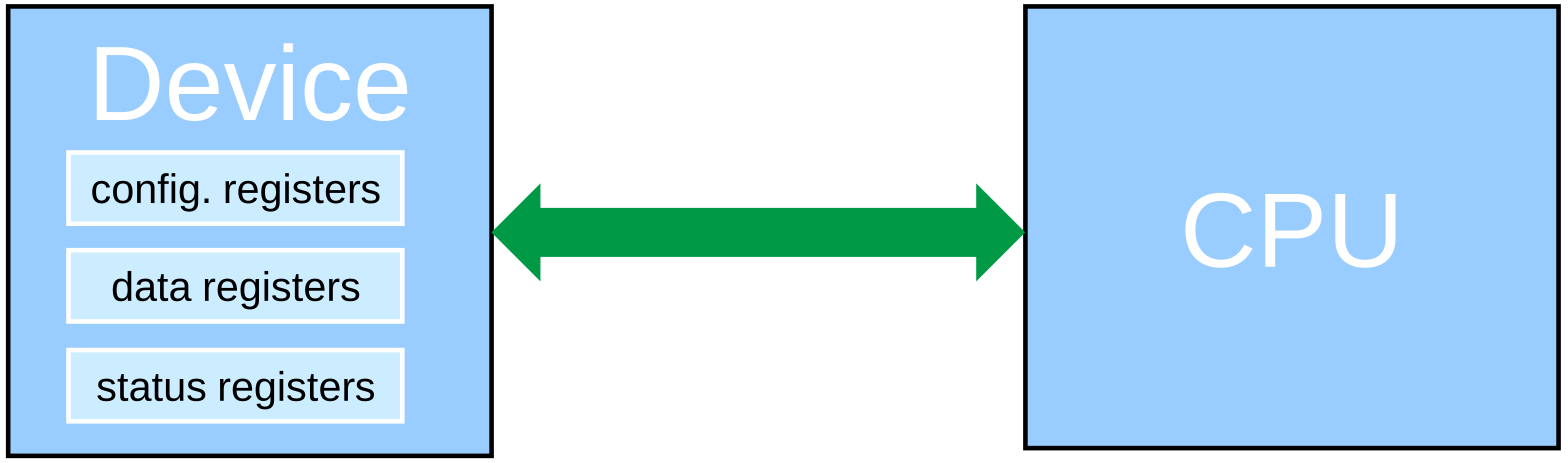
Architecture of Digital Systems II

3 Microprocessor Interfaces

Input and Output Devices

In embedded systems, the CPU needs to communicate with its environment (e.g., disk drives, graphics displays, electric servo motors, ...). The digital interface between an input or output device ("I/O device") and the CPU is typically a set of registers:



- Configuration registers – written by the software to configure the operation of the device
- Data registers – hold values that represent data exchanged between CPU and device (registers are read or written)
- Status registers – provide status and signaling information

Programming I/O

There are two ways microprocessors support the programming of I/O devices:

- I/O instructions. The architecture provides special instructions for accessing devices (for example: the IN and OUT instructions of the Intel x86 architectures). The I/O address space is separate from the main memory address space.
- Memory-mapped I/O. The device registers are accessed using addresses from the main memory address space. Programs use the normal memory access instructions to read and write the device registers.

Most CPUs today use memory-mapped I/O.

Busy-Wait I/O – "Polling"

A simple way to access input/output devices is called polling or busy-wait I/O: The CPU monitors the status of the I/O device to detect when the device has finished a transaction and is ready to communicate with the CPU. Monitoring is done by repeatedly reading the device's status register.

Depending on the services they provide, I/O devices can be significantly slower than the processor. While a device is busy performing a transaction, the CPU may wait for many cycles before the device becomes ready again.

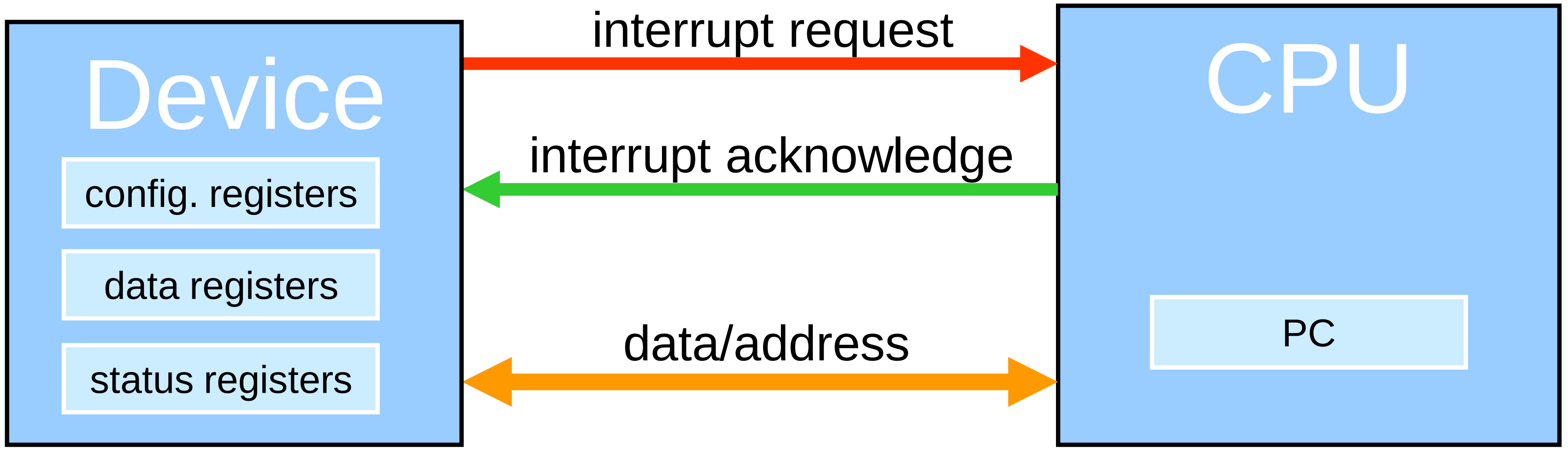
Busy-wait I/O is easily implemented. However, during polling, the CPU does nothing but monitor the device's status.

Interrupts

A more efficient way to implement I/O is using interrupts. While a slower transaction is in progress on a particular I/O device, the CPU can continue working on other tasks, e.g., controlling other I/O devices or performing computations on the data received or to be transmitted.

The basic idea is that a peripheral device signals the CPU that it is ready to send or receive data using the interrupt mechanism. The program currently running on the CPU (the foreground program) is interrupted and a special piece of software, the interrupt handler (also known as device driver) is called in a way very similar to a subroutine call. The handler takes care of the device by sending or receiving the data. In the end, control is returned to the foreground program.

Basic Interrupt Mechanism



Basic mechanism:

- When the I/O device needs service, it asserts the interrupt request signal (often called IRQ).
- When the CPU is ready to handle the interrupt, it asserts the interrupt acknowledge signal (often called IACK) and sets the program counter to the interrupt service routine.

The CPU checks the interrupt request signal at the beginning of execution of every instruction.

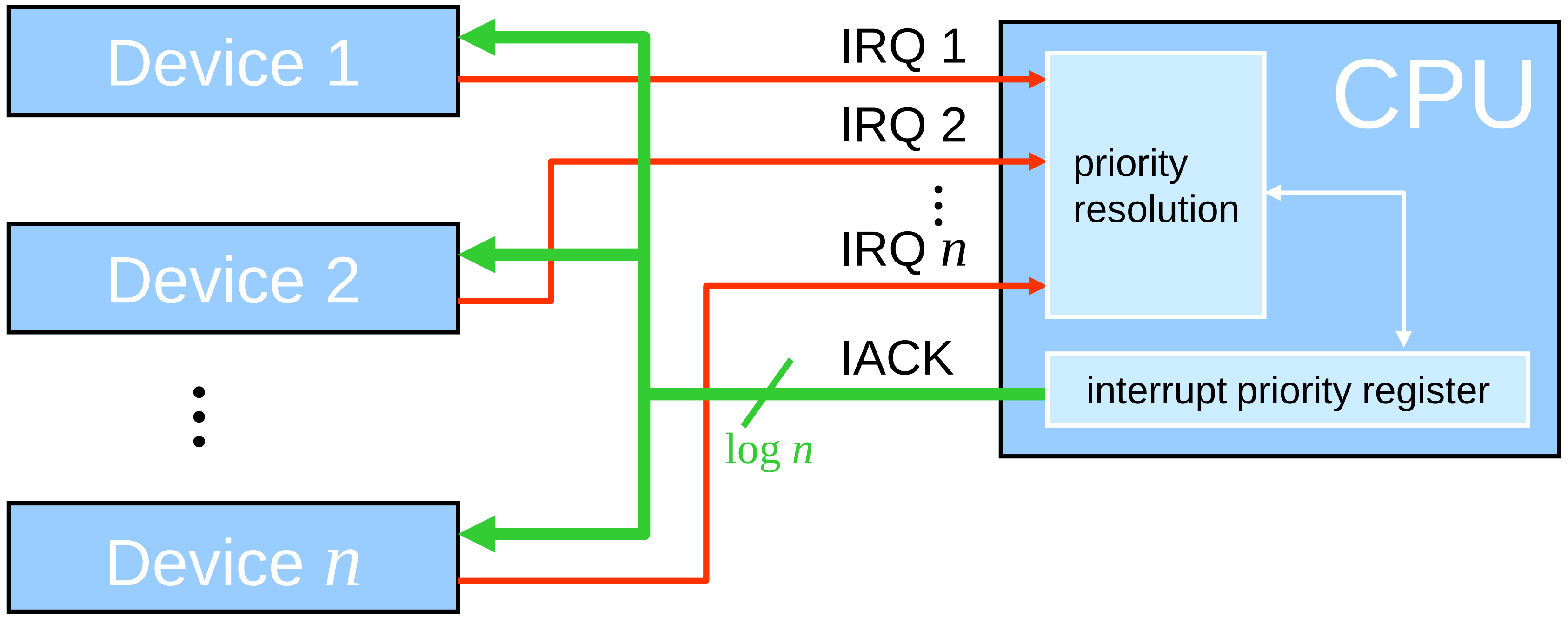
Handling Multiple I/O Devices

Practical systems have more than one I/O device. The interrupt mechanism must be able to distinguish between different sources of interrupts. There are two widely used concepts generalizing the interrupt mechanism for multiple I/O devices:

- interrupt priorities:
  the CPU may treat some I/O service requests as more important than others
- interrupt vectors:
  the I/O device can tell the CPU which service routine should handle its request

(*These two concepts can be used together.*)

Prioritized Interrupts



The CPU provides several interrupt request lines. Typically, the lines with lower numbers are given higher priorities.
The acknowledge signal encodes the winning interrupt priority in binary form – a device knows that its interrupt request is serviced by finding its priority number on the acknowledge lines.

Masking of Interrupts

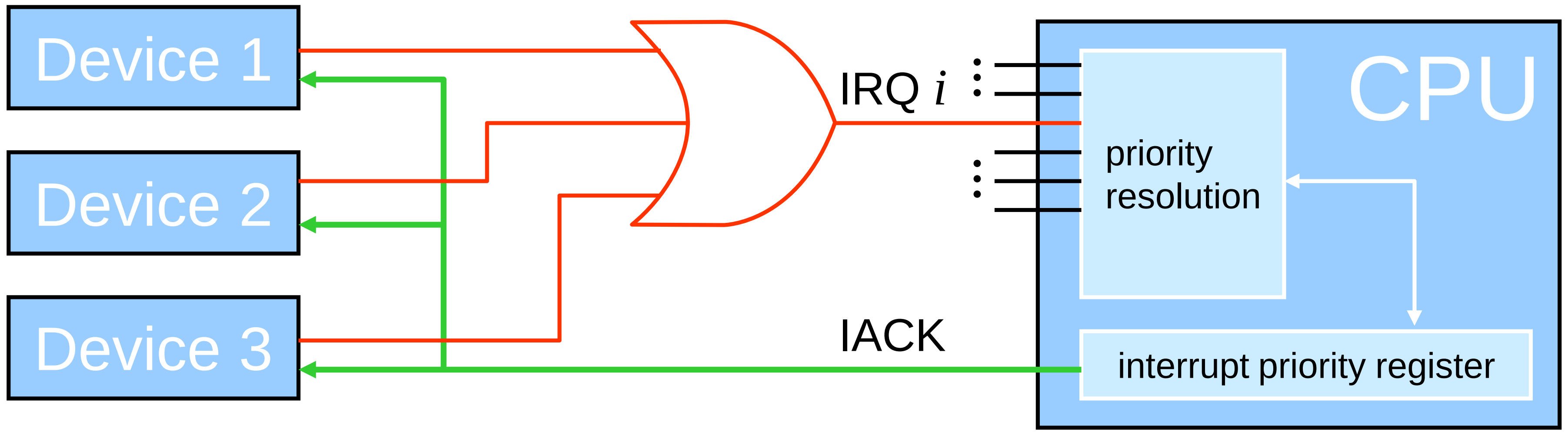
The priority resolution mechanism in the CPU ensures that lower-priority interrupts do not occur while a higher-priority interrupt is being handled: we say, the lower-priority interrupt is masked.

Some architectures offer the possibility to explicitly mask certain interrupts by setting a bit in an interrupt mask register.

The highest-priority interrupt or the interrupt that cannot be disabled by an interrupt mask is usually called the non-maskable interrupt (NMI). It is reserved for situations that require immediate attention like non-recoverable hardware errors (e.g. power failures).

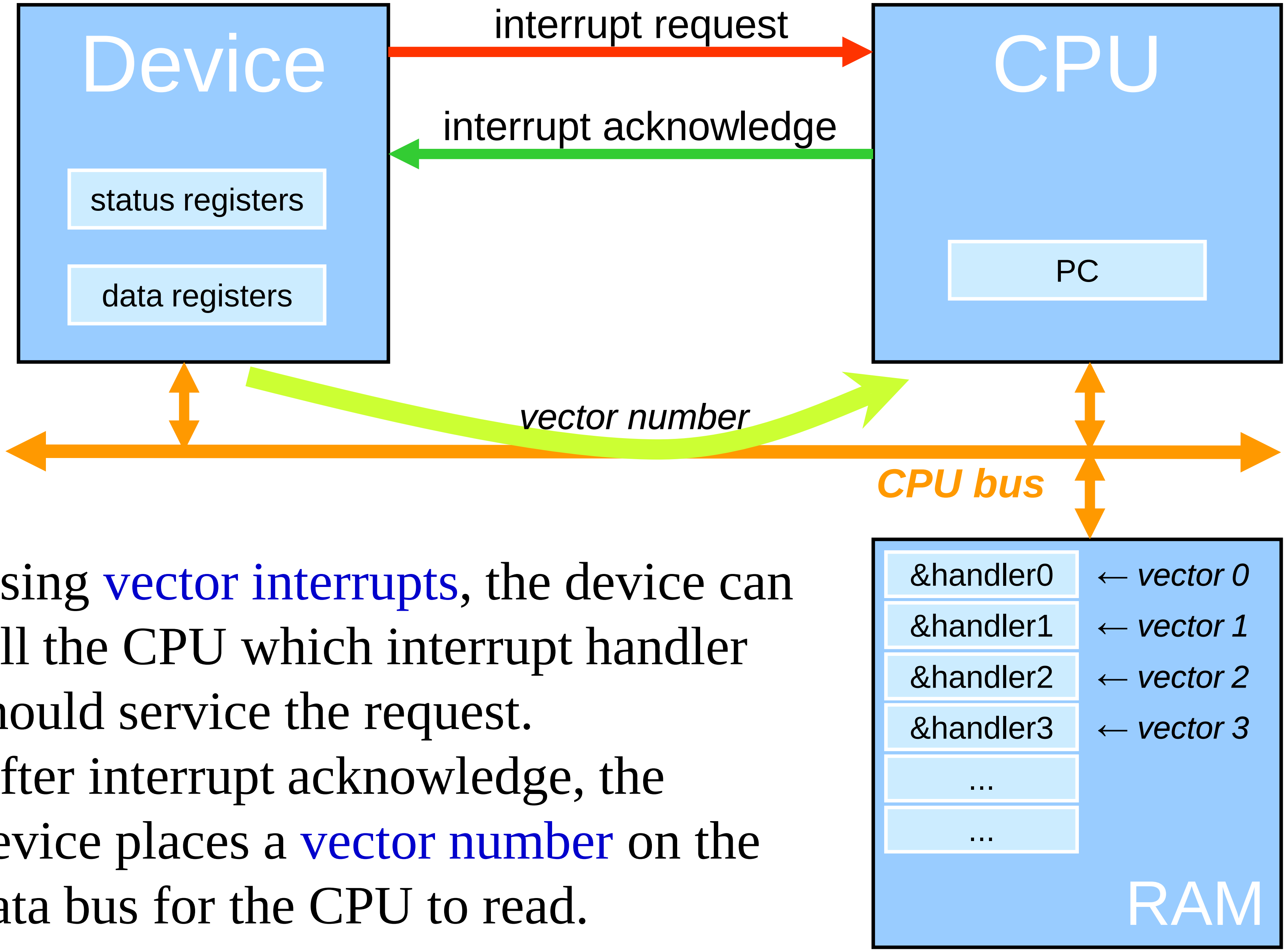
Sharing Interrupt Levels over Several Devices

Often, there are more I/O devices than interrupt lines available. One solution to this problem is to share an interrupt line among several devices with the same priority, using some additional hardware.



The CPU will call the interrupt handler associated with the priority. The handler uses software polling to check the status registers of each device to see which of them is requesting service.

Vector Interrupts



Using vector interrupts, the device can tell the CPU which interrupt handler should service the request.
After interrupt acknowledge, the device places a vector number on the data bus for the CPU to read.

(Vector Interrupts)

The vector number is an index in a table stored in main memory and called the interrupt vector table. The memory location referenced by the index gives the address of the interrupt service routine associated with the vector number. The CPU branches to this routine.
There are a number of advantages to this concept:

- Interrupt handlers may be shared among devices.
- The *device* stores its vector number, not the CPU or the software. The interrupt handler for a device can be exchanged simply by changing the vector number it sends.
- The interrupt vector table can be arbitrarily configured to map between devices and service routines.

Interrupt Procedure, Refined

1. The CPU checks for pending interrupts at the beginning of every instruction.
2. One or more devices send interrupt requests.
3. The CPU compares the highest-priority interrupt with its interrupt priority register. If priority is higher, the interrupt is answered by sending an interrupt acknowledge.
4. The device receives the acknowledge and sends the CPU its interrupt vector number.
5. The CPU calls the interrupt handler pointed to by the vector number in the interrupt vector table. The CPU saves the current PC and possibly some more CPU state.
6. The handler (device driver) possibly saves more CPU state, services the interrupt, restores the saved CPU state, and returns.
7. The return-from-interrupt instruction restores PC and any automatically saved CPU state and sets control flow back to the code that was interrupted.

Interrupt Overhead

The interrupt mechanism involves some processing overhead:

- The jump into the interrupt handler changes the PC. In pipelined processors, this incurs a branch penalty.
- Automatic saving of CPU state requires extra cycles.
- Time is needed to acknowledge the interrupt and to obtain the vector number from the requesting device.
- The interrupt handler may save and restore registers that were not automatically saved by the CPU.
- Also, the return-from-interrupt instruction incurs a branch penalty.
- Restoring of automatically saved CPU state costs time.

The amount of interrupt overhead varies greatly from processor to processor and application to application.

Exceptions

Interrupts, as discussed so far, are mechanisms for the CPU to react to external events. Exceptions and traps deal with internal events and they are handled in a way very similar to interrupts.

An exception is an unexpected internal event, as, for example, an illegal opcode. Exception handling is usually implemented much in the same way as interrupt handling. Both, prioritization and vector tables are used.

- Prioritization is needed because a single instruction can cause several exceptions, such as illegal operands and illegal memory access.
- Vector tables are used so that the software can react appropriately to individual exceptions using specialized exception handlers.

Traps and Supervisor Mode

A trap, also called software interrupt, is an instruction that explicitly generates an exception condition. Trap handling works in the same way as exception handling.

In many cases, traps are used to enter a supervisor mode which has certain privileges over user mode. For example: supervisor mode allows the execution of system code to access peripheral devices or to manage memory allocation in systems with virtual memory. Normal programs run in user mode which protects the system against buggy or malicious code. Attempting to perform privileged actions in user mode leads to an exception.

Whenever a system function is needed, the user program issues a software interrupt to request service from the system software. The CPU switches to supervisor mode and calls the corresponding trap handler.

Example: ARM Software Interrupt

For example, in ARM, the instruction

```
        SWI code_1
```

puts the CPU into supervisor mode.

The argument code_1 is a 24-bit immediate value that can be evaluated by the trap handler.

The CPU saves the return address (next PC value) in register r14_svc, copies the status register CPSR into register SPSR and then jumps to the trap handler in the vector table at location 0x08. After the software interrupt, the CPU state is restored and control is returned to the user program.

