DMA

Direct Memory Access
Accès Direct en Mémoire

rene.beuchat@epfl.ch
rene.beuchat@hesge.ch

Contents

- Problem
- Minimum Computer System
- Polling / Interruption
- DMA
- Transfer's Types
- Dual-port Memory
- DMA Controller

Problem

- In a computer system, the peripheral access (through programmable interface) can be realize by processor transfer's instructions.
- Status registers are used in the programmable interface to specify to the processor if data transfer can be done.

Problem (2)

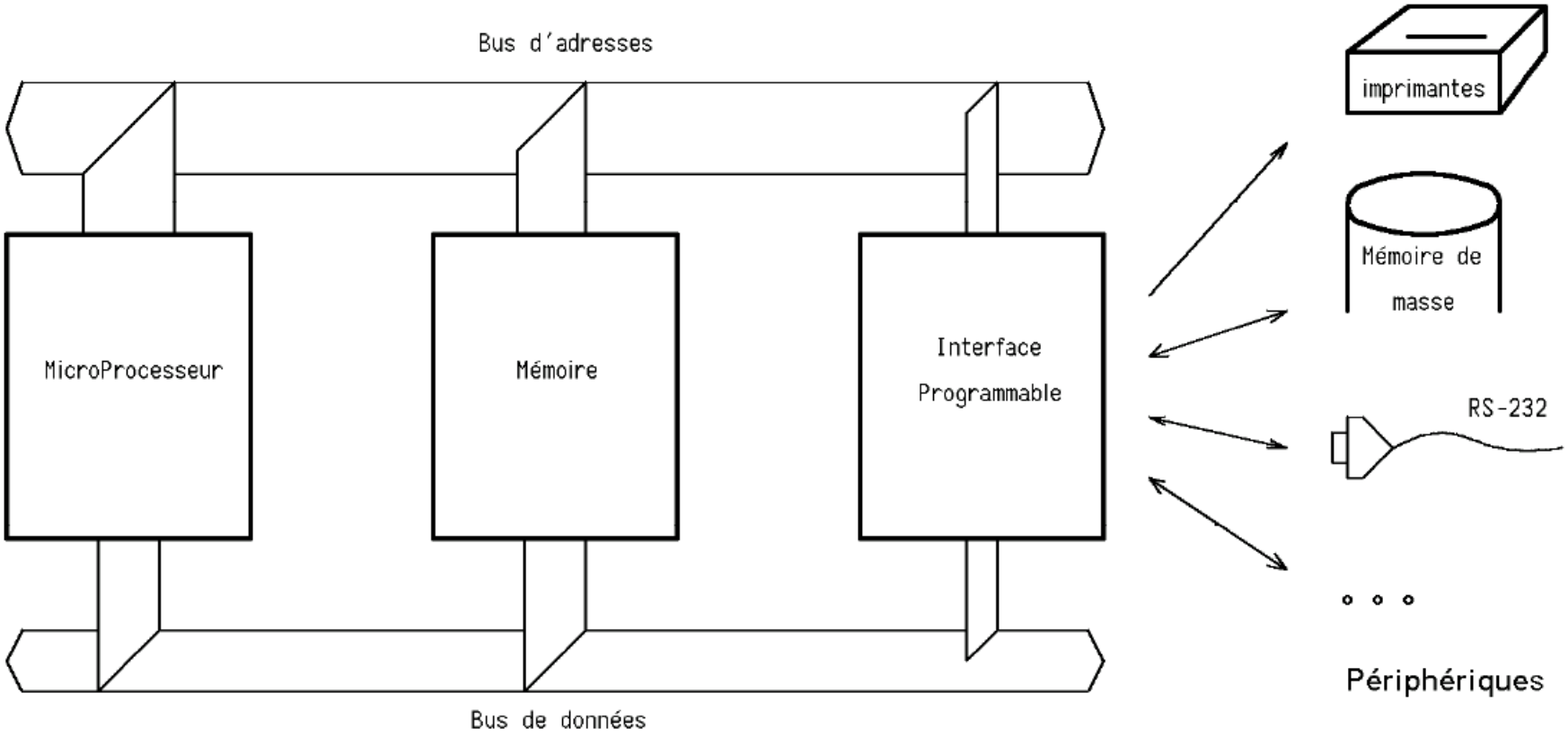
- By **polling** of status register, the program can know when the interface is ready for the transfer:

  - Wait (status_transfer == OK)
    - Do the transfer

Or

  - If (status_transfer == OK)
    - Do the transfer
  - EndIf

Polling


Bus d'adresses
MicroProcesseur
Mémoire
Interface
Programmable
imprimantes
Mémoire de
masse
RS-232
Périphériques
Bus de données

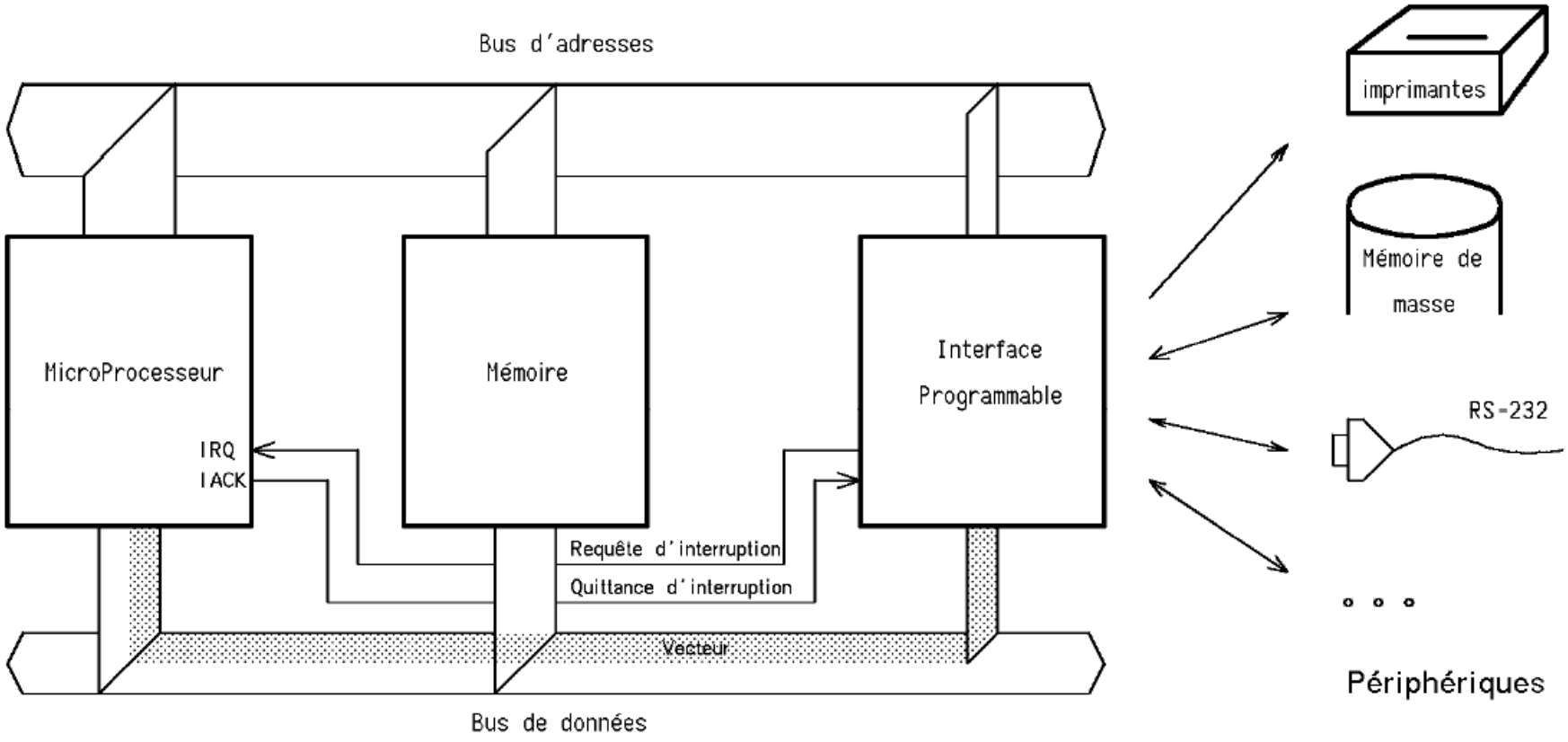
Polling

- With polling, the system is **synchronous**, the program control exactly when it can access a resource (Programmable Interface)
- **Disadvantage,** the program must often test the status register for nothing, and often enough not to miss Data

Interruption

- If we want that the processor does not lose time to poll unnecessarily interfaces, hardware interrupts can use the processor just when service is to be performed to serve the interface.
- The synchronization with the information consumer / producer is to be processed by software (message, semaphore, FIFO, etc.…)

Interruption



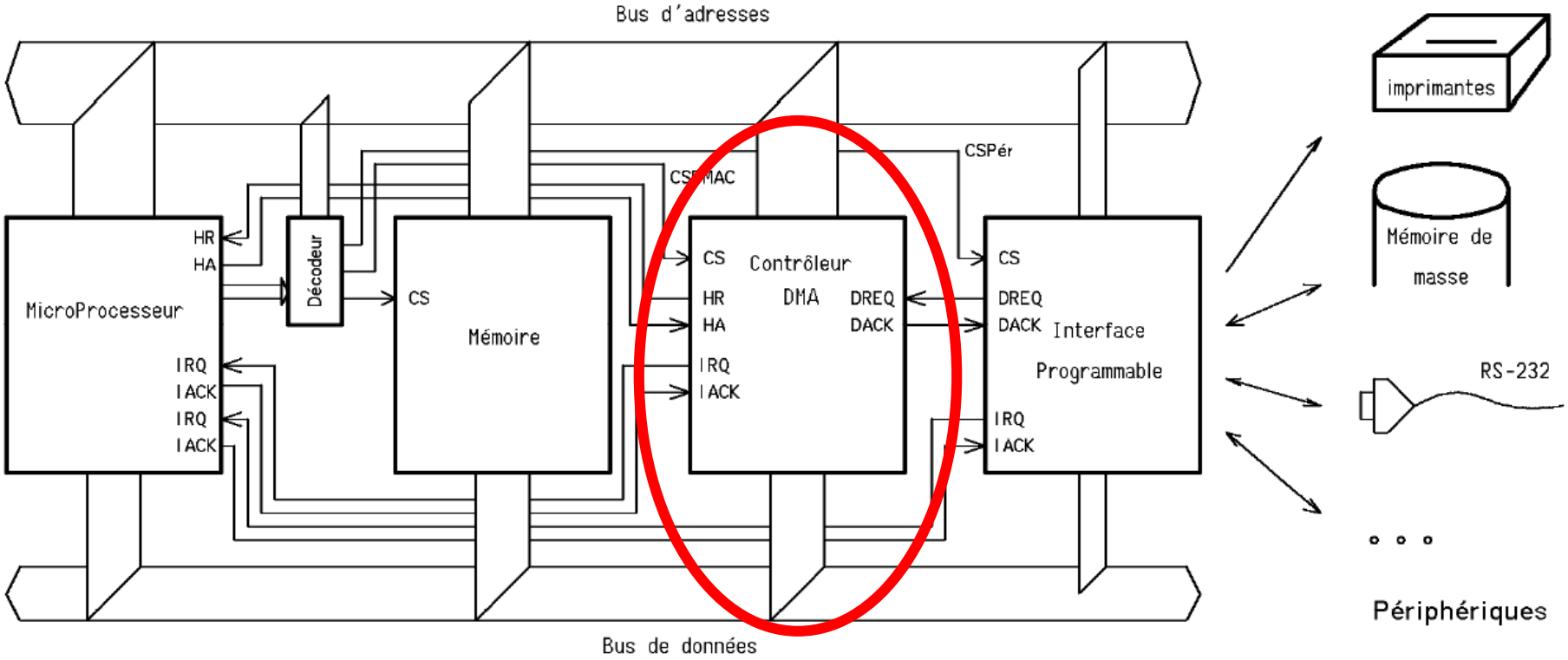
Interruption

- Interruptions need **specialized hardware** in the processor and in the programmable interface. That hardware depend of the processor used (interrupt vector, way to access the interrupt handler function, etc…)
- Some signals are necessary as ***Interrupt Request*** (at least) and sometime ***Interrupt Acknowledge***
- Some instructions needs to be executed to serve the interrupt handler (context saving and switching, request testing, programmable interface servicing and acknowledge)
  → **Limited transfer bandwidth**

DMA

- For systems where the transfer rate between the I / O and memory is high, the polling or interruptions are unusable A more efficient system is needed → **DMA**
- The transfer is carried out by a specialized unit: the **DMA controller**

