Embedded Computing Platform

IL2206 Embedded Systems

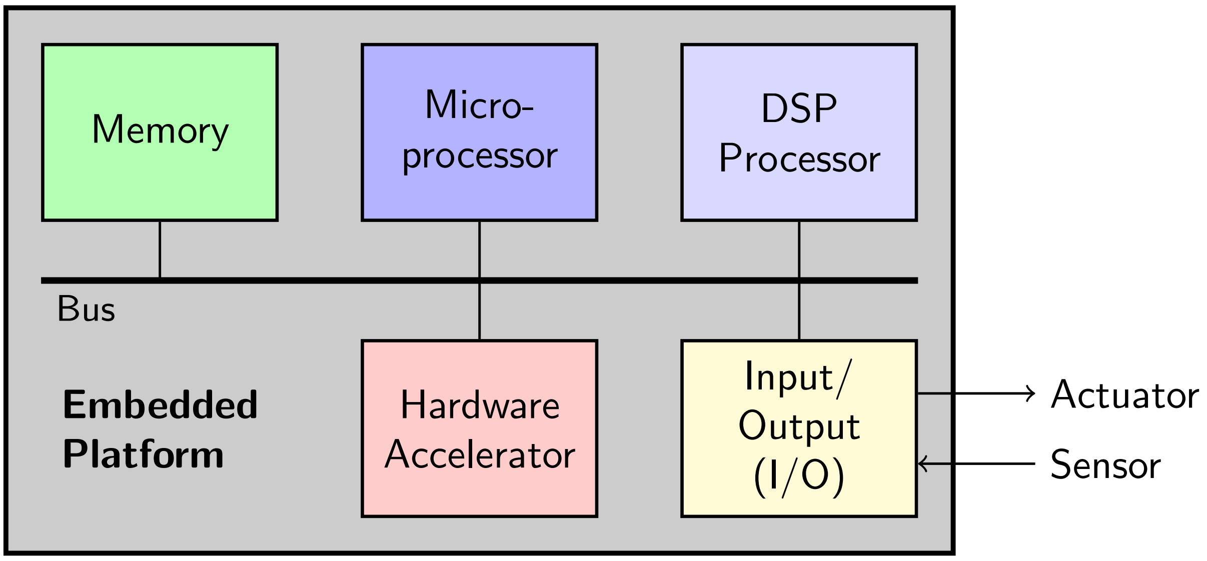
Ingo Sander

KTH Royal Institute of Technology
Stockholm, Sweden
ingo@kth.se

Outline

Outline

Embedded Computing Platform



- A platform is the basic hardware and software architecture needed to implement an embedded application

Outline

Microprocessor

- The microprocessor is the key component in embedded systems
- Modern cars include around 40 to 100 microprocessors
  - Advanced functions (engine control, brake system) need powerful processors (32-bit)
  - Simple functions (window control) need less powerful processors (8-bit)
- Autonomous driving requires very powerful processing capabilities

Microprocessor

Microprocessors are

- flexible and easy to program
- cheap and optimised
- come in several variations
  - 8-bit, 16-bit, 32-bit
  - general purpose microcontroller, DSP, or customised processor

but include overhead in form of instruction decoding and memory access (compared to pure hardware)

Microprocessor

Microprocessors are

- flexible and easy to program
- cheap and optimised
- come in several variations
  - 8-bit, 16-bit, 32-bit
  - general purpose microcontroller, DSP, or customised processor

but include overhead in form of instruction decoding and memory access (compared to pure hardware)

A microprocessor is found in almost any embedded system!

Microprocessors

- Microprocessors are used as key components in an embedded design
- Programmable Logic and ASICs are used for critical parts in a design
- An objective for an embedded system designer is to find the cheapest solution that meets the requirements

Microprocessors

- Microprocessors are used as key components in an embedded design
- Programmable Logic and ASICs are used for critical parts in a design
- An objective for an embedded system designer is to find the cheapest solution that meets the requirements

1. A microprocessor is found in almost any embedded system!

Microprocessors

- Microprocessors are used as key components in an embedded design
- Programmable Logic and ASICs are used for critical parts in a design
- An objective for an embedded system designer is to find the cheapest solution that meets the requirements

1. A microprocessor is found in almost any embedded system!
2. Do not use a powerful 32-bit processor, when you only want to control a simple battery-driven device

Embedded System Processors

- There exist many different microprocessors that are used for embedded systems
  - different sizes and performance
  - processors may either come as
    - integrated circuit component or hard core (like Intel i5 or i7)
    - soft core, which can be instantiated on ASIC or FPGA

Embedded System Processors

- Embedded microprocessors follow general concepts
  - similar architecture
  - similar instruction set
- Embedded programs are usually written in C or another high-level programming language, only very critical parts are programmed in assembly language

Programming in Assembly

- The assembly language reflects the instruction set (almost one to one)
  - One instruction per line
  - Labels provide names for addresses (usually in first column)
- Example (Nios II assembly language):

```
      ...
      movi r1, 1         # Move 1 into register R1
loop: addi r1, r1, 1     # Add 1 to R1, store result in R1
      bge  r2, r1, loop  # Branch to loop if R2 >= R1
cont: ...
```

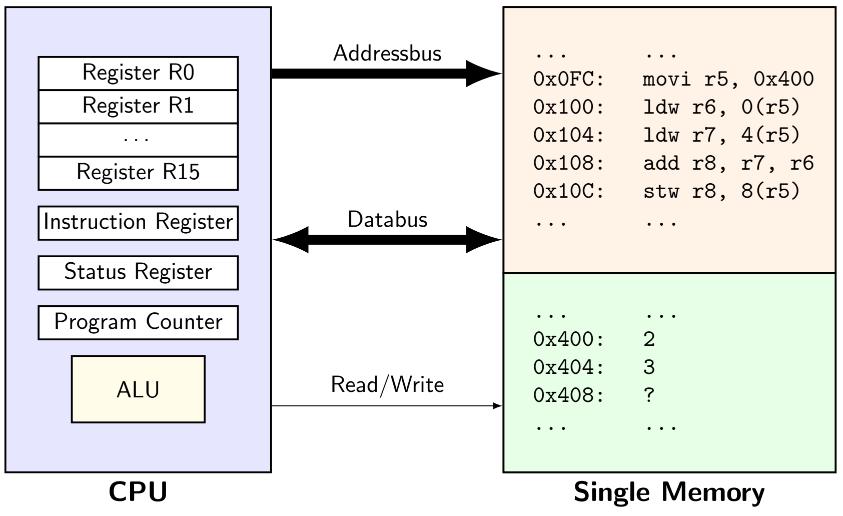
Another Assembly Program

```
start:
     movi r5, 0x400
     ldw r6, 0(r5)
     ldw r7, 4(r5)
     add r8, r7, r6
     stw r8, 8(r5)
     ...
```

How is this program executed?

von Neumann Architecture

- CPU with registers and one single memory
- Memory contains instructions and data