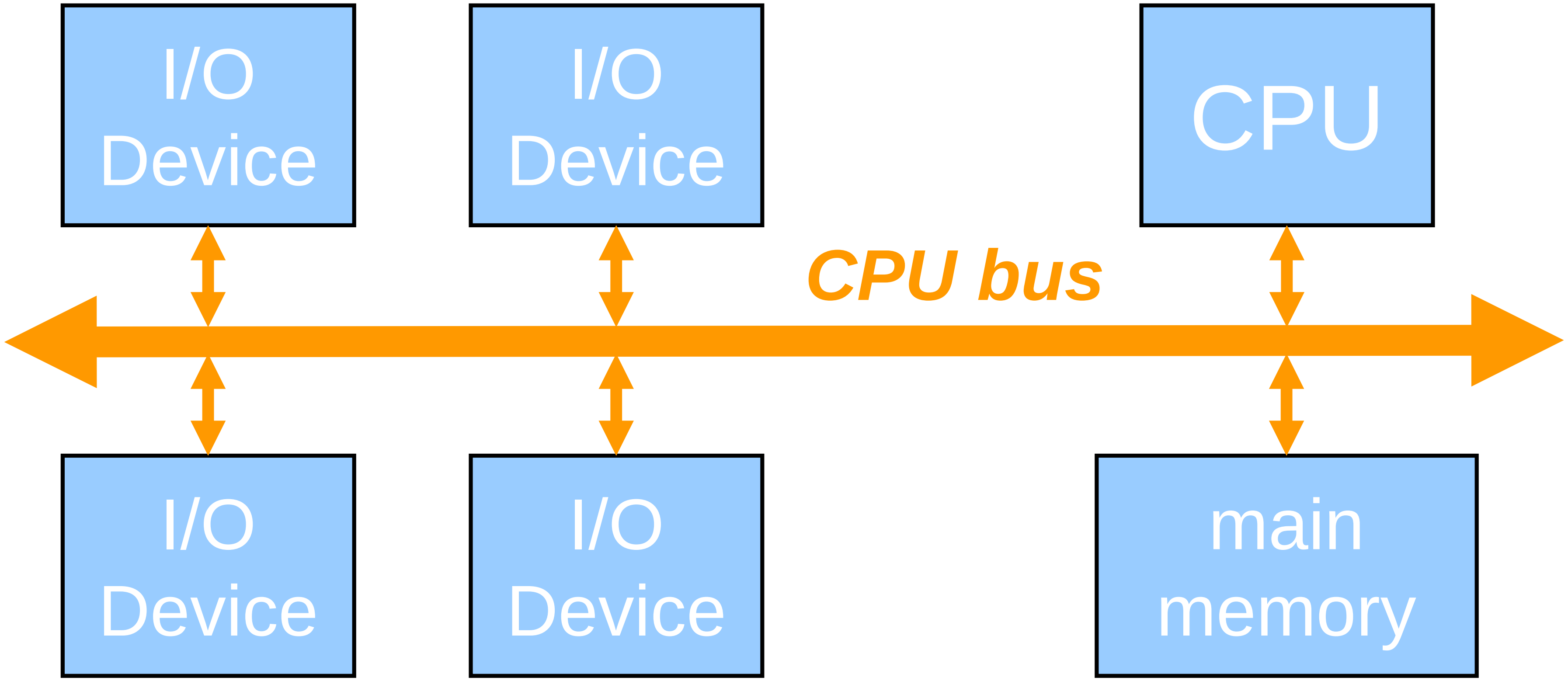
Interrupts, Exceptions, Traps – Example: ARM

Exception	Prio.	Mode	Vector Addr.
Reset	1	SVC	0x00000000
Undefined opcode	6	UND	0x00000004
Software Interrupt (SWI)	6	SVC	0x00000008
Prefetch Abort (Memory error when loading an instruction)	5	Abort	0x0000000C
Data Abort (Memory error when loading data)	2	Abort	0x00000010
IRQ (external interrupt)	4	IRQ	0x00000018
FIQ (fast external interrupt)	3	FIQ	0x0000001C

The vector table does not contain addresses but branch instructions to the exception handlers.
IRQ and FIQ can be masked (using I-bit and F-bit in the CPSR).

The CPU Bus

The CPU exchanges data with the main memory and with I/O devices using the CPU bus. A bus is a collection of wires, some of which can be used bi-directionally. Associated with a bus is a protocol that defines the sequences of signals transmitted over the wires.



Bus operations and signals

The purpose of the bus is to transfer data between the participants. The fundamental operations are reading and writing. Data transfer usually happens between two participants, the master and the slave:

A bus master (e.g., the CPU) may initiate data transfers. It sends a request to a slave (e.g., a memory device) to read or write data at a certain location given by an address.
A bus slave responds to master requests by receiving or returning the requested data.

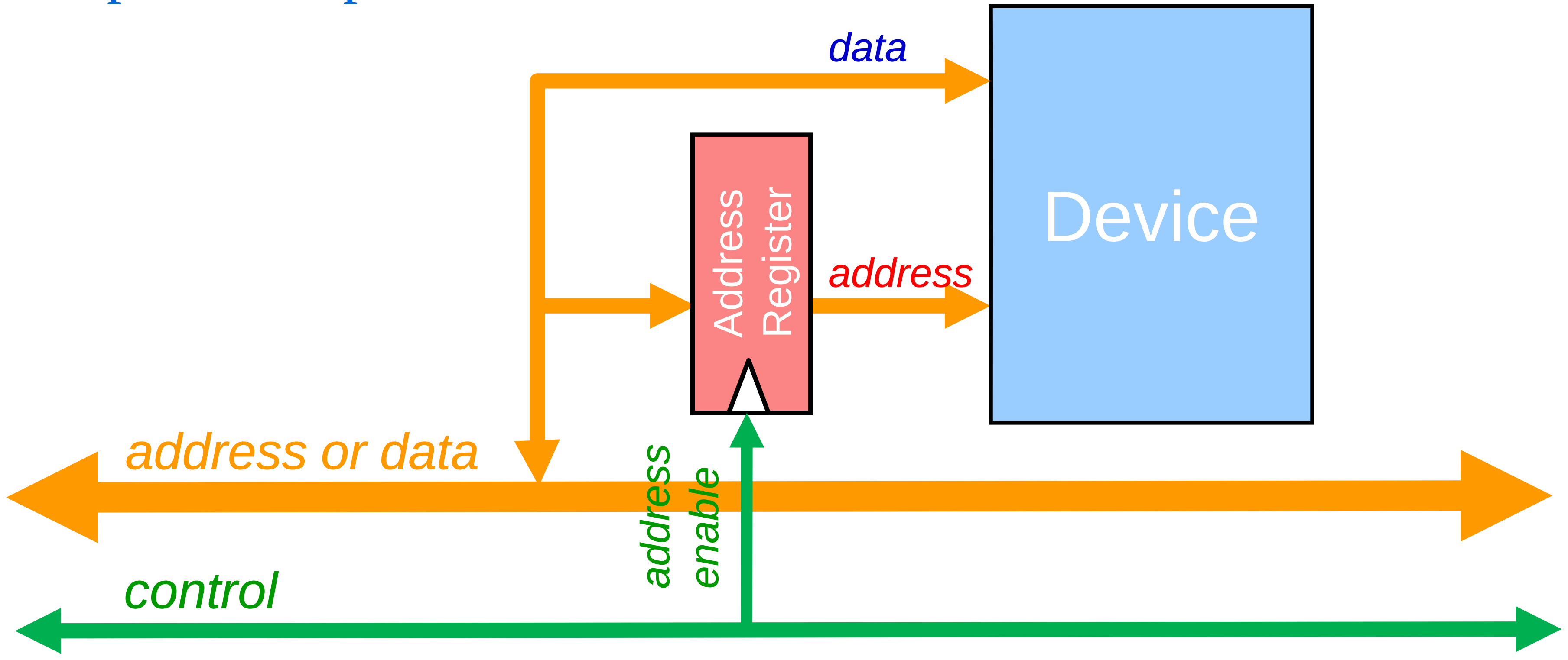
The bus signals (wires) can be categorized into

- data signals
- address signals
- control signals (e.g., to define a clock, to signal read or write, burst, request, acknowledge, etc.)

Multiplexed Bus vs. Split Bus

Address and data may be transmitted over separate wire bundles (split bus) or they may be transmitted sequentially over the same bundle, in two separate phases (multiplexed bus).

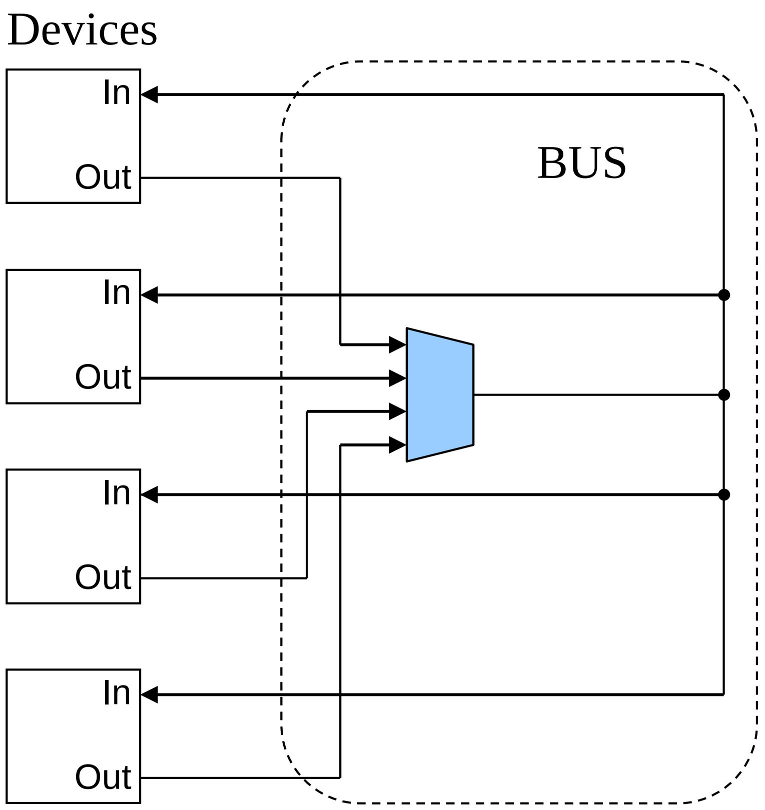
Example: multiplexed bus



Bus Interconnect Implementations

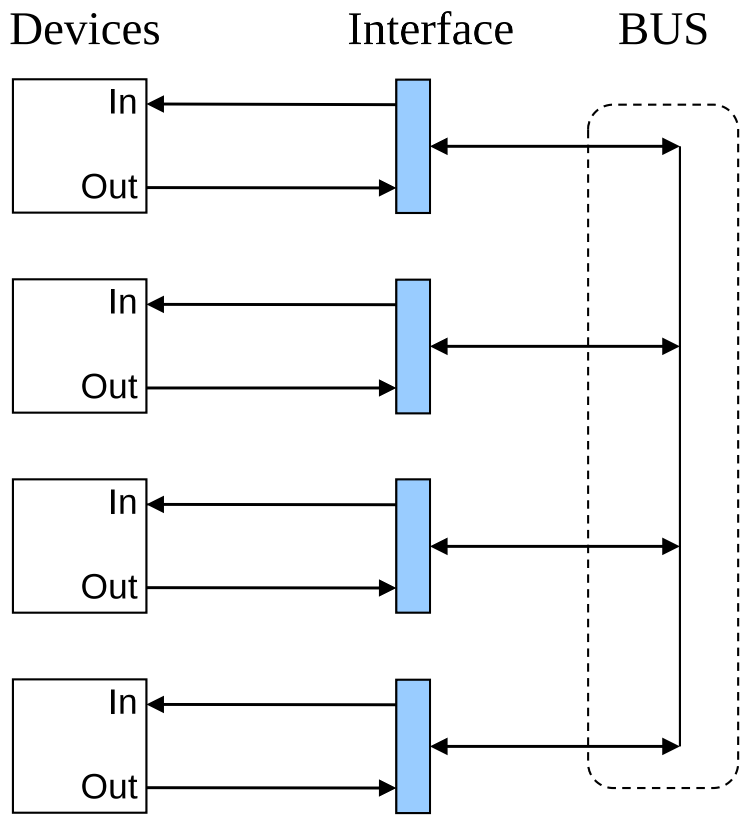
The main characteristic of a bus is that, with some exceptions, its control, address and data lines are used by all participants of the bus. Some lines are bidirectional (e.g., the data lines). The implementation of the bus must allow to selectively couple and decouple devices with/from the bus. There are different ways to physically implement this.

One solution is to use multiplexers. This allows for fast switching, however, it requires a substantial amount of hardware and wiring.



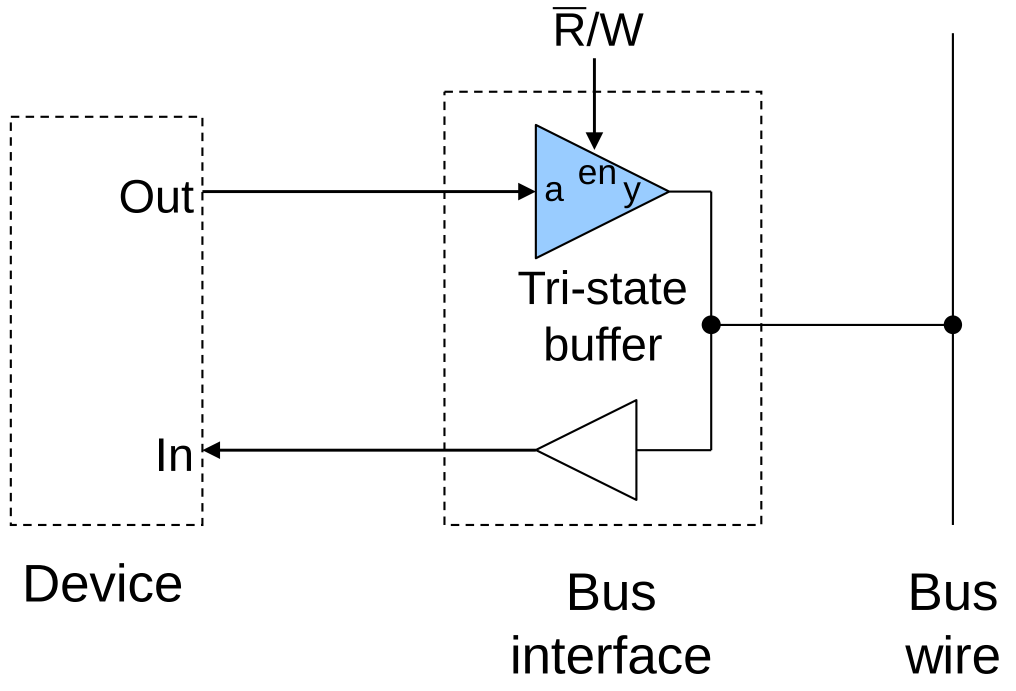
(Bus Interconnect Implementations)

Wiring and hardware overhead can be reduced by connecting devices to the bus via dedicated bus interfaces (e.g., tri-state buffers, open-collector circuits, etc.).



(Bus Interconnect Implementations)

Example of a bus interface using tri-state buffers:



Z: high-impedance output

Tri-state buffer

en	a	y
0	0	Z
0	1	Z
1	0	0
1	1	1

Transaction Timing

The slaves cannot answer the requests of masters arbitrarily fast. Memory and I/O devices need a certain amount of time to process the requests. Therefore, transactions on the bus must follow certain rules, i.e., obey certain sequencing and timing constraints. This is defined by the bus protocol.

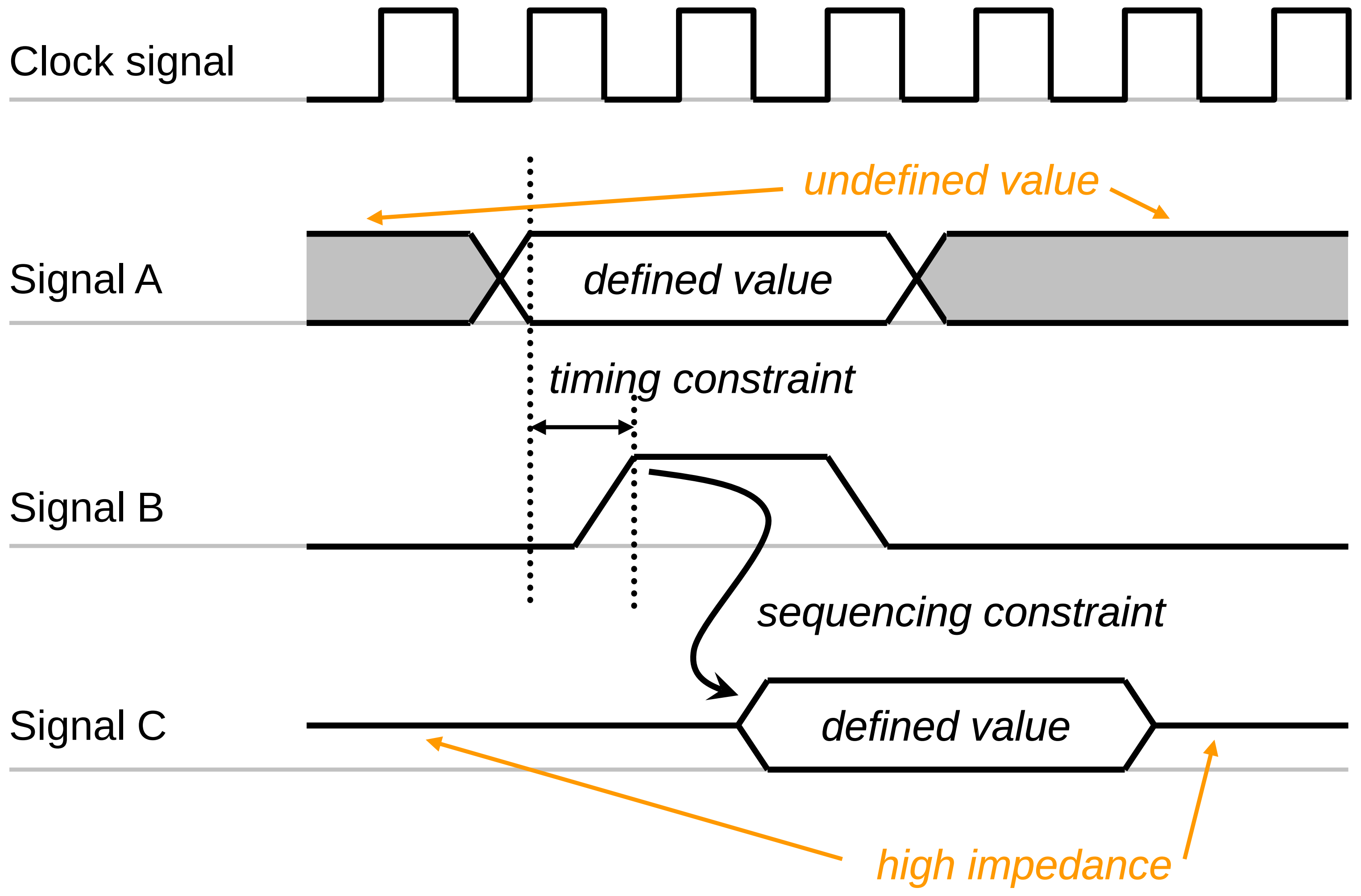
For example, the bus protocol ensures that

- the master supplies the address signal long enough for the slave to evaluate it
- the data is kept on the data lines until it has been properly received

Timing Diagrams

In specification documents, timing diagrams are used to describe bus protocols and to define timing constraints.

Notation examples:



Synchronous vs. Asynchronous Bus

A synchronous bus contains a special signal, the clock signal, to synchronize the bus interfaces of all bus masters and slaves. (This implies that all bus interfaces must run at the same clock speed.) The bus signals are evaluated at the clock event time points, e.g., at the rising or falling edge of the clock signal. Synchronous busses can be very fast. However, the wires cannot be very long as clock skew may result in timing errors.

An asynchronous bus does not have a dedicated clock signal. The internal clocks of the bus participants may run at different speeds. Time points for safe data transfer are signaled by events on some control lines. Most often, handshaking protocols are used for signaling. Asynchronous buses can be used over longer distances because clock skew is not a problem.