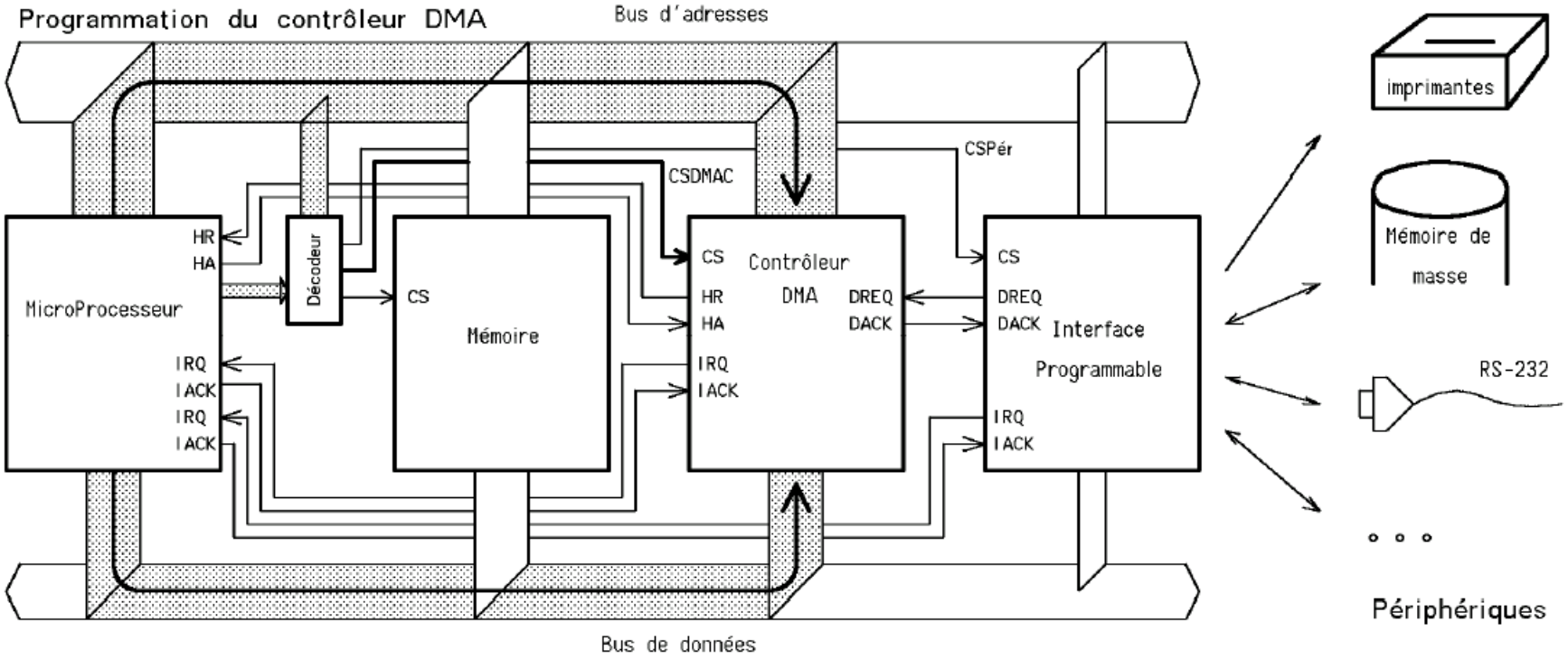
DMA


Bus d'adresses
CSMAC
CSPér
HR
HA
MicroProcesseur
IRQ
IACK
IRQ
IACK
Décodeur
CS
Mémoire
CS
Contrôleur
HR
DMA
DREQ
HA
DACK
IRQ
IACK
CS
DREQ
DACK
Interface
Programmable
IRQ
IACK
imprimantes
Mémoire de
masse
RS-232
Périphériques
Bus de données

DMA

- The DMA controller performs transfers **in place of the processor**
- It must have control of the bus:
  - Address
  - Data
  - Control transfers
- Before performing a transfer, **an arbitration** must occur :
  - **Only one master can access a slave unit at a given time**

DMA Controller:
a programmable interface

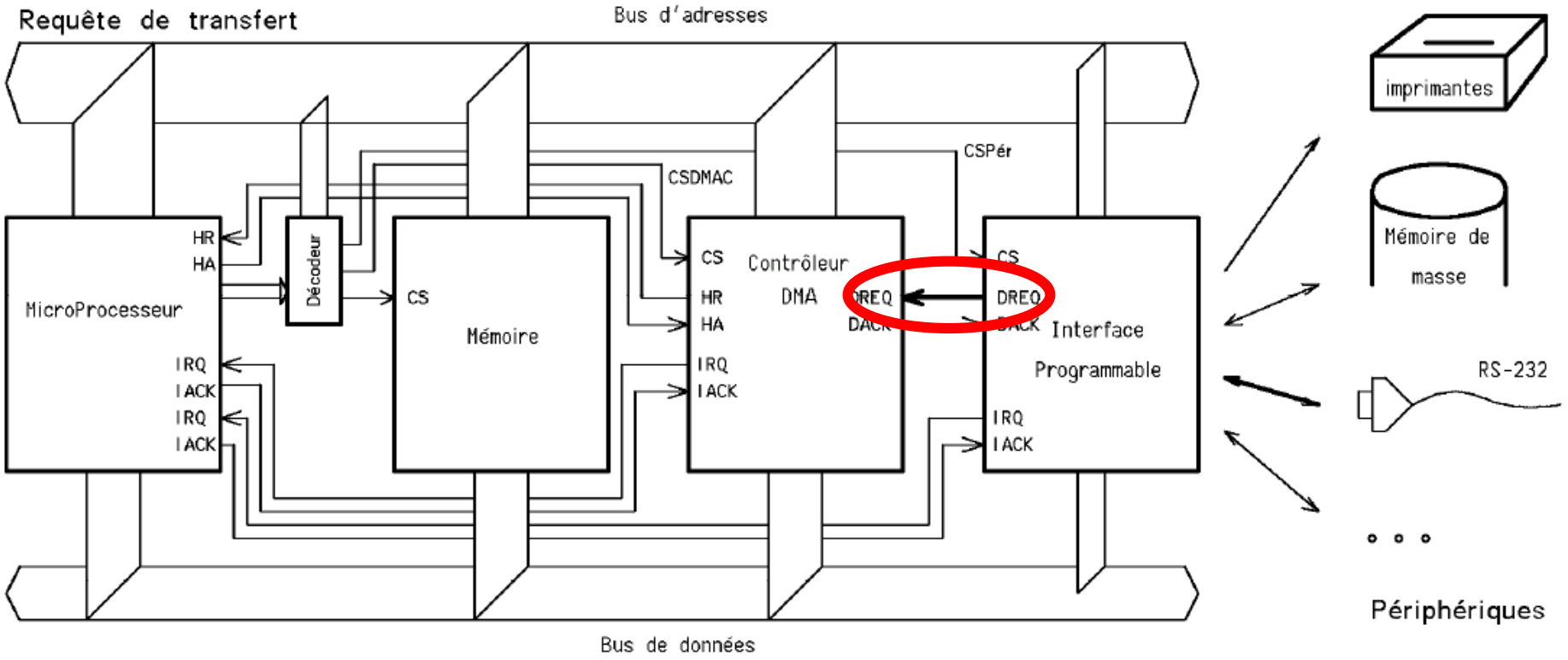


DMA Controller:
a programmable interface (2)

- The DMA controller is a programmable interface that must be programmed by the processor before it is operational
- Example of transfer with dual cycle
  - **I/O → memory :**
    1. I/O → DMA
    2. DMA → memory

