

ADS2 Zusammenfassung

Matthias Jung

14. September 2009

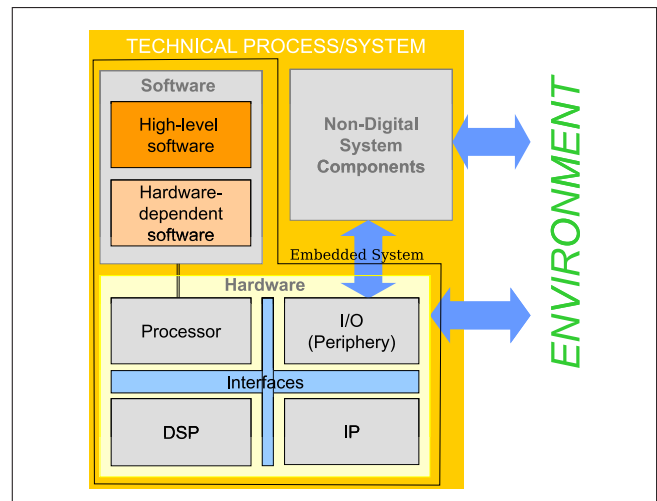
1 Embedded Systems:

Definition:

A device that includes a programmable computer but is not itself intended to be a general-purpose computer:

- The processor is integrated into a larger system (cell phone, car...)
- The device interacts with the system environment (sensors, actuators...)
- It cannot be used as a general-purpose computer.
- It fulfills application-specific tasks

More than 90% of all processors sold world-wide are used in embedded applications



Embedded Systems are Everywhere:

- Household appliances
- Telecommunications/networking
- Consumer electronics (DVD, MP3...)
- Building control
- Avionics
- Space
- Industrial automation
- Military
- Medical care

Embedded Systems in Car:

After the microprocessor was available, automobile makers started using it almost immediately because the devices are programmable so the hardware designs can be reused.

- Engine and emission control
- Antilock brake system (ABS)
- Electronic stability program (ESP)
- Airbag
- ...

1.1 Requirements on a Embedded System

Embedded computing systems have to fulfill demands different from those of general-purpose computing:

Real-Time Constraints: (may safety-critical)

Many embedded computing systems need to perform in real time (Output must be produced by a certain deadline)

Hard Realtime: if violated, the system fails completely (Airbag...)

Soft Realtime: if violated, the system does not fail completely but performance is degraded. (Video...)

The requirements for the design of an embedded system may contain: *maximum latency, minimum throughput*

Safety-Critical Systems:

Many embedded applications directly influence the safety of the systems and of the people using them (aviation, automotive, traffic control...)

Design correctness:

Formal verification and System simulation of HW and SW

Fault Tolerance / Reliability:

Built-in self-test, redundancy...

Power and Heat Constraints:

Obviously mobile applications demand an efficient usage of the available energy (PDAs, cell phones...). Also heat dissipation is important because the heat increases with the transistor density.

Cost Constraints:

The total cost of making a product is one important factor for success in the market.

- **Development cost:** processor, fpga, ip, design automation... ?
- **Manufacturing cost:** types of HW, number of components, size
- **Life-cycle costs:** maintenance

1.2 Component Choice:

Microprocessor:

- efficient implementations
- programmability
- SW is easier to maintain and update as HW
- Different product families
- software can be re-used

ASIC: Application-Specific Integrated Circuit

ASICs allow to meet performance constraints that otherwise cannot be met ASIC production only is worth at higher production volumes. Bugfixes are expensive!

FPGA: Field Programmable Gate Array

- Allows to meet performance constraints that cannot be met with a microprocessor (concurrency)
- Re-configurable

ASIC: Application-Specific Integrated Circuit

ASICs allow to meet performance constraints that otherwise cannot be met ASIC production only is worth at higher production volumes. Bugfixes are expensive!

ASIP: Application Specific Instruction Set Processor

An ASIP is an Application-Specific Instruction Set Processor. It is a special form of IP (Intellectual Property), i.e., a piece of RTL code in a hardware description language describing a special-purpose CPU.

