Lecture 5

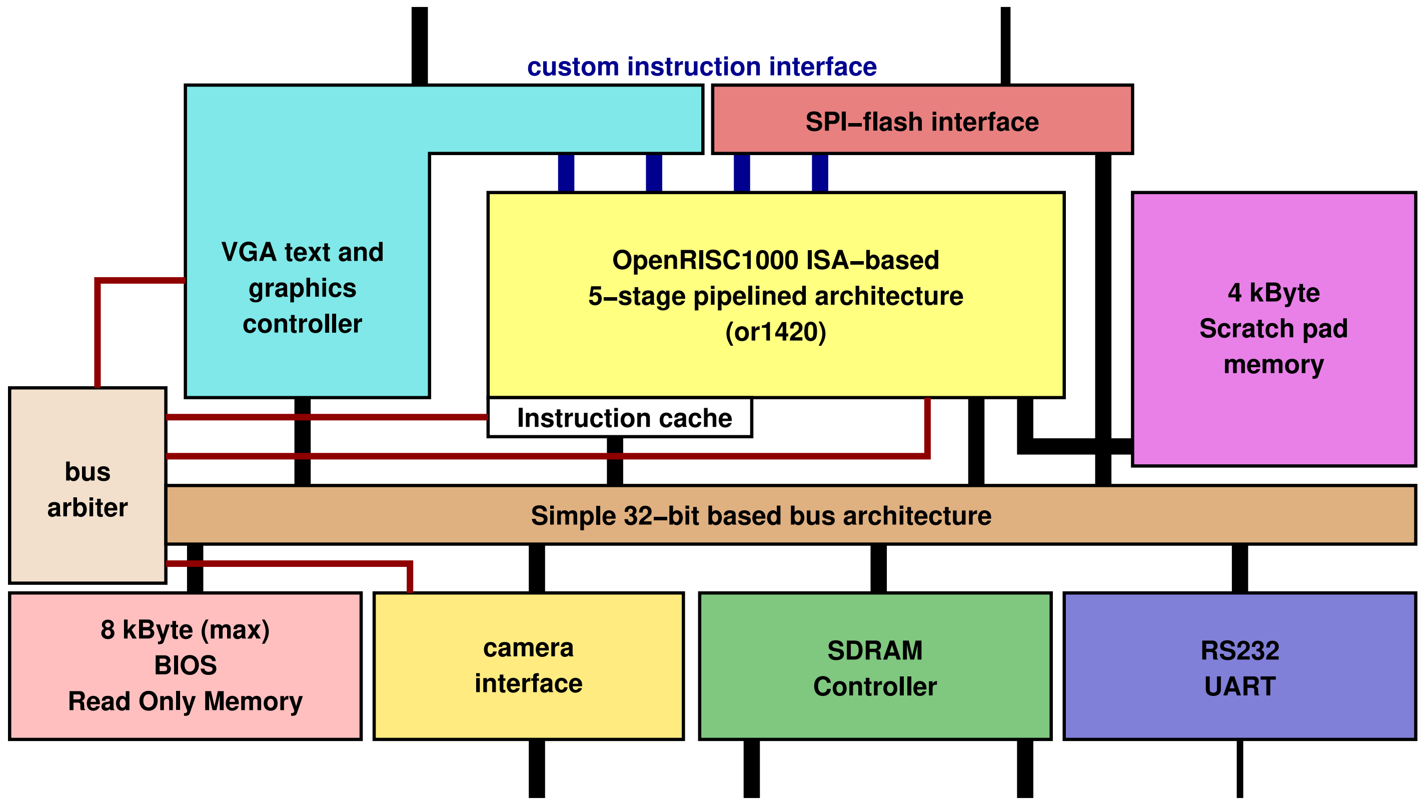
Embedded system design

Bus architectures

CS476 - ESD
April 8, 2024

Dr. Theo Kluter
EPFL

Introduction

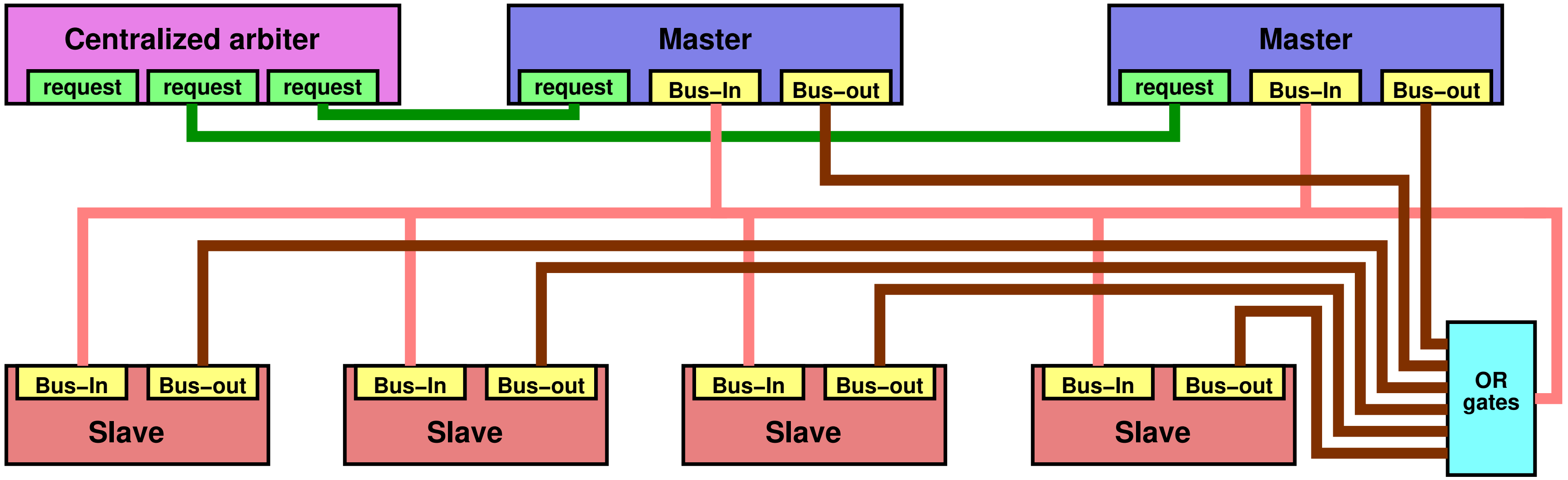

custom instruction interface
VGA text and graphics controller
SPI–flash interface
OpenRISC1000 ISA–based 5–stage pipelined architecture (or1420)
4 kByte Scratch pad memory
Instruction cache
bus arbiter
Simple 32–bit based bus architecture
8 kByte (max) BIOS Read Only Memory
camera interface
SDRAM Controller
RS232 UART

- We have already seen a lot of parts of our embedded system.
- This week we are going to dive into the bus system.

Simple bus architectures

- Let’s start out with the basic idea of a bus system.
- We need to exchange information from (a) *master* device(s) to (a) *slave* device(s).