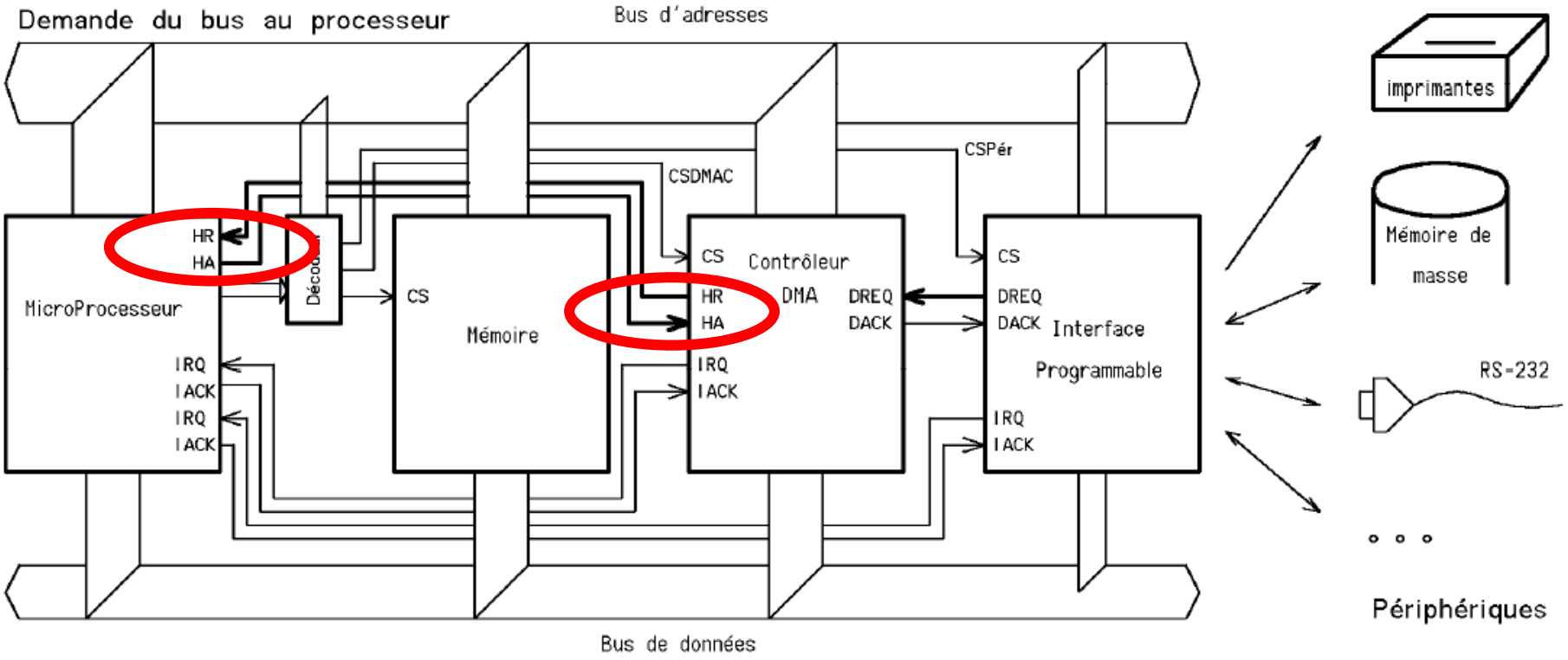
DMA : I/O → memory

Transfer Request



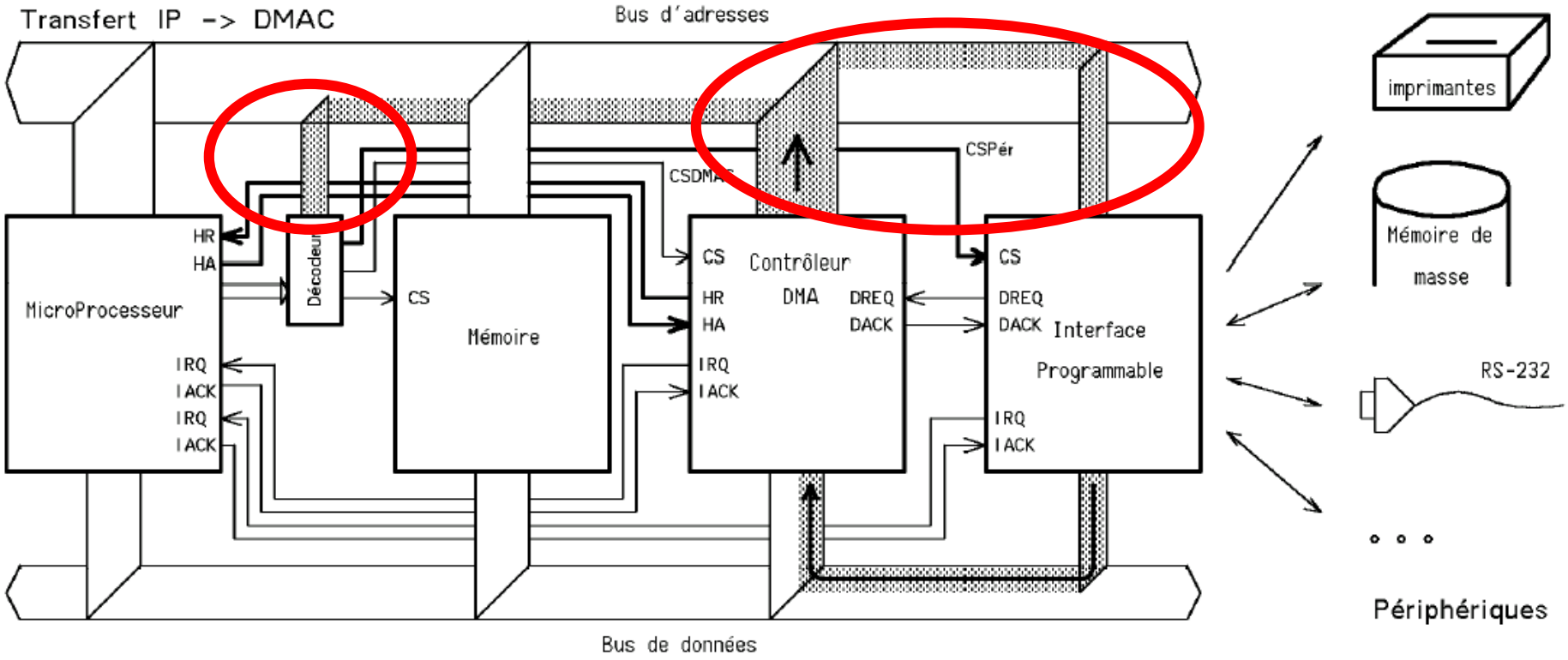
DMA : I/O → memory

Request of the Bus to the processor



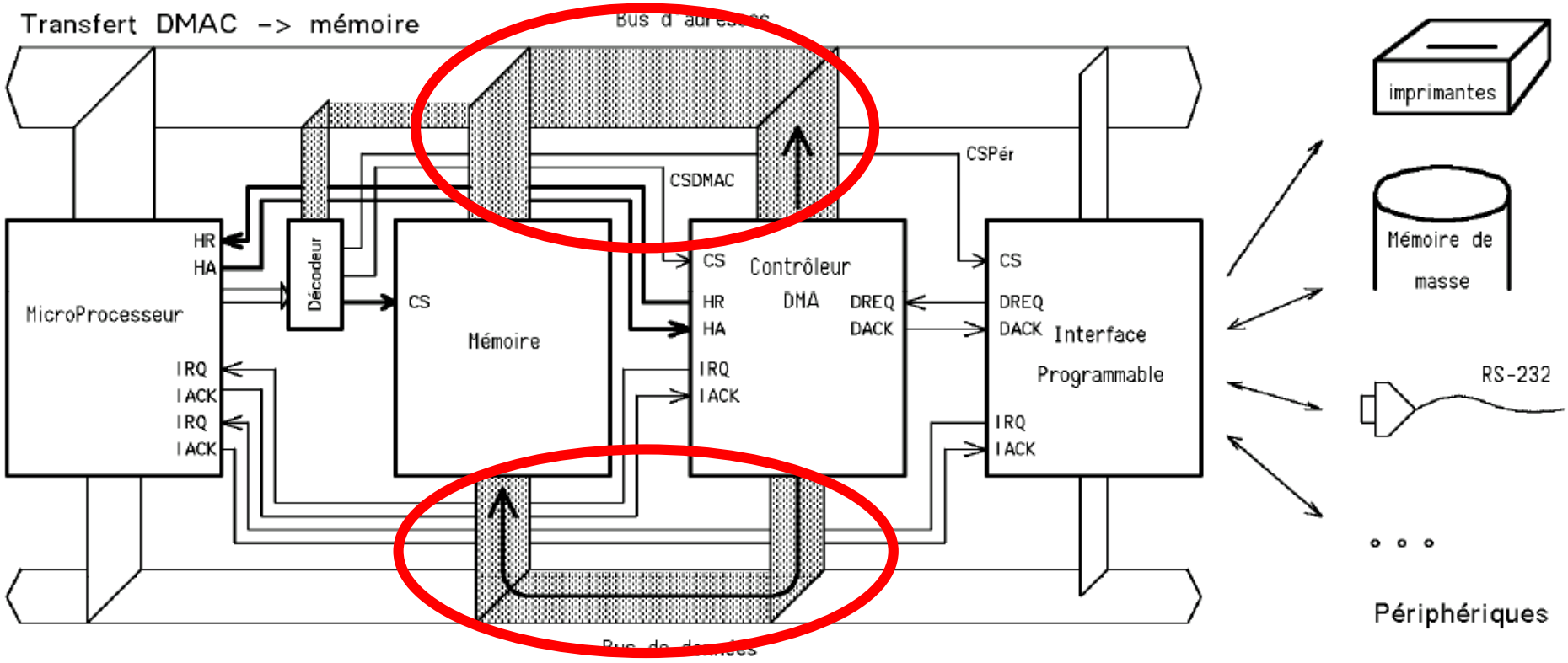
DMA : I/O → memory

Transfer Interface Prog. → Ctrl DMA



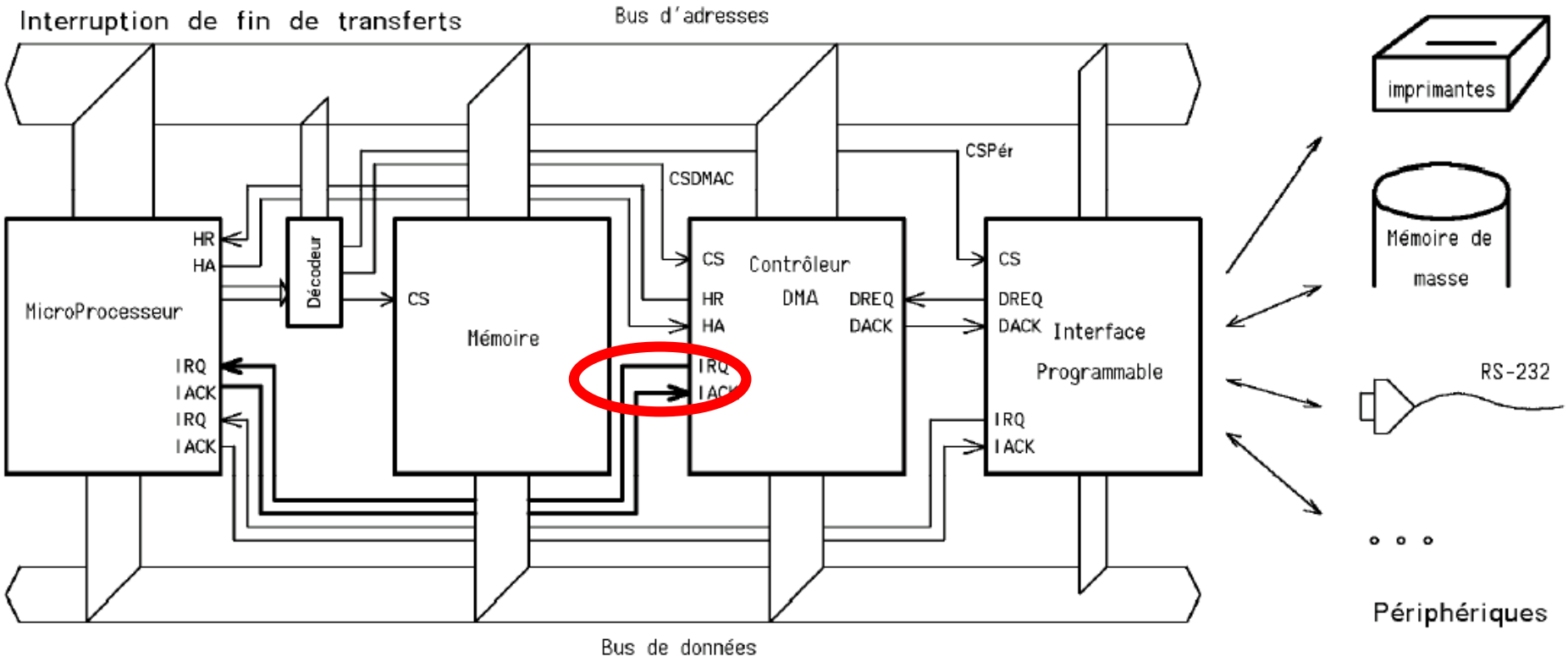
DMA : I/O → memory

Transfer Ctrl DMA → Memory



DMA : I/O → memory

End of transfers interrupt



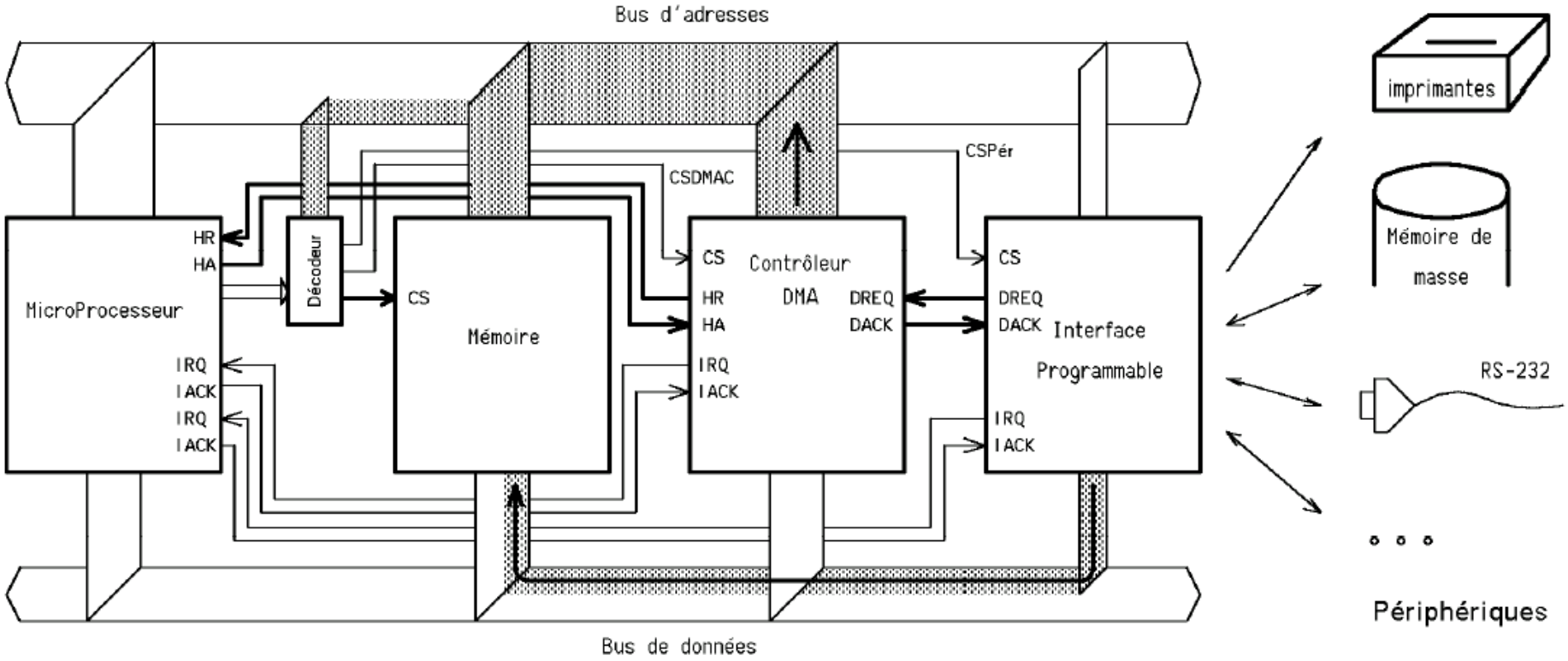
DMA fin de transfert(s)

- When a data packet has been transferred, the processor is notified by interruption or it can use the **polling** of a **status register**
- For the DMA to be useful, we need a **certain amount** of data to transfer, not only one byte, as we need to initialize the DMA controller before using it.

Simple cycle Transfer

- For greater efficiency transfer, access by the interim controller is not always necessary.
- A direct access from **I/O → memory** is possible, it's the simple cycle transfer

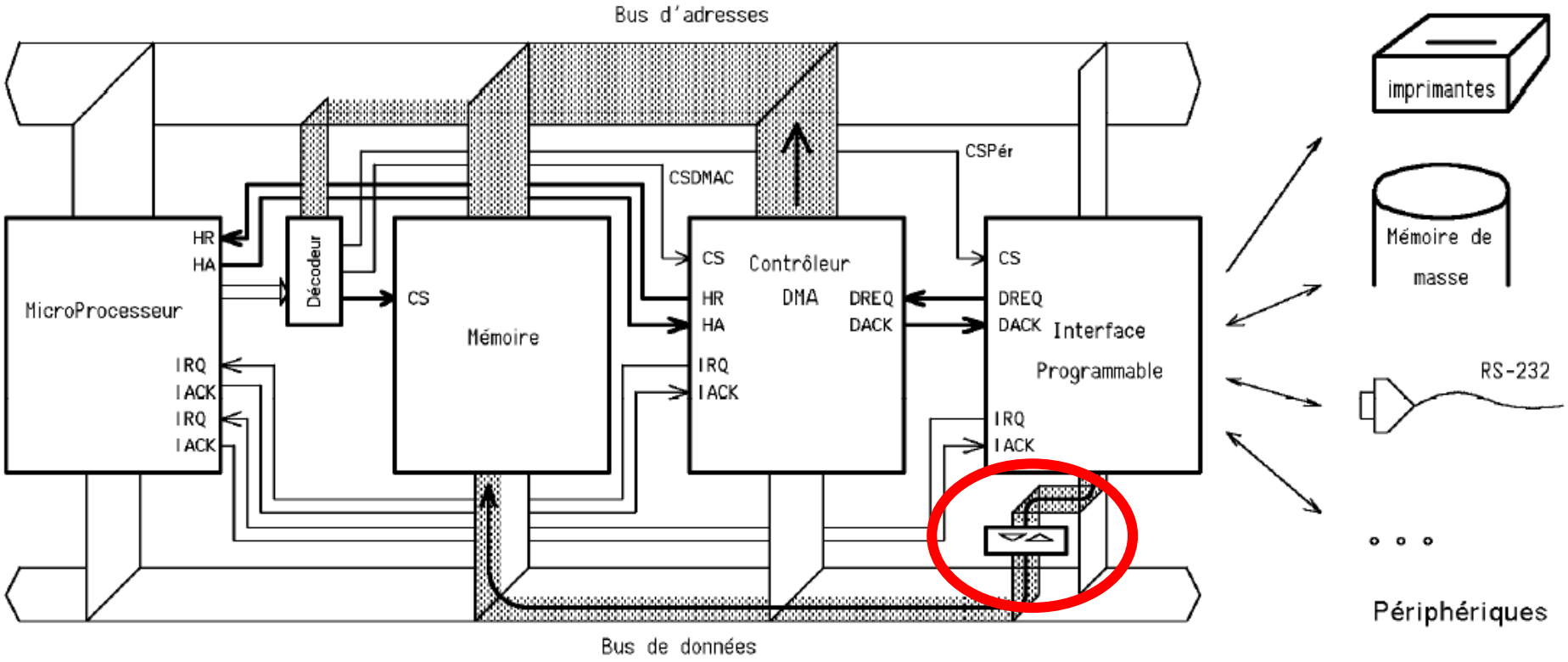
DMA, simple cycle



Simple cycle

- A problem could be arises if the data bus width of memory and Programmable Interface are not the same
- The data stored in memory must be accessed by the processor addresses contiguously. If the programmable interface source width bus is lower than the width of the memory, alignment drivers on the data bus should be added (or bidirectional multiplexers).