1.3 Design Flow

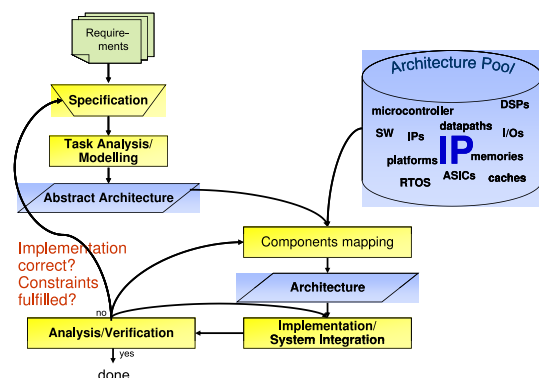
The design flow is an iterative refinement process:

Question for Design:

- How much hardware do we need?
- How do we meet deadlines and performance constraints?
- How do we minimize power consumption?
- How do we design for the future?
- How do we ensure safety, reliability and robustness?

Development is difficult.
Debugging is hard.
Testing is complex.

Design Flow:



1. Requirement Definition:

Collect the requirements for the system as an informal description. The requirements serve as a starting point for specification.

Functional Requirements:

system behavior and user interface

Nonfunctional Requirements:

performance, physical size and weight, power consumption...

2. Specification:

The specification is a more precise description of the system using technical terms. It is used as a reference during the design process. It must be written carefully and unambiguously to avoid problems during design. There are informal languages (english...) or formal languages (UML...)

3. Abstract Architecture:

The specification says what the system does. The architecture explains how! Typically the architecture is specified as a block diagram without giving too much implementation detail.

4. Final Architecture

Refinement leads to the final architecture definition given in two block diagrams: one for the hardware architecture and another for the software architecture.

6. Designing HW and SW components:

Hardware components are described on the Register Transfer Level (RTL) using HDLs like VHDL. HW components may be pre-designed (IP). Software is written in a high level language or in assembler. Here also may be standard pre-designed code snippets.

6. System Integration:

Put all components together and bring system to work. Often in this phase bugs appear but it is very difficult to discover and fix them. Re-spin the design will be very expensive! So it is much more effective to find bugs already during the design of the individual modules (Verification!)

Goal: Highly automated design Flows.

Today an efficient design flow integrates a number of design tools

- **estimators** for exploring design space and for making architectural choices
- **synthesis tools** for generating the hardware description
- **simulators** to validate the specification and verify the implementation
- **formal verification** for verifying implementation correctness
- **compilers/assemblers** for generating machine code
- **profilers** for gaining statistical information
- **debuggers** for fixing bugs in SW and HW

...

2 Microprocessor interfaces:

2.1 Interrupts Exceptions and Traps

Input and Output Devices :



In embedded systems the CPU needs to communicate with its environment (disk, graphics...). The digital interface between an input or output device and the CPU is typically a set of registers.

Programming I/O:

I/O instructions: The architecture provides special instructions for accessing devices. The I/O address space is separate from the main memory address space.

Memory-Mapped The device registers are accessed using addresses from main memory address space. Programs use the normal memory access instructions to read and write the device registers.

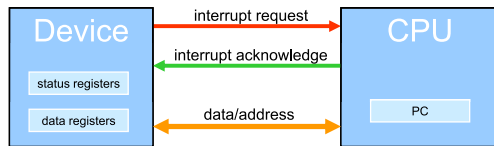
Busy-Waiting - Polling:

A simple way to access I/O devices is called polling. The CPU monitors the device by reading its status register frequently. Polling is easy to implement but the CPU does nothing but polling...

Interrupts:

The I/O device signals the CPU that it is ready to send or receive data. The Program currently running on the CPU is interrupted and a special piece of software (interrupt handler ISR) is called. Later return to the foreground program.

Basic Interrupt Mechanism:



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

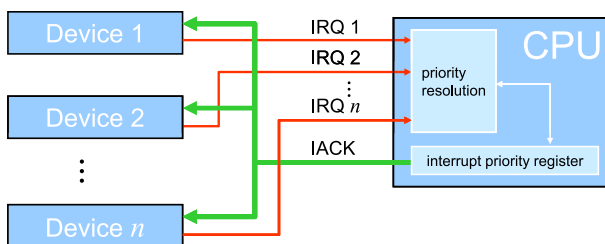
Interrupt Masking:

The priority resolution mechanism in the CPU that lower priority interrupts do not occur while a higher priority interrupt is being handled.