- this information consists of:
  - The memory address of the access.
  - The type of access (read or write).
  - The data (to/from the master).
  - Some handshake signals.
- There are many different ways how we can set-up this transfer of information, let’s start with the bus realized in our system, a transaction based multi-master burst-enabled shared bus system.

Simple bus architectures



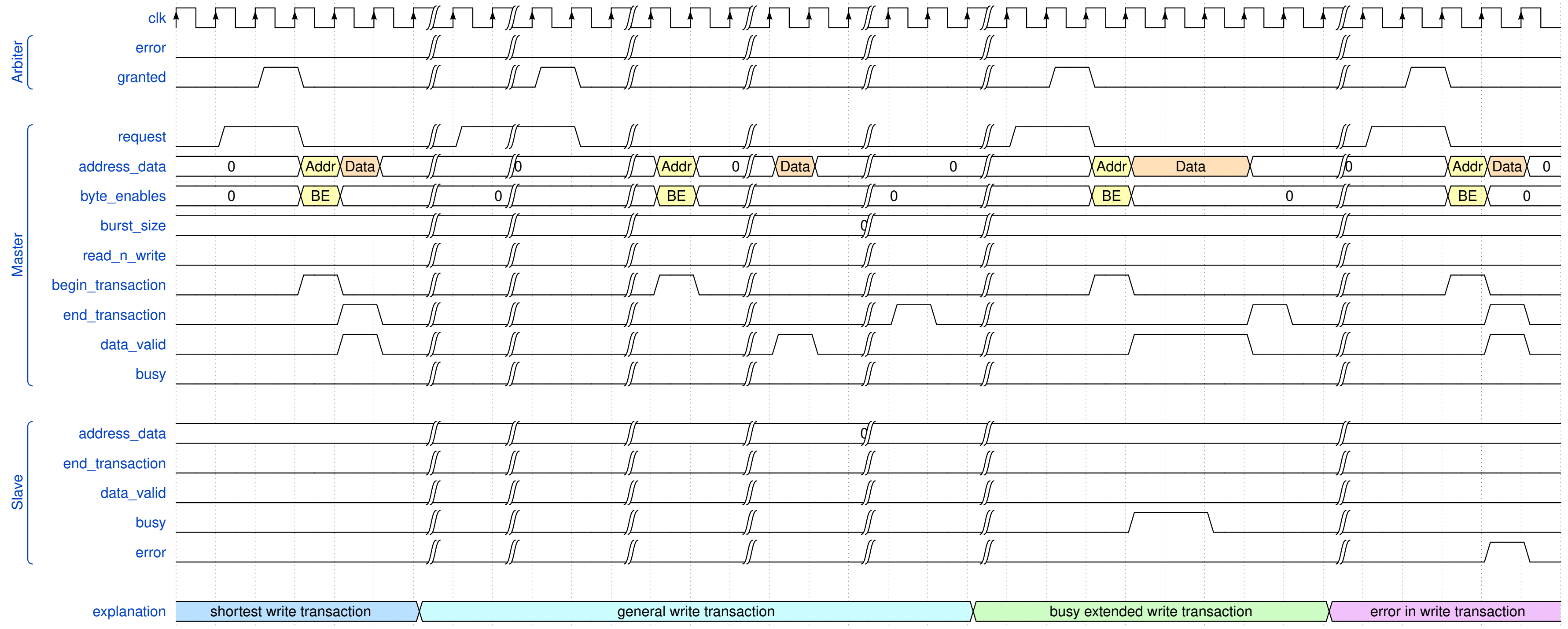
- The block diagram of the applied bus-system is shown above.
- This bus is working with a 74.25MHz clock.
- Note the OR-gates (sometime realize with AND-gates), this is typical for on-chip buses, as we do not apply tri-state (bi-directional) buses as:
  - They are slow (tri-state capacitance, etc.)
  - They may cause short circuits if improper used.