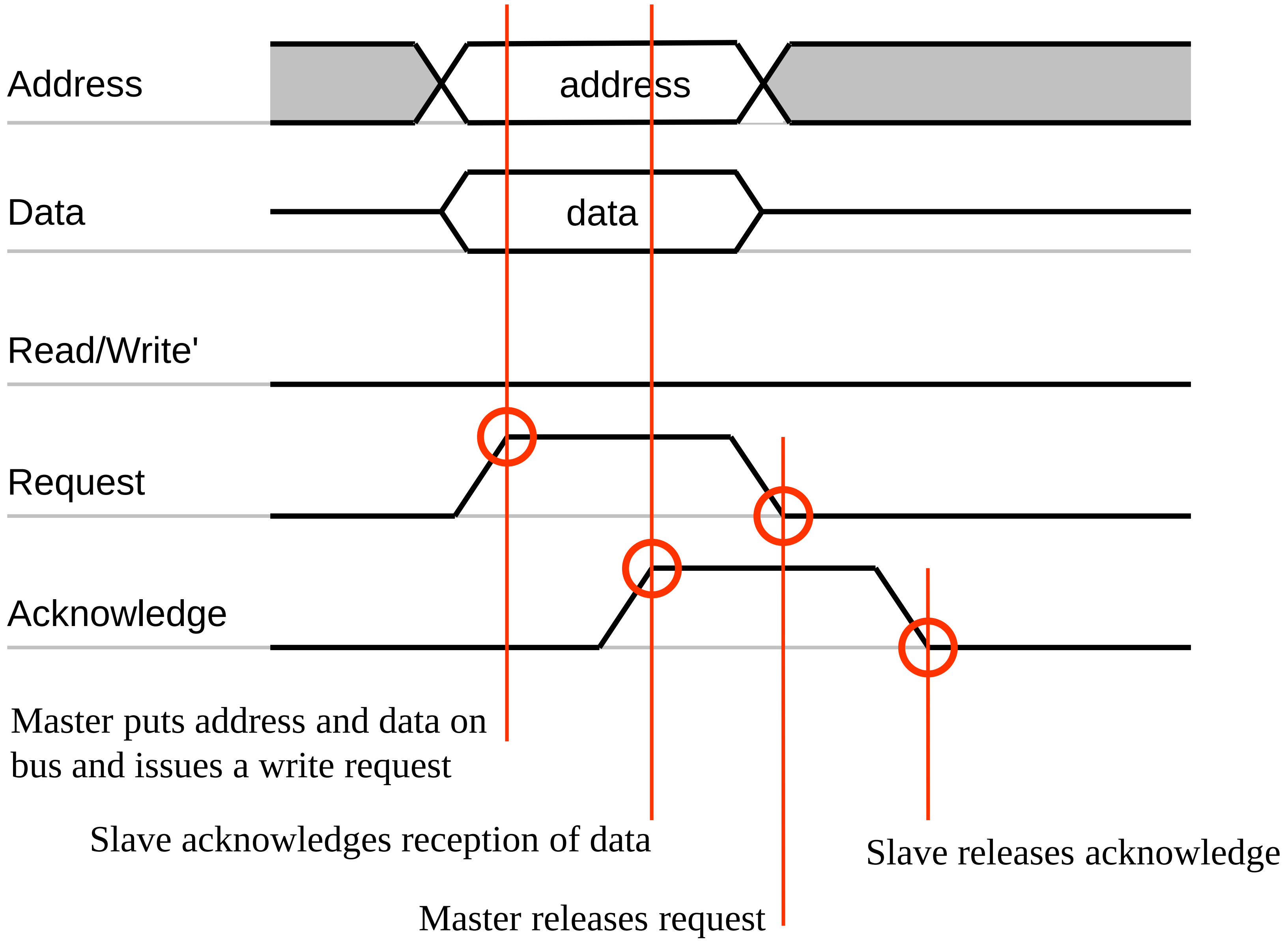
Asynchronous Bus: Four-cycle Handshake

A typical signaling mechanism for asynchronous communication is the four-cycle handshake. It uses two control signals, a *request* signal for the master, and an *acknowledge* signal for the slave. The events are signaled with the rising and falling edges of these signals. The transaction is carried out in four phases. Assuming that both signals are active high, the procedure is as follows:

1. The master issues a request with the rising edge of the request signal. The address and other signals controlled by the master must be stable at this point.
2. The slave processes the request. Once it is ready, it signals that with the rising edge of the acknowledge signal.
3. The master releases the request with the falling edge of the request signal.
4. The slave releases the acknowledge with the falling edge of the acknowledge signal.

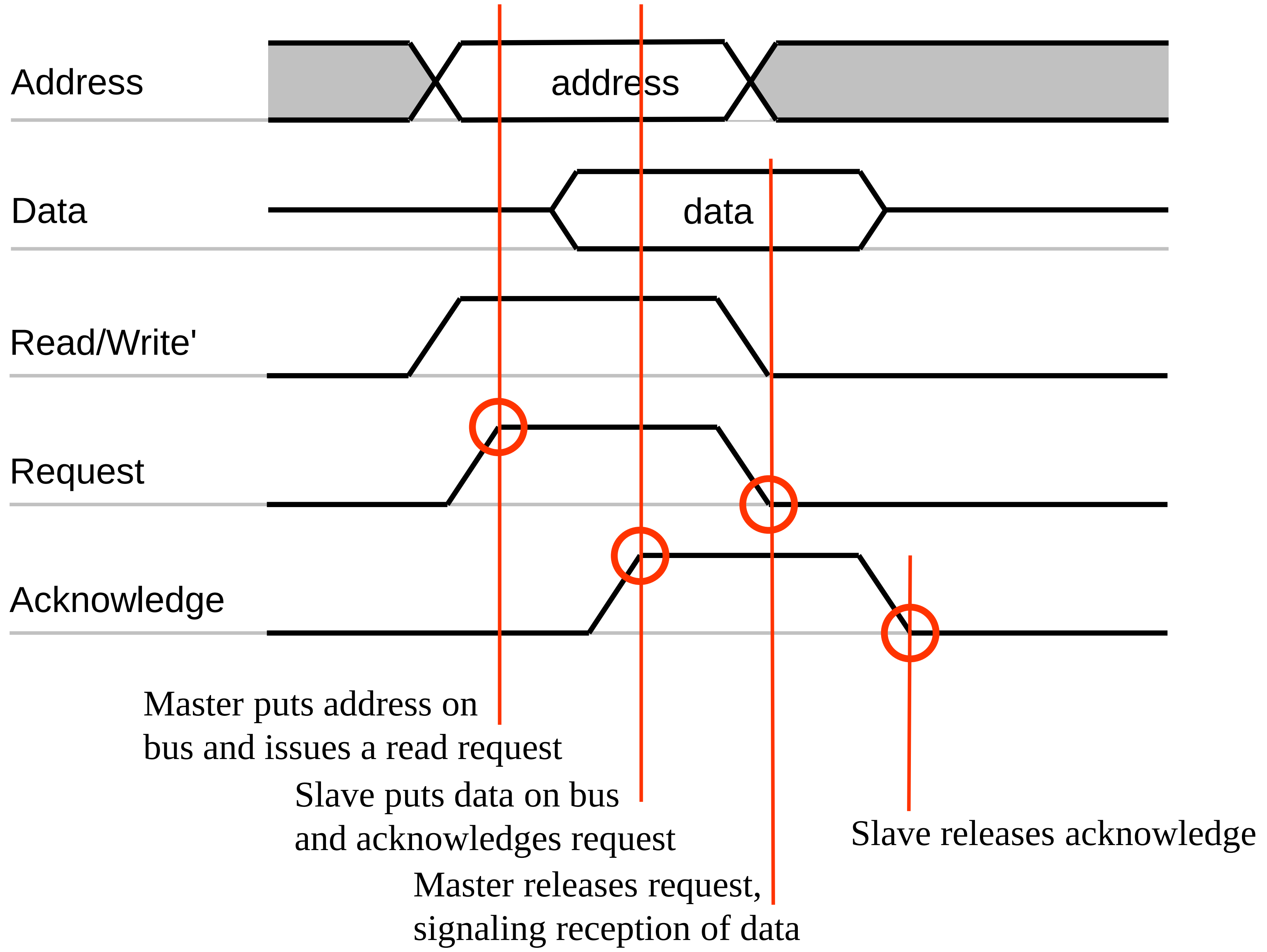
(Asynchronous Bus: Four-cycle Handshake)

Example of an asynchronous bus *write* transaction:


Address
address
Data
data
Read/Write'
Request
Acknowledge
Master puts address and data on bus and issues a write request
Slave acknowledges reception of data
Master releases request
Slave releases acknowledge

(Asynchronous Bus: Four-cycle Handshake)

Example of an asynchronous bus *read* transaction:



Address
address
Data
data
Read/Write'
Request
Acknowledge
Master puts address on bus and issues a read request
Slave puts data on bus and acknowledges request
Master releases request, signaling reception of data
Slave releases acknowledge

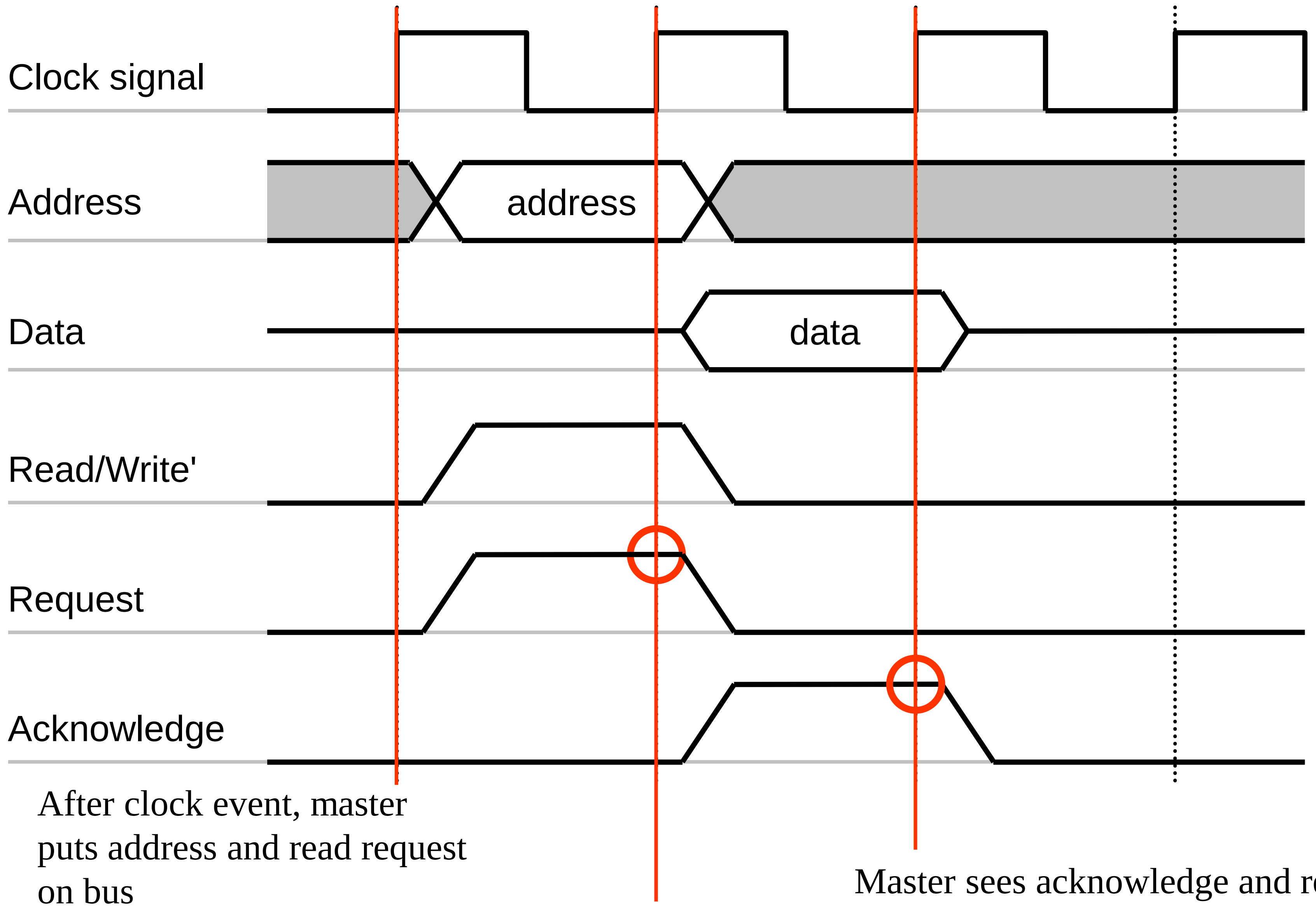
Synchronous Bus

In a synchronous bus, synchronization is not done using handshaking. Instead, the time points for "handing over" address and data between the communication partners are defined by the rising or falling edge of the clock signal. A protocol may even use both edges for synchronization.

In the following, we consider an (academic) example of a synchronous bus protocol. A read or write transaction in this protocol has two phases: an *address phase* and a *data phase*. The data is expected one clock cycle after the address, allowing the slave an additional clock cycle to access the data location.

(The example is quite similar to ARM's AMBA AHB bus.)

Synchronous Bus: Example: Read Transaction

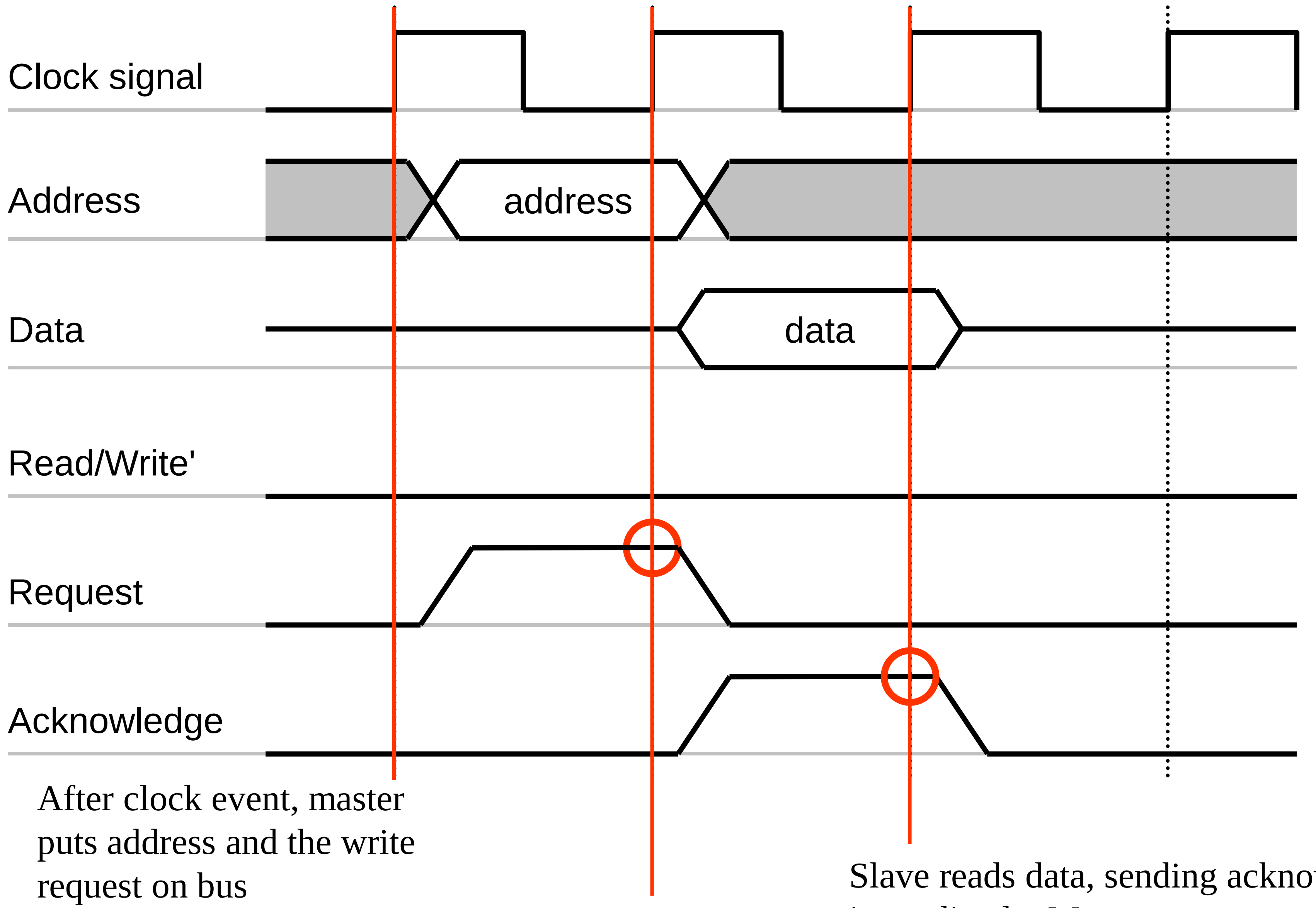


After clock event, master puts address and read request on bus

Slave receives read request, puts data on bus and sends acknowledge

Master sees acknowledge and reads data.

Synchronous Bus: Example: Write Transaction



After clock event, master puts address and the write request on bus

Slave receives write request and address. Master sends data.

Slave reads data, sending acknowledge immediately. Master may start next transaction.

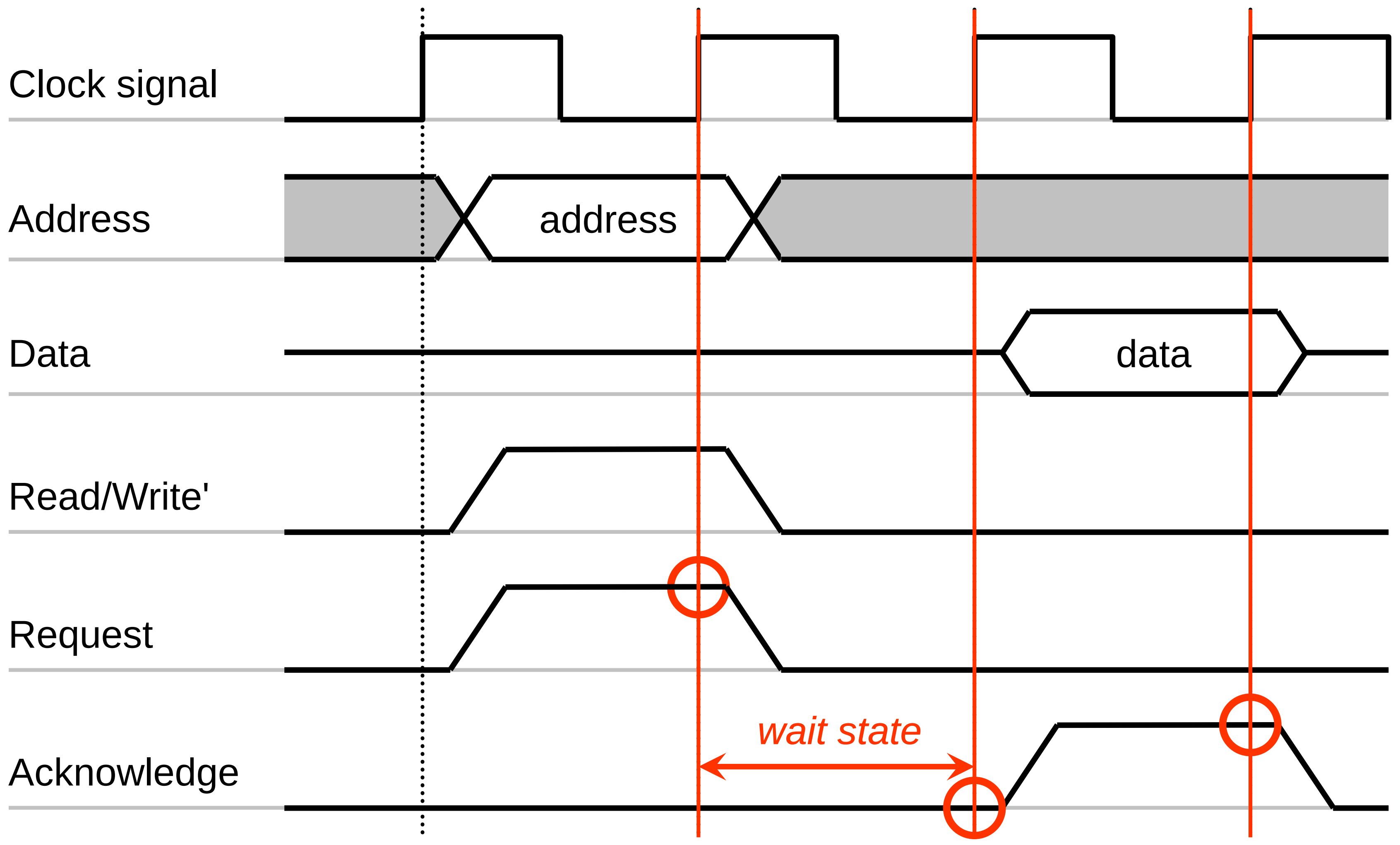
Synchronous Bus: Wait States

Some slaves may not be fast enough to respond to a master request within one clock cycle (for example, if the slave is a slow memory device). This problem can be solved by wait states:

The slave device signals to the master that it is still processing the request by simply not sending an acknowledge until it has its data ready. Every cycle after a master request with de-asserted acknowledge signal is a wait state.