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).

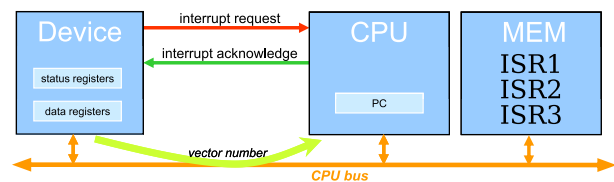
Interrupt Priorities :

the CPU may treat some I/O service requests as more important than others



Interrupt Vectors :

the I/O device can tell the CPU the service routine that should handle its request.



The device stores its vector number. The interrupt handler for a device can be exchanged simply by changing the vector number it sends.

Interrupt Overhead:

- The jump into the interrupt handler changes the PC. (also return from interrupt)
- Save the state of the CPU
- Time is needed to acknowledge the interrupt and obtain the vector number from the requesting device.
- The interrupt handler save and restore registers.

Exceptions:

An exception is an unexpected internal event. (e.g. division by zero) Exception handling is usually implemented much in the same way as interrupt handling. Prioritisation is needed because a single instruction can cause several exceptions.

Traps:

A trap is a software interrupt, an instruction that explicitly generates an exception. In many cases traps are used to enter a supervisor mode which has certain privileges over user mode (protecting the system against buggy or malicious code).

Exceptions:

An exception is an unexpected internal event. (e.g. division by zero) Exception handling is usually implemented much in the same way as interrupt handling. Prioritisation is needed because a single instruction can cause several exceptions.

2.2 Bus

Definition:

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.

There are two types of participants, the master and the slaves. The master initiates data transfers with the master

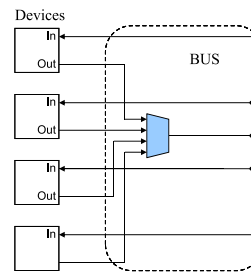
There are 3 types of wires: data, address, control.

Transaction Timing:

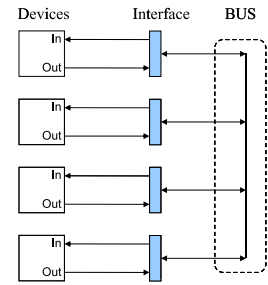
Transactions on the bus must follow certain rules and timing constraints. This is defined by the bus protocol. (e.g. the master supplies the address signal long enough for the slave...)

Bus Interconnect Implementations:

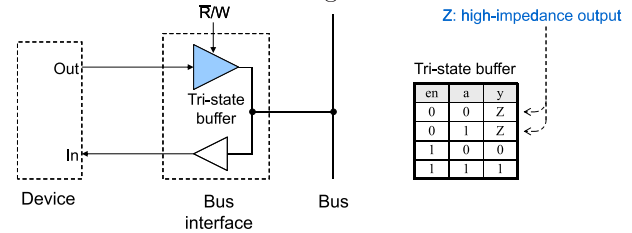
Multiplexers:



Bus Interfaces:



Bus interface using tri-state buffer:



Synchronous Bus:

A synchronous bus contains a clock signal to synchronise the bus interfaces among each other. All interfaces run with the same clock speed. Synchronous buses can be very fast. The wires cannot be very long because of clock skew.

Asynchronous Bus:

An asynchronous bus does not have a dedicated clock signal. The internal clocks of the interfaces may run at different speeds. Synchronisation is done with control signals (handshaking). Asynchronous buses can be used for longer distances.

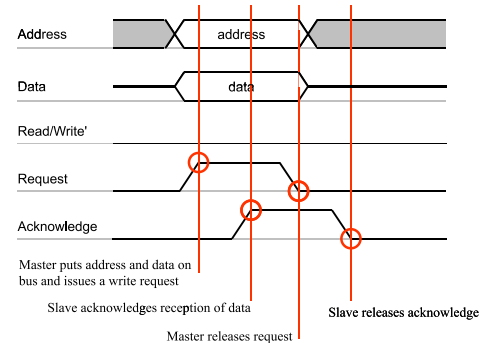
Asynchronous Four-cycle Handshake:

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 cycles. 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.

