: the Nios-II custom-instruction interface
12. register files for Nios-II custom instructions
13. references

outline:

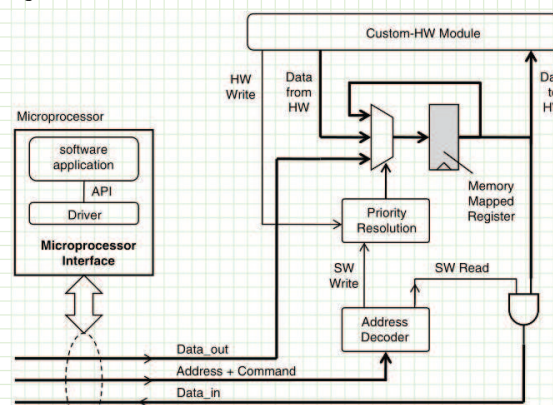
- memory-mapped interfaces
  - memory-mapped register
  - mailbox
  - FIFO queue
  - handshake protocols
  - shared memory
- coprocessor interfaces
- ASIP design flow
- custom-instruction interfaces

example: the Nios-II custom-instruction interface

## memory-mapped register

memory-mapped interfaces are the most general type of HW/SW interface  
in programming they are supported through the use of pointers, e.g.:

```
volatile unsigned int *MMRegister = (unsigned int *) 0x8000;
// write the value '0xFF' into the register
*MMRegister = 0xFF;
// read the register
int value = *MMRegister;
```

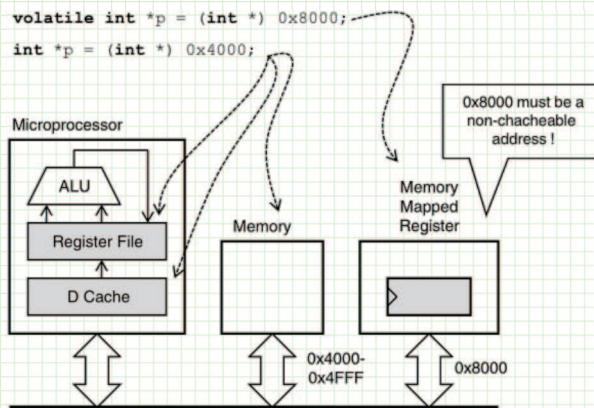


Schaumont, Figure 11.1 - A memory-mapped register

## integration in the memory hierarchy

why must the pointer be a volatile pointer?

- generally, the value stored at an `int *` can appear in three different locations in the memory hierarchy: in main memory, in the cache memory, and in a processor register
- by defining the pointer as a volatile `int *`, the compiler will avoid maintaining a copy of the memory-mapped register in the processor registers



Schaumont, Figure 11.2 - Integrating a memory-mapped register in a memory hierarchy

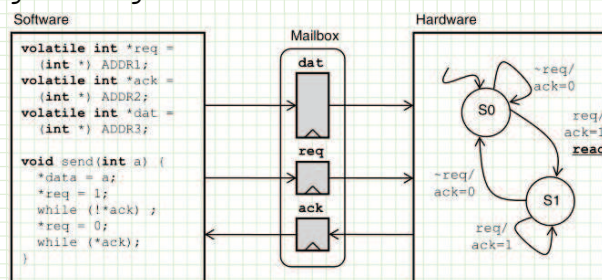
however, defining a memory-mapped register with a volatile pointer will *not* prevent that memory address from being cached!

two approaches to deal with this problem:

- allocation into a non-cacheable memory area, if the processor has a configurable cache (e.g. a Microblaze)
- use of specific cache-bypass instructions of the processor (e.g. a Nios-II)

## mailbox

simple extension of a memory-mapped register with a handshake mechanism, whereby the communicating parties signal the register state to each other



Schaumont, Figure 11.3 - A mailbox register between hardware and software

the protocol shown in the figure has two synchronization points, viz. just after `req` and `ack` taking the same value

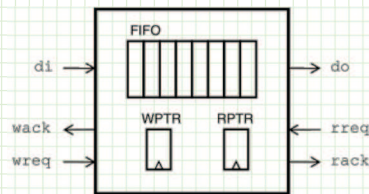
this means that it should be quite easy to double the throughput of this protocol... how?

two main disadvantages of this protocol:

- tight coupling, because of interlocked write/read operations
- high overhead in terms of bus transfers, to check the mailbox status

## FIFO queue

the use of a FIFO queue compensates temporary imbalances between the read and write throughputs