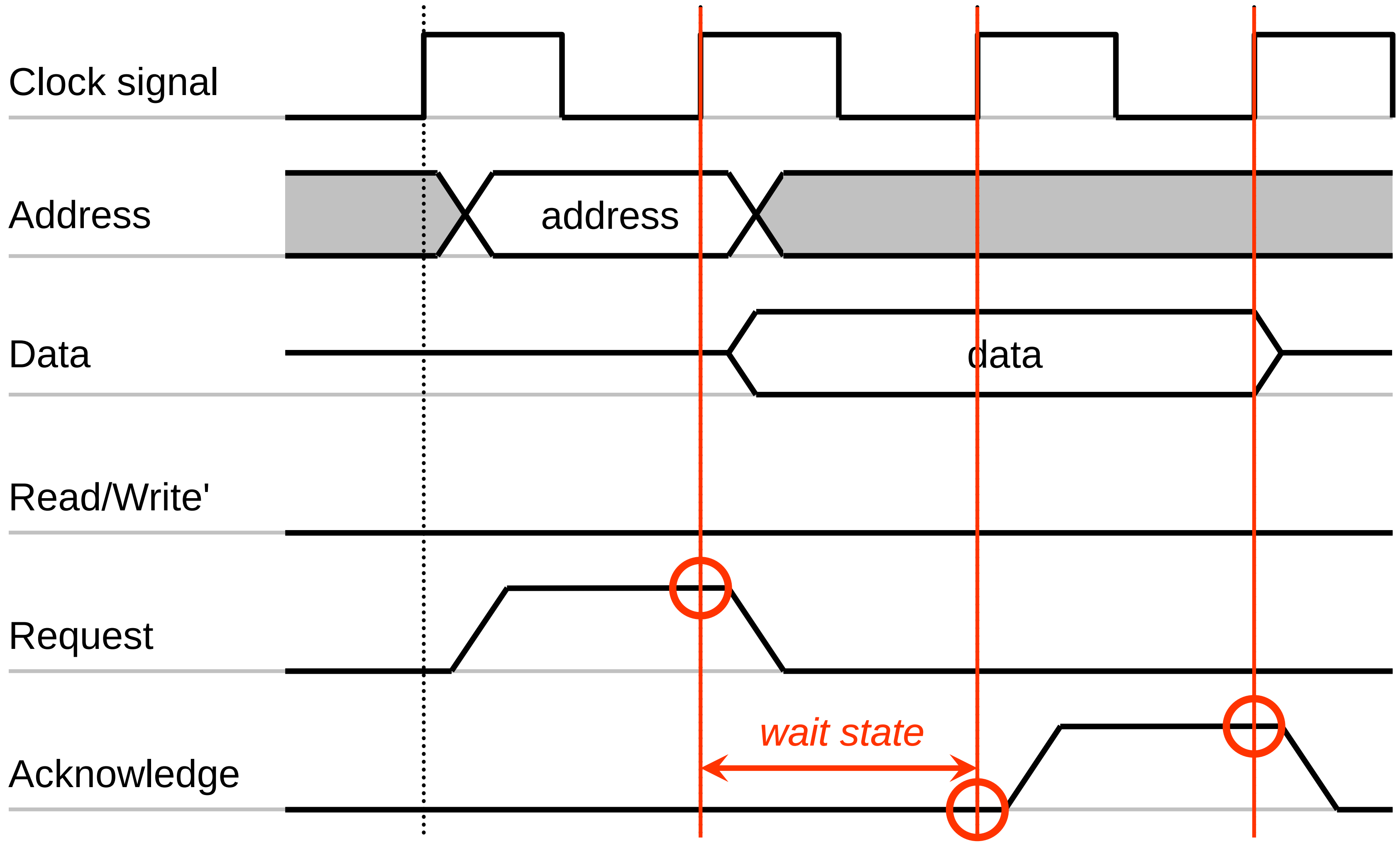
Synchronous Bus: Example: Wait State

Example: Bus *read* transaction with wait state



(Synchronous Bus: Example: Wait State)

Example: Bus *write* transaction with wait state

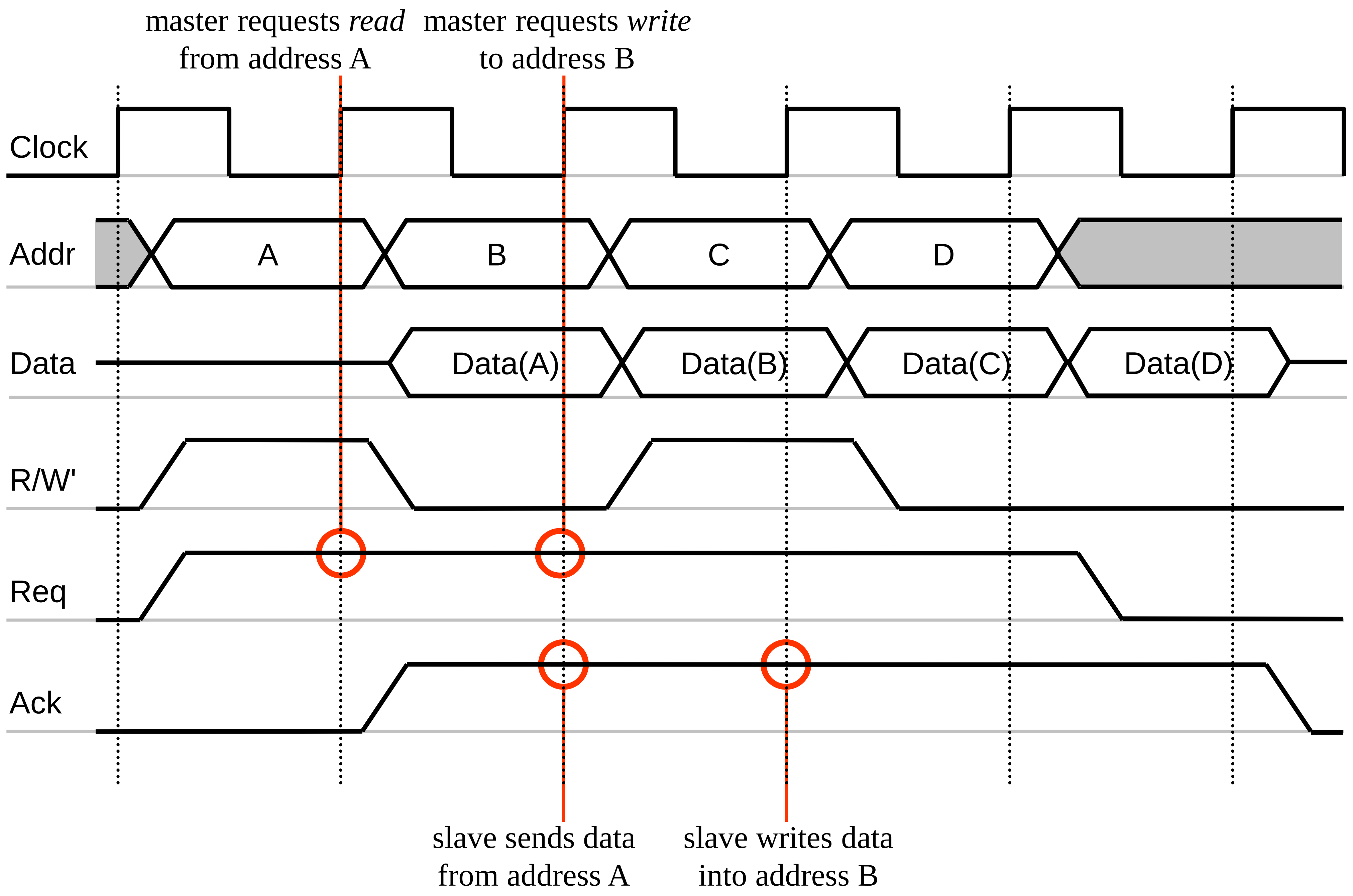


Pipelining a Synchronous Bus

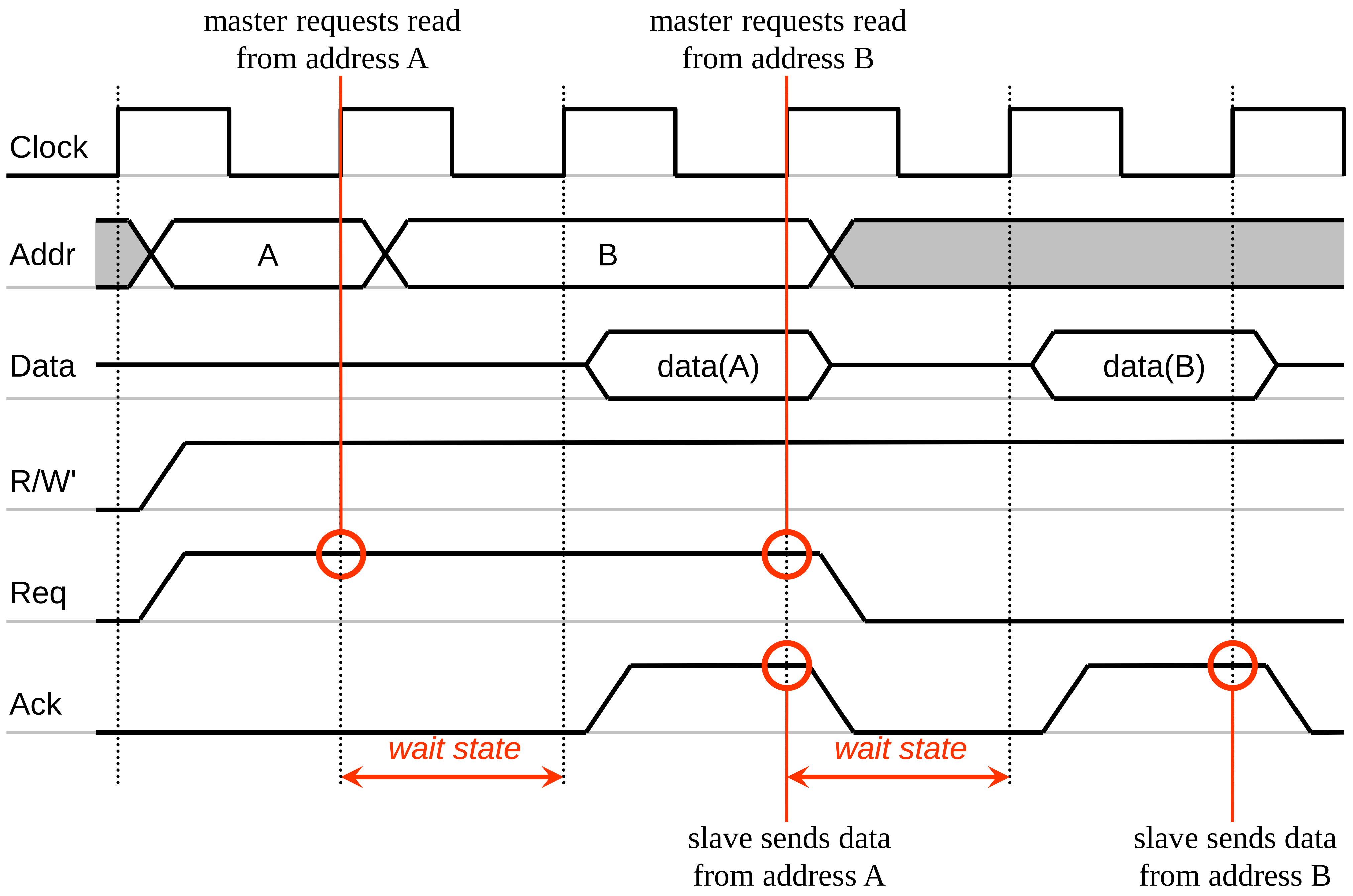
If a synchronous bus has separate lines for data and address (i.e., a “split bus”), the address of a new transaction may already be issued when the data of the current transaction is being transmitted.

Consider a read transaction in our example bus protocol:
The address sent by the master is valid only when the request signal is asserted. When the slave finally sends the data (possibly after one or more wait states), the value of the address lines at that time is irrelevant. The slave needs to remember (latch) the requested address until it is able to service the request.
Idea: Overlap address phase and data phase of two transactions – *pipeline the bus transactions!*
This increases “duty cycle” of address and data lines,
i.e., it increases throughput.

Example: Pipelined Bus Transactions (Reads and Writes)


master requests *read* from address A
master requests *write* to address B
Clock
Addr
A
B
C
D
Data
Data(A)
Data(B)
Data(C)
Data(D)
R/W'
Req
Ack
slave sends data from address A
slave writes data into address B

Ex.: Pipelined Bus Transaction with Wait States


master requests read from address A
master requests read from address B
Clock
Addr
A
B
Data
data(A)
data(B)
R/W'
Req
Ack
wait state
wait state
slave sends data from address A
slave sends data from address B

Synchronous Bus: Burst Modes

Slaves may be able to respond more quickly to pipelined master requests if the addresses point to *consecutive locations*.
A bus may have special control lines used to indicate such situations to the slaves: the protocol includes burst modes.

There may be burst modes of *fixed length*, where the number of transmitted data items is known in advance, and there may be burst modes of *variable length*, where the master needs to signal to the slave when data transfer is to stop.

Burst modes can be either incremental or wrapping.
In incremental burst mode, each address is an increment of the data size over the previous address.
Example: An incrementing burst of four 32-bit words starting at 0x0038 will address locations 0x0038, 0x003C, 0x0040, 0x0044.

(Synchronous Bus: Burst Modes)

In wrapping burst mode, addresses are incremented but “wrapped around” at particular data word boundaries. (Example: a wrapping burst of four 32-bit words starting at 0x0038 will address locations 0x0038, 0x003C, 0x0030, 0x0034).

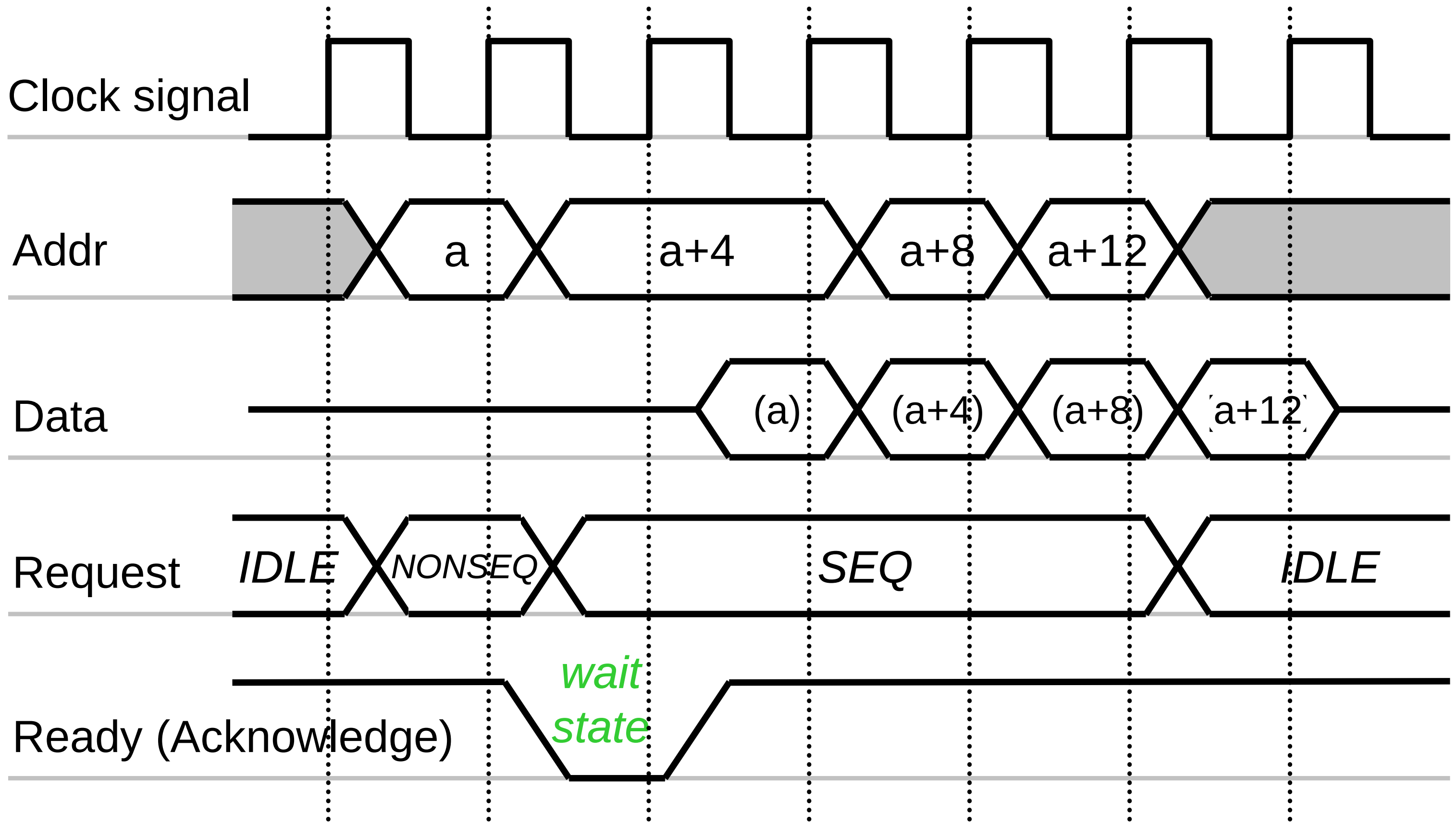
We extend our example of a synchronous pipelined bus with a burst mode:

Instead of a simple master request line, we now have a set of signals encoding the following symbols:

IDLE	no request
NONSEQ	a request to a new address
SEQ	a request to an address in sequence with the previous one

The data size (and address increment) are 4 bytes (32 bits).

Example: Synchronous Pipelined Burst Read



Single-master multiple-slave bus

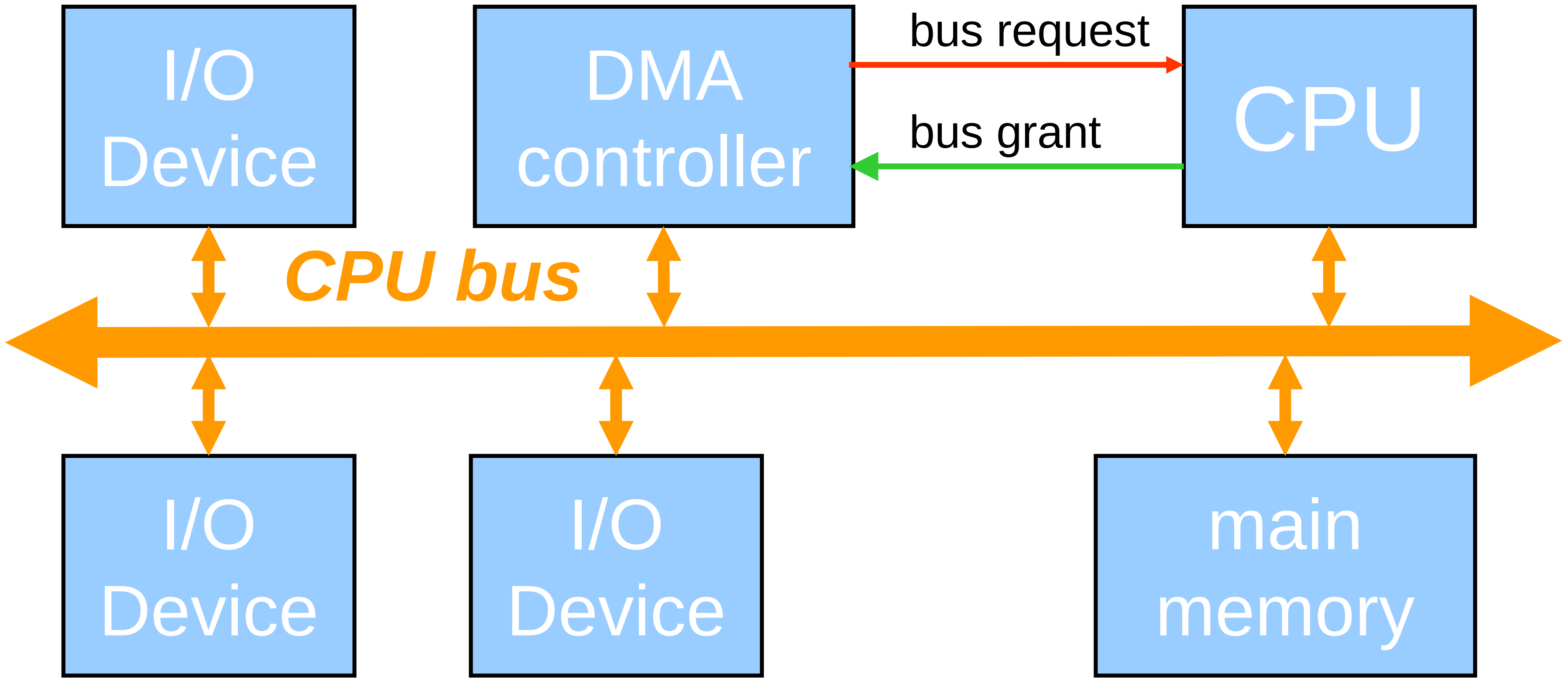
So far, we have considered bus systems with a single master (the CPU) and one or more slaves (memory, I/O devices). All memory transactions on such a bus require the CPU as mediator. This is especially inefficient for data transfers from one slave to another.