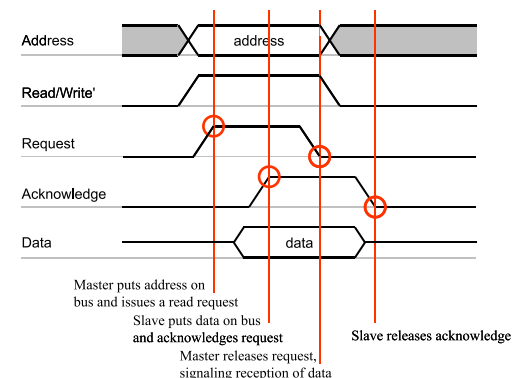
Write:

Example of an asynchronous bus write transaction:



Read

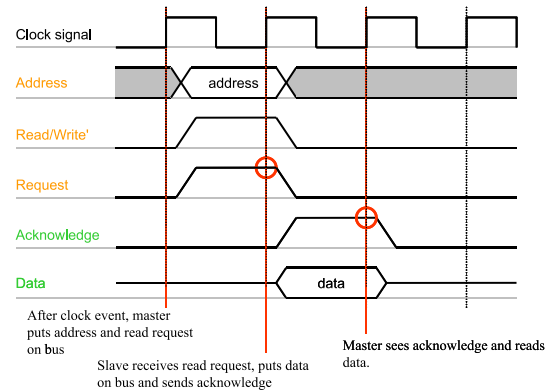
Example of an asynchronous bus read transaction:



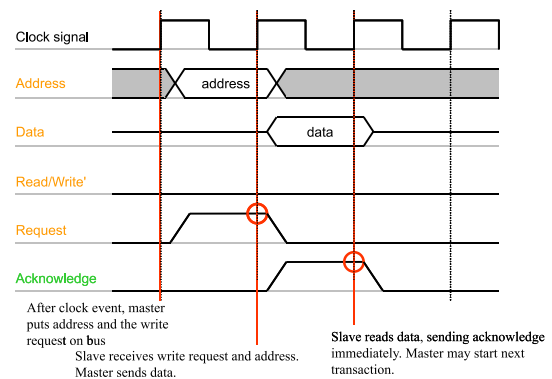
Synchronous Bus:

In a synchronous bus, handshaking is not necessary if the clock period is long enough so that all timing constraints are met. The time points for "handing over" address and data between the communication partners are defined by the rising or falling edges of the clock signal. (A protocol may even be designed to use both edges!) 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.

Read:



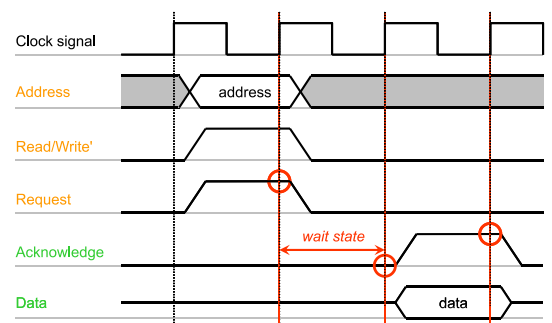
Write



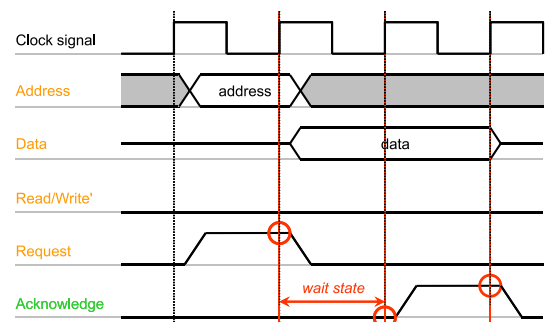
Wait States:

Some slaves may not be fast enough to respond to a master request within one clock cycle (example, if the slave is a large, 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.