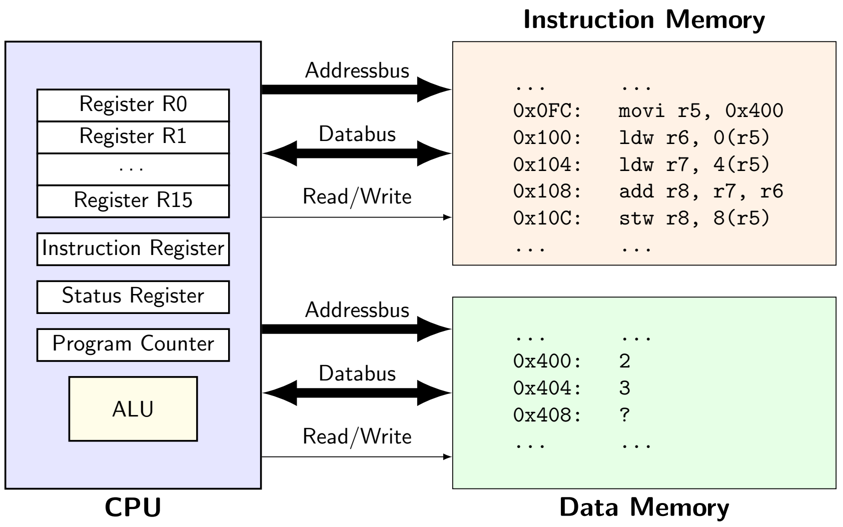


```
...      ...
0x0FC:   movi r5, 0x400
0x100:   ldw r6, 0(r5)
0x104:   ldw r7, 4(r5)
0x108:   add r8, r7, r6
0x10C:   stw r8, 8(r5)
...      ...

...      ...
0x400:   2
0x404:   3
0x408:   ?
...      ...
```

Harvard Architecture

- CPU with registers and two memories
- One memory holds instructions (instruction memory)
- One memory holds data (data memory)

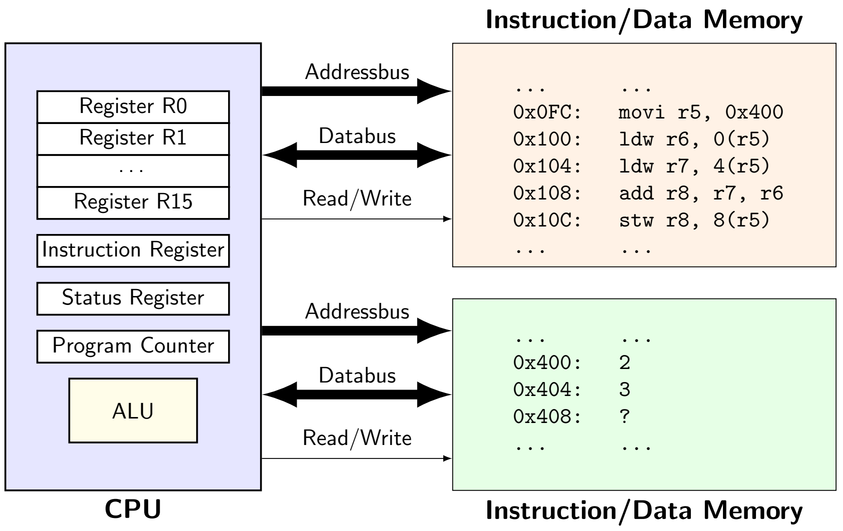


```
...      ...
0x0FC:   movi r5, 0x400
0x100:   ldw r6, 0(r5)
0x104:   ldw r7, 4(r5)
0x108:   add r8, r7, r6
0x10C:   stw r8, 8(r5)
...      ...
```

```
...      ...
0x400:   2
0x404:   3
0x408:   ?
...      ...
```

Modified Harvard Architecture

- CPU with registers and two memories
- Both memories can hold instructions and data



```
...      ...
0x0FC:   movi r5, 0x400
0x100:   ldw r6, 0(r5)
0x104:   ldw r7, 4(r5)
0x108:   add r8, r7, r6
0x10C:   stw r8, 8(r5)
...      ...
```

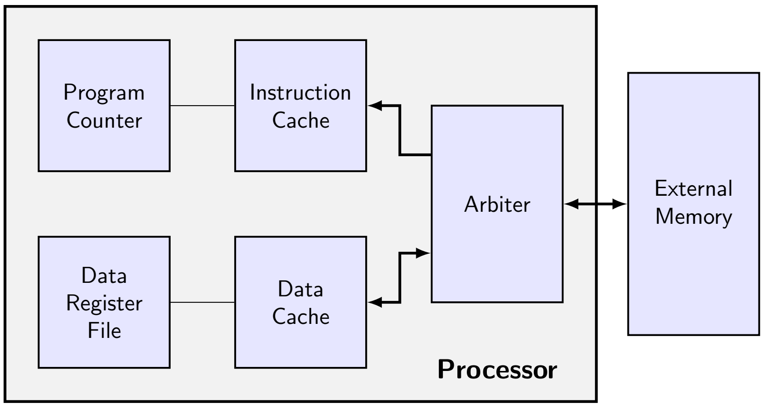
```
...      ...
0x400:   2
0x404:   3
0x408:   ?
...      ...
```

Comparison von Neumann and Harvard Architecture

- Harvard allows two simultaneous memory fetches
- Most DSPs use Harvard architecture internally for streaming data
- Additional hardware, since two address and data busses are needed

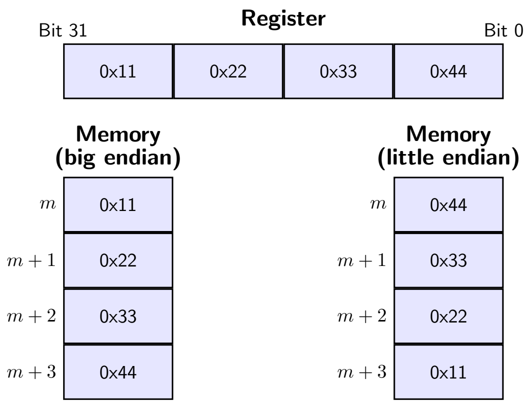
Harvard Architectures and External Memories

- Most embedded systems have an internal Harvard architecture with
  - instruction cache
  - data cache
- but only a single bus to external memory



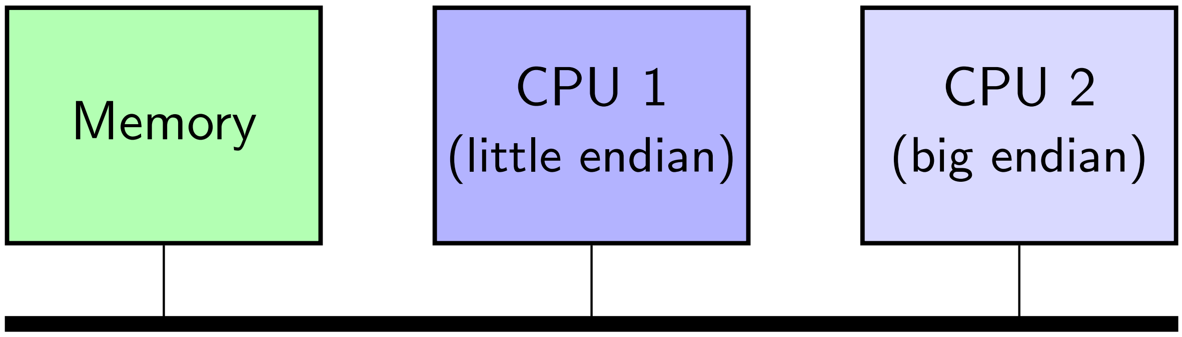
Addresses and Endianness

- Many embedded processors use 32 bit-registers
- A data word is 32 bits (4 bytes) long
- An address refers to a byte in the memory
- Different standards to arrange data words in memory
  - big endianness: most significant byte is placed first (on lowest address) and least significant bit last (on the highest address);
  - little endianness, least significant byte is placed first (on lowest address) and the most significant bit last (on highest address).