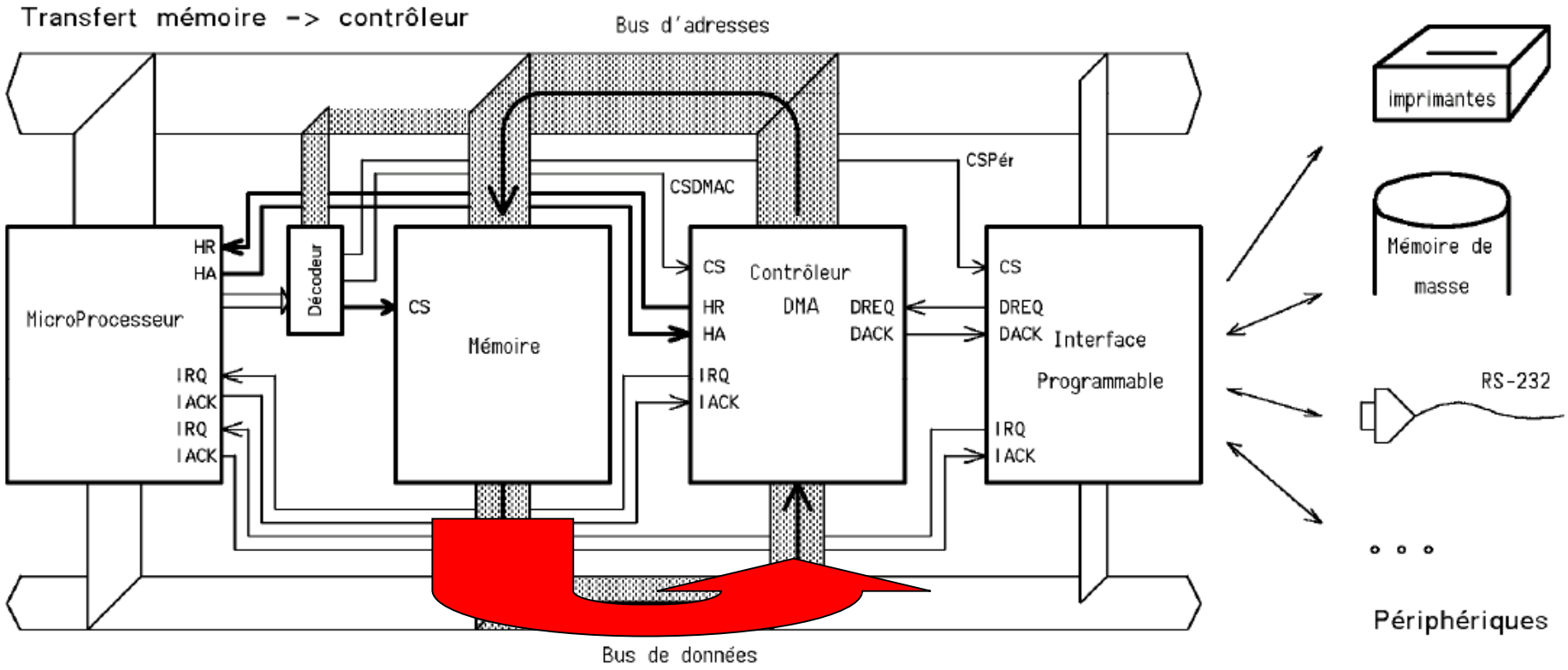
Simple cycle



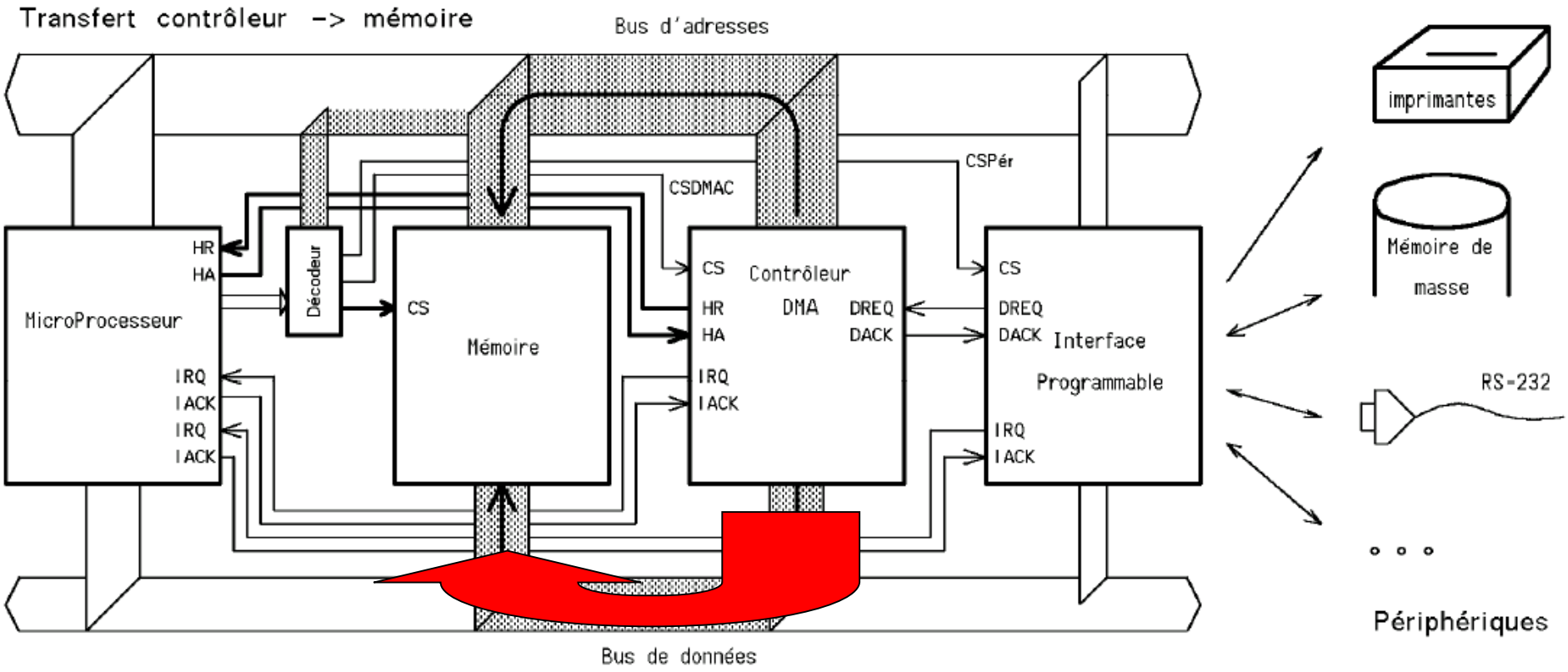
Transfers memory to memory (1)

- The DMA unit can be used to transfer data from memory to memory more efficiently than the processor.
- In this case a passage by an internal DMA controller register usually occurs.

Transfers memory to memory (2)



Transfers memory to memory (3)



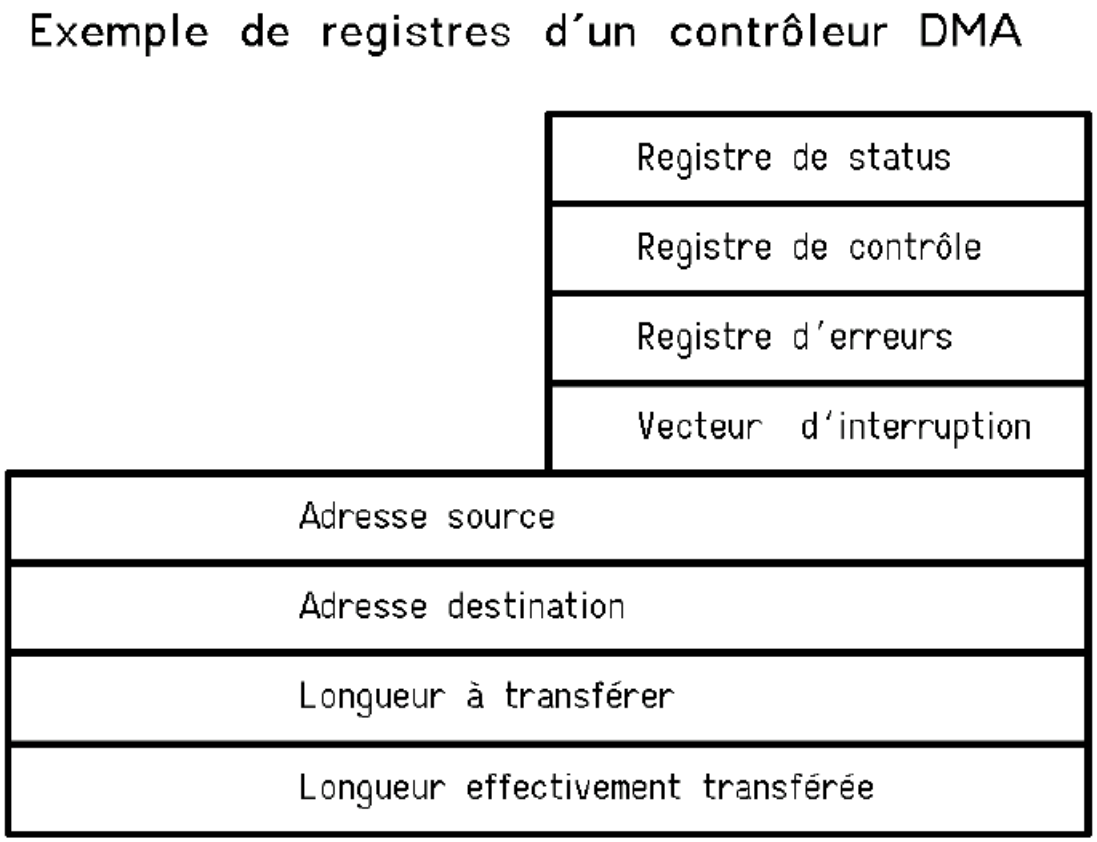
Programmation

- The DMA controller is a programmable interface. It must therefore be initialized prior to use.
- Several methods are possible depending on the circuit used:
  - By direct access to internal DMA registers by the processor
  - By descriptors automatically loaded from memory to the DMA controller by itself

Programmation (2)

- A minimum set of descriptors are available on virtually all controllers DMA:
  - Source Address
  - Destination Address
  - Length of data to transfer / transferred
  - Modes of operations
  - Status of the controller
  - Interrupt control

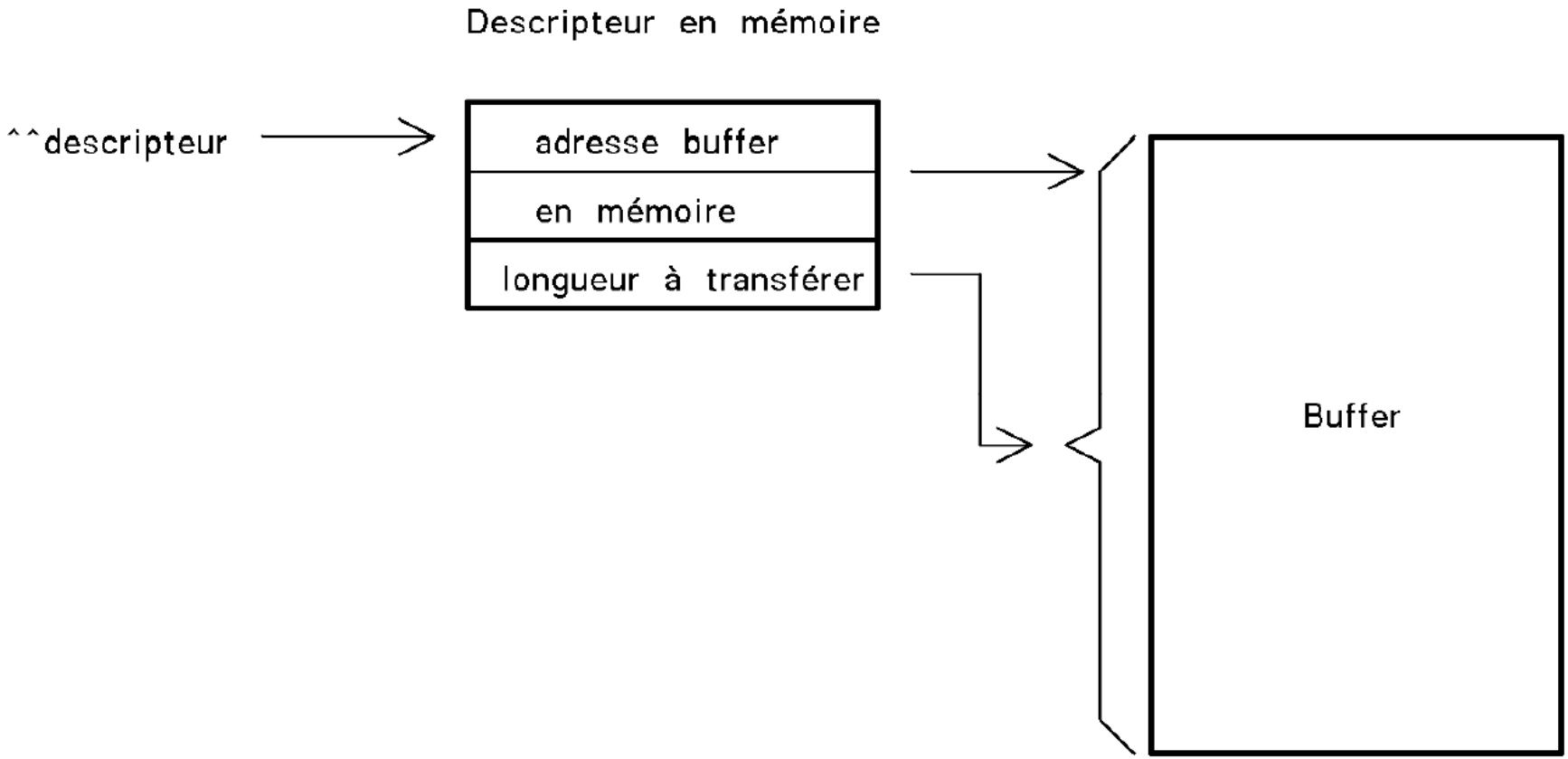
Base Registers



Base Registers

- In general a pointer in the controller point to a descriptor in memory.
- This descriptor contains various parameters describing the transfer and addresses of buffers source / destination
  - If the pointer specifies a **memory** address, the address will be incremented for each access
  - If the pointer specifies a **programmable interface** (I / O), the address will not be changed