Read:



Write



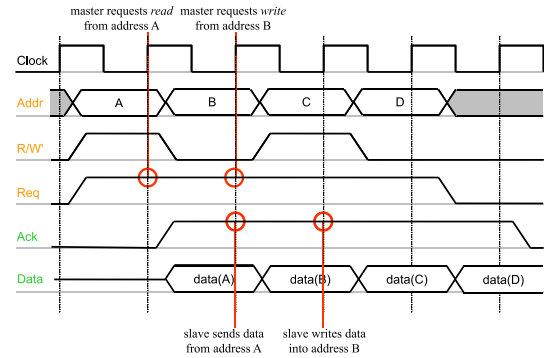
Pipelining a Synchronous Bus:

If a synchronous bus has separate lines for data and address, the address phases and data phases of individual transactions may be independent of each other.

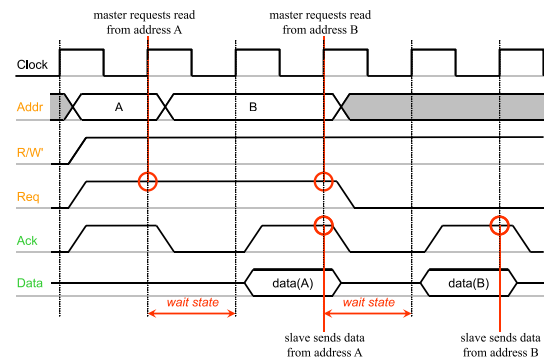
Idea:

Let address phase and data phase of two transactions overlap → pipeline the bus transactions!

Read/Write:



Read/Write With Wait States:

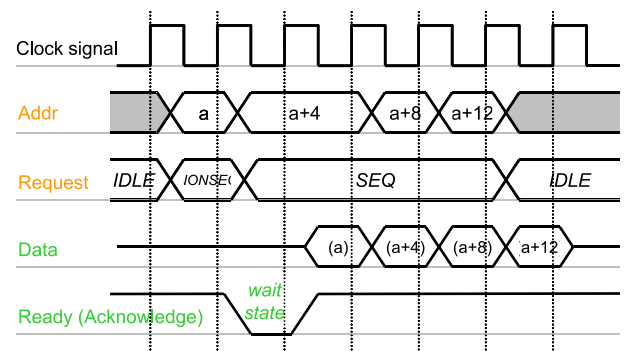


Burst Modes

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 signals to the slave when data transfer is to stop.

Burst modes can be:

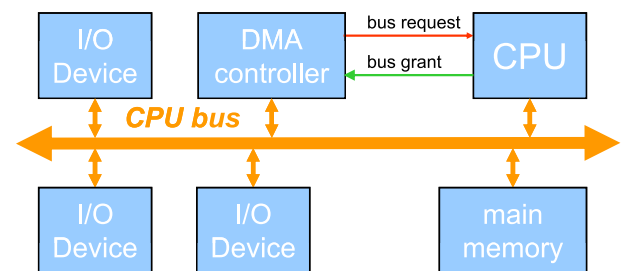
- incremental
- wrapping



DMA Controller:

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 four-cycle handshake on two additional bus signals, bus request and bus grant. Typically, the CPU initiates the DMA transfer. It writes to the registers in the DMA controller (source, destination, length of the block...)

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.



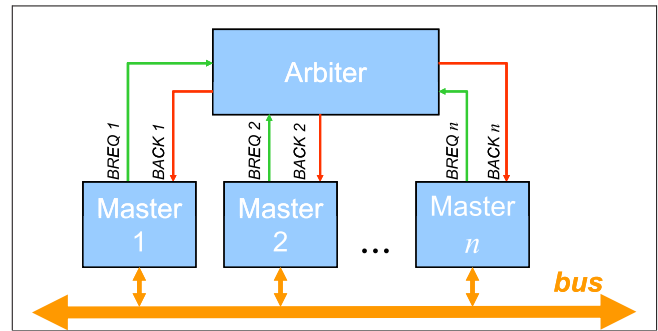
Without DMA all memory transactions on such a bus require the CPU as mediator. This is especially inefficient for data transfers from one slave to another.