Simple bus architectures

- So which signals are defined in our bus?
  - **address_data** : 32-bit channel that transports the address or data.
  - **byte_enables** : 4-bit channel that indicates in a single transfer which bytes are valid.
  - **burst_size** : 8-bit channel that indicates the number of words to transfer (value+1).
  - **read_n_write** : 1-bit channel indicates a read (when 1) or write transaction (when 0).
  - **begin_transaction** : 1-bit channel that indicates the beginning of a transaction.
  - **end_transaction** : 1-bit channel that indicates the end of a transaction.
  - **data_valid** : 1-bit channel that indicates a valid datum on the *address_data* lines.
  - **busy** : 1-bit channel that indicates that the receiver cannot process yet the datum.
  - **error** : 1-bit channel that indicates a bus error.
- All signals (50-bits) are active-high and should be forced to 0 when not in use (due to the or-gates).

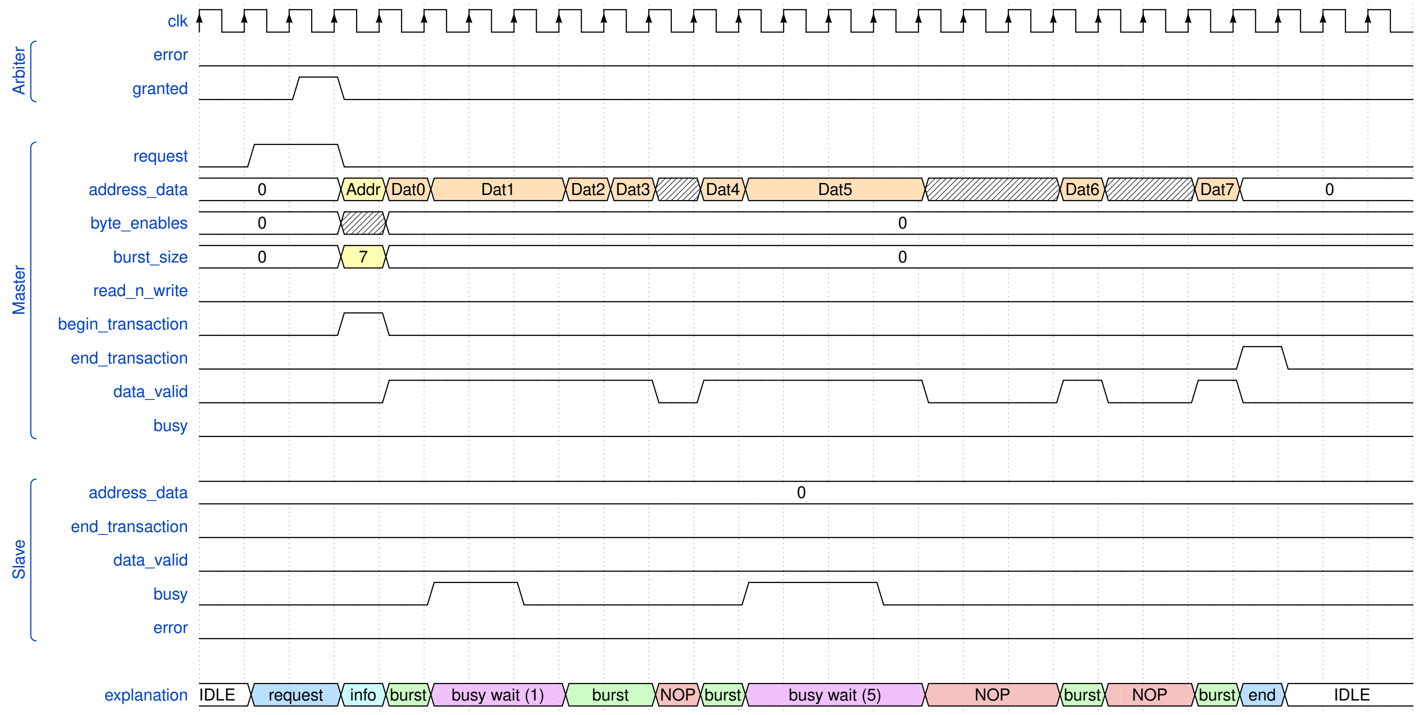
Channel:	**master**		**slave**	
	Bus-in	**Bus-out**	**Bus-in**	**Bus-out**
address_data	required	required	required	required
byte_enables	X	required	required	X
burst_size	X	required	required	X
read_n_write	X	required	required	X
begin_transaction	X	required	required	X
end_transaction	required	required	required	required
data_valid	required	required	required	required
busy	required	optional	required	optional
error	required	X	X	optional