Be aware of Endianness. . .

- Embedded systems has many devices communicating with each other
- Different endianness can cause bugs and disastrous problems
  1. CPU 1 writes `0x11223344` into a memory location `0x1000`
  2. CPU 2 reads from same memory location `0x1000`, but interprets the value as . . . ?

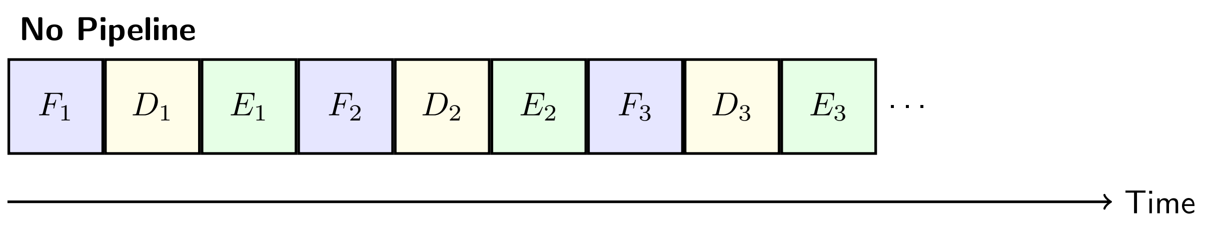


Pipelining

- Idea of pipeline is to increase execution speed
- Several instructions are executed simultaneously at different stages of completion
- Various conditions can cause pipeline stalls that reduce execution speed, e.g. branches

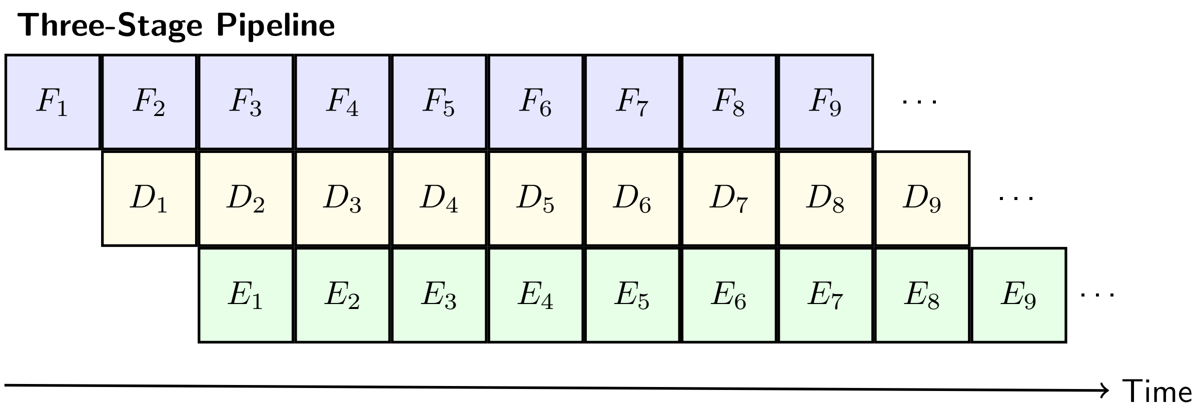
Processor without Pipeline

- Processor will either
  - fetch an instruction (F),
  - decode an instruction (D),
  - execute an instruction (E)
- Processor produces a result every third cycle

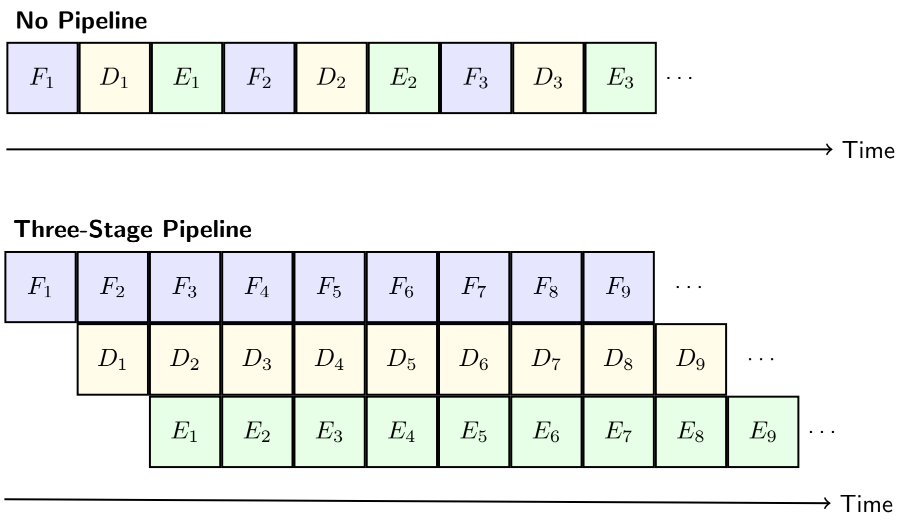


Processor without Three-Stage Pipeline

- Processor will simultaneously
  - fetch an instruction (F),
  - decode an instruction (D),
  - execute an instruction (E)
- Processor produces a result every cycle


Three-Stage Pipeline
F_1 F_2 F_3 F_4 F_5 F_6 F_7 F_8 F_9 $\cdots$
D_1 D_2 D_3 D_4 D_5 D_6 D_7 D_8 D_9 $\cdots$
E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8 E_9 $\cdots$
Time

Pipeline: Latency and Throughput


No Pipeline
F_1 D_1 E_1 F_2 D_2 E_2 F_3 D_3 E_3 $\cdots$
Time
Three-Stage Pipeline
F_1 F_2 F_3 F_4 F_5 F_6 F_7 F_8 F_9 $\cdots$
D_1 D_2 D_3 D_4 D_5 D_6 D_7 D_8 D_9 $\cdots$
E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8 E_9 $\cdots$
Time

Latency	Time required to go through all stages of the pipeline
Throughput	Number of instructions executed per time unit

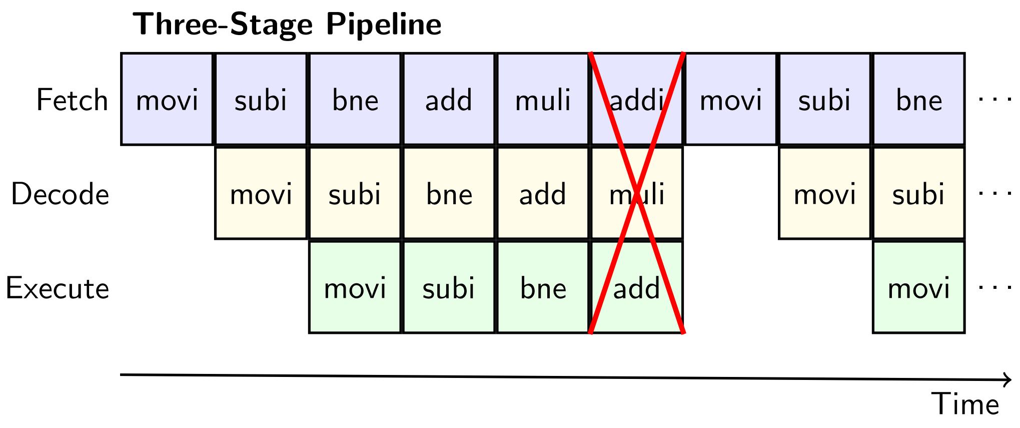
Pipeline Stalls

- In a perfect pipeline
  - all stages take same amount of time
  - fetched instruction will also be executed
- Pipeline stall can occur, if these prerequisites are not met