Bus Arbitration:

In computer systems, several devices connected to the bus may be required to become bus master. Whenever there is more than one possible bus master, access to the bus must be allowed by a arbitration scheme:

Local Arbitration: CPU is the arbiter.

Global Arbitration: External arbitration logic

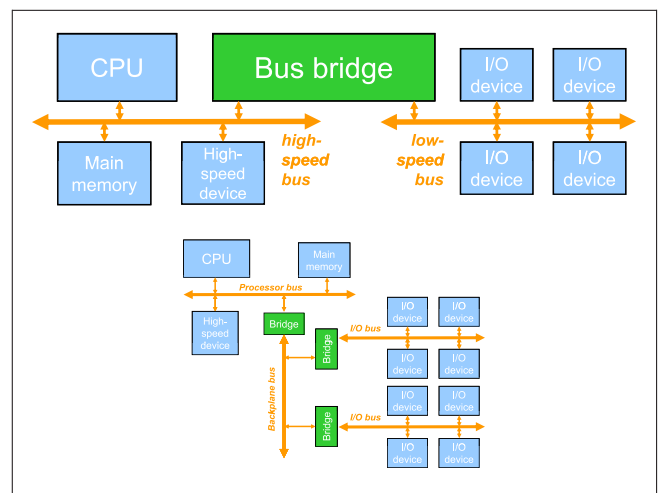


Split Transactions:

Some multi-master bus protocols allow split transactions. The address phase and the data phase of a transaction may be separated in time. The master requests a transaction as usual. The slave, instead of answering with a number of wait states, sends a SPLIT as answer over a separate signal. The bus is free again for other masters' transactions. When, finally, the slave is ready to answer the request, the master re-requests the bus to complete the transaction.

Bus Bridge:

All devices in a synchronous bus need to run at the same clock rate. Even low-speed devices need a high-speed bus interface or fast interfaces have to speak slow. This is inefficient and expensive. The CPU needs to communicate with high speed at least with main memory. For a high speed bus, the wires may 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 low-speed bus. The connection between the high-speed CPU bus and the lower-speed bus is made using a special device called bus bridge.



Bus Bridge Transactions:

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:

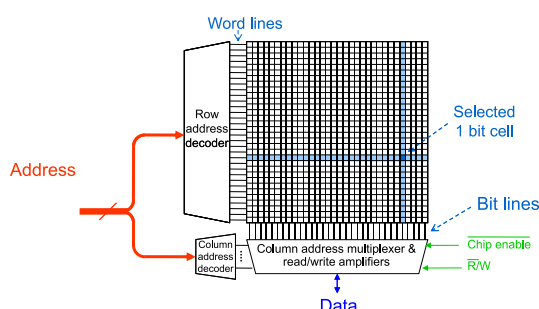
In a write operation, the bridge can store address and data in an internal buffer and immediately acknowledge the write to the master. When the buffer is full, the bridge signals a SPLIT transaction.

Read operation:

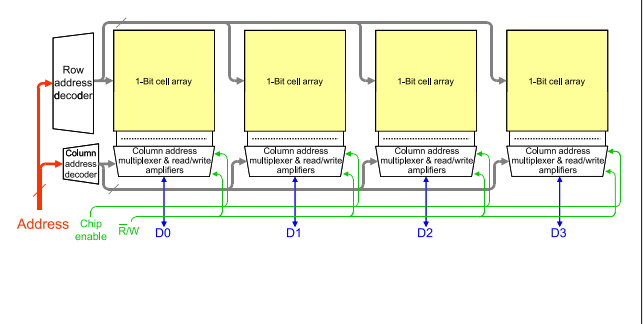
In a read operation the bridge signals a SPLIT transaction on the fast bus. As soon as the data has been read from the slow bus the transaction on the fast bus can be resumed and completed. (burst....)