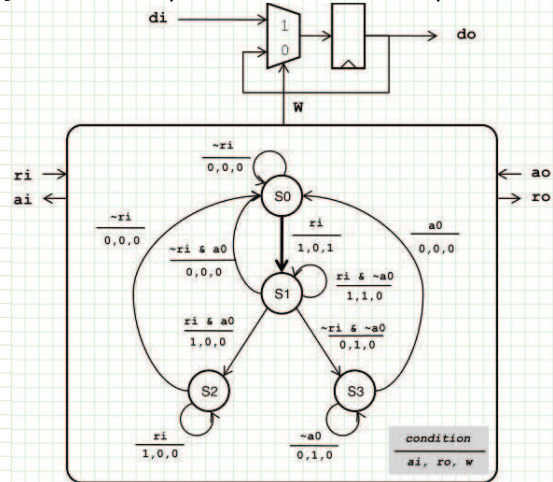


```

dp fifo(in di : ns(8);
        in wreq : ns(1);
        out wack : ns(1);
        out do : ns(8);
        in rreq : ns(1);
        out rack : ns(1)) {
    sig read, write : ns(1);
    reg rp, wp : ns(3);
    use dualport_mem(di, wp, write, // write port
                    do, rp, read); // read port
    always {
        read = (rreq & (wp != rp)) ? 1 : 0;
        write = (wreq & ((wp + 1) != rp)) ? 1 : 0;
        wp = write ? wp + 1 : wp;
        rp = read ? rp + 1 : rp;
        wack = write;
        rack = read;
    }
}
    
```

Schaumont, Figure 11.4 - A FIFO with handshakes on the read and write ports

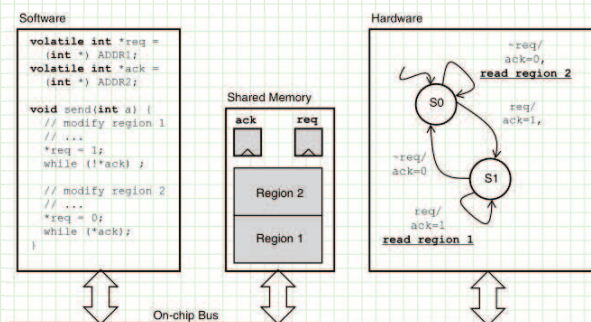
a FIFO may be built by chaining multiple FIFO sections, each acting as a slave on input and as a master on output



Schaumont, Figure 11.5 - A one-place FIFO with a slave input handshake and a master output handshake

## shared memory

instead of controlling access to a single register, a single handshake can also be used to control access to a region of memory

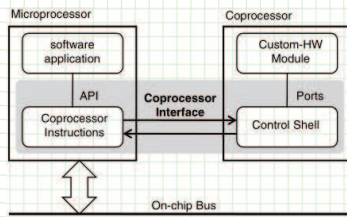


Schaumont, Figure 11.6 - A double-buffered shared memory with a memory-mapped request/acknowledge handshake

in one phase of the protocol in figure, changes are allowed to region 1 of the memory, while in the other phase of the protocol, changes are allowed in region 2 of the memory

this scheme implements the pipelining of read/write operations, which, for applications that require data-reorganization in between processing stages, may lead to substantial performance improvements





Schaumont, Figure 11.7 - Coprocessor interface

#### coprocessor interfaces

when high data-throughput between the software and the custom hardware is needed, a dedicated processor interface outperforms memory-mapped interfaces

a coprocessor interface does not make use of the on-chip bus, it uses a dedicated port on the processor, driven by coprocessor instructions

both the coprocessor instruction set and the specific coprocessor interface depend on the type of processor—not all processors have a coprocessor interface

the decision of using a specific coprocessor interface locks the custom hardware module into a particular processor, thus it limits the reusability of that custom hardware module to systems that also use the same processor

main advantages of a coprocessor interface over an on-chip bus:

- higher throughput: because not constrained by the wordlength of the bus on chip, nor by the load/store transfer mechanism
- fixed latency: a coprocessor bus is a dedicated, point-to-point connection with stable, predictable timing

#### custom instruction interfaces

the integration of hardware and software can be considerably accelerated as follows:

1. reserve a portion of the opcodes from a microprocessor for new instructions
2. integrate the custom-hardware modules directly into the micro-architecture of the micro-processor
3. control the custom-hardware modules using new instructions derived from the reserved opcodes