> In such a case the CPU needs to repeatedly load data from a device 1 into an internal register and then store it into a device 2. If buffered I/O is used then even a third slave (a memory device) is involved in the process. The overhead can become significant if large blocks of data need to be transferred.

A common solution to this problem is Direct Memory Access (DMA). It allows data transfers that are not controlled by the CPU and do not involve the CPU.

Direct Memory Access (DMA)

Direct Memory Access is controlled by a DMA controller. The DMA controller needs to become a *bus master*. It negotiates with the CPU over bus mastership using (synchronous) four-cycle handshake on two additional bus signals, *bus request* and *bus grant*.



DMA operation

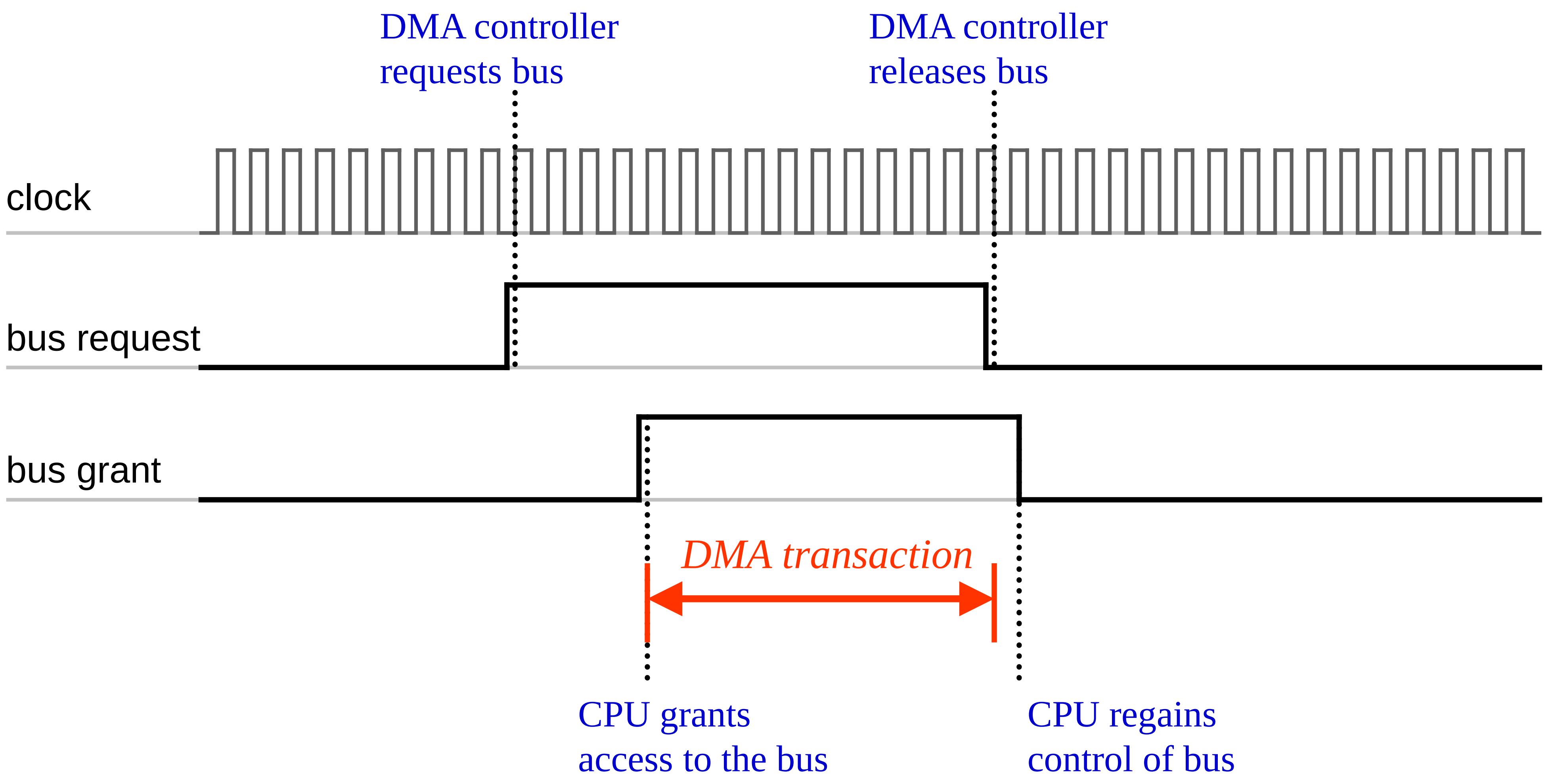
Typically, the CPU initiates the DMA transfer. It writes to the registers in the DMA controller

- the source and destination start addresses where the block transfer is to begin,
- the length of the block (number of words) to transfer,
- control information in the status register to start the transfer.

The DMA controller then takes over the bus until the end of the block transfer. Afterwards, the DMA controller sends the CPU an interrupt to signal that the transfer is complete.

Most DMA controllers do not transfer large blocks in a single operation but split the transfer up in smaller transactions of 4, 8 or 16 words at a time. In between these transactions, control of the bus is returned to the CPU and re-requested after a certain time.

DMA Bus Arbitration Cycle


DMA controller requests bus
DMA controller releases bus
clock
bus request
bus grant
DMA transaction
CPU grants access to the bus
CPU regains control of bus

During the DMA transaction, the DMA controller is master of the bus. The slaves cannot distinguish between CPU and DMA transactions.

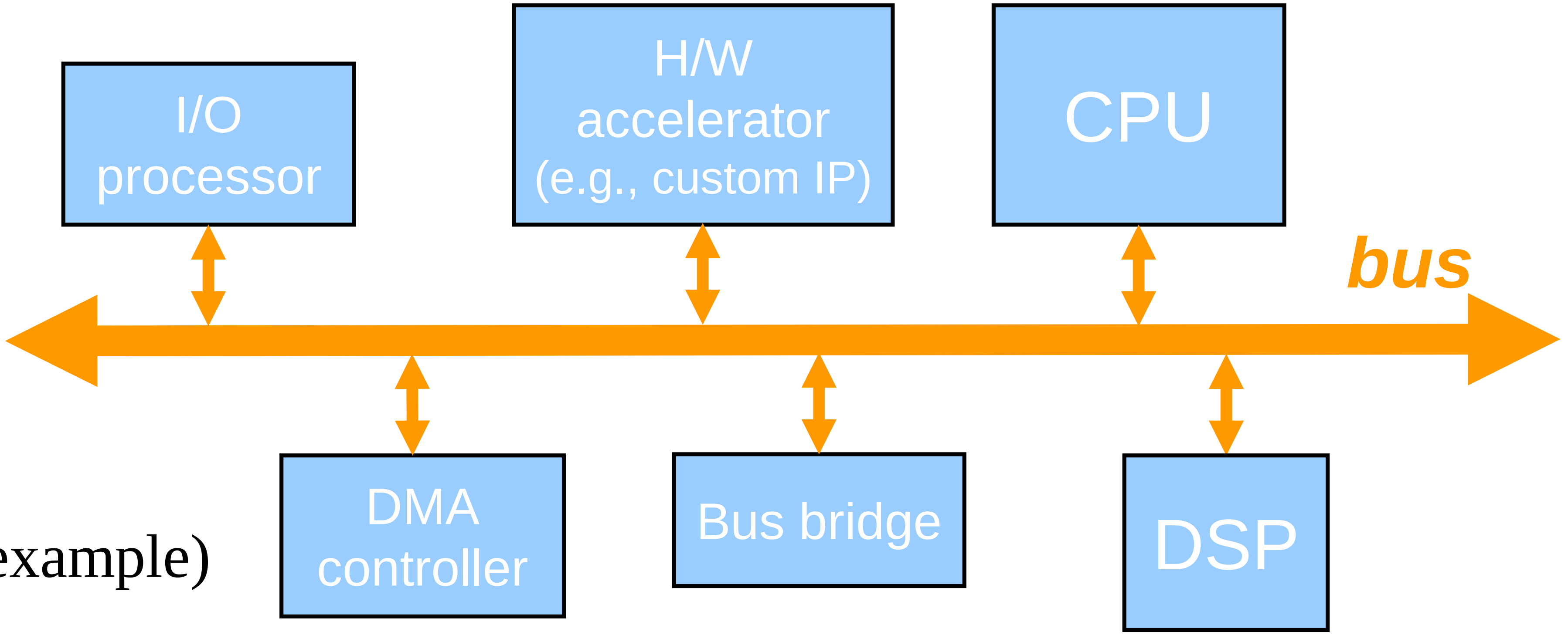
DMA Controllers, I/O Processors

Sophisticated DMA controllers allow to handle several transactions simultaneously. Each transaction is assigned to a DMA channel. DMA transactions can be interleaved: if a device is not ready, the bus can be used for another transaction in the meantime.

Sophisticated systems today use I/O processors which are, basically, programmable DMA controllers. They have an instruction set similar to a CPU and internal memory to store I/O programs. The CPU initiates an I/O transfer by sending the I/O processor a command with arguments (e.g., starting address, size of block to be transferred). The I/O processor executes the command and sends the CPU an interrupt upon completion.

Multi-master Bus

In computer systems, several devices connected to the bus may be required to become bus master.



(example)

Whenever there is more than one possible bus master, access to the bus must be allowed by a bus arbitration scheme.

Bus Arbitration Schemes

Local Arbitration

There is a default bus master (e.g., the CPU). It handles bus requests from other masters.

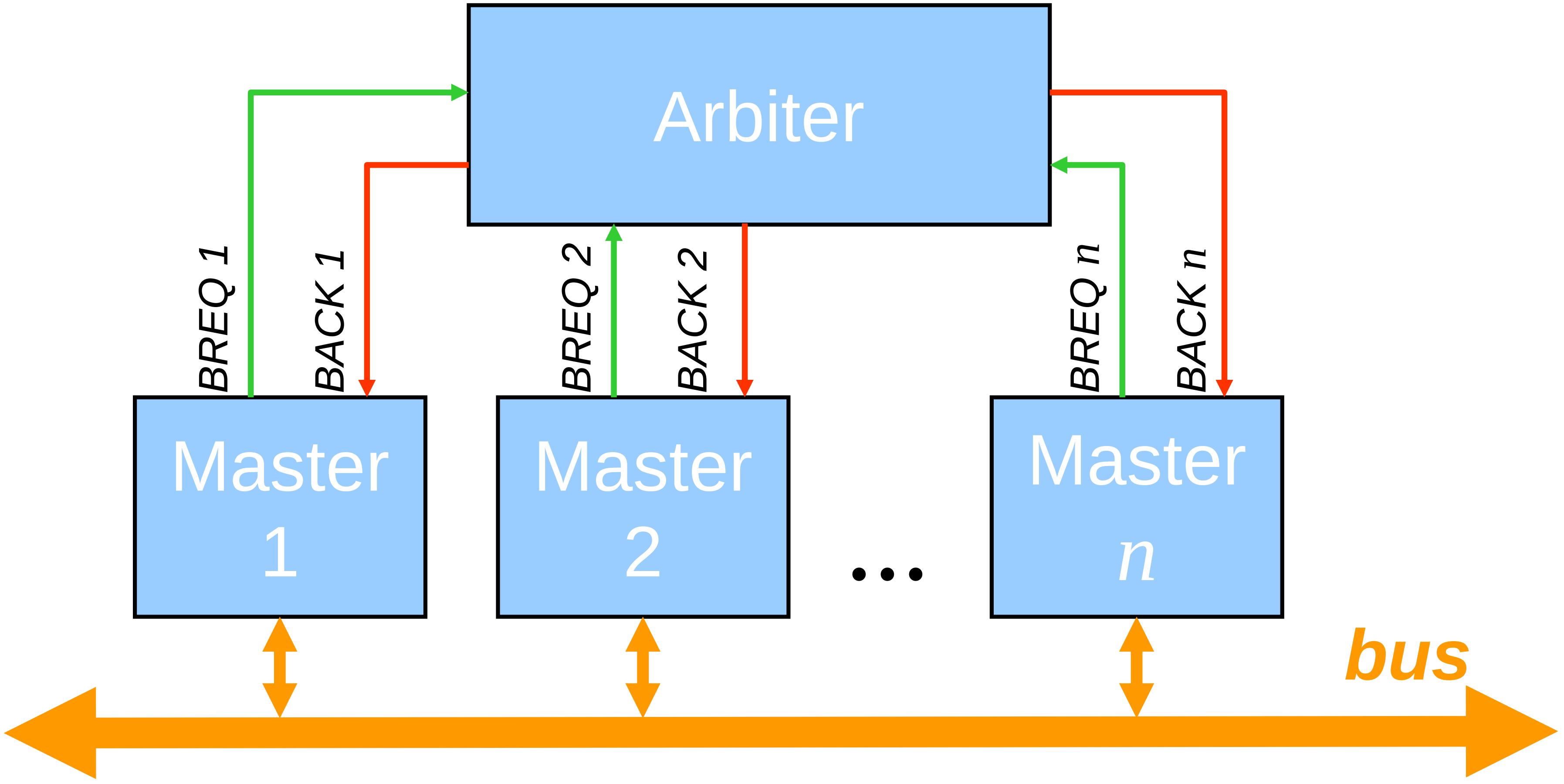
Global Arbitration

There is no default bus master. Control over the bus is granted to a master by the arbitration logic upon request.

The arbitration logic may be centralized in a bus arbiter circuit, or it may be distributed (e.g., the bus interfaces of the masters include some arbitration logic).

Arbitration usually uses prioritization.

Bus Arbiter



Arbitration using a central arbiter allows flexible arbitration schemes to be implemented. However, it incurs substantial wiring and hardware cost.
Example busses using arbiters: PCI, AMBA AHB.

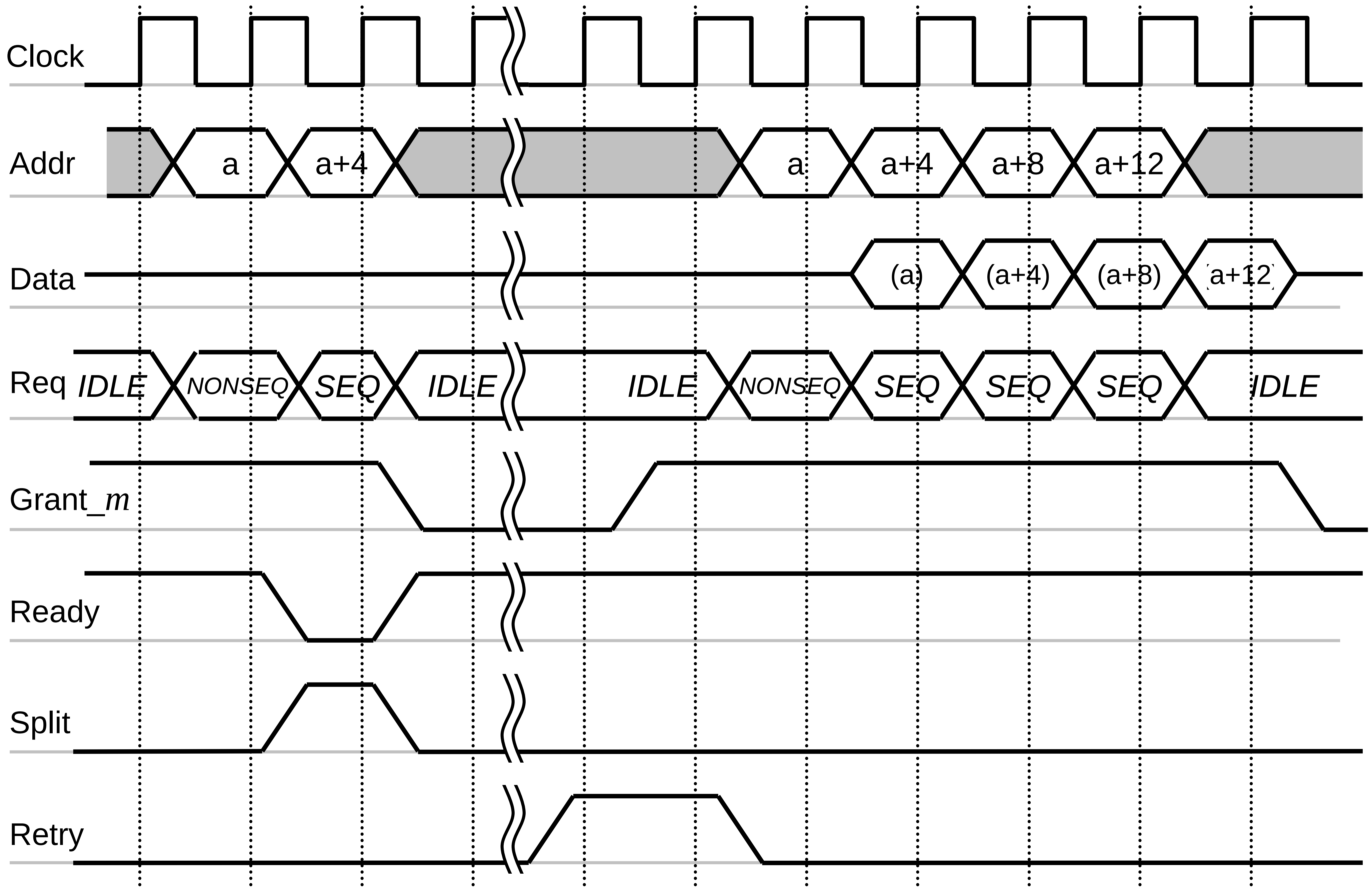
Optimizing Bus Performance – Split Transactions

In a multi-master bus, a slow transaction between a master and a slave may stall other masters waiting to gain access to the bus. While a slave is still busy fetching the data to answer a request, the bus could be used by other masters.

Some multi-master bus protocols support split transactions. The master requests a transaction as usual. If a slave is not able to respond immediately, instead of answering with a number of wait states the slave sends a *SPLIT* over a separate signal. The bus is free again for other masters' transactions. When, finally, the slave is ready to answer the request, the slave sends a *RETRY* and the master repeats its request. This time, the transaction completes.

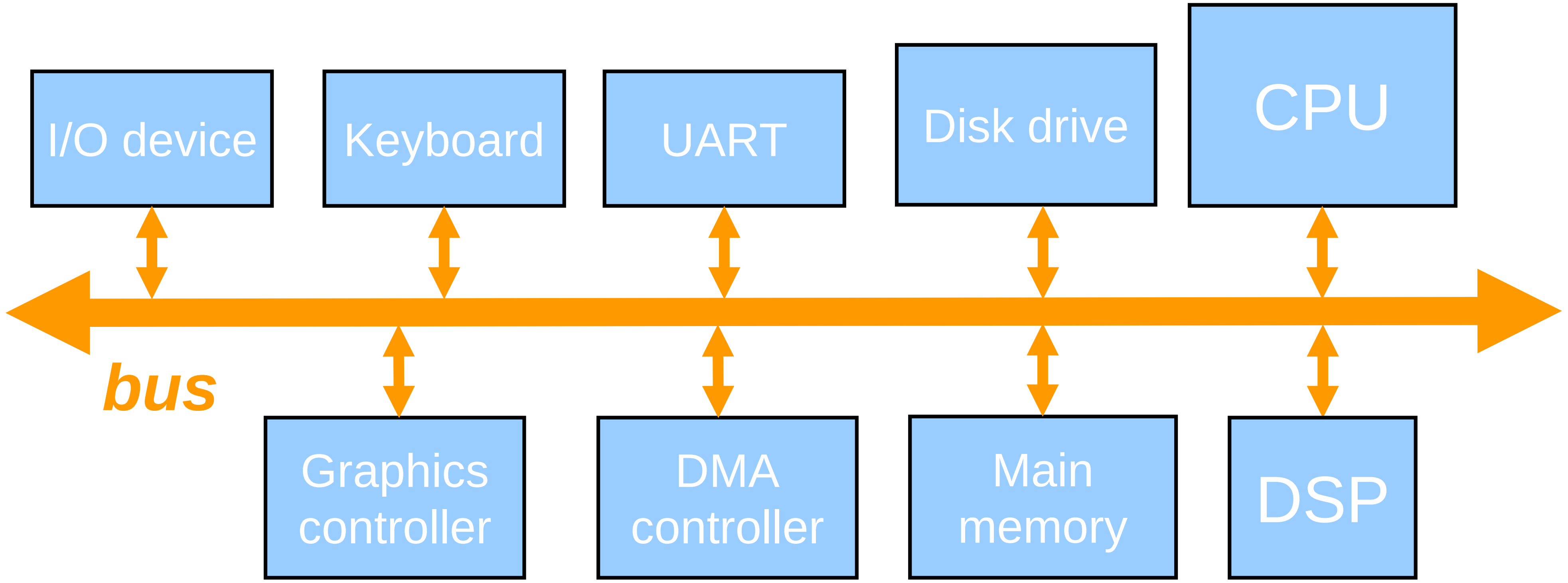
The arbitration scheme must be designed to incorporate handling of split transactions.

Split Transaction – Example



System Bus Configurations

Consider a system with many masters and slaves connected to a high-speed synchronous bus. All devices need to run at the same clock rate. Even low-speed devices need a high-speed bus interface. This is inefficient and expensive.