Pipeline Stalls

- In a perfect pipeline
  - all stages take same amount of time
  - fetched instruction will also be executed
- Pipeline stall can occur, if these prerequisites are not met

```
loop: movi r2, 5
      subi r2, r2, 1
      bne loop
      add r5, r3, r4
      muli r6, r5, 10
      addi r6, r6, 2
```



Outline

Memory Bottleneck

- Most instructions in a RISC processor can execute in a single clock cycle
- Problem: Access to main memory (typically in DRAM) is slow
- If memory access time can be shortened the system would perform considerably better

Memory Performance

Memory Bandwidth Rate at which information can be transferred from the memory system (bytes/second)

Latency Time between the following two time instances

- time instance where the processor issues a request to the memory
- time instance where the requested data arrives and is available for use by processor

Memory Types

Volatile Memories Memories lose their information, when they are disconnected from power

- static RAM (SRAM)
- dynamic RAM (DRAM)

Non-Volatile Memories Memories keep their information, when they are disconnected form power

- Flash
- EPROM, EEPROM

Embedded systems require usually both volatile and non-volatile memory

Flash Memory

- non-volatile memory
- low-cost and low-power (compared to hard-drive)
- can be erased and updated, but takes considerably long-time

Applications Flash Memory

Keeping any information that is needed in case of power failure (such as boot code for the microprocessor or the configuration of the FPGA)

SRAM (Static Random Access Memory)

- volatile memory
- simple memory interface
- fast memory (compared to DRAM), since it does not need to be refreshed
- higher costs (compared to DRAM)
- memory cell takes much more space than a DRAM cell

Applications SRAM

- buffers and look-up-tables, where access time shall be small
- main memory for a small application running on a CPU that has no cache

DRAM (Dynamic Random Access Memory)

- volatile memory
- memory must be refreshed to keep its content
- memory cell is very small compared to SRAM (cheap and large memories)
- DRAM interface is much more complex and requires a more sophisticated controller
- Latency is larger than in SRAM

Applications DRAM

- Storing large blocks of data
- Main memory for large programs

Memory Access Times and Costs

Memory Technology	Typical Access Time	Cost per GB in 2004
SRAM	0.5 ns - 5 ns	$4000 - $10000
DRAM	50 ns - 70 ns	$100 - $200
Magnetic Disk	5 ms - 20 ms	$0.5 -$2

[Source: Patterson and Hennessy, 2004]

Embedded System Memories

- Large fast memories are very expensive
- Embedded systems have to be produced at a low cost
  - single SRAM main memory is in general too expensive
  - combination of fast and slow memories is often still feasible

Key Challenge

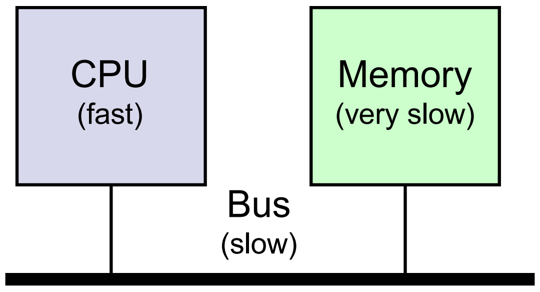
Selecting an efficient memory architecture in the embedded system design process

Caches

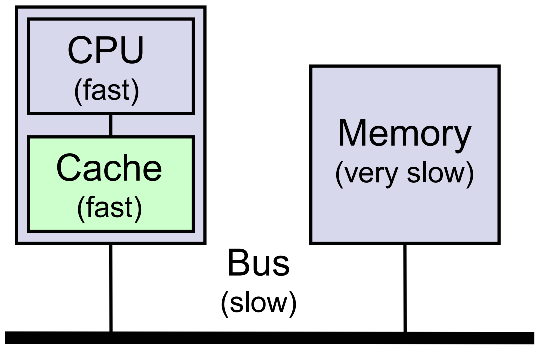
- Large fast memories are too expensive, but small fast memories are feasible
- A cache memory is a small, but fast memory that is located near the CPU to reduce memory access times
- Ideally the processor does only need to access the cache and not the main memory
- There can be dedicated caches for data, instruction, or unified caches (data and instruction)

Memory is Bottleneck

While the CPU is fast, each memory access takes long time and slows down the system



Caches can increase the performance, if most memory requests do not need to access the main memory



Principal of Locality

Programs access a relatively small portion of their address space at any instant of time

- Temporal locality: If an item is referenced, it will tend to be referenced again soon
- Spatial locality: If an item is referenced, items whose addresses are close by tend to be referenced soon

Principal of Locality

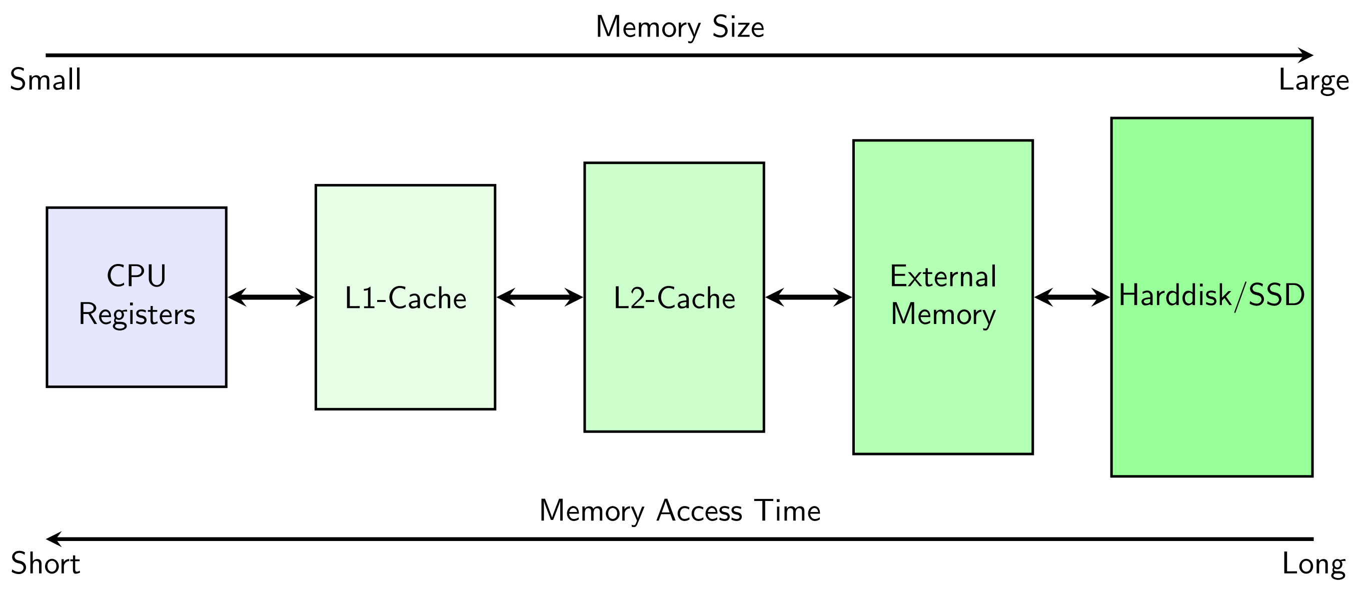
Programs access a relatively small portion of their address space at any instant of time