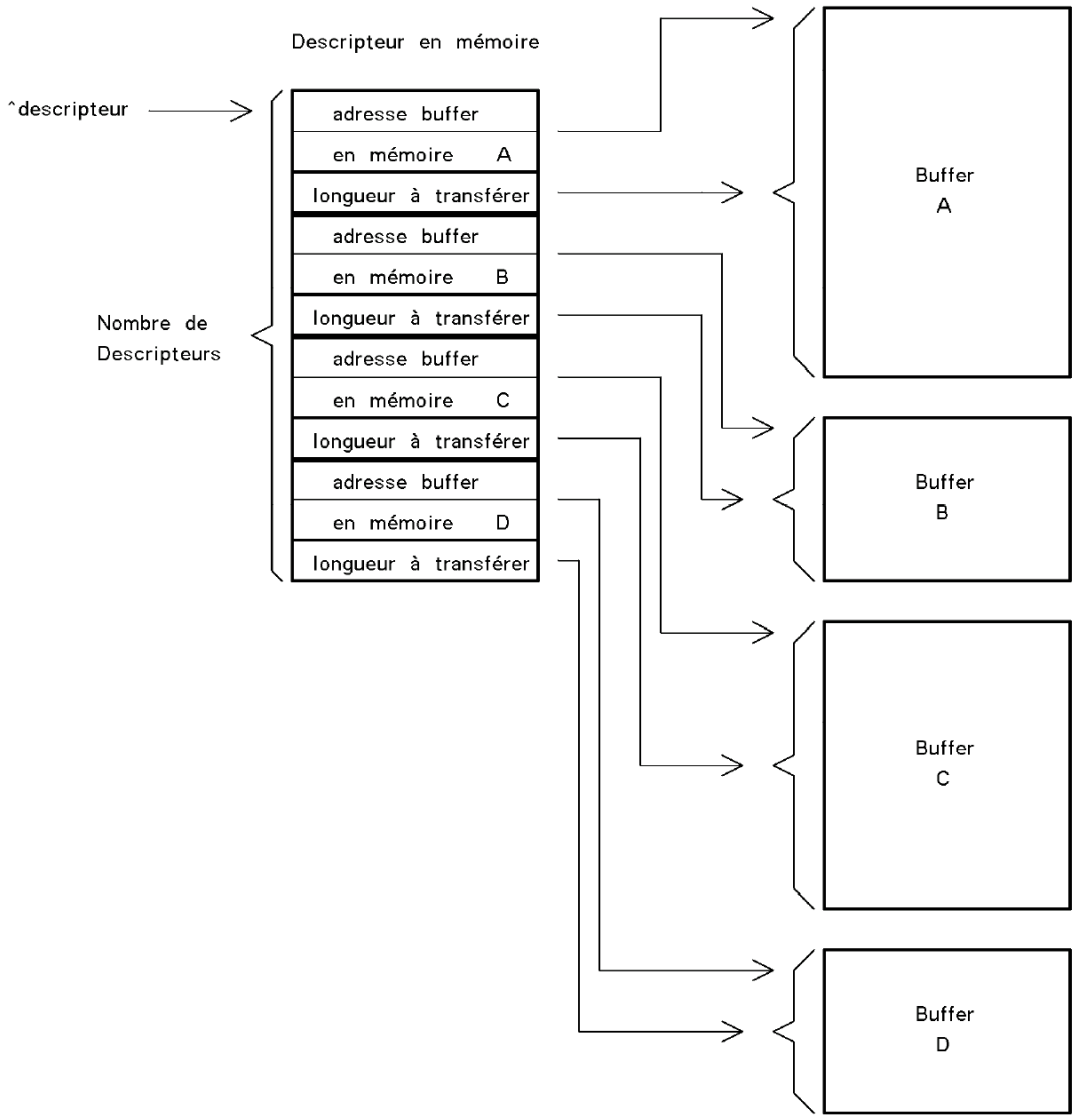
Base Registers



Base Registers

- A single buffer descriptor is limited if new data are received and that the buffer precedent is not yet released, the data will be lost.
- A table of descriptors is generally proposed

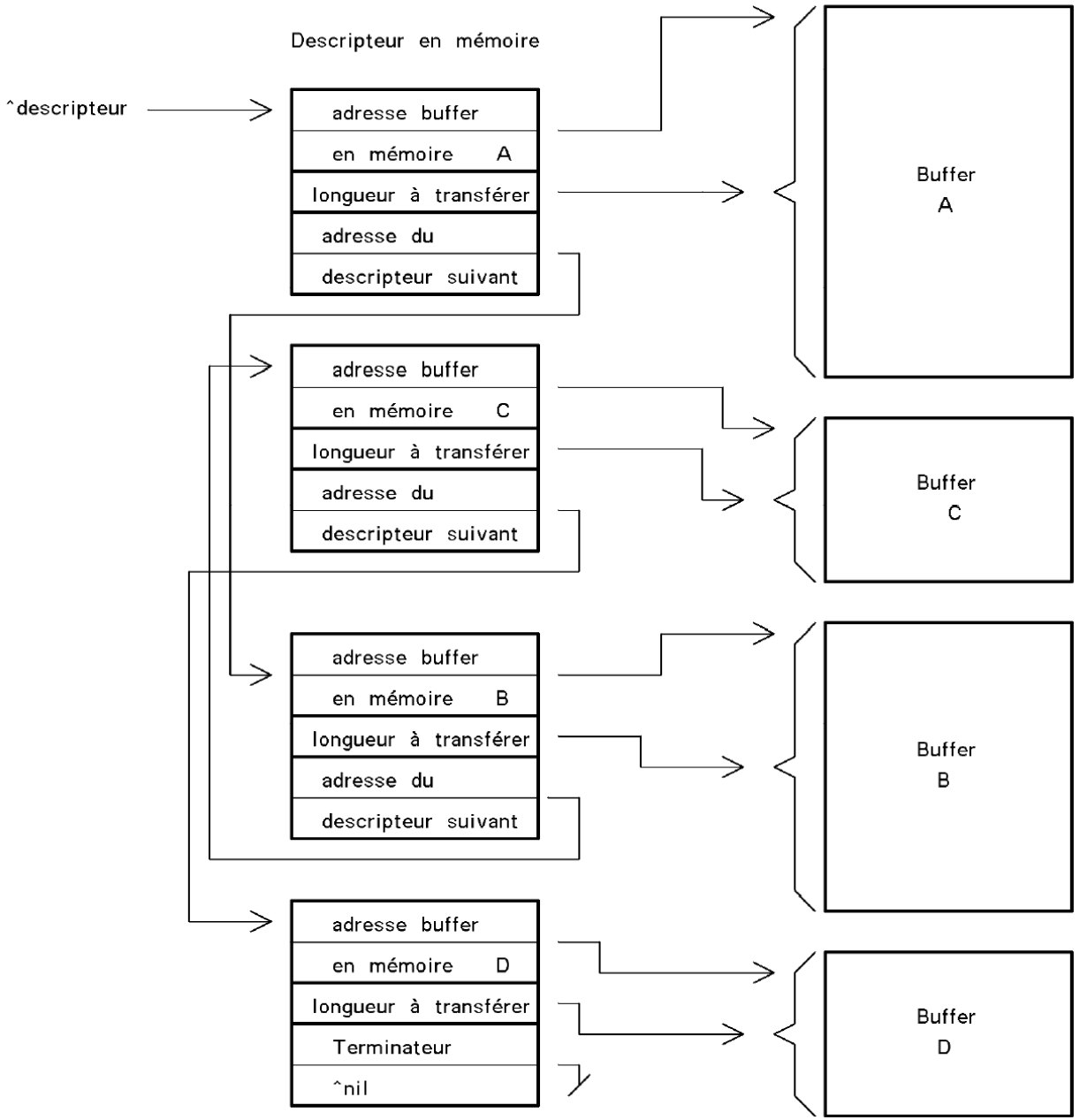
Descriptors in memory



Descriptors in memory

- With an array of descriptors, the number of buffer is static. A buffer management is needed to maintain order in the buffer descriptors.
- A linked list of descriptors allows greater flexibility in the management of descriptors, and buffer number can easily be dynamic

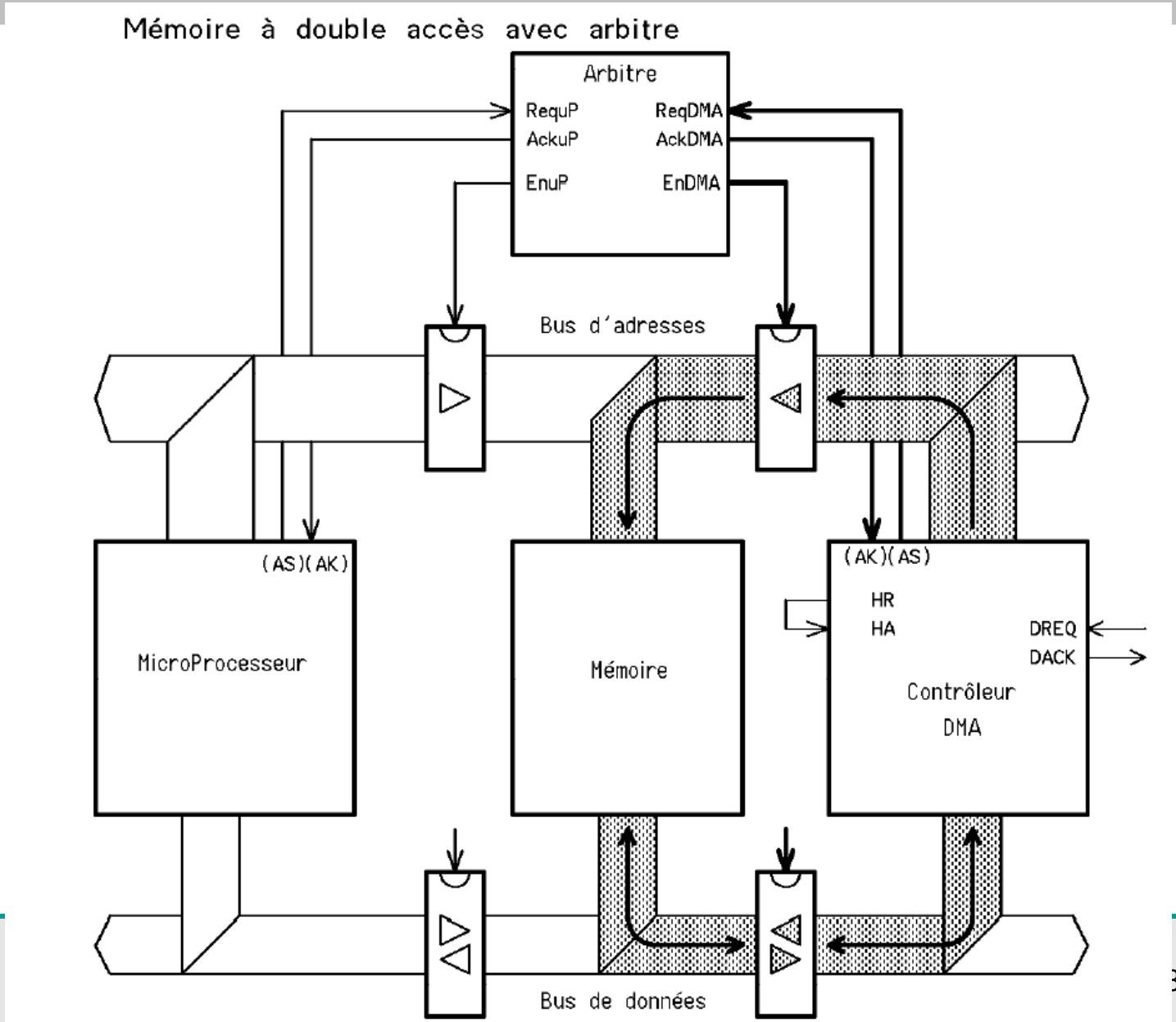
Descriptors in memory



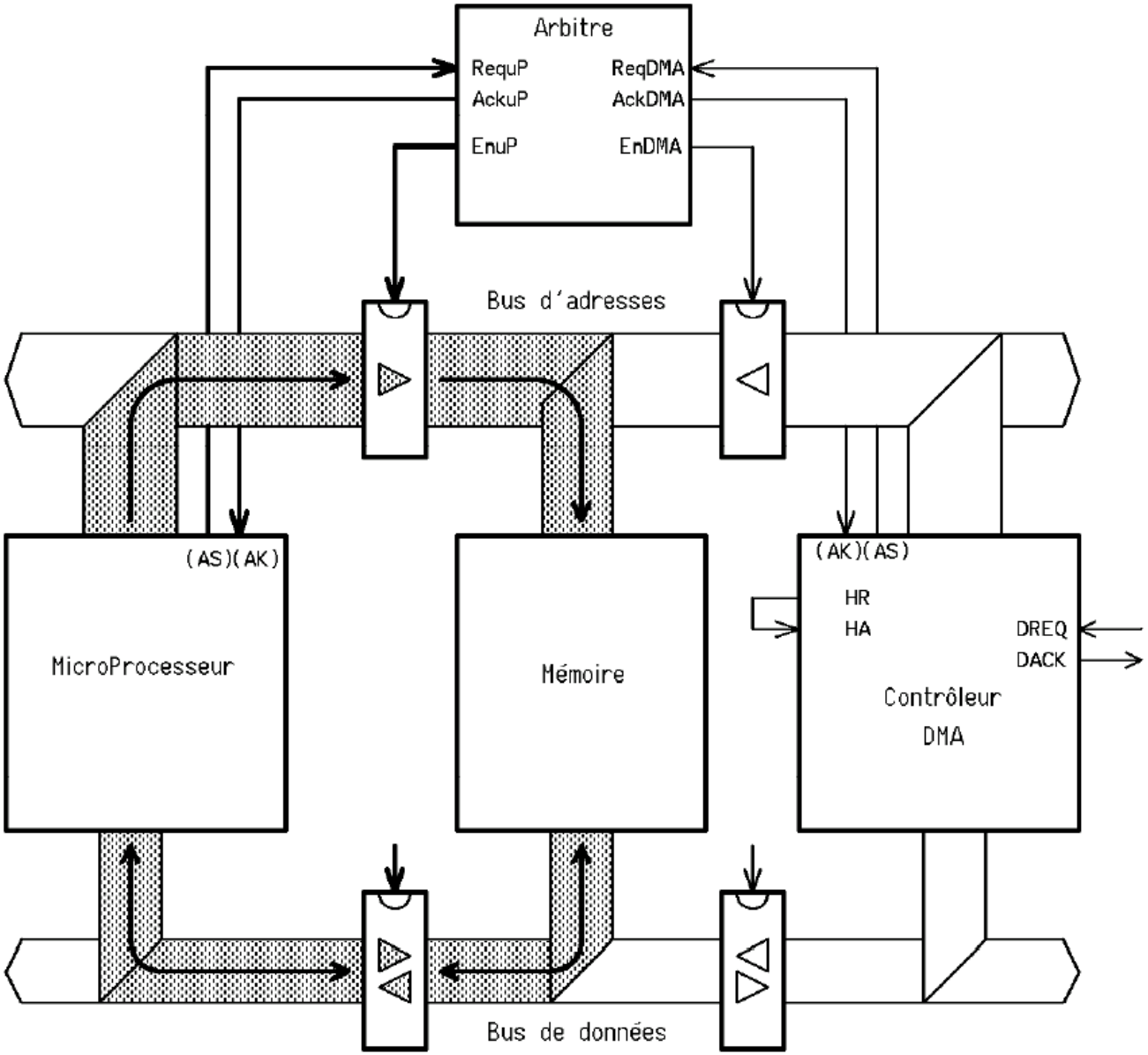
Double access Memories

- To make DMA transfers in parallel with the use of a processor, separate buses are to be used
- The memory is seen as a memory with double (triple, or more) accesses
- The multiplexed bus can be realized with conventional buffers and / or external arbitrator dedicated circuits