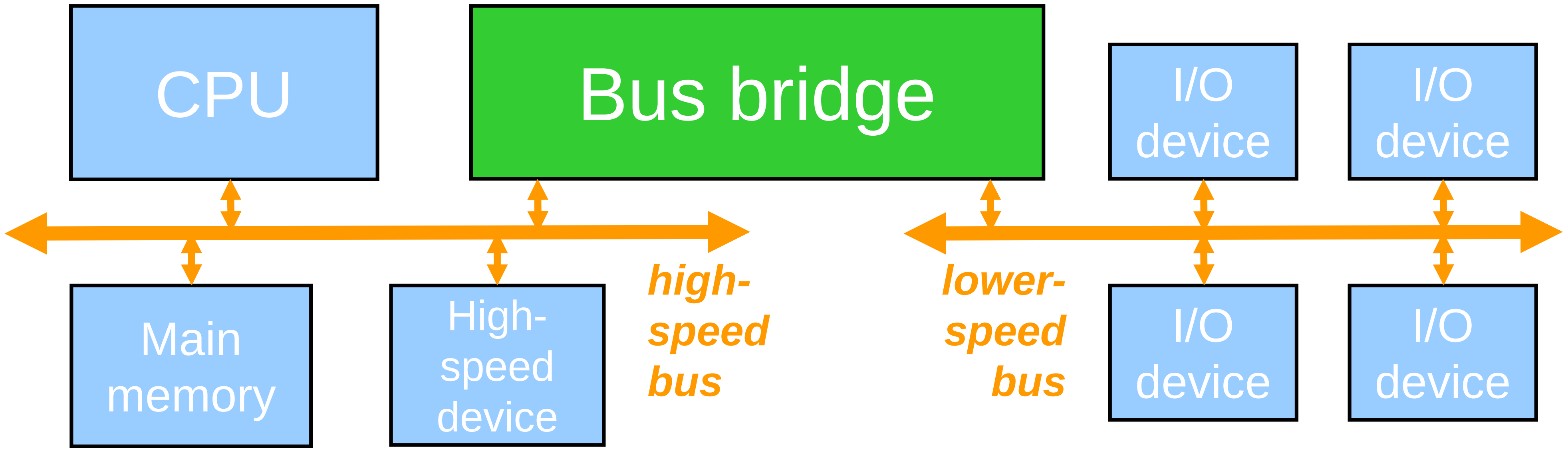


(Example)

Bus Bridge

The CPU needs to communicate at high speed at least with main memory. For a high speed bus, the wires must not be too long and the number of devices needs to be small.

Therefore, I/O devices with lower performance requirements are connected to a separate lower-speed bus. The connection between the high-speed CPU bus and the lower-speed bus is made using a special device called bus bridge.



High-Speed Bus vs. Low-Speed Bus

Typical characteristics of a high-speed bus are

- wide data connections
- separate data lines for reading and writing (no tri-state drivers necessary)
- (often) synchronous operation
- high clock rate

The high performance involves higher implementation cost and power consumption.

Typical characteristics of a low-speed bus are

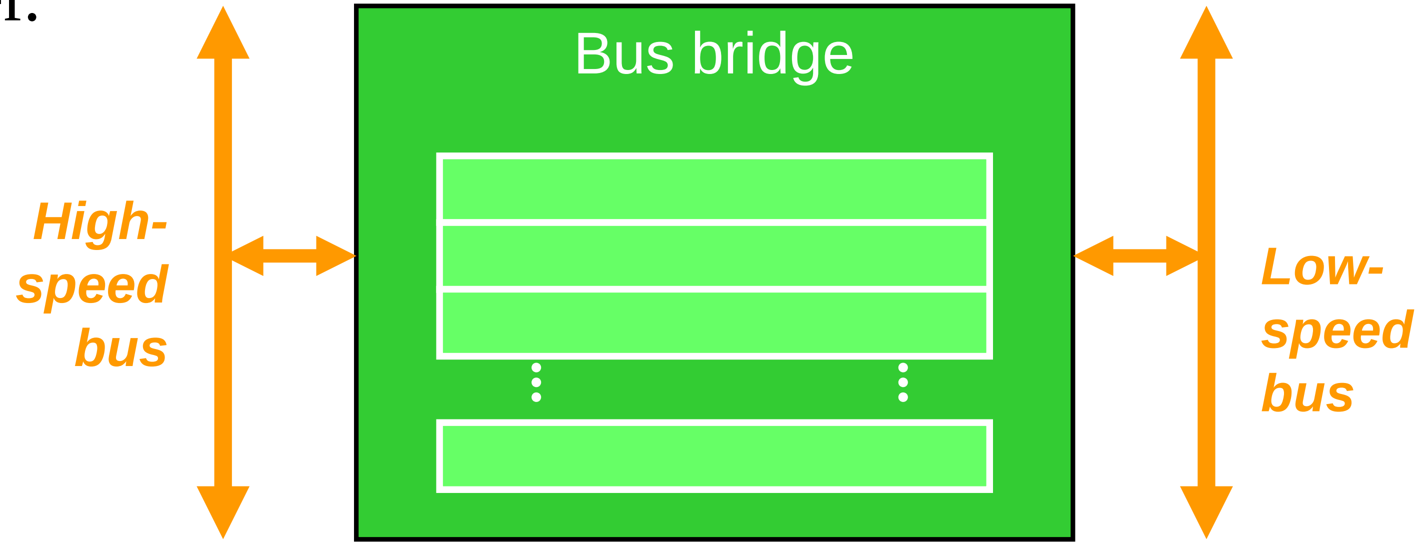
- bidirectional data lines (i.e., tri-state bus drivers)
- sometimes multiplexed address and data lines
- lower clock rate
- (often) asynchronous operation
- no pipelining

Bus Bridge Operation

The bridge translates between the protocol on the high-speed side and the protocol on the low-speed side. It acts as a slave on the fast bus and as a master on the slow bus.

Write operation

In a write operation, the bridge can store address and data in an internal buffer and immediately acknowledge the write to the master.



When the buffer is full, the bridge signals a *SPLIT* transaction.

(Bus Bridge Operation)

Read operation

In a read operation, the bridge signals a *SPLIT* transaction on the fast bus. As soon as the data has been read from the slow bus the transaction on the fast bus can be resumed and completed.

Burst transactions

A bridge can be designed to use the buffer for burst transactions. In a burst write, the buffer is filled from the fast bus and then sent out in individual writes on the slow bus. Similarly, a burst read transaction can be used to transfer the complete buffer content over the fast bus.

Practical bus bridges are often highly sophisticated, and are sometimes integrated together with DMA and interrupt controllers.

Bus Hierarchies

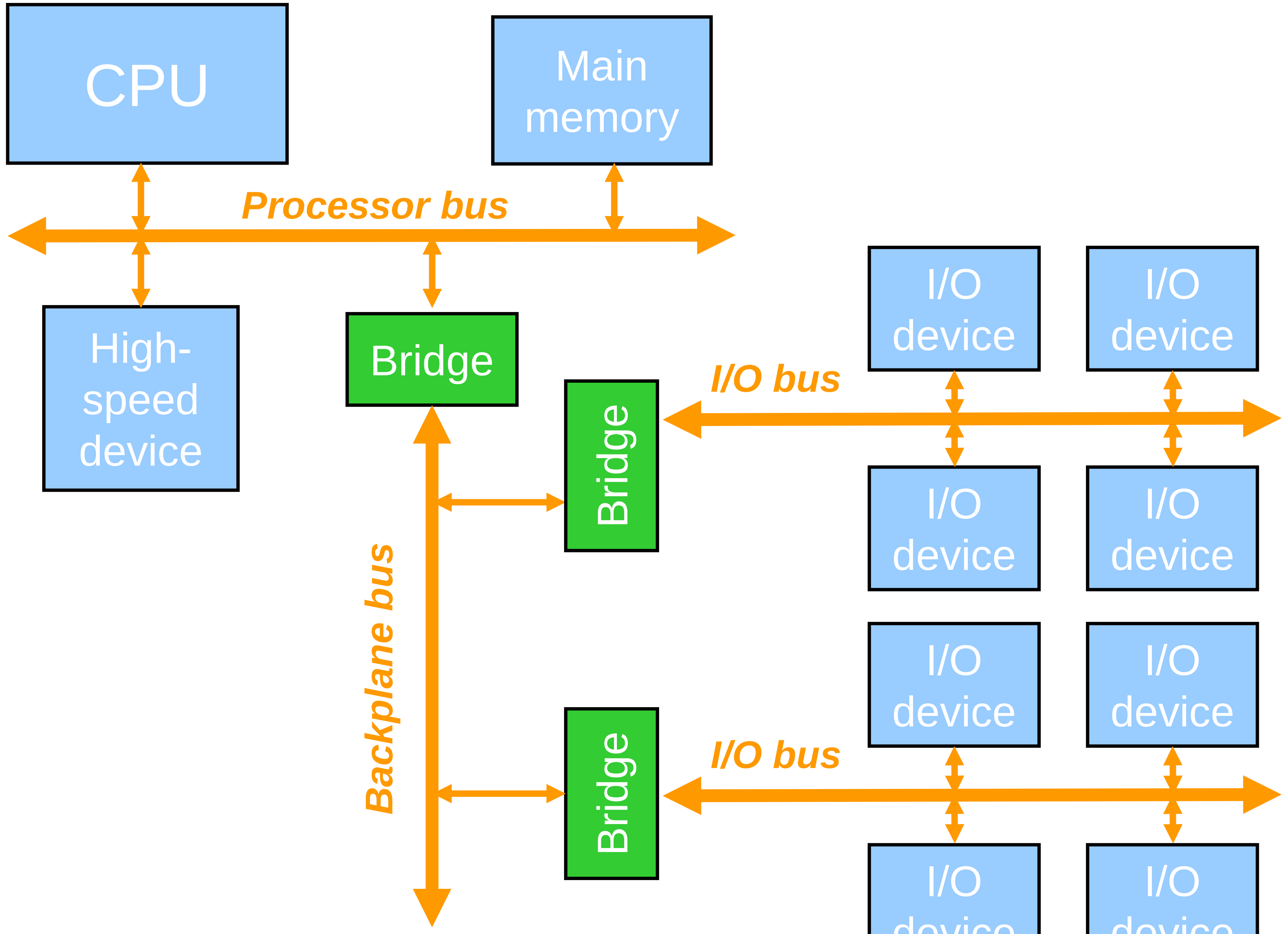
In practice, bridges can be used to build a hierarchy of busses.

Processor busses are used to connect the CPU with memory. The protocols are often proprietary and architecture-dependent.

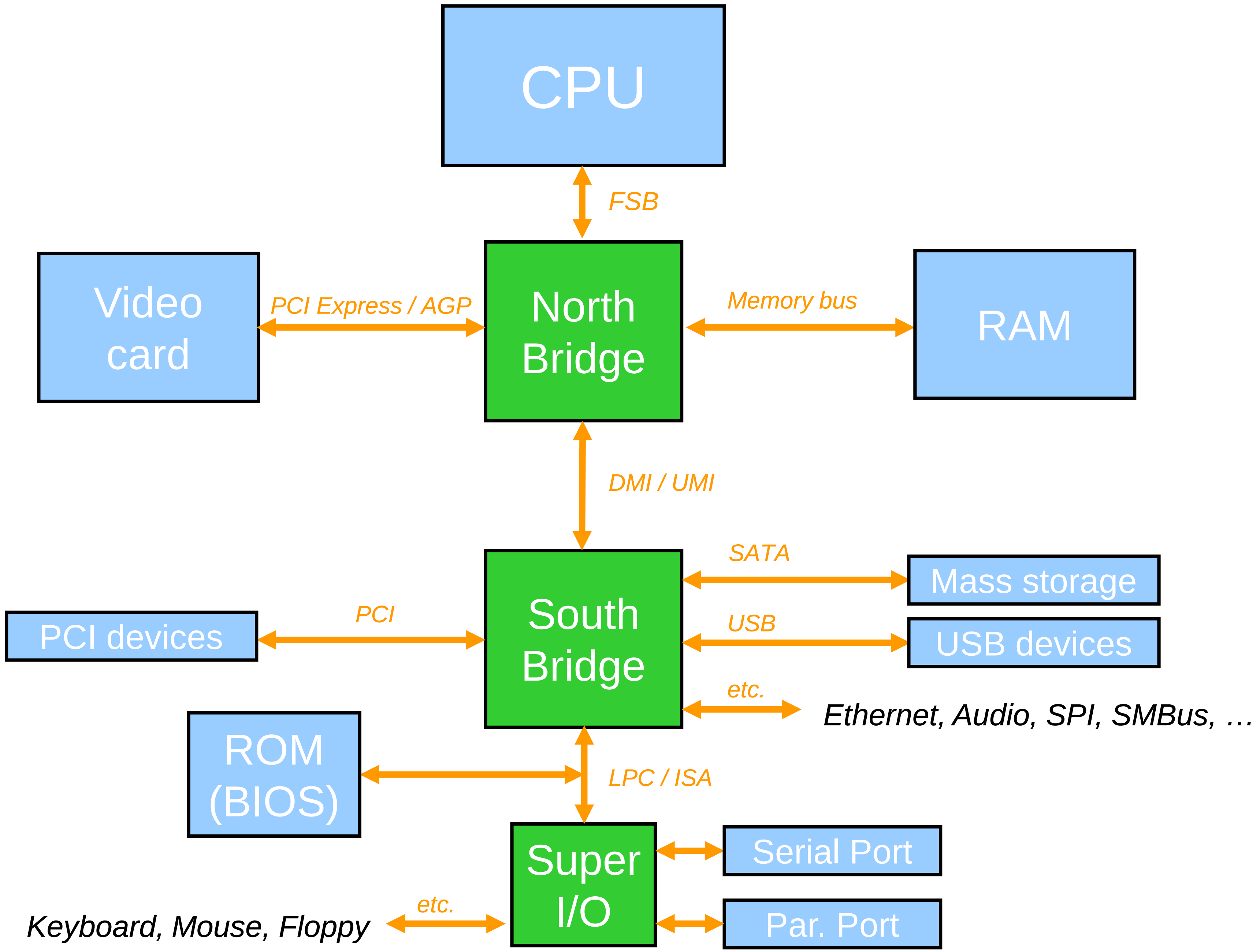
I/O busses or Peripheral Busses connect I/O devices to the system. There is a variety of standard protocols used, e.g., USB, SCSI, IEEE 1394.

Backplane Busses are used in computer systems as intermediary busses to connect I/O busses with the processor bus. Example: PCI.

Bus Hierarchy: Example



Classical PC architecture (Example)



Memory Technology

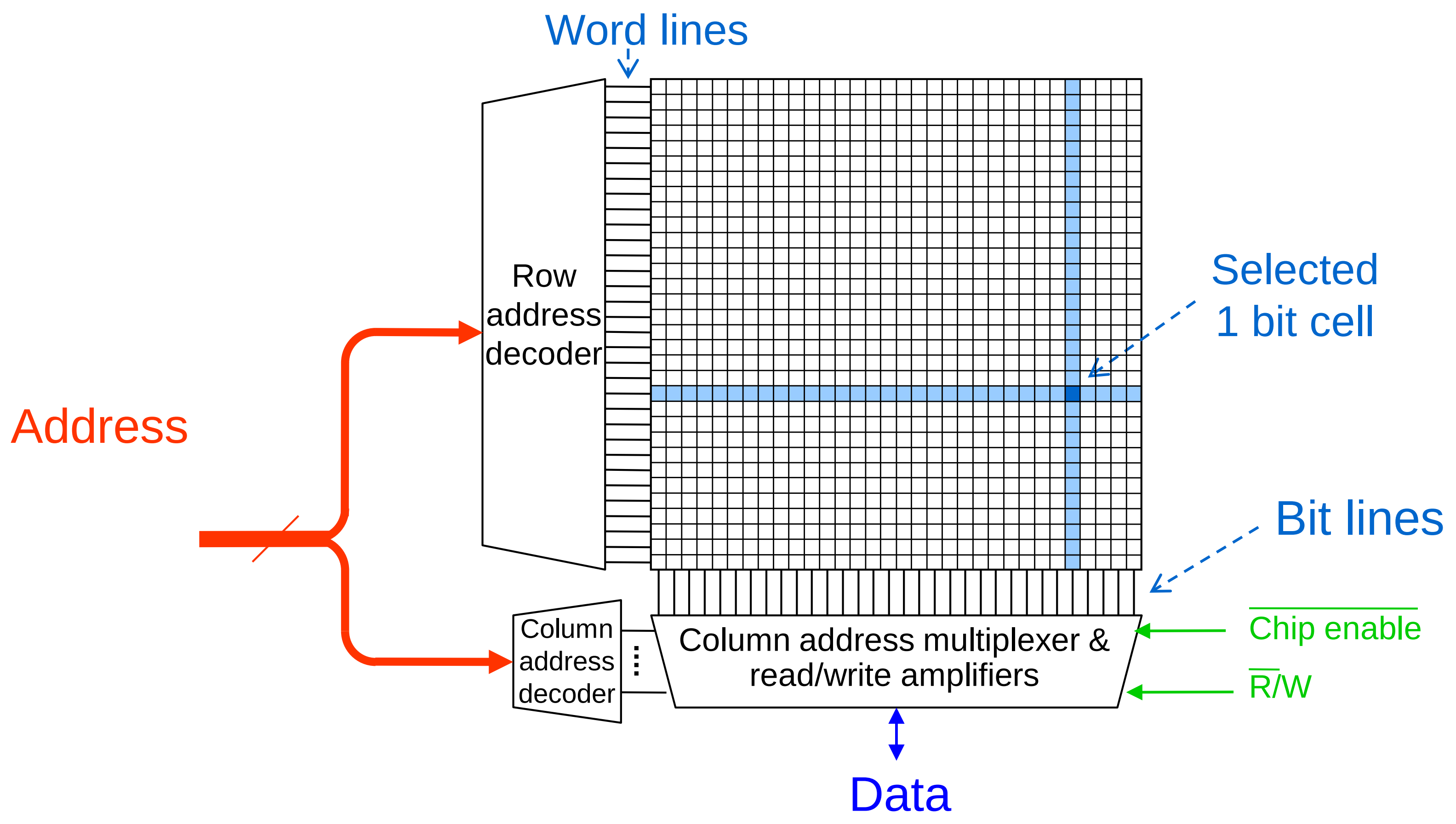
In the remainder of this chapter we discuss the basic types of random access memory technology found in computer systems.

Computer memory is organized in arrays of memory words. A word is a collection of memory cells, each storing 1 bit of information.
Each word in memory has a unique address.

Random Access Memory (RAM) is called *Random* Access Memory because the access time for a specific memory location is independent of the address – locations can be accessed in an arbitrary, random order.

RAM devices are organized in two-dimensional arrays. The address of a cell can be decomposed into a row address (e.g., most significant bits) and a column address (remaining least significant bits).

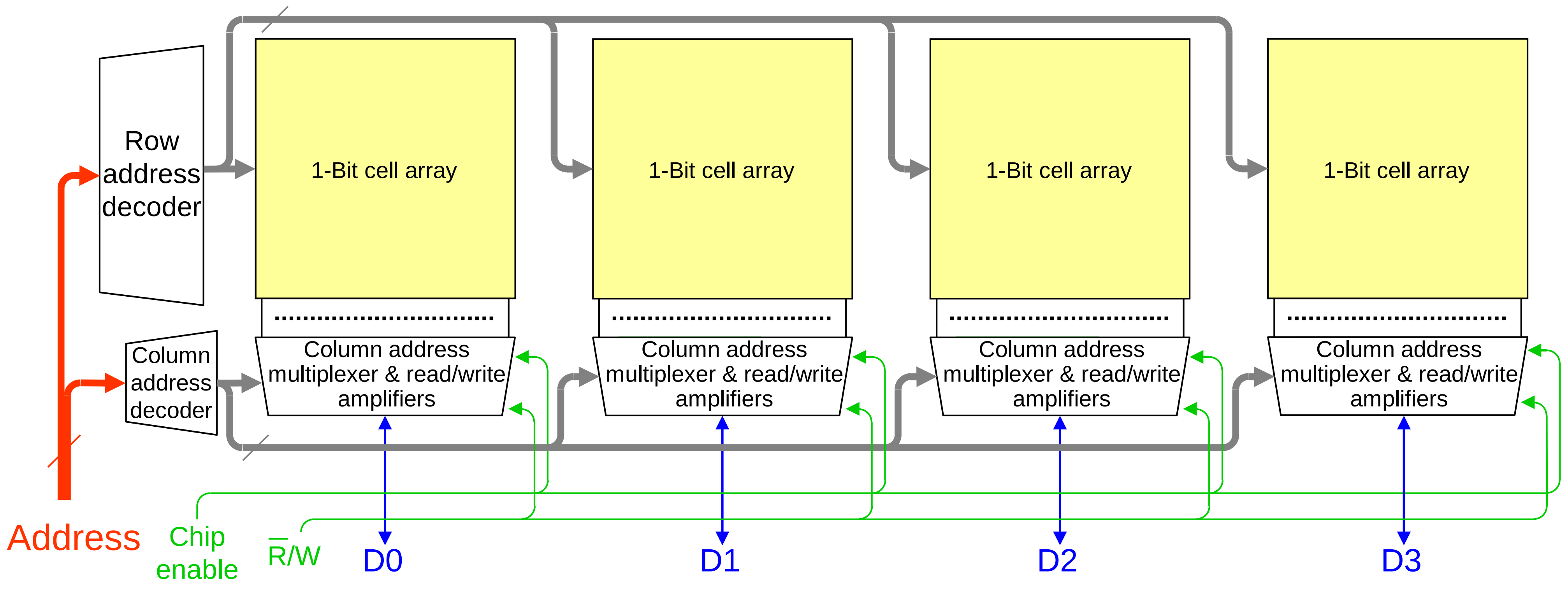
Memory Organization – Example



A row is activated by a word line.
A column is activated by a bit line.
The data port is bidirectional
– its signal flow direction is controlled by the R/$\overline{\text{W}}$ signal.