- Temporal locality: If an item is referenced, it will tend to be referenced again soon
- Spatial locality: If an item is referenced, items whose addresses are close by tend to be referenced soon

Ideal Strategy

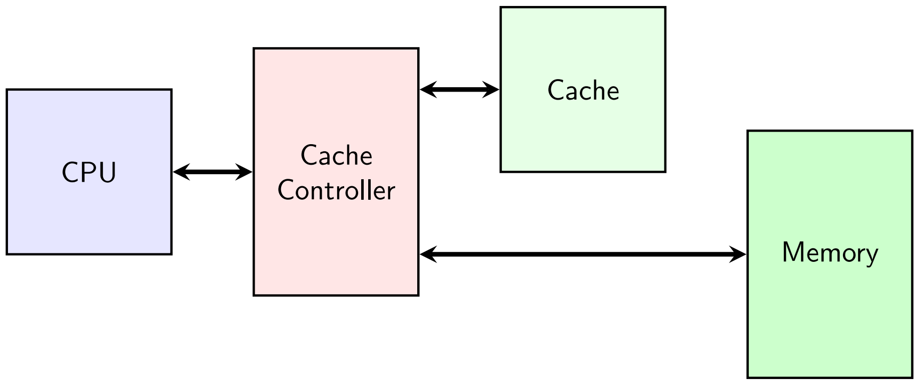
Keep the locations in the cache that are most often accessed

Memory Hierarchy



Cache - Memory Access

- Cache controller tries to find address in cache
- Miss implies additional penalty since memory transaction has to access main memory
- Access time very difficult to determine, because difficult to predict contents of cache
- In practice access time is non-deterministic



Hit Rate and Miss Rate

- Impossible to keep all locations in the cache
- Aim is to optimise the hit rate
  - Hit: Data that is referenced is located in the cache
  - Miss: Data that is referenced is not located in the cache

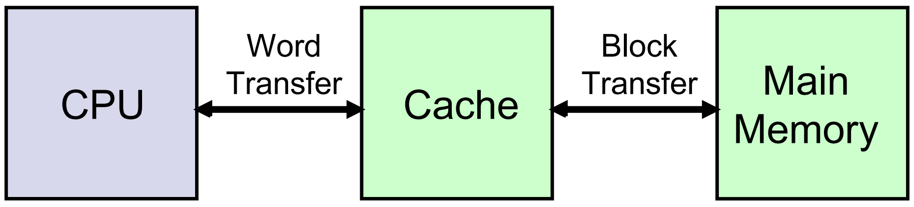
Hit Rate and Miss Rate

$$\text{hit rate} = \frac{\text{number of hits}}{\text{number of requests}}$$

$$\text{miss rate} = 1 - \text{hit rate}$$

Data Transfer to Cache

- Words are transferred between cache and processor
- Blocks (of multiple words, given by the block size) are transferred between cache and memory
  - Based on spatial locality principle
  - Sequential accesses are faster after first access (burst)



Cache Terms

- Cache Hit – required location is in cache
- Cache Miss – required location in not in cache
- Working Set – set of locations used by a program during an interval of time
- Compulsory (Cold) Miss – location has never been accessed before
- Capacity Miss – working set is too large
- Conflict Miss – two or more locations in working set map to the same cache entry

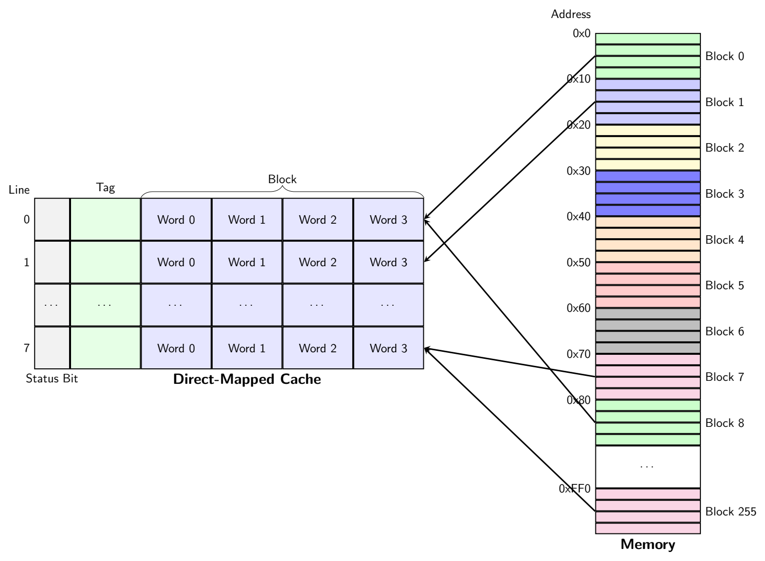
Cache Organisation

Caches organisation trade flexibility versus implementation costs

- Direct-mapped cache: each memory location maps onto exactly one cache entry in a dedicated cache line (set)
- *N*-way set-associative: each memory location can be mapped to *N* different entries (ways) in a dedicated set
- Fully-associative: each memory location can be stored anywhere in the cache

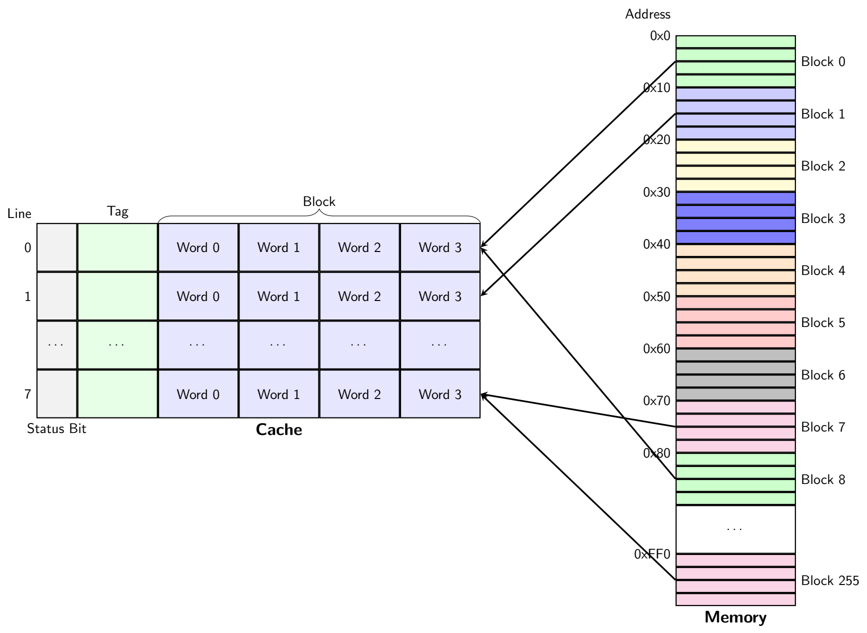
Direct Mapped Cache

- Cache lines: 8
- Block size: 4 words of 4 bytes
- Status bit: indicates if cache block is valid, invalid or modified
- Tag: indicates which memory block is in cache line



Direct Mapped Cache

- Memory:
  - 2^{12} bytes = 4096 bytes
  - 2^{10} words = 1024 words
- Cache:
  - Cache lines: 8
  - Block size: 4 words of 4 bytes



Cache Line = (Memory block number) modulo (Number of cache lines)

Why is the Tag needed in a Cache?