Single word write transactions



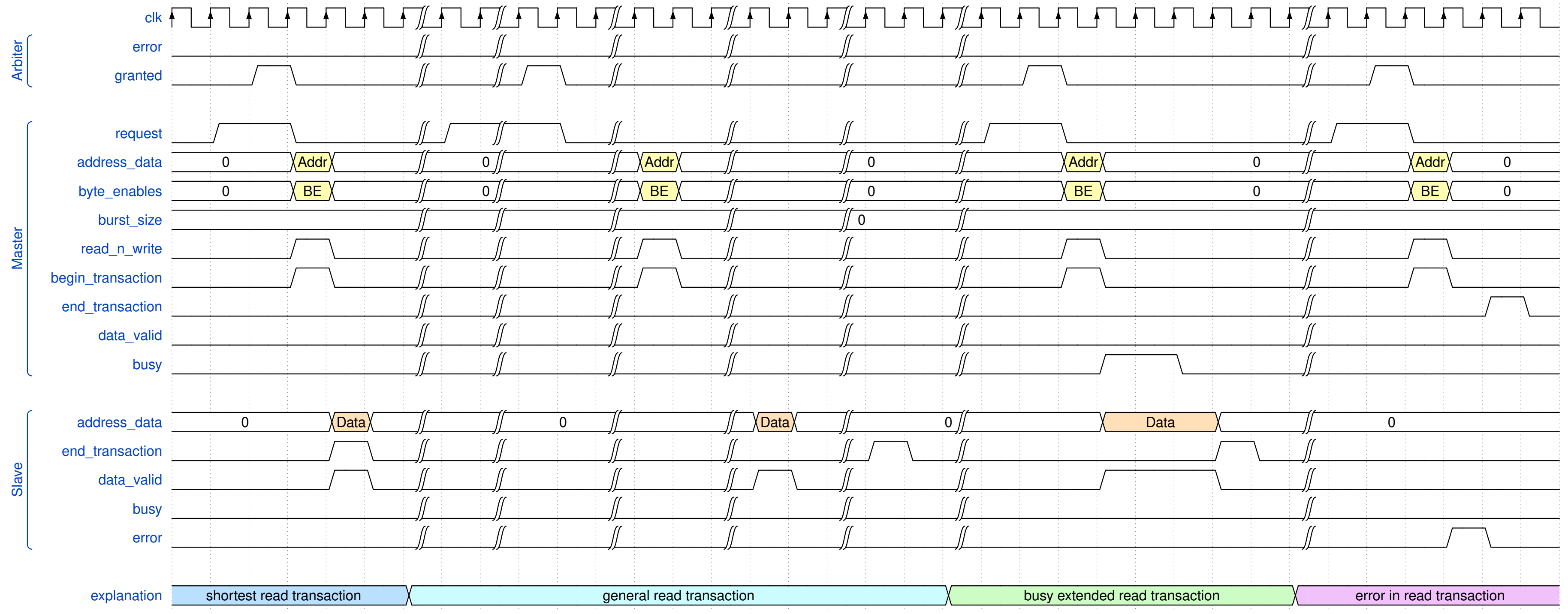
- The signals above are the signals seen at the Bus-out ports (the Bus-in ports see the or-ed version of these signals).
- At the beginning of the transaction (yellow) all information is provided.
- In case of an “error” the master must end the transaction.
- *Note:* The minimal time of a transaction is 5 clock-cycles.

Multiple words write transactions



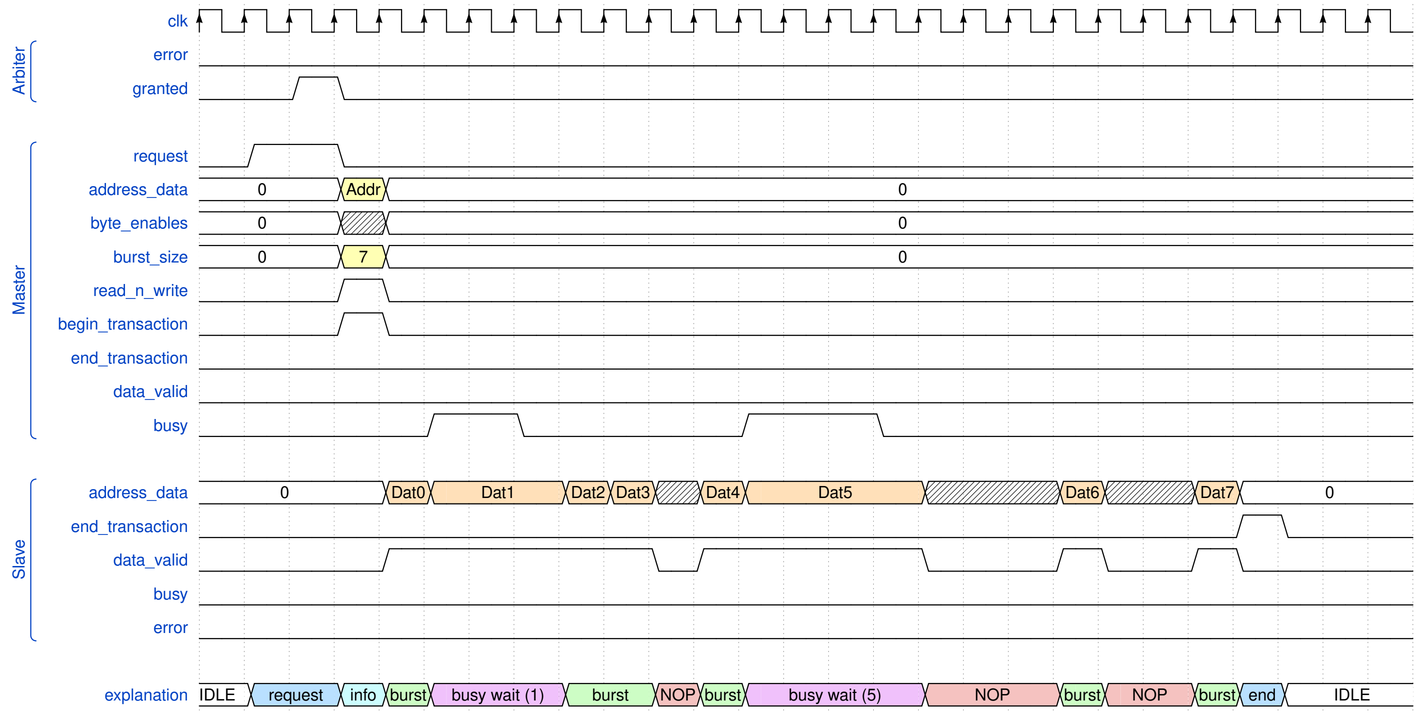
- The signals above are the signals seen at the Bus-out ports (the Bus-in ports see the or-ed version of these signals).
- *Note:* the minimal time of a transaction is 3+NrOfWords clock-cycles.