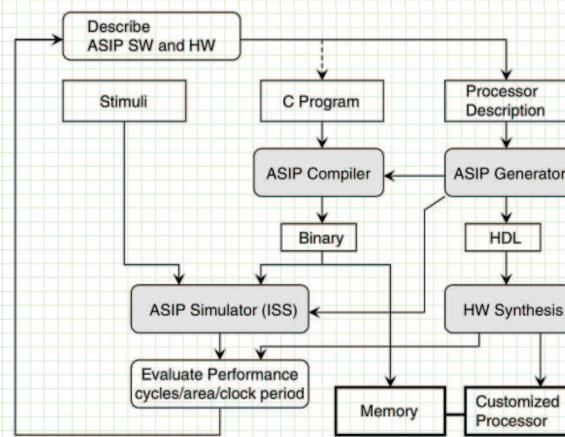
the resulting design is called an *Application-Specific Instruction-set Processor (ASIP)*

while a coprocessor instruction set is part of a microprocessor, an ASIP instruction set is defined by the application

ASIP design automates some of the more difficult aspects of HW/SW codesign:

- the instruction-fetch and dispatch mechanism in the micro-processor ensures that custom-hardware and software remain synchronized
- design of an ASIP proceeds in an incremental fashion, that helps the designer to avoid drastic changes to the system architecture while exploring different options

### ASIP design flow



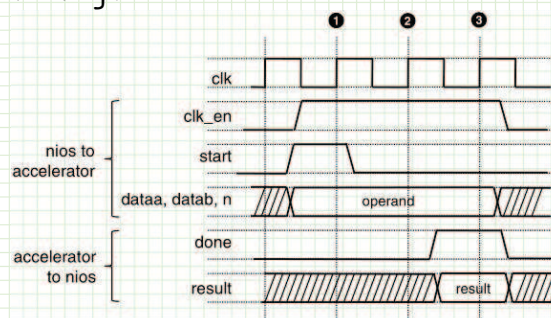
Schaumont, Figure 11.12 - ASIP design flow

sequential ASIP design does not generally deliver better performance than SoC design based on custom hardware modules, yet it does deliver less error-prone results

significant progress has been made on design tools that support the ASIP design flow—all of the shaded boxes in the figure can be obtained as commercial tools

### example: the Nios-II custom-instruction interface

the Nios-II softcore processor has a coprocessor interface whereby custom instructions may be defined and hardware modules may be attached to



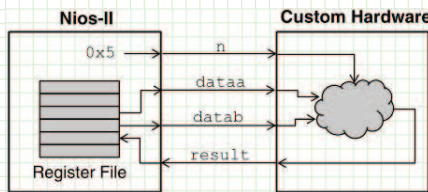
Schaumont, Figure 11.15 - Nios-II custom-instruction interface timing

the interface supports variable-length execution of custom instructions through a two-way handshake the **clk\_en** input is used to mask off the clock to the custom hardware when the instruction is inactive in software, the custom instruction is executed through the instruction custom, e.g.:

custom 0x5, r2, r3, r5

assigns the value 0x5 to n and associates the dataa, datab, result ports with registers r2, r3, r5, respectively—use of n: to multiplex different custom instructions in the hardware module

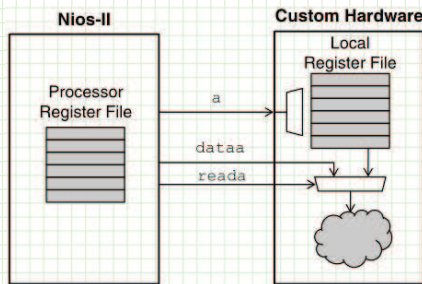
the use of a local register file in the custom hardware module is also supported



Schaumont, Figure 11.16a - Nios-II custom-instruction integration with processor register file

a custom instruction may take operands from either register file: registers prefixed with *r* are located in the processor, while registers prefixed with *c* are located in the custom hardware

instructions that use both are allowed, such as custom 0x5, c2, c3, r5



Schaumont, Figure 11.16b - Nios-II custom-instruction integration with local register file

figure 11.16b shows the case for the first input operand only: the control signal *reada* selects either the processor's or the local register file

- in the former case, the operand is provided through the *dataa* port, that is associated with a processor's register
- in the latter case, the input *a* selects the local register to use as operand

## references

recommended readings:

Schaumont (2012) Ch. 11, Sect. 11.1.1-11.1.5, 11.2.0, 11.3.0-11.3.1, 11.3.3

for further consultation:

Schaumont (2012) Ch. 11, Sect. 11.1.6, 11.2.1-11.2.2, 11.3.2, 11.3.4