- Different blocks map to the same cache line ⇒ Need to know, which block is in the memory
- What happens, if memory location `0x3F4` is accessed and has not previously been stored in the cache?

Why is the Tag needed in a Cache?

- Different blocks map to the same cache line ⇒ Need to know, which block is in the memory
- What happens, if memory location `0x3F4` is accessed and has not previously been stored in the cache?

b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	1	1	1	1	1	1	0	1	0	0

0x07	7	1	0
Tag	Line	Word	Byte

Memory System

Why is the Tag needed in a Cache?

- Different blocks map to the same cache line ⇒ Need to know, which block is in the memory
- What happens, if memory location `0x3F4` is accessed and has not previously been stored in the cache?

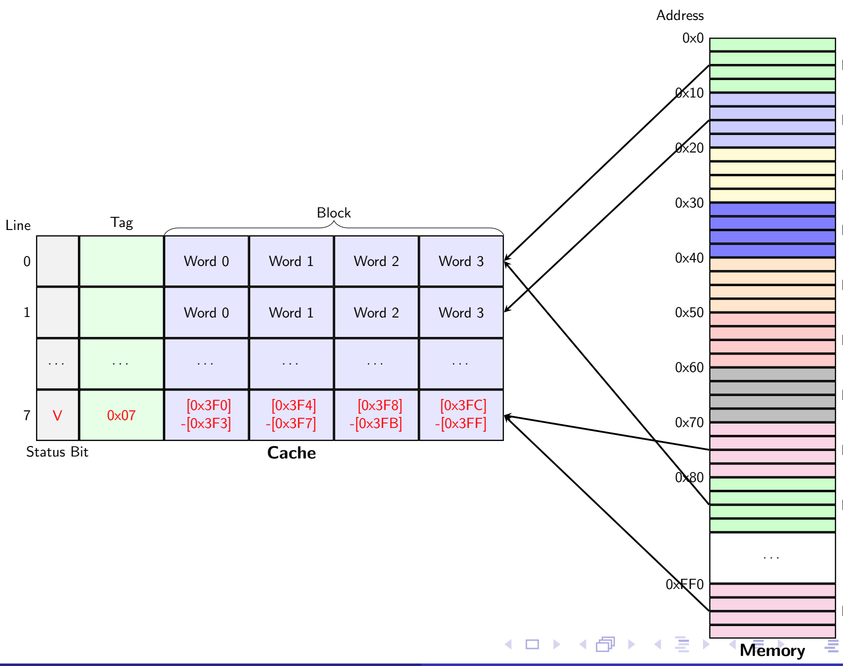
b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	1	1	1	1	1	1	0	1	0	0

0x07	7	1	0
Tag	Line	Word	Byte

The full block (addresses 0x3F0 to 0x3FF) containing the address 0x3F4 is loaded into the cache line 7. The tag 0x07 is stored to differentiate from other blocks that map onto the same cache line.

What happens, if memory location 0x03F4 is accessed, and has not previously been stored in the cache?

- The notation [0x3F0] - [0x3F3] means that the content of the memory locations 0x3F0, 0x3F1, 0x3F2, and 0x3F3 are stored in this word of the cache block.
- The tag has the value 0x07
- The status bit is set to V (Valid)

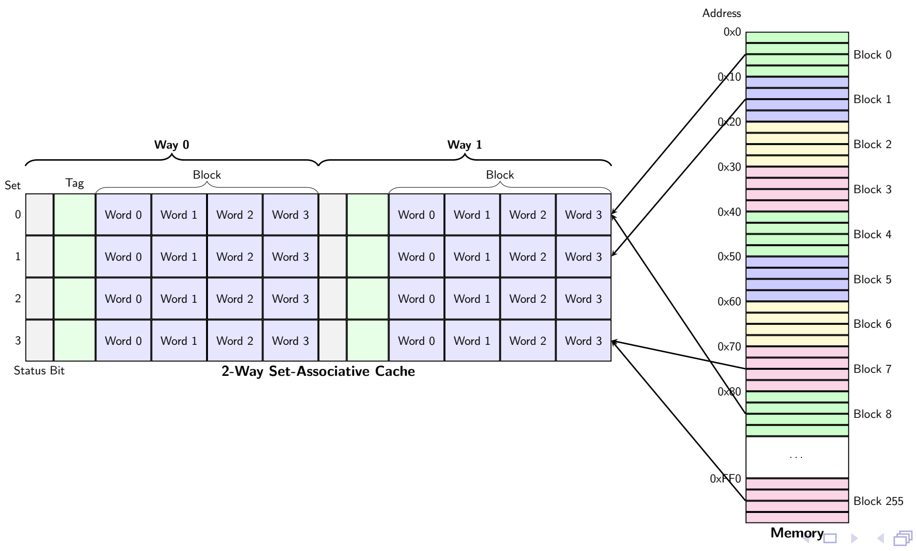


Conflict Misses in Direct-Mapped Caches

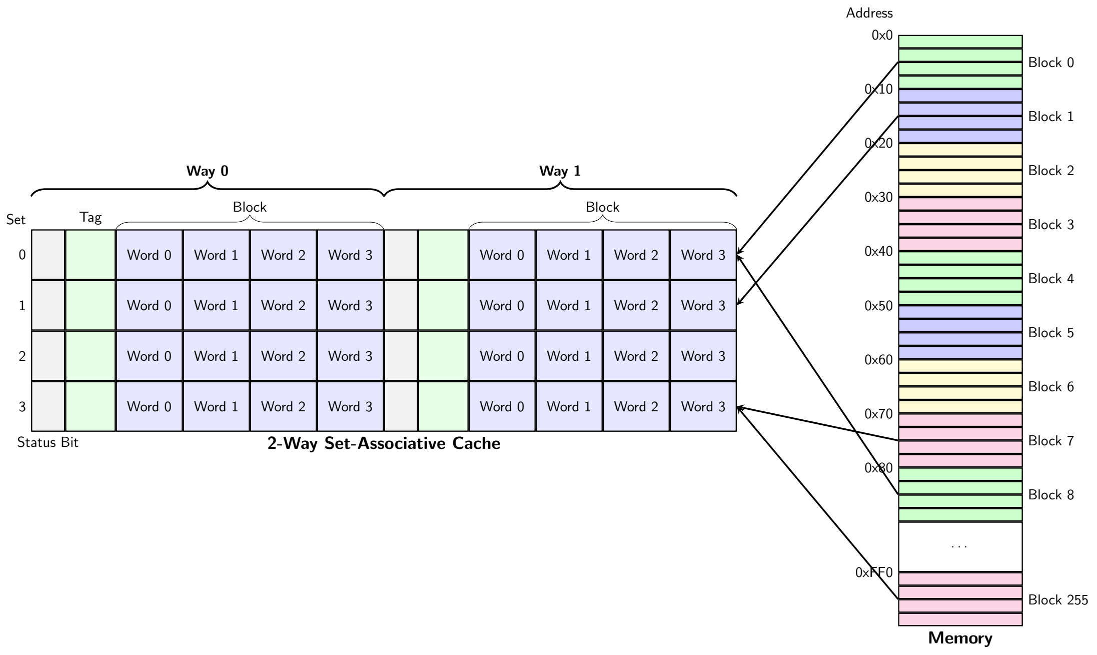
- Many locations map onto the same cache block
- Conflict misses are easy to generate:
  - Array a[] uses locations 0, 1, 2, . . .
  - Array b[] uses locations 1024, 1025, 1026, . . .
  - Both a[i] and b[i] map not only to the same cache line 0, but also to the same location in the cache block
  - Operation a[i] + b[i] generates conflict misses and a loop with this will have a very low performance (only cache misses)