Single word read transactions



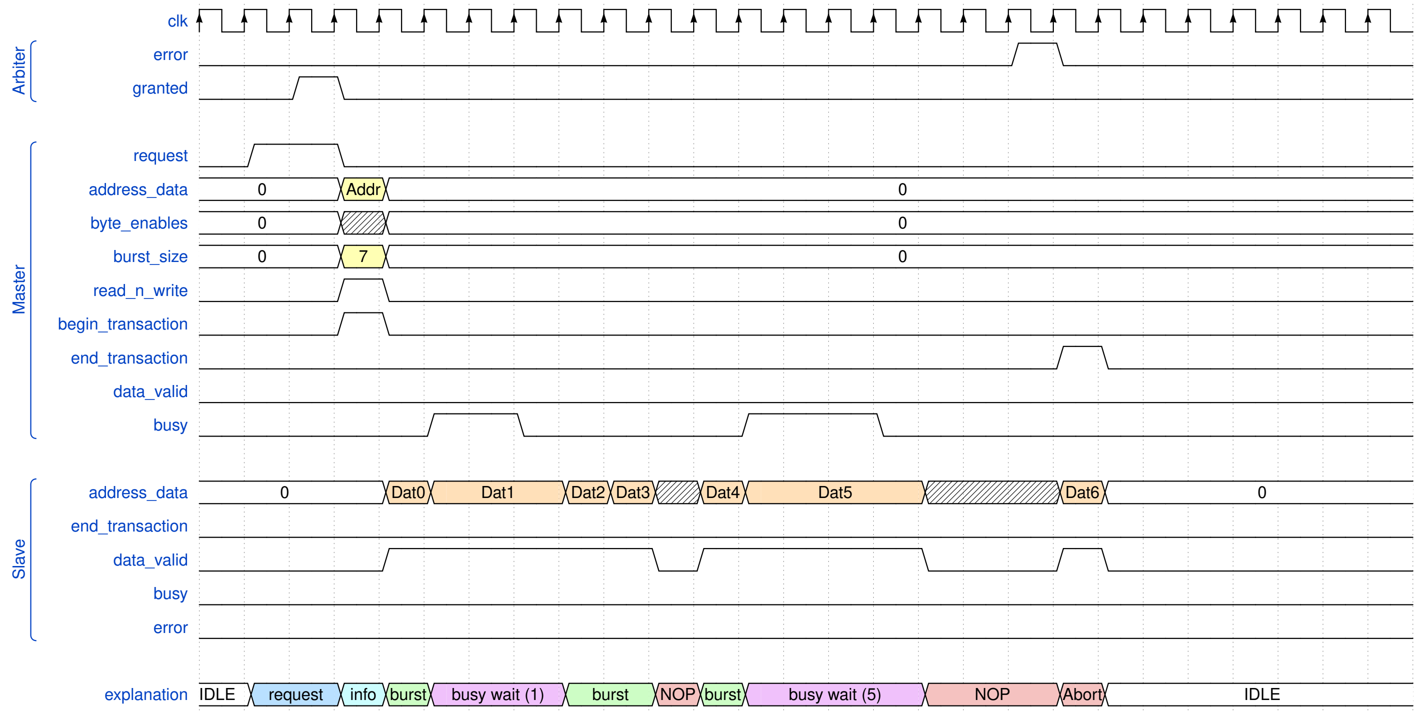
- The signals above are the signals seen at the Bus-out ports (the Bus-in ports see the or-ed version of these signals).
- In case of an "error" the master must end the transaction. Otherwise the slave ends the transaction.
- *Note:* The minimal time of a transaction is 5 clock-cycles.

Multiple words read transactions



- The signals above are the signals seen at the Bus-out ports (the Bus-in ports see the or-ed version of these signals).
- *Note:* the minimal time of a transaction is 3+NrOfWords clock-cycles.