2.3 Memory

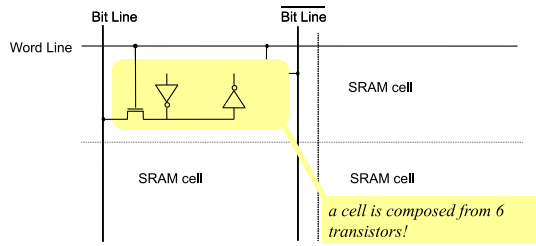
Cell Array:



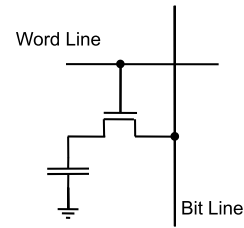
Memory Organization:



SRAM:



DRAM:



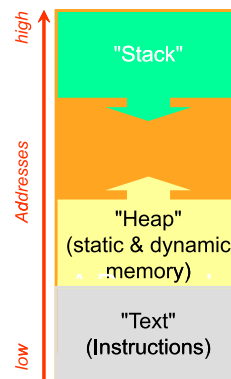
2.4 Processes and Operating Systems

Co Routine:

An early approach to multi-tasking was the co-routine. A co-routine is similar to a subroutine, however, it has several entry points. Each task is implemented as a co-routine. The co-routines decide when to yield the CPU to another task, explicitly jumping into the corresponding co-routine.

- + easy to implement
- + no OS
- + very fast switching
- hardcoded → inflexible

Process:



Switching between tasks is handled by a specialized system component - the scheduler. Tasks are handled by processes. The scheduler decides when to assign the CPU to a task. A process is the unique execution of a program. A process is defined by its program code and its data. The data is in the main memory as well as in the CPU registers.

Threads:

Threads are light-weight processes often used in embedded systems. They have their own distinct sets of values for CPU registers but share the same main memory space. (Communication is simple and fast, but they can destroy the data of each other)

Example Fork and Execv:

```
pid_t pid = fork();
if (pid == 0) { execv("prog", childargs); }
else if (pid == -1) { /* error */ }
else { parent_stuff(); wait(&childstat); }
```

Context Switch:

In a context switch, the operating system needs to replace the context of one process by that of another. The context is stored in the process control block PCB.

- PC
- Stackpointer
- general-purpose registers
- status registers
- return address registers

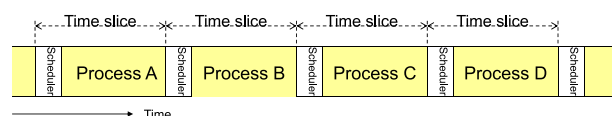
Cooperative Multitasking:

In a cooperative multitasking system processes give up the CPU to other processes voluntarily. A process decides autonomously when to initiate a context switch by calling the scheduler. This is more flexible than co routine.

Problem: if a single process doesn't give up the CPU the system may lock up, or real-time constraints may be violated

Preemptive Multitasking:

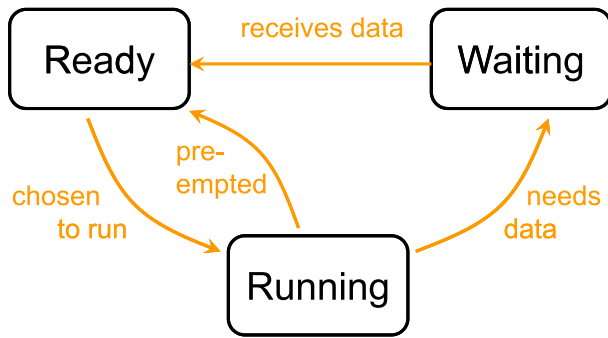
A system timer triggers an interrupt to the CPU which call the process scheduler of the operating system to switch the context.



Round Robin Scheduling:

Is a simple scheduling policy. All process control blocks are kept in a linked list. The processes are activated one after the other. After the end of the list is reached the scheduler starts with the first process again,

Process States:



Rate Monotonic Scheduling:

All process run with a constant period and constant execution time. Assign priorities according to their period. *the shorter the period of a process, the higher its priority*

RMS is therefore a static scheduling policy. It provides high CPU utilization while ensuring that all processes meet their deadlines.

Earliest Deadline First Scheduling:

It assigns priorities in the order of deadline. The highest priority is assigned to the process whose deadline approaches next. Priorities must be recalculated at every termination of a process (dynamic scheduling policy). Implementation of EDF is more complex than that of RMS since the pool of processes must be kept sorted in order of deadline. This requires additional computation time in the scheduler.