Double access Memories



Double access Memories



Double access Memories

- The dual access memory model is similar to a multiprocessor system with a common shared memory.
- This model corresponds to a multi-masters backplanes bus

Conclusion

- The DMA units allow the processor to offload tasks of packet data transfer
- The transfer is "wired" rather than instructions executed by a processor → transfers speeds increased
- Assembling of Data is possible
- Data filters is possible
- Operations on Data is possible

Conclusion

- The end of a transfer can be handled by polling or interruption by the processor.
- Currently many programmable interfaces directly integrates DMA controller (networks, SCSI disk,…)

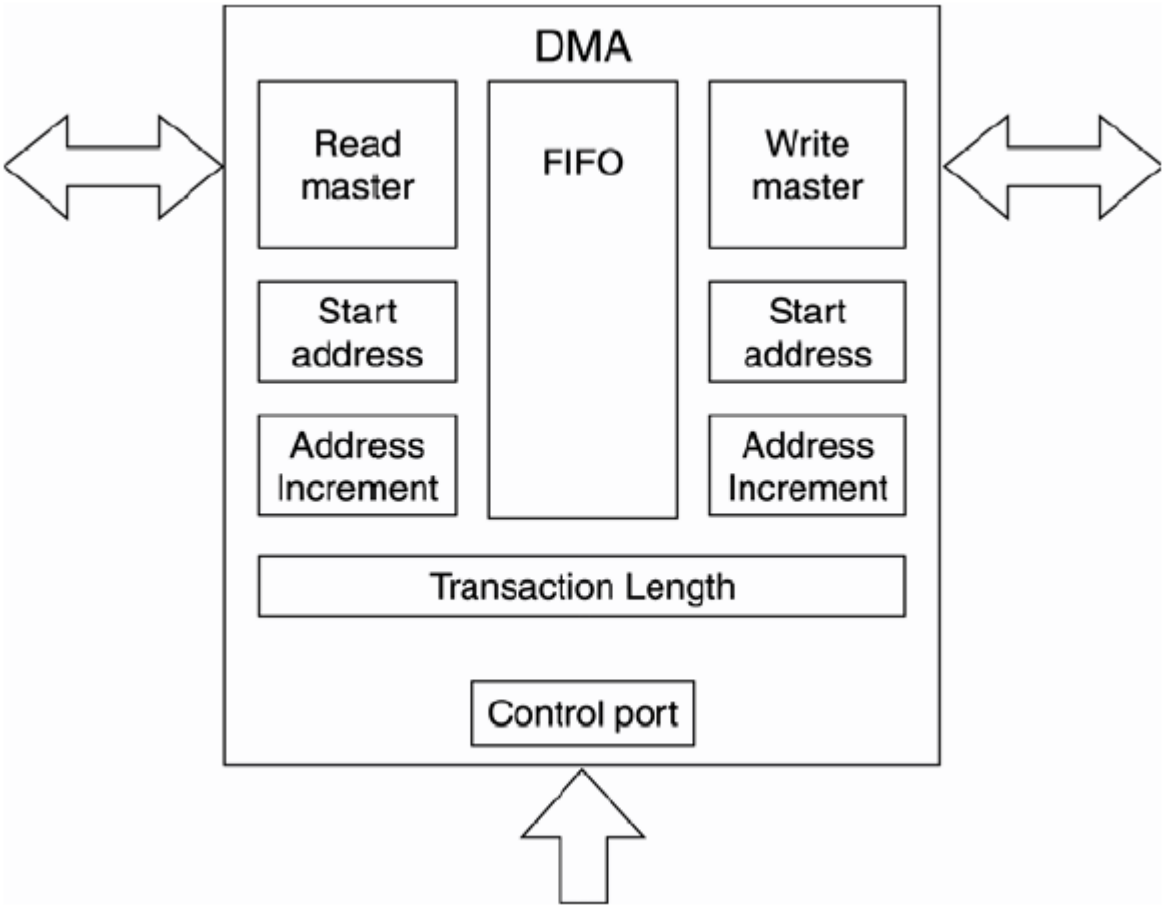
DMA on FPGA

- Core DMA in HDL (VHDL / Verilog) are feasible or available for FPGA
- Synthesizable IP to integrate on programmable logic

  Example: DMA for Avalon bus

DMA for Avalon

- DMA unit architecture
- Read bus
- Internal FIFO
- Write bus
- Programmable control unit



DMA registers

- Seen by the processor (NIOS) as
  8 * 32 bits registers

A2..A0	Register Name	R/W	Description/Register Bits											
			31	. . .	9	8	7	6	5	4	3	2	1	0
0	status(1)	RW								len	weop	reop	busy	done
1	readaddress	RW	Read master start address											
2	writeaddress	RW	Write master start address											
3	length	RW	Length in bytes											
4	reserved1	–	Reserved											
5	reserved2	–	Reserved											
6	control	RW			wcon	rcon	leen	ween	reen	i_en	go	word	hw	byte
7	reserved3	–	Reserved											

Status Register

- Information on controller status
- A write access clear
  len, weop, reop, and done bits

Bit Number	Bit Name	Description
0	done	A DMA transfer is completed.
1	busy	A DMA transfer is in progress.
2	reop	Read end of packet occurred.
3	weop	Write end of packet occurred.
4	len	A DMA transfer is completed and the requested number of bytes are transferred.

Status

- ***done*** is activated at the end of the transfer
- An interrupt can be generated if it's enabled
- Bits ***len***, ***weop***, and ***reop*** allows to know the cause of the transfer end.
- When ***done*** is deactivated by a write to this register, the interrupt request is deactivated too

Control Registers

- *Readaddress, writeaddress, length* specify the source, destination addresses and the length of the transfer
- *length* is defined in the number of bytes
- The widths of the registers are specified at the DMA unit creation

Control Register

- The control register specify modes and enabling functions

Bit Number	Bit Name	Description
0	byte	Byte (8-bit) transfer.
1	hw	Half-word (16-bit) transfer.
2	word	Word (32-bit) transfer.
3	go	Enable DMA.
4	i_en	Enable interrupt.
5	reen	Enable read end of packet.
6	ween	Enable write end of packet.
7	leen	End DMA transfer when `length` register reaches 0.
8	rcon	Read from a fixed address.
9	wcon	Write to a fixed address.

Control

- *rcon* et *wcon* specified if the read or write address is fixed (‘1’) or to increment (‘0’)
- depending the transfer width and *_con* specified, the addresses are incremented by 0, 1, 2 or 4

Bit Name	Transfer Width	Increment
byte	byte	1
hw	half-word	2
word	word	4

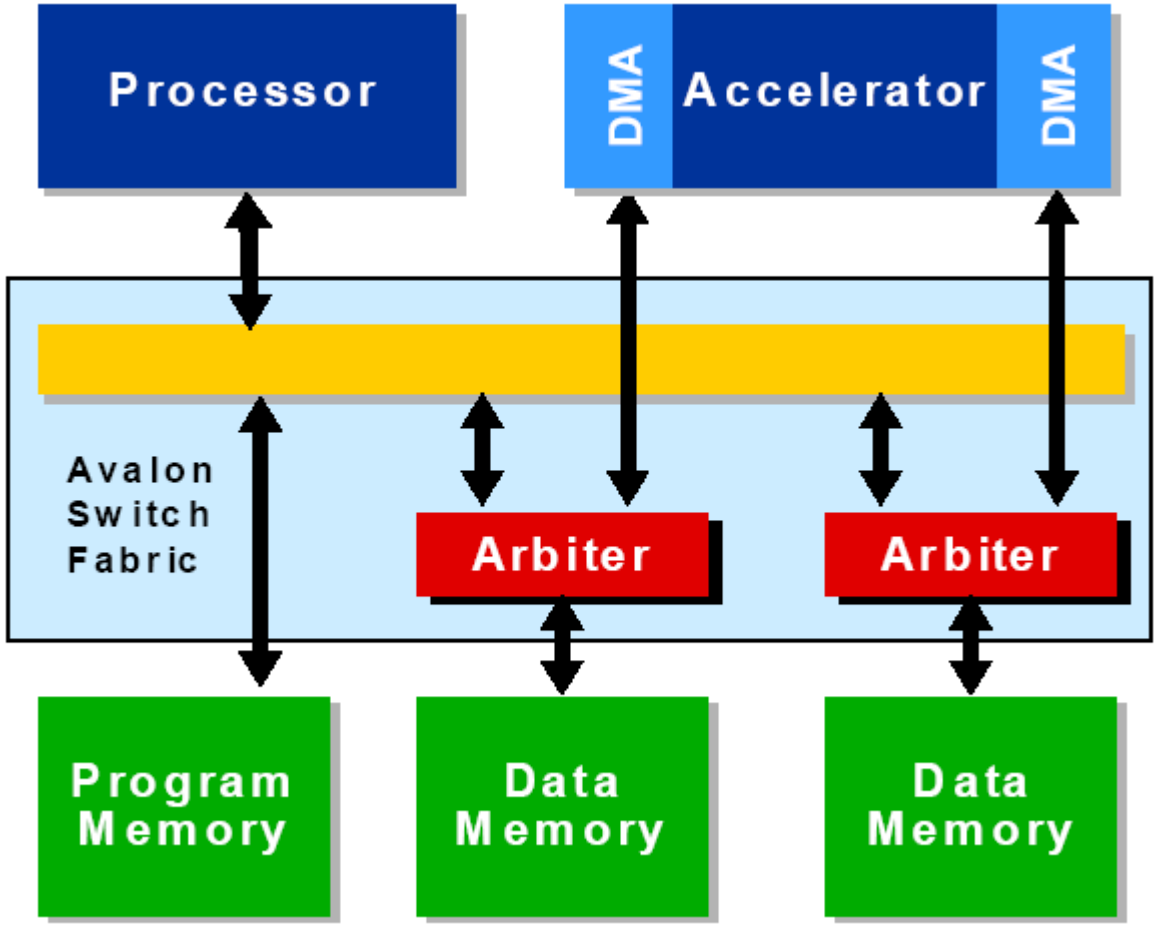
DMA programming

1) Clear mode
2) Set up everything except the go-bar

```
dma->np_dma_status = 0;
dma->np_dma_read_address = (int)source_address;
dma->np_dma_write_address = (int)destination_address;
dma->np_dma_length = transfer_count * bytes_per_transfer;
```

3) Construct the control word... to start
4) Wait until it's all done !! Polling or interrupt!!

NIOS II Processor, Hardware accelerator



- A hardware accelerator is a master unit with at least 2 DMA channels :
- One (or more) to read data
- One (or more) to write result

- To realize a DMA unit, a **master Avalon** unit has to be designed
- It has to provide the address of the data to access and to generate the data transfers
- The Avalon **WaitRequest** signal is mandatory to synchronize the end of the transfer cycle.