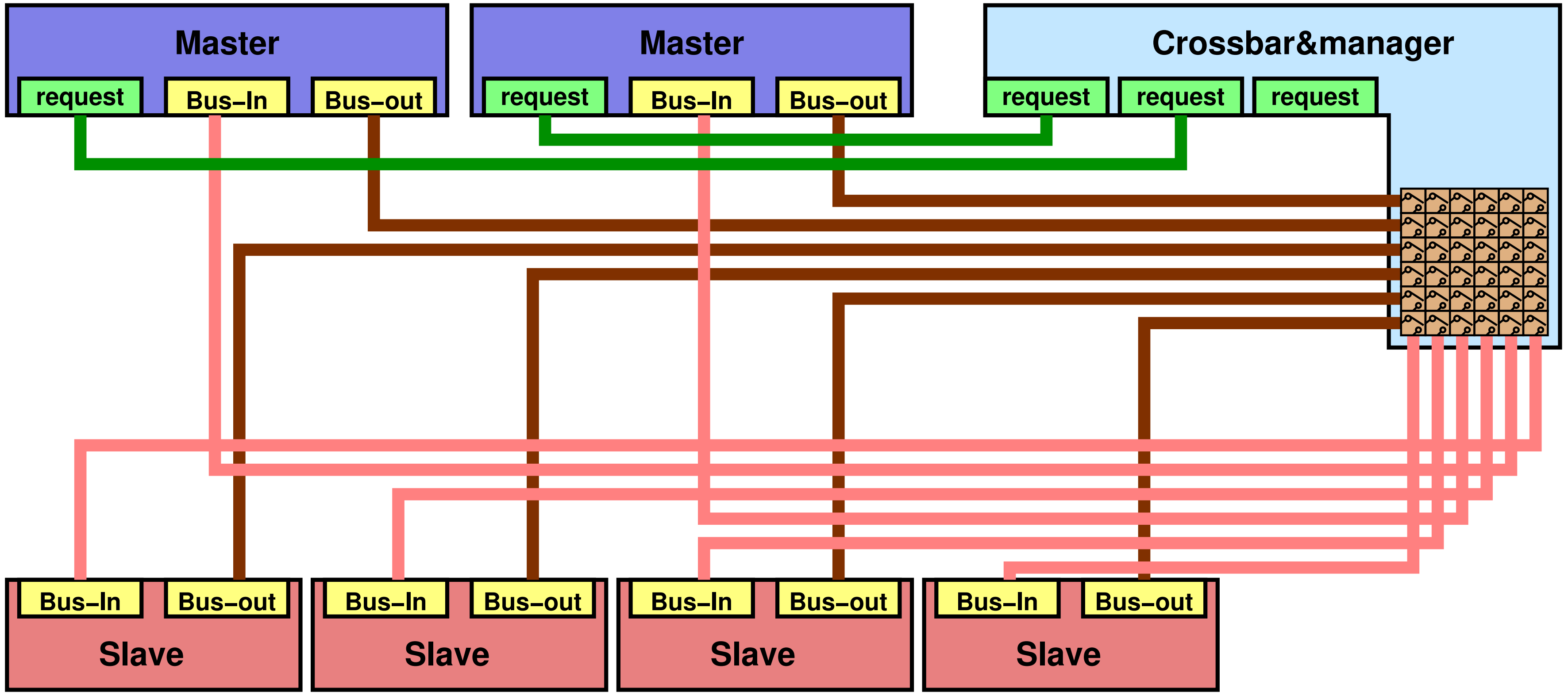
Multiple words read aborted transaction



- In case an error is detected the master must end the transaction.
- If the slave sees an end of transaction before the burst/single read is finished it must end the ongoing transaction and release the bus.

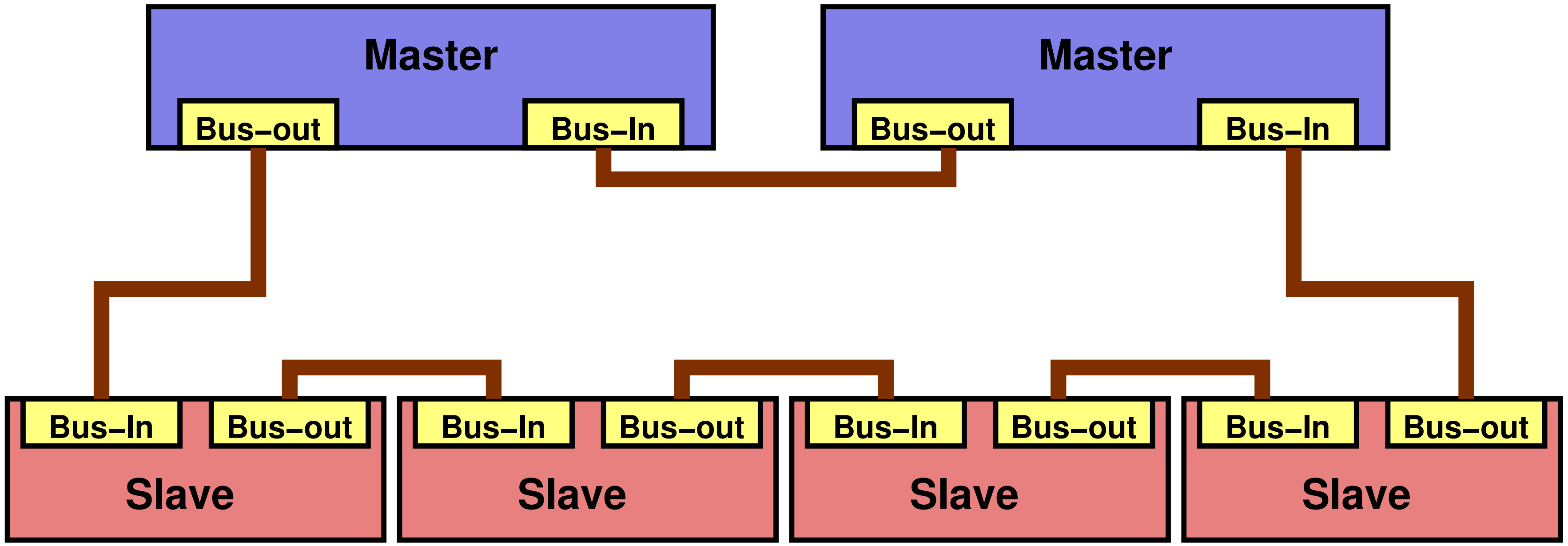
Crossbar architectures

- Is this the only bus architecture, of course not, this is the one we started out with (the most simple one).
- We will visit some more advanced architecture, the first one is the cross-bar (sometimes referred to as point-to-point):



Ring architectures

- The next one is the ring architecture.
- This architecture is sometimes also called *streaming interface* or *network-on-chip (NOC)*.