(Memory Organization)

Memory chips are organized in data words of various sizes.
Example: memory device with 4-bit words:


Row address decoder
1-Bit cell array
1-Bit cell array
1-Bit cell array
1-Bit cell array
Column address decoder
Column address multiplexer & read/write amplifiers
Column address multiplexer & read/write amplifiers
Column address multiplexer & read/write amplifiers
Column address multiplexer & read/write amplifiers
Address
Chip enable
$\overline{R}/W$
D0
D1
D2
D3

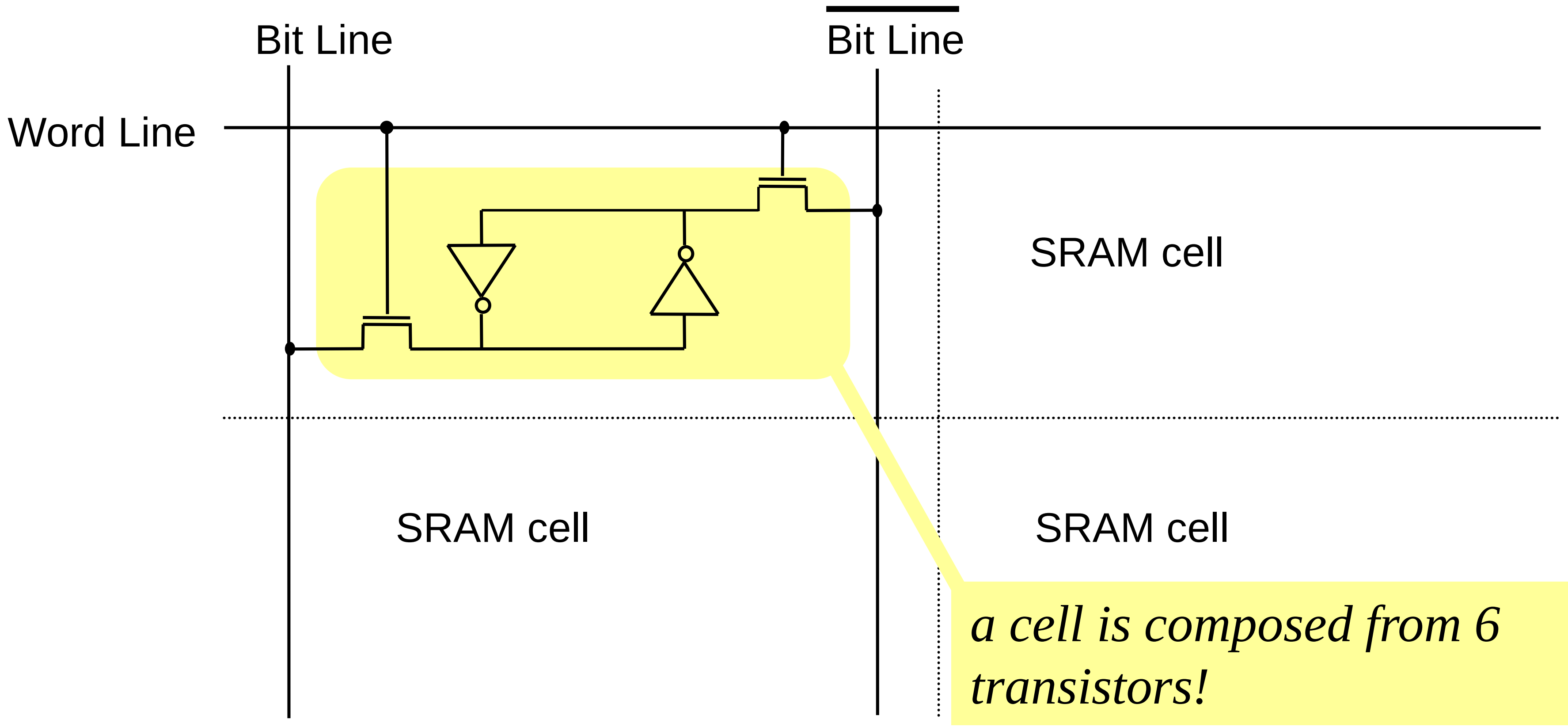
RAM Technology Basics – SRAM and DRAM

The basic storage element for binary information is the flip-flop or latch. It can be used as the basic storage cell in a memory array. Flip-flops keep the information until they are explicitly set or reset. Random-Access Memory constructed using flip-flops is called Static RAM (SRAM). An SRAM cell is constructed from 6 transistors.

Information can also be stored using less hardware: with only a capacitor and a transistor. The capacitor has two distinguishable states: *charged* or *uncharged*. The capacitor is charged and discharged through the transistor. Reading the information destroys it – the information must be rewritten after a read. More severely, the capacitor looses its charge over time. The information needs to be refreshed periodically (e.g., every 64ms). This type of memory is called Dynamic RAM (DRAM).

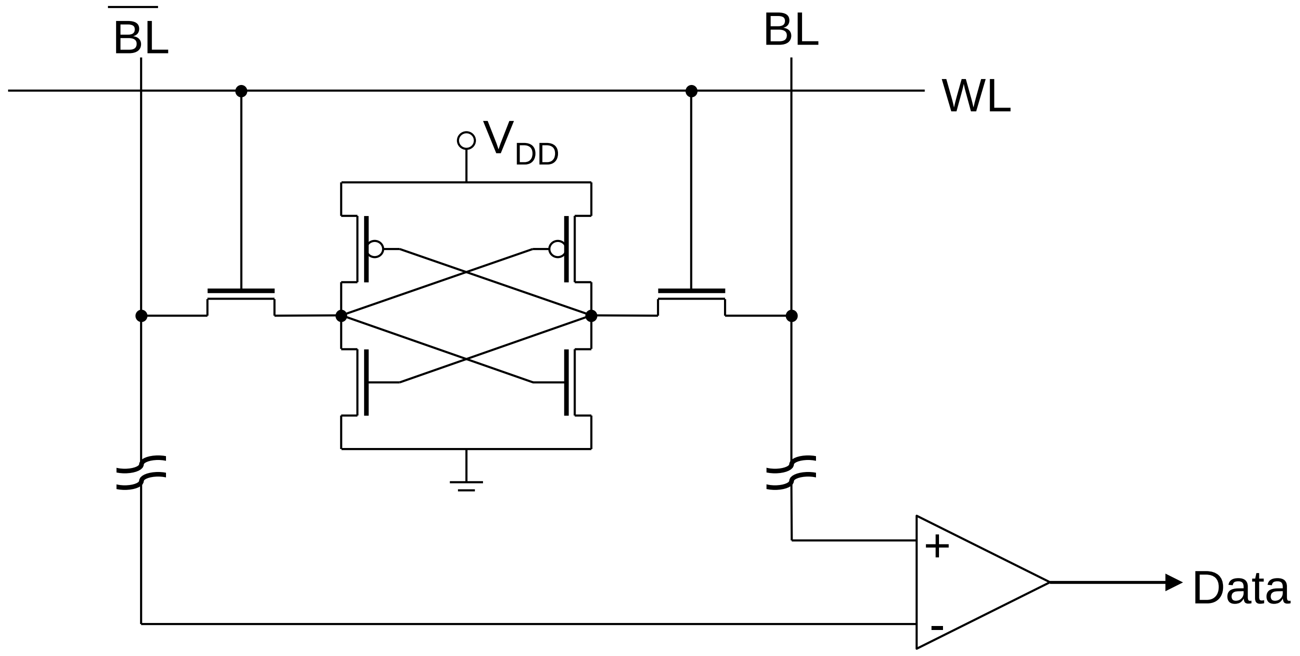
Static RAM (SRAM)

In an SRAM cell the information is stored in a ring of inverters. Two pass transistors connect the inverters with the bit lines.



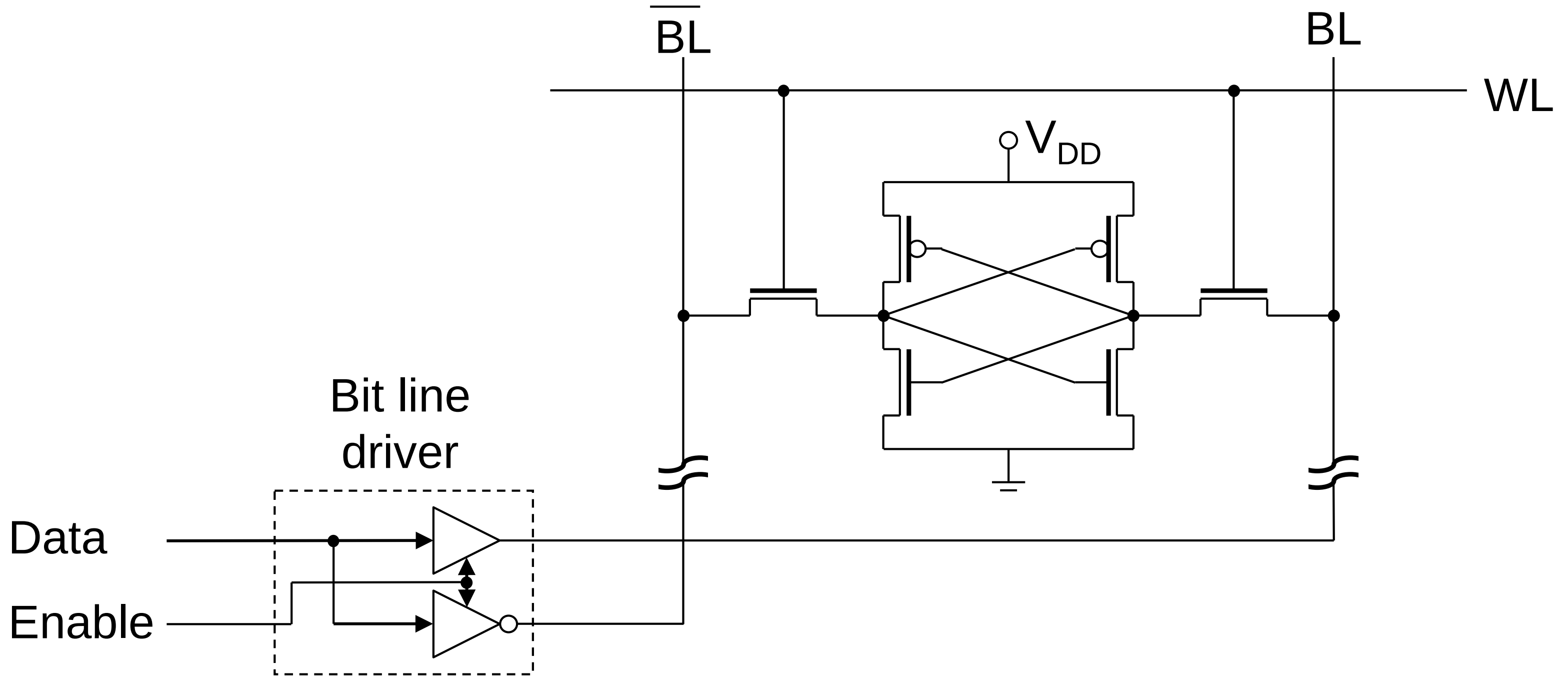
a cell is composed from 6 transistors!

SRAM – Read Operation



When reading the memory cell, the word line (row) is activated and the two inverter outputs are connected to a read amplifier to produce the data output.

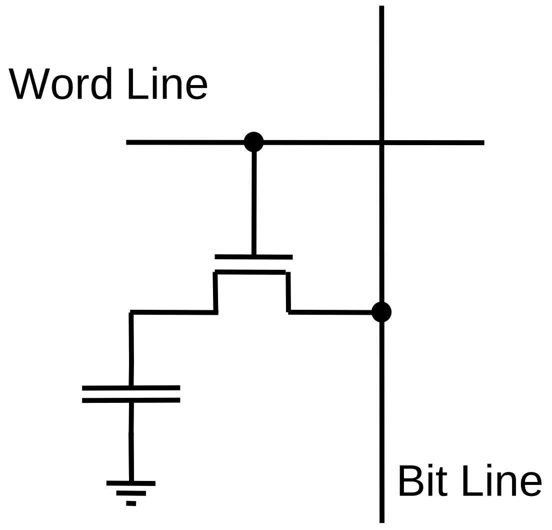
SRAM – Write Operation



When writing into the memory cell, the word line (row) is activated and the bit lines are forced to opposite values by the bit line drivers.

Dynamic RAM (DRAM)

In a DRAM cell the information is stored in a capacitor.
A pass transistor, activated by the word line, connects the cell with the bit line.



A DRAM cell requires much less chip area than an SRAM cell.
DRAM memory can be built with significantly higher density than SRAM memory.

DRAM organization

Reading a DRAM cell is a two-phase process:

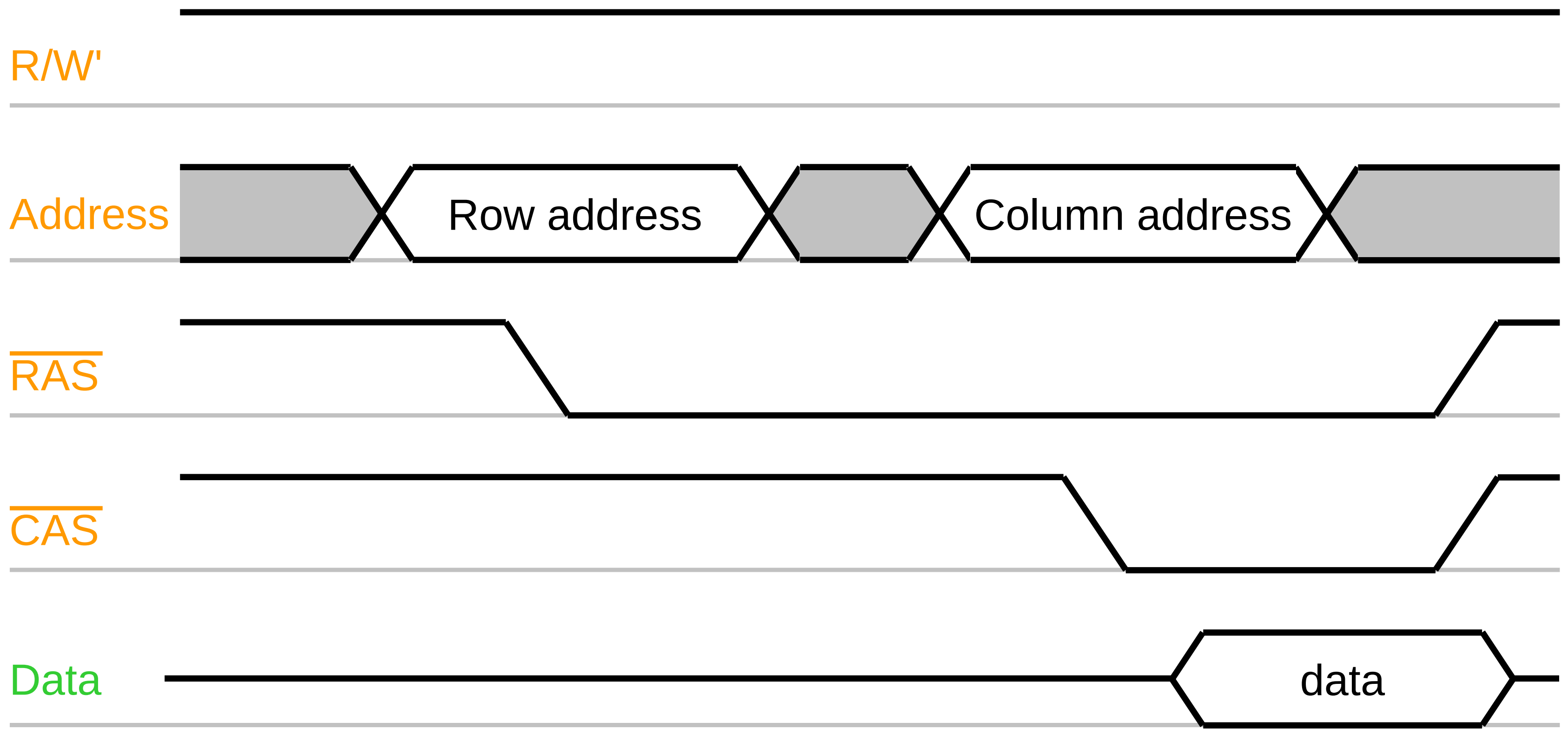
1. set up the word line decoder (row address)
2. after a minimum amount of time (row-to-column latency) connect the bit lines (column address) to the sense amplifier, activate the word line to read out the data and write it back

Since row address and column address are not required simultaneously, they can be read into the DRAM chip sequentially (time multiplex). This saves I/O pins on the device.

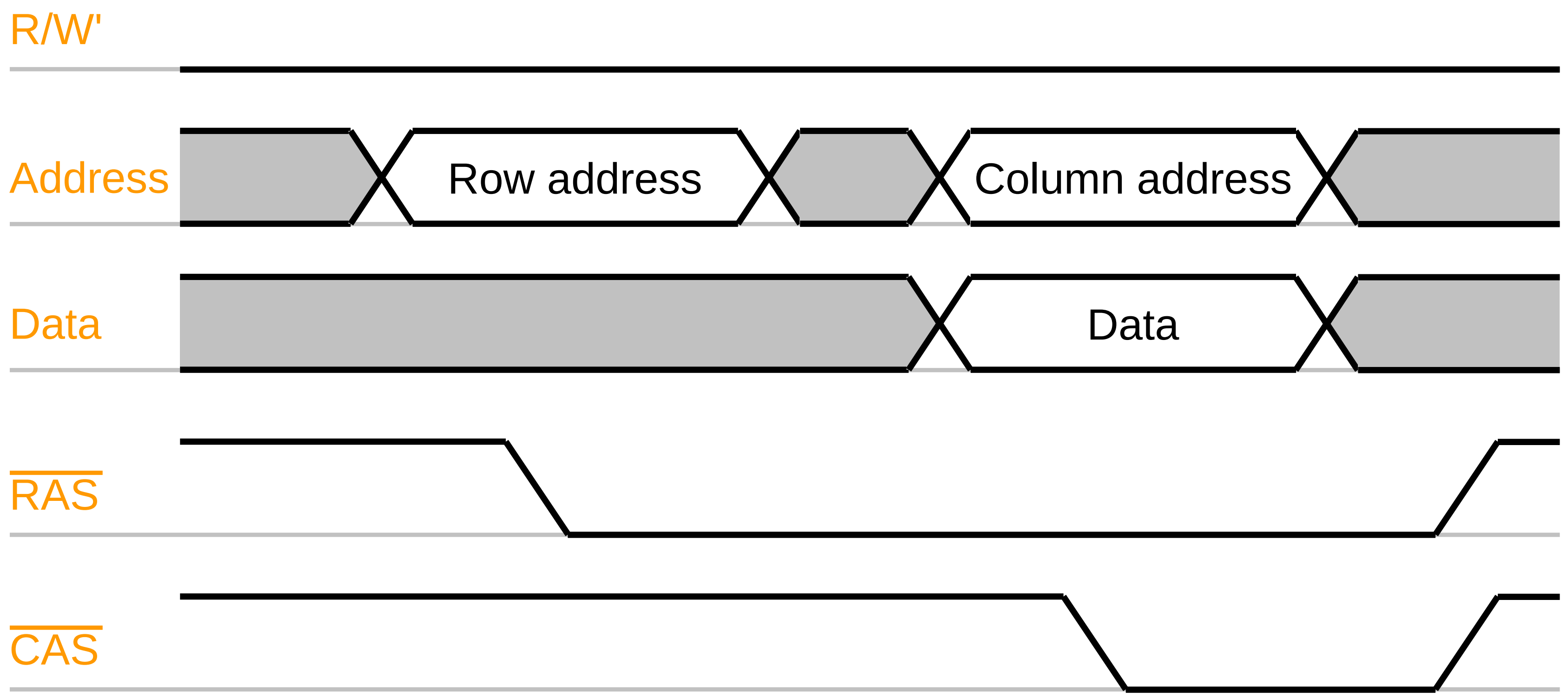
A DRAM chip has two control inputs to select the address component:

- RAS (row address strobe) selects the word line
- CAS (column address strobe) selects the bit lines.