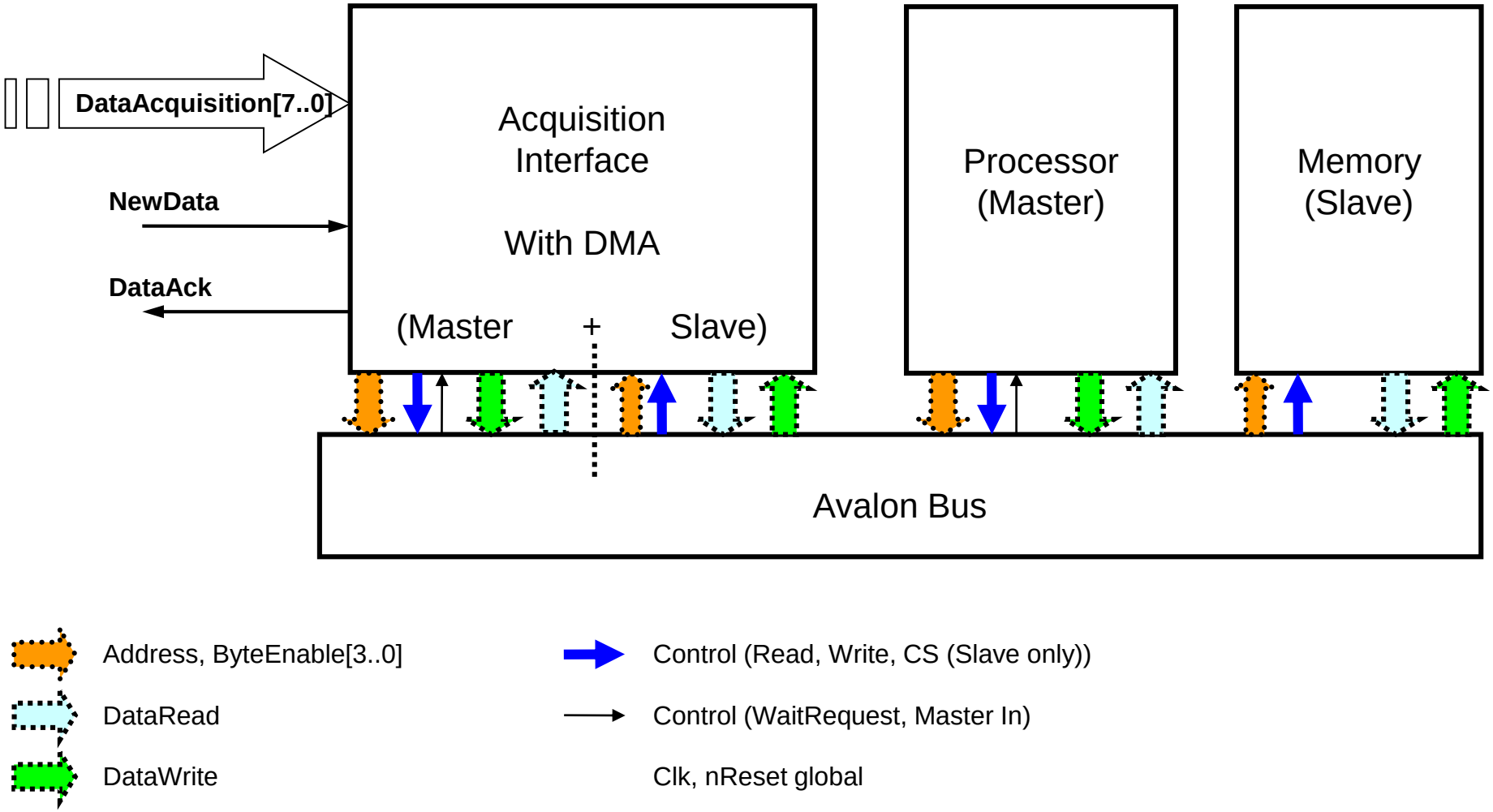
Example:
Master Interface on Avalon Bus in VHDL

- A data acquisition system has to be realized in a FPGA. A clock (**Clk**) synchronizes all the system. A specialized module receives data on an external 8 bits bus named **DataAcquisition[7..0].** A signal called **NewData** specifies when a new data is received on this bus.
- This signal stay activated until acknowledged by the **DataAck** signal generated by the module. This signal means that the data has been accepted and a new one can be received.
- Once **DataAck** is activated, **NewData** is deactivated at the next rising edge of the clock cycle.
- At the next clock, le **DataAck** signal can be deactivated.
- And the cycle can start again.

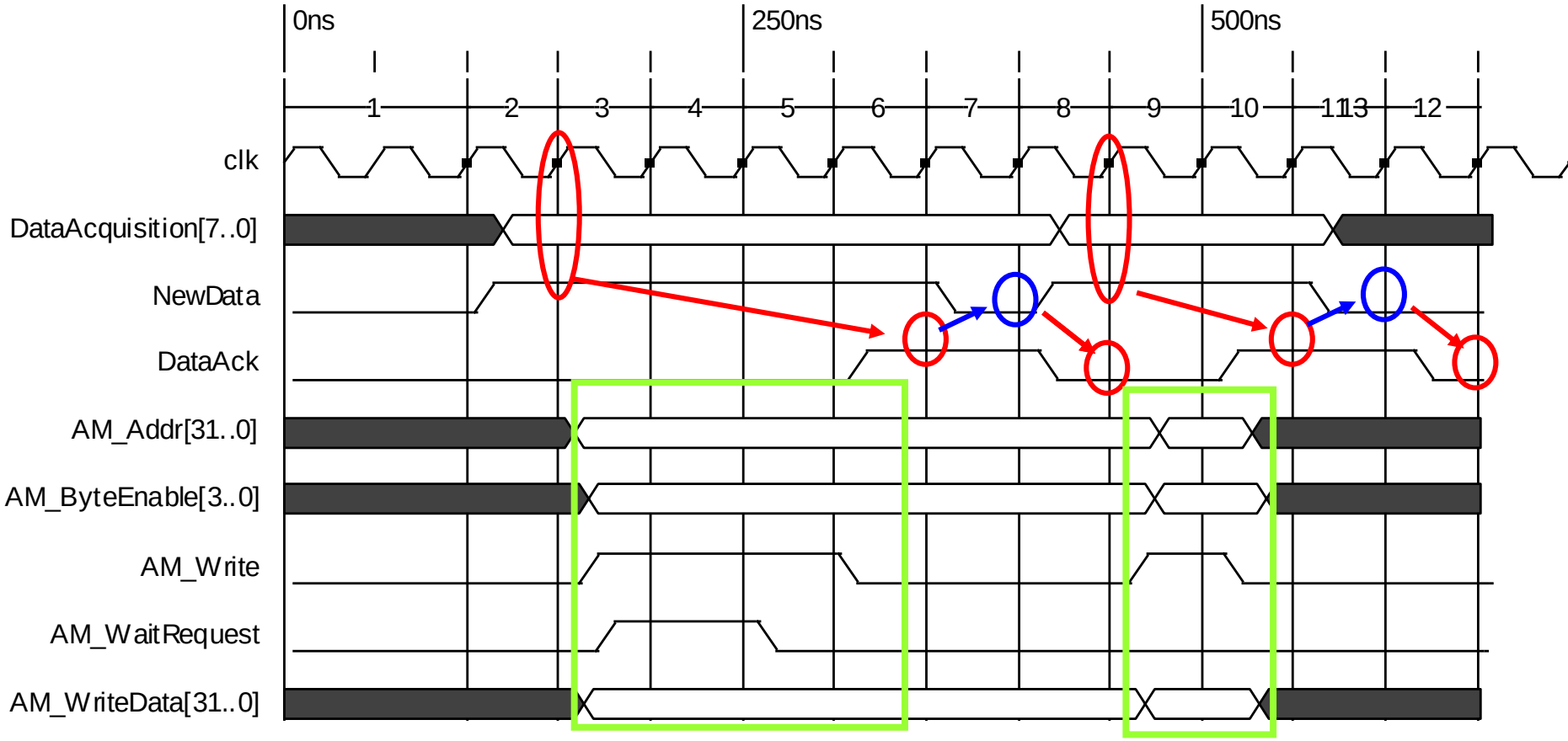
Example:
Master Interface on Avalon Bus in VHDL

- With this mechanism, a DMA unit on the Avalon bus will take the data and copied it in memory. With every new data, the next memory position will be used until all the specified length of data is receive.
- The **start address** of write data in memory and the **length** of the data transfer are seen as **registers** in the programmable DMA controller to design.
- The transfer **start when the length is different of 0**.
- When all the specified memory length is full, the start address is used again and the process continue until the processor send a stop command by writing a '0' to the length register.

Example:
Master Interface on Avalon Bus in VHDL



Example:
Master Interface on Avalon Bus in VHDL



Example:
Master Interface on Avalon Bus in VHDL

2 registers to realize the interface:

1.Address register: ***AcqAddress***
2.Length register: ***AcqLength***

Those registers are seen by the processor in the native (register) mode on the Avalon bus

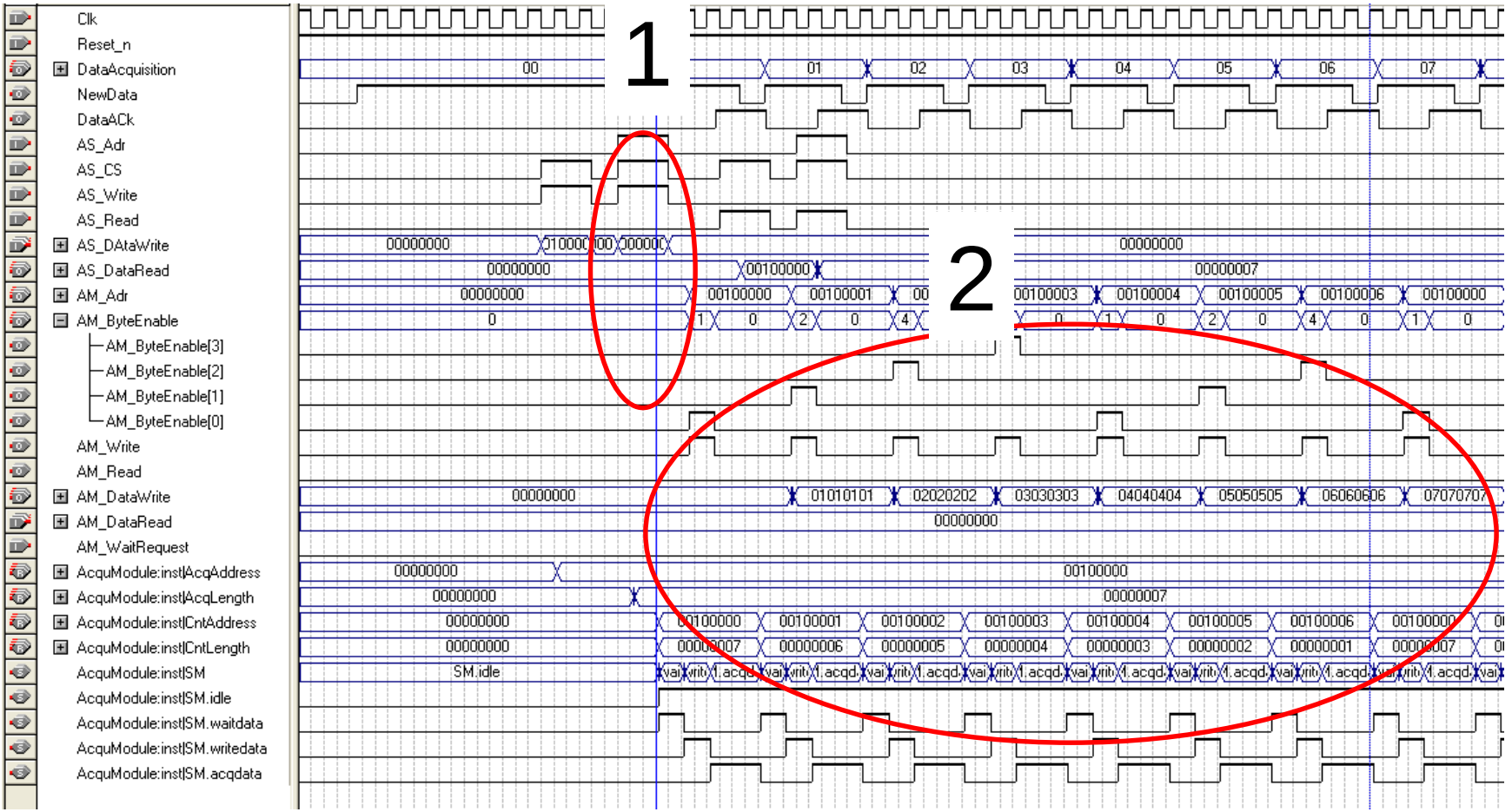
Name	Offset Interface	Offset uP	Data width	Function
AcqAddress	0	0	32 bits	Memory address where to start to put the data
AcqLength	1	4	32 bits	Length of the memory to stock the data

Once those registers programmed the transfers can started if the Request is send to the module by the NewData signal.
The next figure show transfers (AM_xx accesses (2)) after initialization of the module by a processor (AS_xx accesses (1)).

In this non optimized design, a transfer is done for each byte request.

As write accesses only are provided by the master, the AM_read and ReadData[..] bus are not provided.