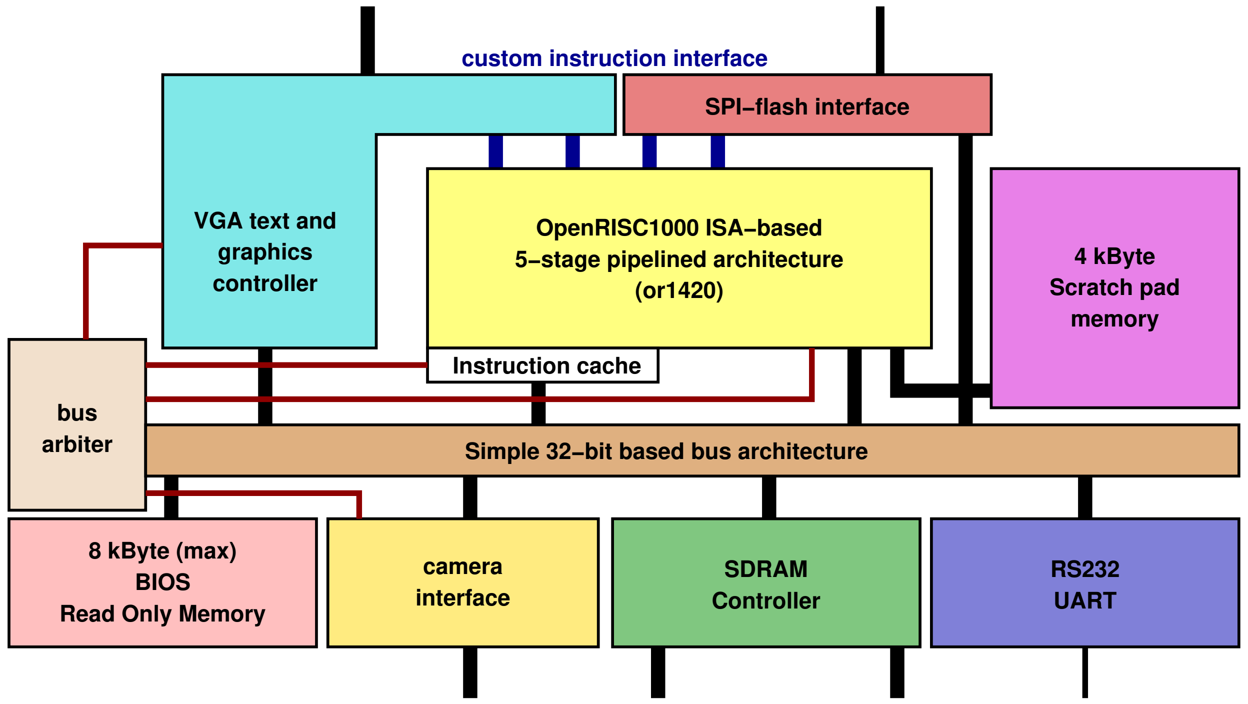


Summary

- There exists a lot of different on-chip bus-systems that apply one or multiple of the shown topologies, some well known are:
  - Arm's AMBA bus that has all of the above topologies.
  - IBM's CoreConnect bus that is a bus-architecture.
  - Altera/Intels Avalon bus that is a special version of a cross-bar architecture.
  - Open Source Hardware's Wishbone bus that allows for all of the above architectures.
- Now that we know how to transfer information let us look into some constructs that are often used:

```
void* memset(void* dest, register int val, register size_t len);
void* memmove(void* s1, const void* s2, size_t n);
void* memcpy(void* dst0, const void* src0, size_t length);
```

Efficiency


custom instruction interface
SPI–flash interface
VGA text and graphics controller
OpenRISC1000 ISA–based 5–stage pipelined architecture (or1420)
4 kByte Scratch pad memory
Instruction cache
bus arbiter
Simple 32–bit based bus architecture
8 kByte (max) BIOS Read Only Memory
camera interface
SDRAM Controller
RS232 UART