2-Way Set-Associative Cache

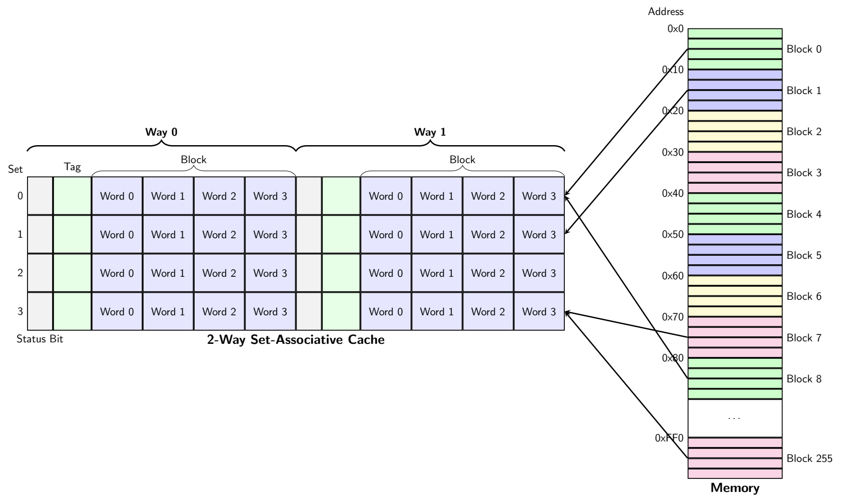
- Idea is to 'double' the memory space in each cache line (set), so that two conflicting memory locations can be mapped simultaneously to same cache set.
- 4 cache sets, where each set has 2 entries (ways) with blocks of size of 4 words of 4 bytes



2-Way Set-Associative Cache



2-Way Set-Associative Cache



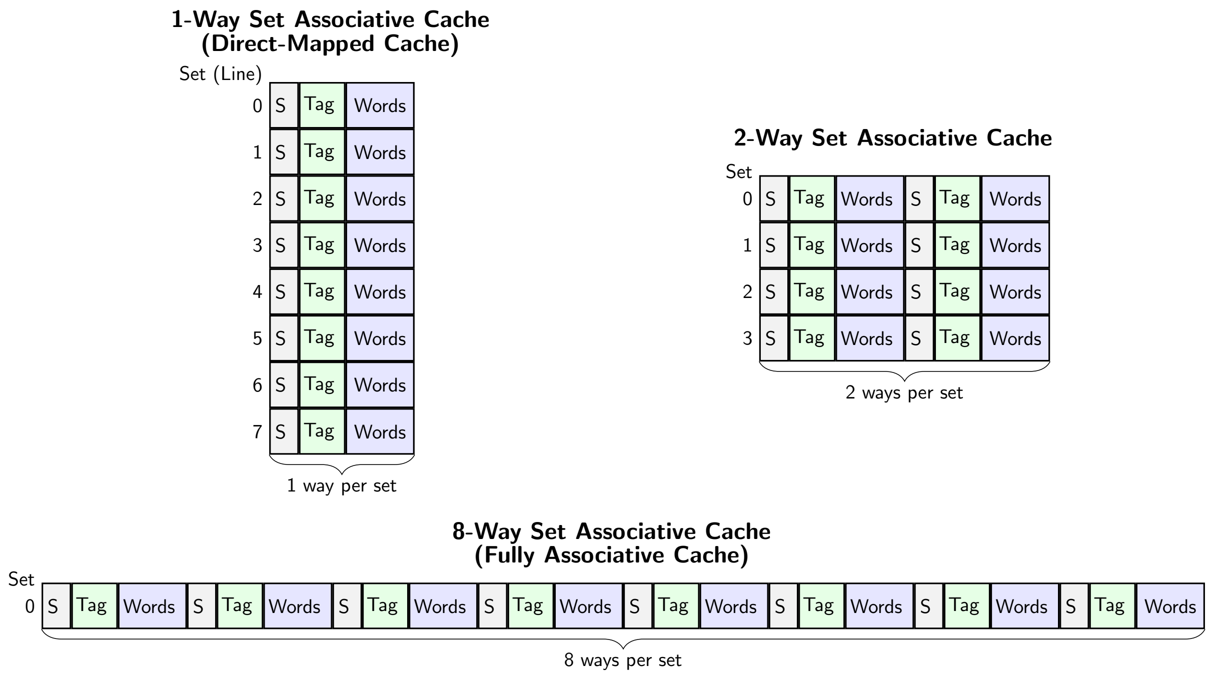
b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	1	1	1	1	1	1	0	1	0	0

0x0F Tag	3 Set	1 Word	0 Byte

Fully-Associative Cache

- There is a complete freedom, where to place a block in the cache
- But all blocks have to be searched for the correct tag pattern
- In order to have an acceptable performance, the tags must be searched in parallel $\Rightarrow$ Very high implementation costs

N-Way Set-Associative Caches


1-Way Set Associative Cache (Direct-Mapped Cache)
Set (Line)
0 S Tag Words
1 S Tag Words
2 S Tag Words
3 S Tag Words
4 S Tag Words
5 S Tag Words
6 S Tag Words
7 S Tag Words
1 way per set
2-Way Set Associative Cache
Set
0 S Tag Words S Tag Words
1 S Tag Words S Tag Words
2 S Tag Words S Tag Words
3 S Tag Words S Tag Words
2 ways per set
8-Way Set Associative Cache (Fully Associative Cache)
Set
0 S Tag Words S Tag Words S Tag Words S Tag Words S Tag Words S Tag Words S Tag Words S Tag Words
8 ways per set

Trade-Off: Caches

- There is no optimal cache
  - A large hit rate can be achieved by an *N*-way set-associative cache with large *N*
  - Implementation cost of a direct-mapped cache is lower compared to a set-associative cache with the same size
- Since both cost and performance are design constraints for embedded systems, the right trade-off has to be chosen

Cache: Write Operations

- Write-through: immediately copy write to main memory
  - May cause unnecessary memory communication
  - Memory has always a valid copy of the cache block
- Write-back: write to main memory only when location is removed from cache
  - Tries to minimise communication with memory
  - Memory may have an invalid copy of the cache block. Must be updated, when a cache block is replaced. Then this cache block is marked as modified (dirty).

Cache Replacement Strategies

- Replacement policy: strategy for choosing which cache entry to delete from the cache to make room for a new memory location.
- Two popular strategies:
  - Random
  - Least-recently used (LRU)
- In case of a modified cache entry in a write-back cache replacement means also to write the contents of the dirty cache entry back to the memory. Thus a cache miss can be expensive!