Example:
Master Interface on Avalon Bus in VHDL

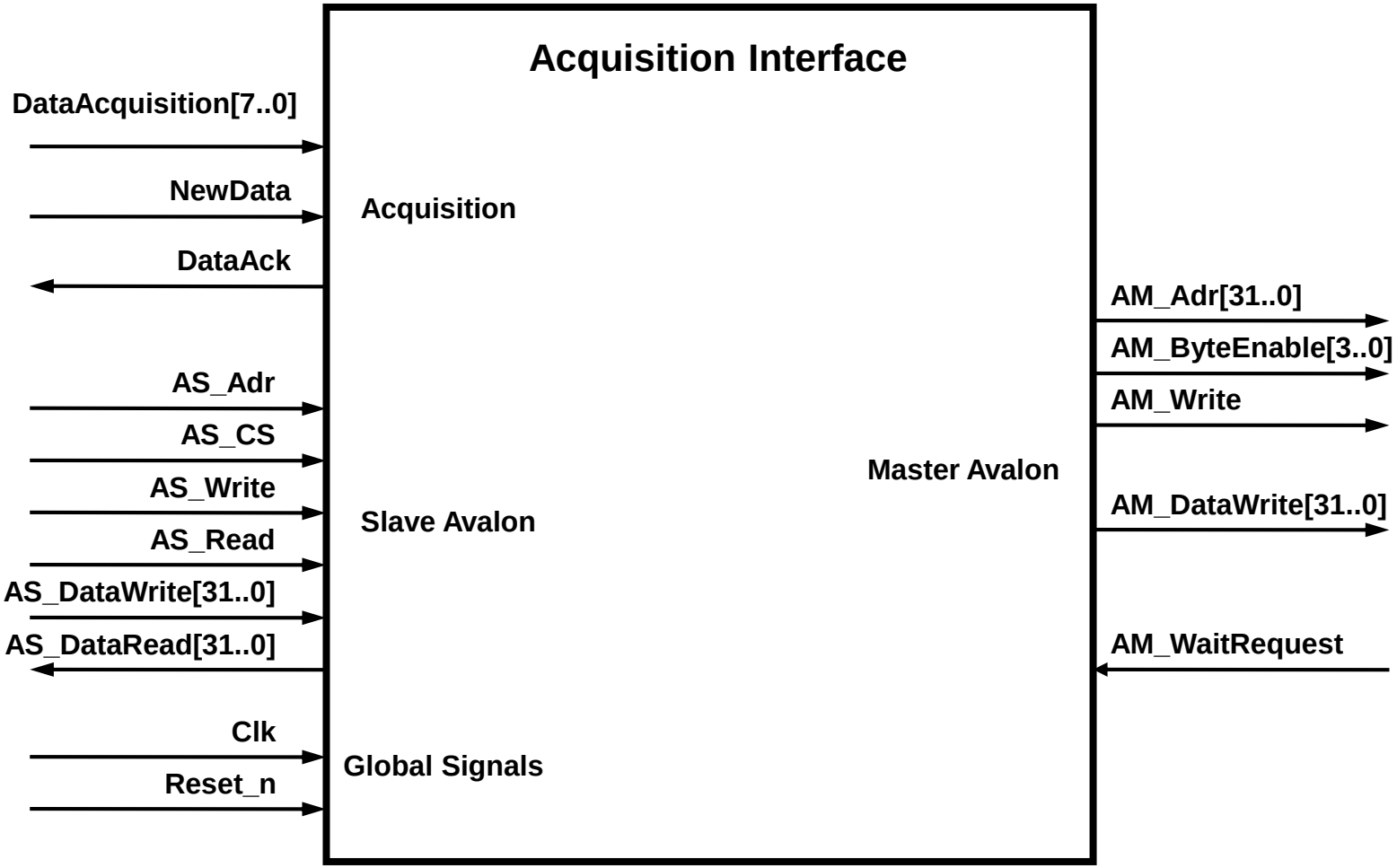


Example:
Master Interface on Avalon Bus in VHDL

- Problem decomposition
- *There are 3 principals units in the design:*

1. *Programmable interface module (interface registers)*
   - *Interface with the Avalon bus in slave mode, receive the memory start address and length. It's possible to read them back.*
2. *Acquisition module*
   - *Receive the data and acknowledge them*
3. *DMA Module, Avalon Master*
   - *Make the transfers with the Avalon bus as a Master. Transfer the data byte by byte, initialize the start address at the end of the bloc transfer.*

Example:
Master Interface on Avalon Bus in VHDL



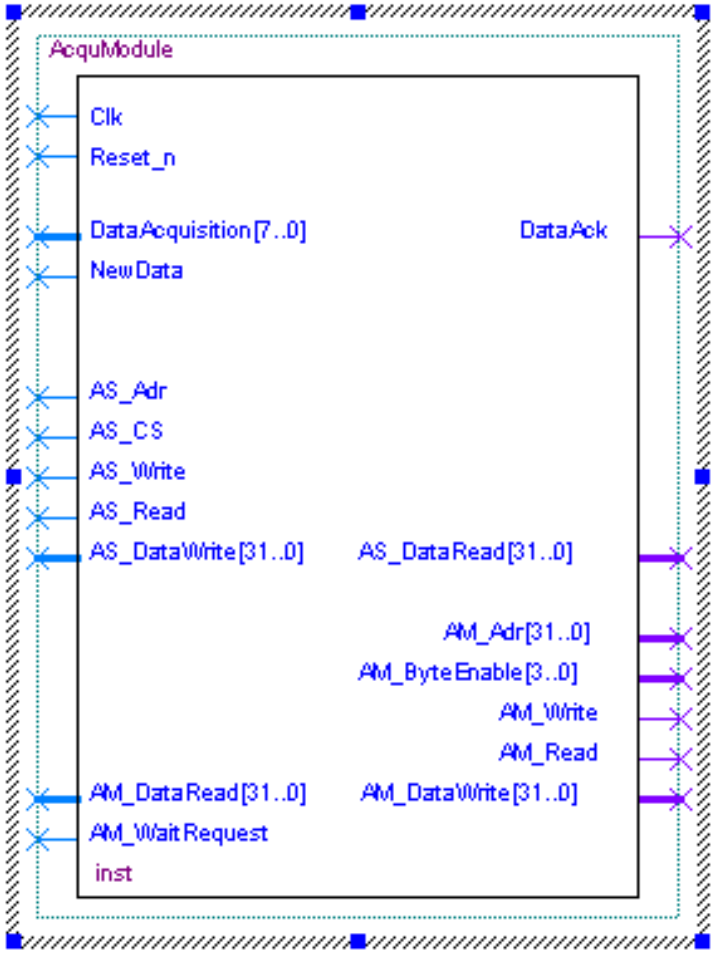
Example:
Master Interface on Avalon Bus in VHDL

```
Entity AcquModule is
Port(
        Clk                 :       IN      STD_LOGIC ;
        Reset_n             :       IN      STD_LOGIC ;
-- Acquisition
        DataAcquisition     :       IN      STD_LOGIC_VECTOR(7 downto 0) ;
        NewData             :       IN      STD_LOGIC ;
        DataAck             :       OUT     STD_LOGIC ;
-- Avalon Slave :
        AS_Adr              :       IN      STD_LOGIC;
        AS_CS               :       IN      STD_LOGIC ;
        AS_Write            :       IN      STD_LOGIC ;
        AS_Read             :       IN      STD_LOGIC ;
        AS_DataWrite        :       IN      STD_LOGIC_VECTOR(31 downto 0) ;
        AS_DataRead         :       OUT     STD_LOGIC_VECTOR(31 downto 0) ;
-- Avalon Master :
        AM_Adr              :       OUT     STD_LOGIC_VECTOR(31 downto 0) ;
        AM_ByteEnable       :       OUT     STD_LOGIC_VECTOR(3 downto 0) ;
        AM_Write            :       OUT     STD_LOGIC ;
        AM_DataWrite        :       OUT     STD_LOGIC_VECTOR(31 downto 0) ;
        AM_WaitRequest      :       IN      STD_LOGIC
) ;
End AcquModule;
```

Example:
Master Interface on Avalon Bus in VHDL

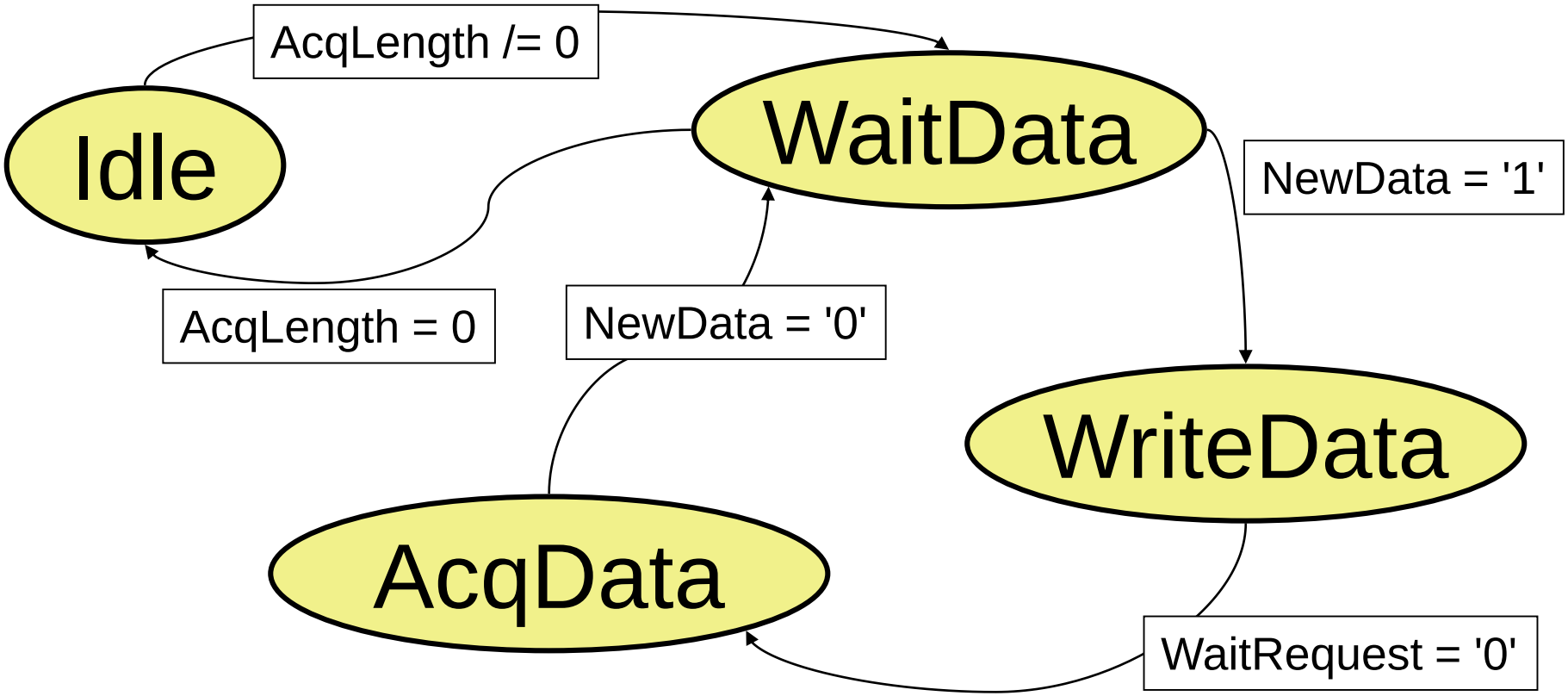
```
LIBRARY ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

Entity AcquModule is
Port(
        …
) ;
End AcquModule;

Architecture Comp of AcquModule is
…
End Comp;
```

Example:
Master Interface on Avalon Bus in VHDL



Example:
Master Interface on Avalon Bus in VHDL

- The sequencer for Acquisition and Write to Avalon bus could be:



Example:
Master Interface on Avalon Bus in VHDL

Architecture Comp **of** AcquModule **is**

TYPE AcqState **IS** (Idle, WaitData, WriteData, AcqData);

```
Signal AcqAddress:      unsigned(31 downto 0);
Signal AcqLength:       unsigned (31 downto 0);
Signal CntAddress:      unsigned(31 downto 0);
Signal CntLength:       unsigned (31 downto 0);
Signal SM:              AcqState;
```

Begin

Example:
Master Interface on Avalon Bus in VHDL

```
-- Interface Registers Accesses
-- Write cycle, 0 wait cycle

pAvalon_Slave_Write:
Process(Clk, Reset_n)
Begin
      if Reset_n = '0' then
            AcqAddress <= (others => '0');
            AcqLength  <= (others => '0');
      elsif rising_edge(Clk) then
            if AS_CS = '1' and AS_Write = '1' then
                  case AS_Adr is
                        when '0' => AcqAddress <= unsigned (AS_DataWrite);      -- Register the adresse
                        when '1' => AcqLength  <= unsigned (AS_DataWrite);      -- Register the length
                        when others => null;
                  end case;
            end if;
      end if;
End Process pAvalon_Slave_Write;
```

Example:
Master Interface on Avalon Bus in VHDL

```
-- Interface Registers Accesses, read cycle
-- !!! Synchronous read with 1 wait cycle !!

pAvalon_Slave Read:
Process(Clk)
Begin
    if rising_edge(Clk) then
        if AS_CS = '1' and AS_Read = '1' then
            case AS_Adr is
                when '0' => AS_DataRead <= std_logic_vector(AcqAddress);-- Read back the Acquisition start
    address
                when '1' => AS_DataRead <= std_logic_vector(AcqLength);-- Read back the length
                when others => null;
            end case;
        end if;
    end if;
End Process pAvalon_Slave_Read;
```

Example:
Master Interface on Avalon Bus in VHDL

```
-- Acquisition process
pAcquisition:
Process (Clk, Reset_n)
Variable Indice : Integer Range 0 to 3;
Begin
    if Reset_n = '0' then          -- Default values at Reset
        DataAck <= '0';
        SM <= Idle;
        AM_Write <= '0';
        AM_ByteEnable <= "0000";
        CntAddress <= (others => '0');
        CntLength  <= (others => '0');
```

Example:
Master Interface on Avalon Bus in VHDL

```
elsif rising_edge(Clk) then                          -- !!! RISING EDGE of Clk !!!!!
   case SM is
        when Idle =>
             if AcqLength /= X"0000_0000" then       -- Start if Length /=0
                  SM <= WaitData;
                  CntAddress <= AcqAddress;
                  CntLength <= AcqLength;
             end if;
        when WaitData =>
             if AcqLength = X"0000_0000" then        -- Idle if Length =0 -> go Idle
                  SM <= Idle;
             elsif NewData = '1' then                -- Receive new data ?
                  SM <= WriteData;
                  AM_Adr <= CntAddress;
                  AM_Write <= '1';
                  AM_DataWrite(7 downto 0)   <= DataAcquisition;
                  AM_DataWrite(15 downto 8)  <= DataAcquisition;
                  AM_DataWrite(23 downto 16) <= DataAcquisition;
                  AM_DataWrite(31 downto 24) <= DataAcquisition;
                  AM_ByteEnable <= "0000";
                  Indice := To_Integer(CntAddress(1 downto 0));      -- 2  low addresses bit as offset activation
                  AM_ByteEnable(Indice) <= '1';
             end if;
```

Example:
Master Interface on Avalon Bus in VHDL

```
            when WriteData =>                                   -- Write on Avalon Bus
                if AM_WaitRequest = '0' then
                    SM <= AcqData;
                    AM_Write <= '0';
                    AM_ByteEnable <= "0000";
                    DataAck <= '1';
                end if;
            when AcqData =>                                     -- Wait end of request
                if NewData <= '0' then
                    SM <= WaitData;
                    DataAck <= '0';
                    if CntLength /= 1 then                      -- Not End of buffer → new address
                        CntAddress <= CntAddress + 1;
                        CntLength <= CntLength - 1;
                    else                                        -- Yes → roll over
                        CntAddress <= AcqAddress;
                        CntLength <= AcqLength;
                    end if;
                end if;
        end case;
    end if;
End Process pAcquisition;
End Comp;
```

Example:
Master Interface on Avalon Bus in VHDL