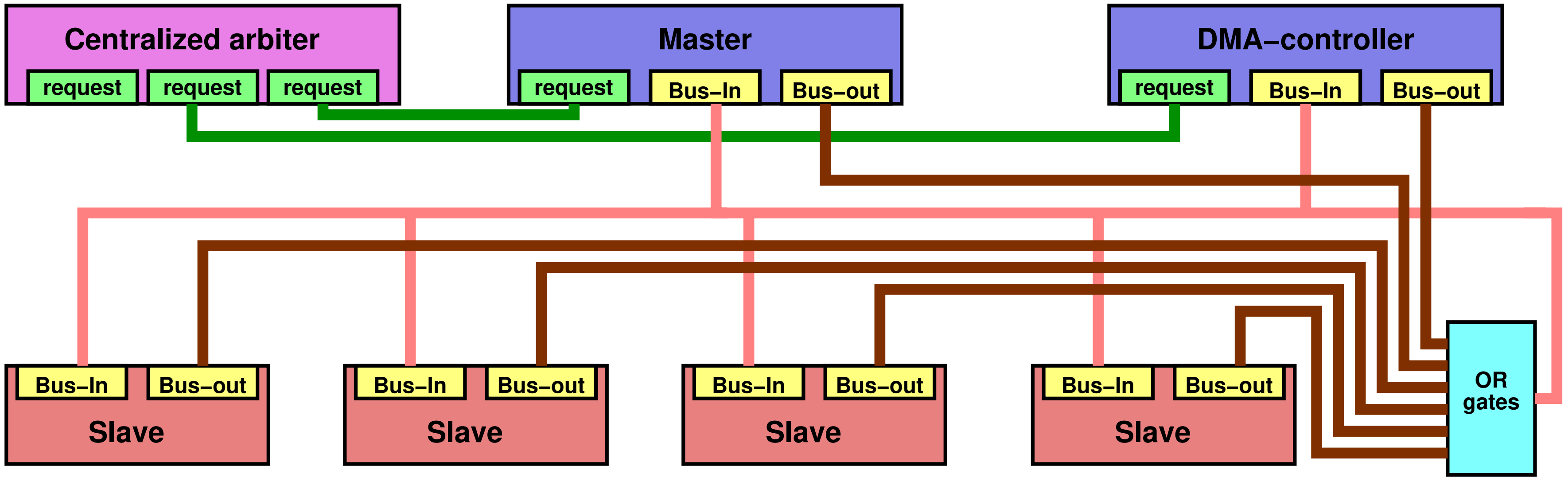
- These constructs execute very inefficient on a CPU...

```
void* memset(void* dest, register int val, register size_t len);
void* memmove(void* s1, const void* s2, size_t n);
void* memcpy(void* dst0, const void* src0, size_t length);
```

Direct Memory Access (DMA)

- Arguably one of the reasons to accelerate such operations let to the invention of the *Direct Memory Access (DMA)*.
- A DMA-controller is a host that is connected to the bus.
- There are basically two types of DMA-controllers:
  - General purpose DMA-controllers.
  - Build-in peripheral DMA-controller.
- We start with the general purpose DMA-controller.

Direct Memory Access (DMA)



- The general purpose DMA-controller basically has two phases of transfer:
  1. Transfer the data from the source device to an internal buffer.
  2. Transfer the data from the internal buffer to the destination device.
- Both transfers are done in a programmable burst-size for efficiency (remember the SDRAM).
- Note: In case of a cross-bar where the source and destination are not the same slaves, or NOC bus-architecture, both phases can be performed in parallel (timely-shifted).
- The *build-in peripheral DMA-controller* only has a single phase, either a transfer to a destination device, or a transfer from a source device.

Direct Memory Access (DMA)

- To be able to use the DMA-controller, it has to be set up by the CPU.
- The minimal information the DMA-controller needs to have/provide is:
  - The source Address.
  - The destination Address.
  - The amount of data to transfer.
  - The mode of operation.
  - The amount of data already transferred.
  - The status of the controller.
  - The interrupt control (later more on this).
- This information can either be provided in special purpose registers of the CPU, or
- as a register map in the memory region (hence the DMA-controller is both a master and a slave device).

Direct Memory Access (DMA)

- A DMA-controller supports different modes of operations:
  - Single address to single address: In this case both the source- and destination address are kept constant.
  - Single address to memory block: In this case the source address is kept constant, and the destination address is auto-incremented.
  - Memory block to single address: In this case the source address is auto-incremented, and the destination address is kept constant.
  - Memory block to memory block: In this case both the source- and destination address are auto-incremented.
- Depending on the source and destination device one of these modi might be required.
- But how does the CPU know when the operation is completed?

Polling

- As the DMA-controller provides (a) status register(s), the CPU can know the status of the DMA-transfer.
- By reading this register over and over again, it can see if the transfer has finished.
- We call this method *polling*.
- Of course this method is very inefficient as:
  - Each request (poll) consumes energy.
  - The CPU reads often exactly the same datum (busy).
  - The CPU is busy with waiting instead of doing some "real work", defeating partially the purpose of a DMA-controller.
- A solution to this might be to poll with lower frequency, however, this could lead to:
  - Loosing data, as the next DMA-transfer is not started fast enough.
  - Loosing performance, as the DMA-controller is ready directly after a poll.
  - ...

Interrupt driven

- A better method is the *interrupt driven* approach.
- In this case the DMA-controller is programmed by the CPU to raise an *interrupt (IRQ)* the moment there is an error and/or the transfer has finished.
- An interrupt-service routine can then handle the next transfer.
- Also this method can have some draw-backs, as:
    1. We have an interrupt latency (the time it takes between the IRQ and the CPU starts the interrupt-service routine).
    2. We have an interrupt-service-routine latency (the number of cycles the CPU requires to take the exception, run the interrupt-service routine, and return to the interrupted program).
    3. We have the IRQ-repetition rate (the frequency the IRQ's come in).
- What can happen is:
    - The CPU is only handling IRQ's, hence not doing anything any more on the main program.
    - IRQ's are "missed" as the CPU is still in an interrupt-service-routine when the next IRQ comes in.
    - The latency's are longer than the time it takes to copy the data by the CPU, hence we "loose".