DRAM – Read Cycle



DRAM – Write Cycle



DRAM Refresh

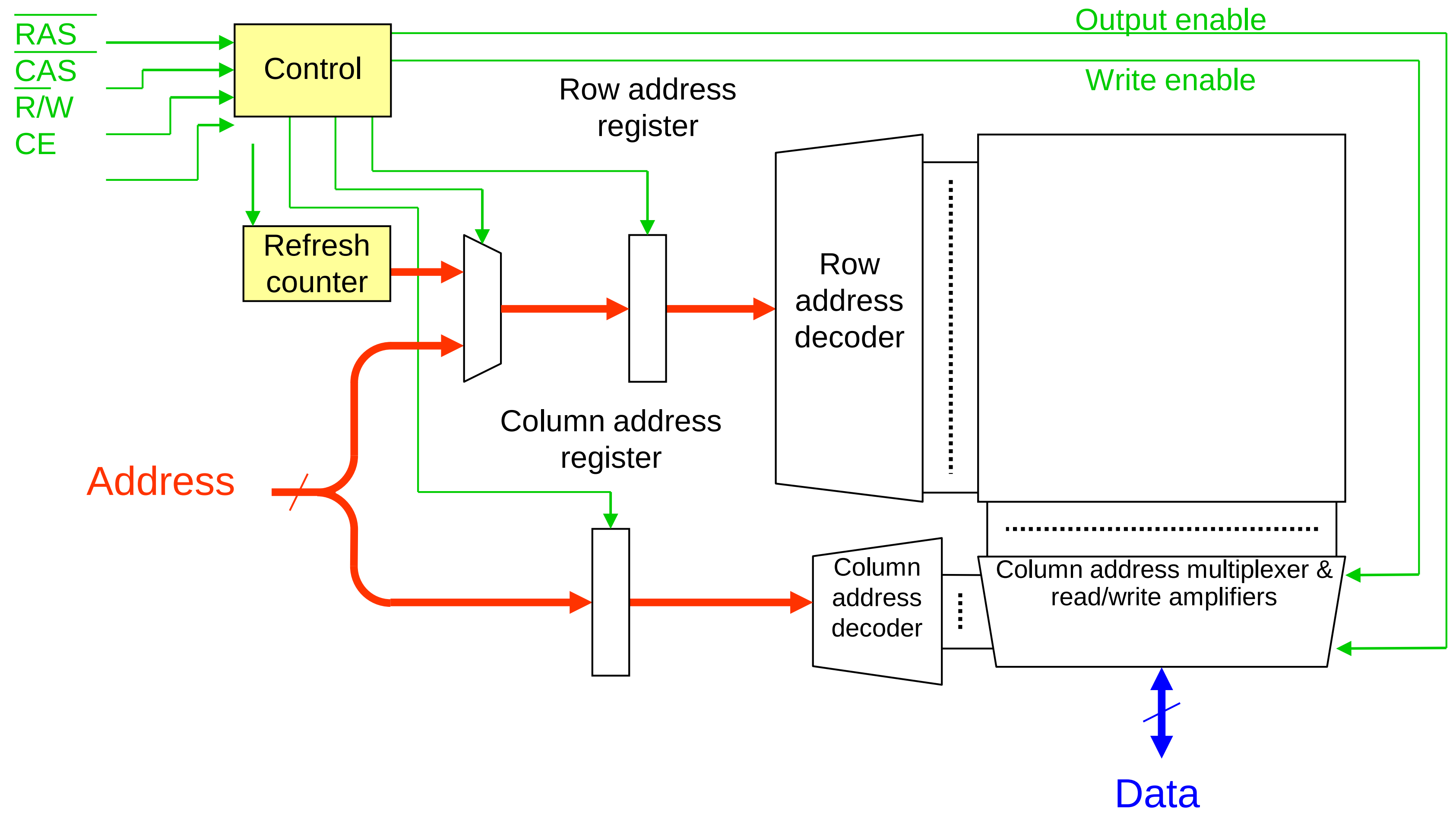
The readout electronics in a DRAM chip are built to refresh the cells of a word immediately after they are read. However, even without reading the charge in the capacitors leaks away within milliseconds. Every data word in a DRAM chip must be periodically refreshed. This is done by an external refresh controller.

DRAM chips can simplify refreshing schemes with built-in refresh modes. For example, the CAS-before-RAS refresh mode uses an internal counter for the row address. Activating the CAS signal before the RAS signal tells the DRAM chip to refresh the row addressed by the current counter value and to increment the counter.

The DRAM controller needs to insert wait states if memory requests come in during refresh cycles.

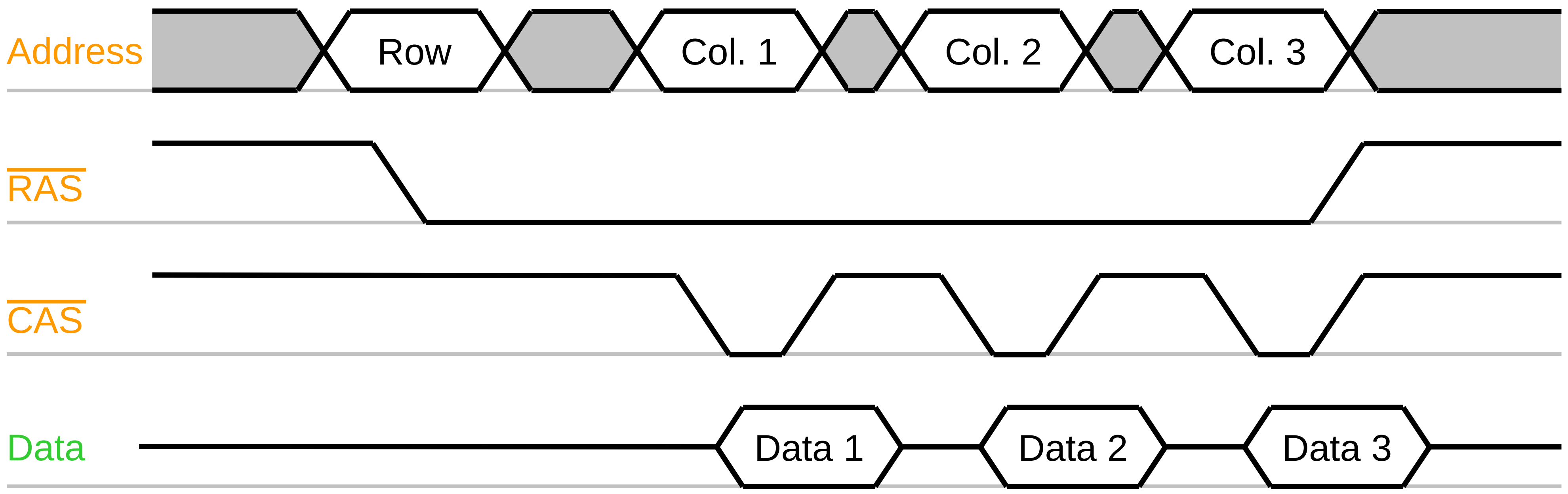
DRAM: CAS-before-RAS refresh logic

The row address to be refreshed next is kept in the refresh counter.



DRAM – Page Mode

DRAM performance can be improved by page mode. Programs often access several locations in the same region of memory. In page mode, the row address is supplied only once at the beginning of the transfer, followed by a sequence of column addresses. Page mode is typically supported for both reads and writes.



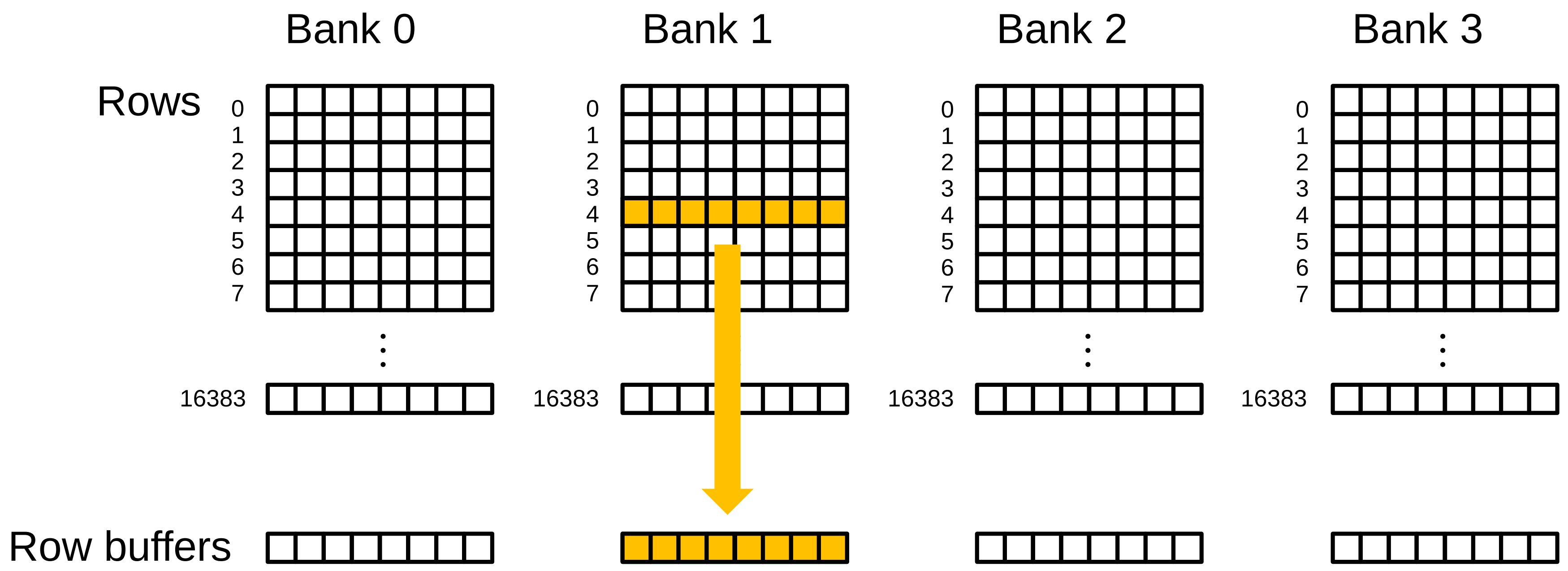
SDRAM (Synchronous DRAM)

Today, main memory of computer systems is, in general, built using SDRAM (Synchronous DRAM) devices. While classic DRAM has an asynchronous interface, an SDRAM chip has a synchronous interface to the memory controller of the system. Synchronous operation allows pipelined accesses to the memory in order to increase the performance. DDR SDRAM uses both the falling and the rising edge of the clock for a data transmission, hence doubling the data rate (DDR = Double Data Rate).

The physics of the memory circuitry requires certain timing constraints on the access to a DRAM array, e.g., a minimum time for opening a row before a read or write access may happen. However, SDRAM memory is organized into multiple *banks*, i.e., separate memory matrices. Memory accesses to different banks may happen concurrently, minimizing delays due to timing constraints.

SDRAM Operation

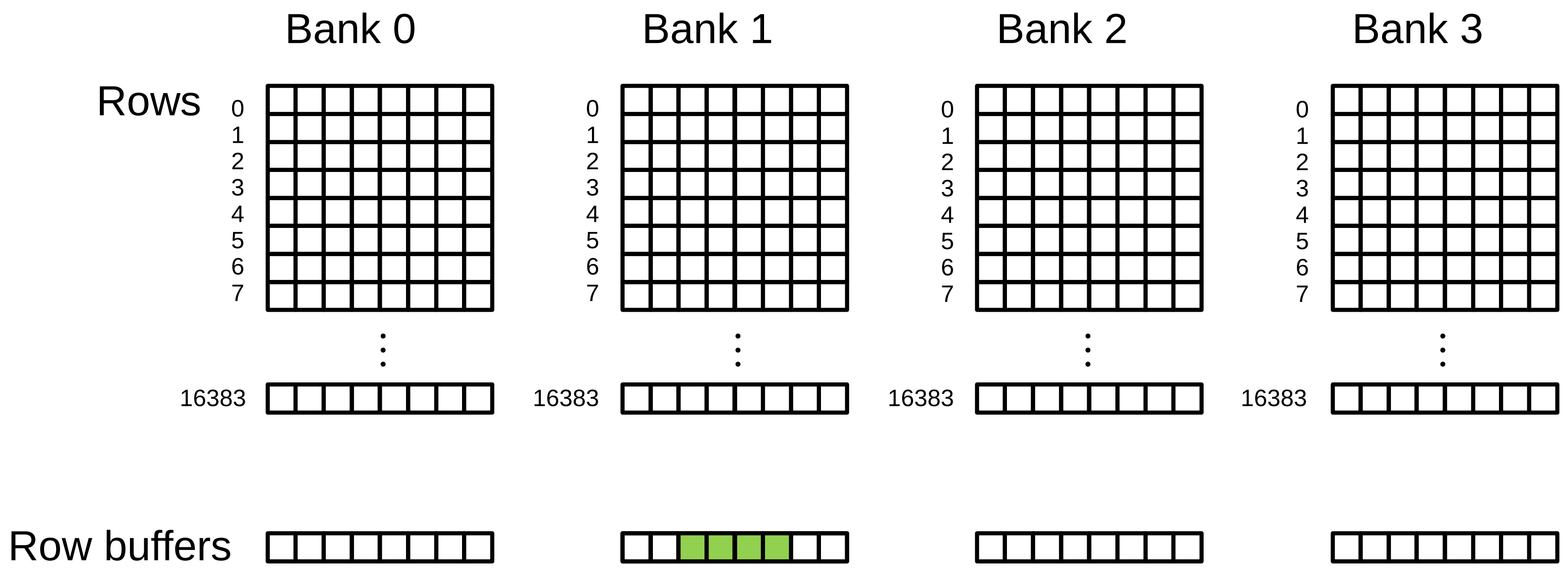
Row Activation



A row from a specific bank is requested. The row is copied into the row buffer.

SDRAM Operation

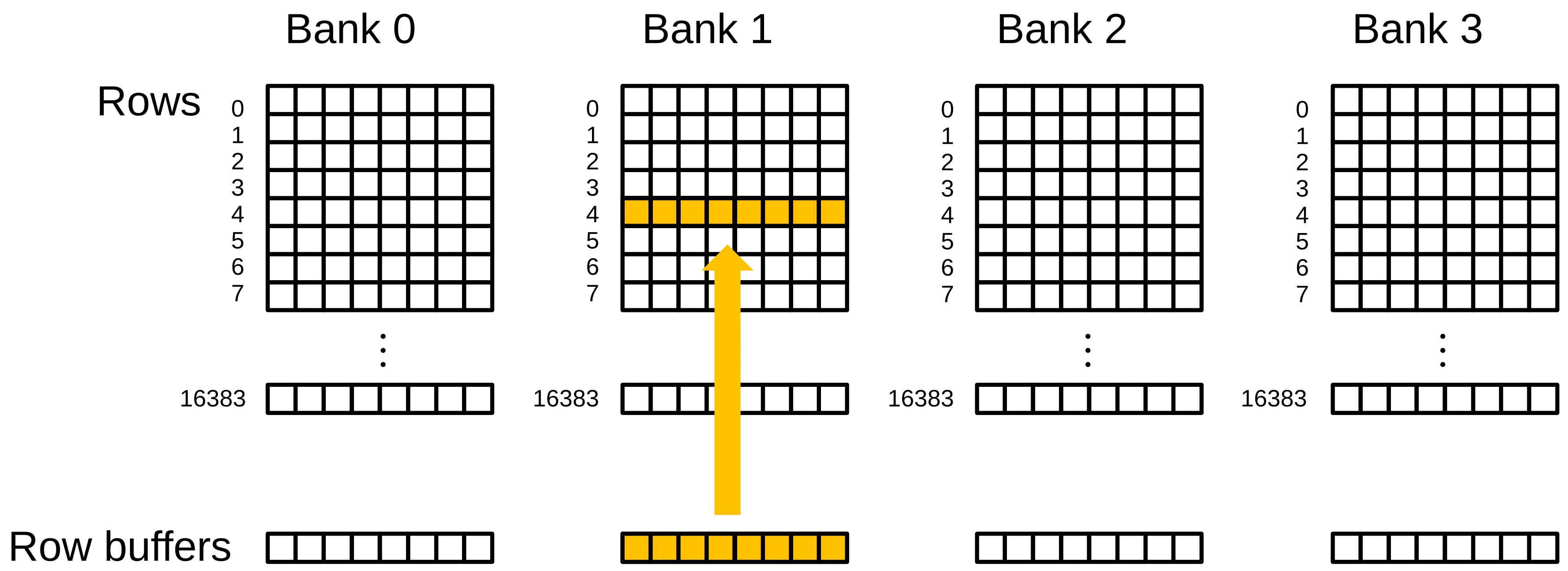
Column Access



Reads and writes are performed in *bursts* on the open row. The burst size n can be configured (typically, *n* = 4, 8, 16).

SDRAM Operation

Row Close (Precharge)



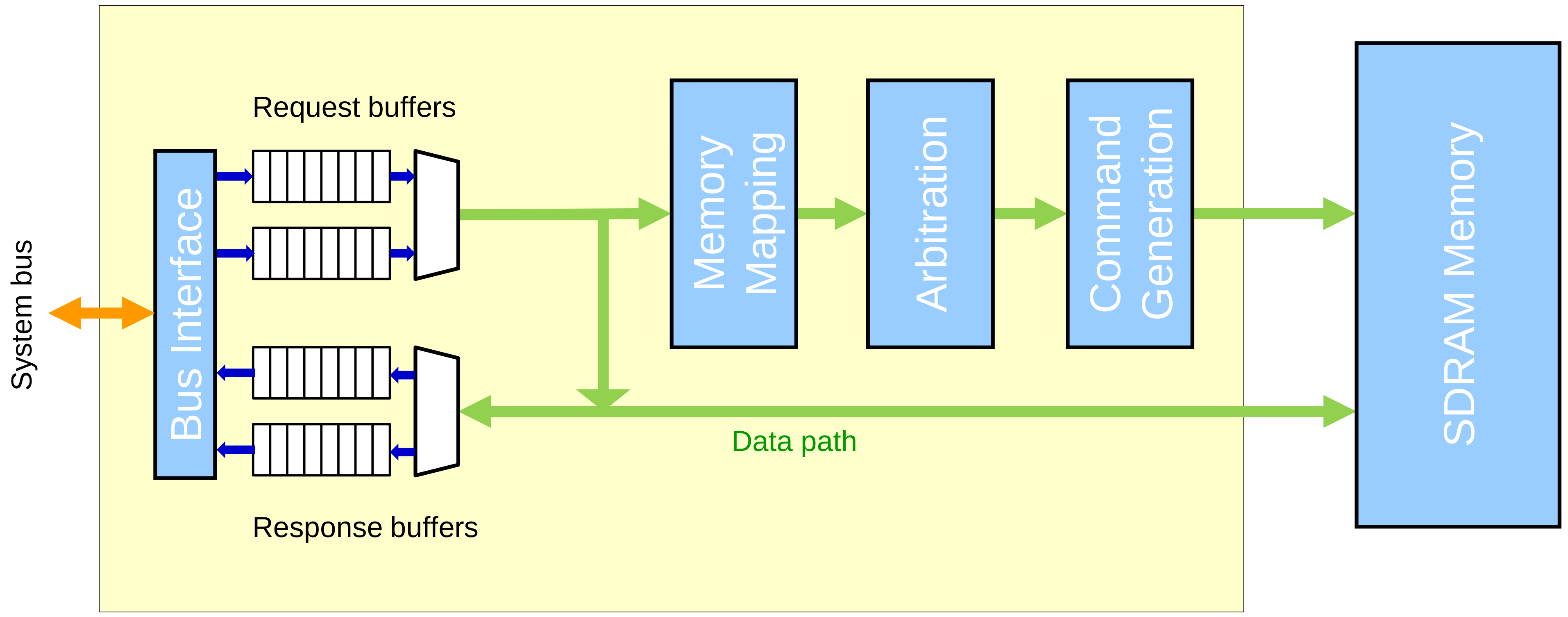
The row is written back into the memory array. The bit lines are precharged to be ready for the next row activation.

SDRAM Operation

Command	Action	\RAS	\CAS	\WE
NOP	No operation	H	H	H
ACT	Activate (open) a row in a particular bank	L	H	H
RD	Read burst from an active row	H	L	H
WR	Write burst to an active row	H	L	L
PRE	Close (precharge) current row in a bank	L	H	L
REF	Initiate a refresh operation	L	L	H

The SDRAM uses the traditional signals \RAS, \CAS, \WE as a command bus. Value assignments to these signals encode commands to the memory device.

Memory Controller



The memory controller takes the CPU bus request and generates the appropriate commands to the memory device. Memory mapping translates physical addresses to banks, rows and columns. Arbitration schedules the memory requests. Command generation is target memory-specific and takes care of correct timing.