Caches in Real-Time Systems

- Caches are designed for average case performance
- Worst case performance is very difficult to predict

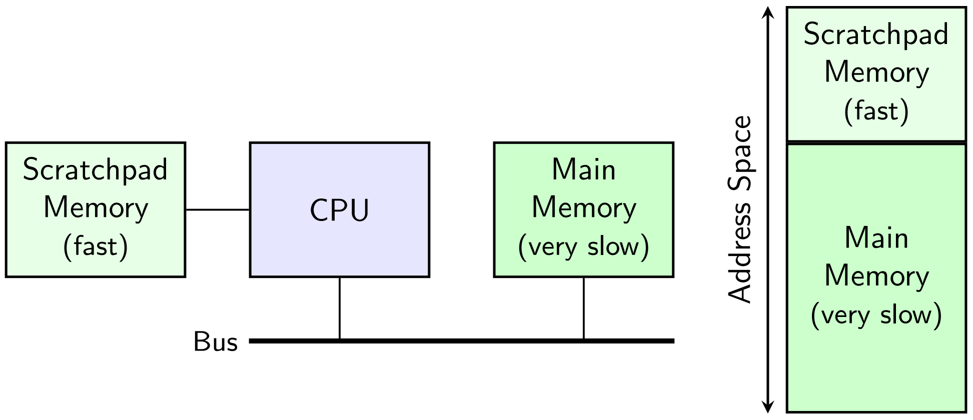
Caches are difficult to exploit caches in hard real-time systems!

“Predictable” Caches

- Some caches allow to “lock” the cache
- Cache content will not be changed
- Execution time is easy to calculate, if the content of the locked cache is known

Scratchpad Memory

- A scratchpad memory is an on-chip memory that is directly connected to the CPU and has a separated address space
- Predictable access time
  - (long) access time to main memory
  - (short) access time to scratchpad memory



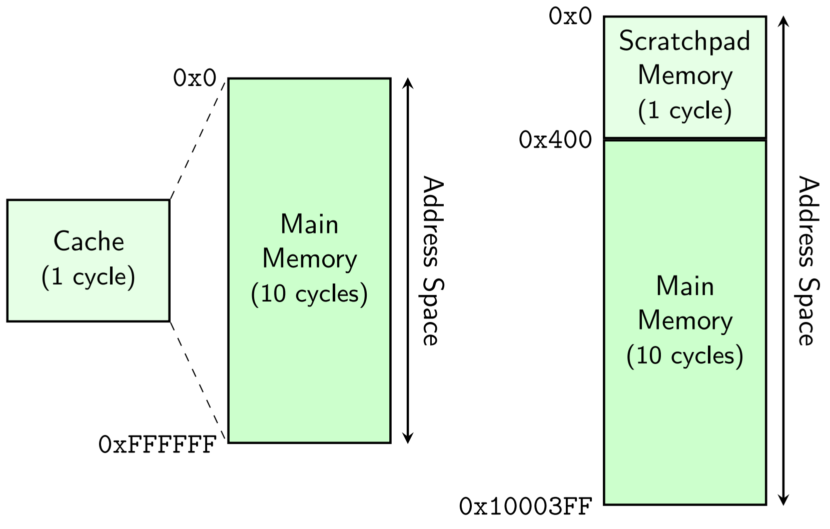
Comparison: Cache vs Scratchpad

Example Program:

```
ldw r2, 0x200
ldw r1, 0x1000
add r3, r1, r2    ; 1 cycle
stw r3, 0x2000
```

Execution Time in Cycles:

Case	Cache (HR=90%)	Scratchpad
Best	4	22
Average	6.7	22
Worst	31	22



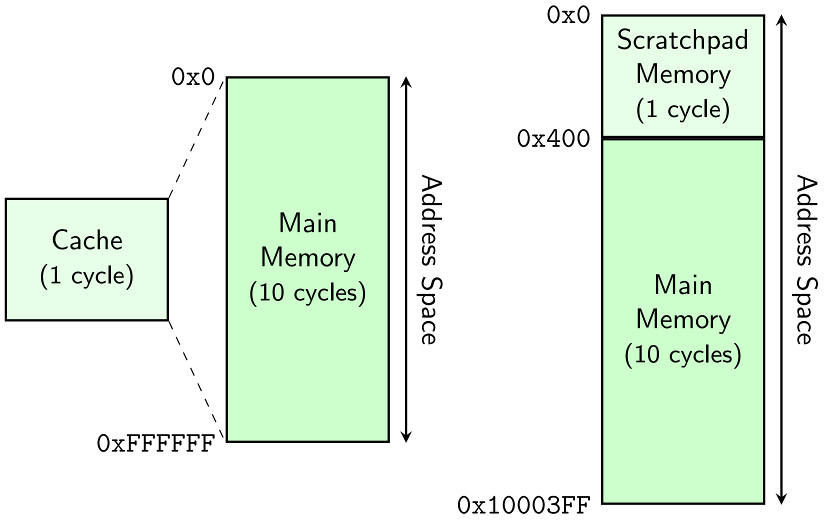
Comparison: Cache vs Scratchpad

Example Program:

```
ldw r2, 0x200
ldw r1, 0x1000
add r3, r1, r2    ; 1 cycle
stw r3, 0x2000
```

Execution Time in Cycles:

Case	Cache (HR=90%)	Scratchpad
Best	4	22
Average	6.7	22
Worst	31	22



Scratchpad memory increases worst case predictability, but has lower average case performance

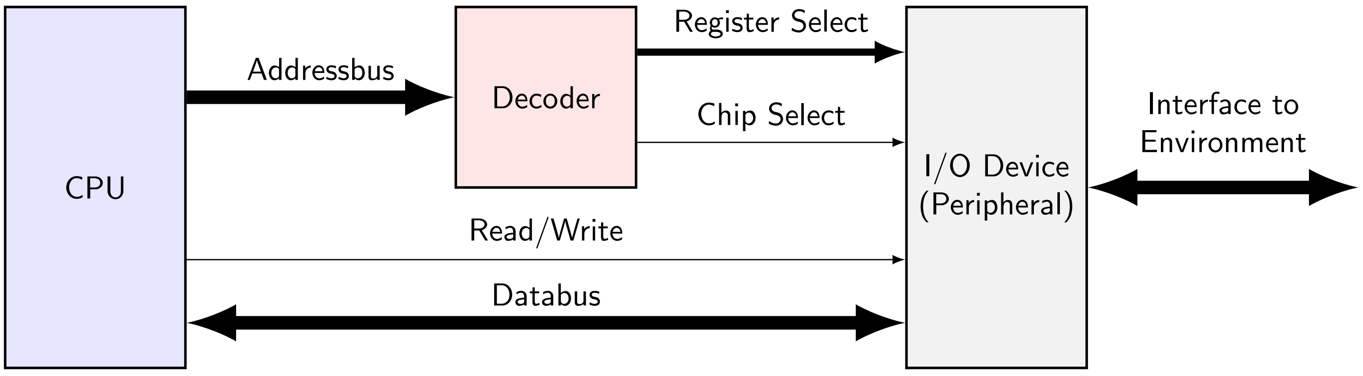
Outline

Input and Output (I/O) Devices

- Input/output devices are used to communicate with the environment
- Peripheral components can be connected to the processor by memory-mapped I/O
  - The components can be reached via a separate address space
  - Memory-mapped I/O requires extra hardware for address decoding

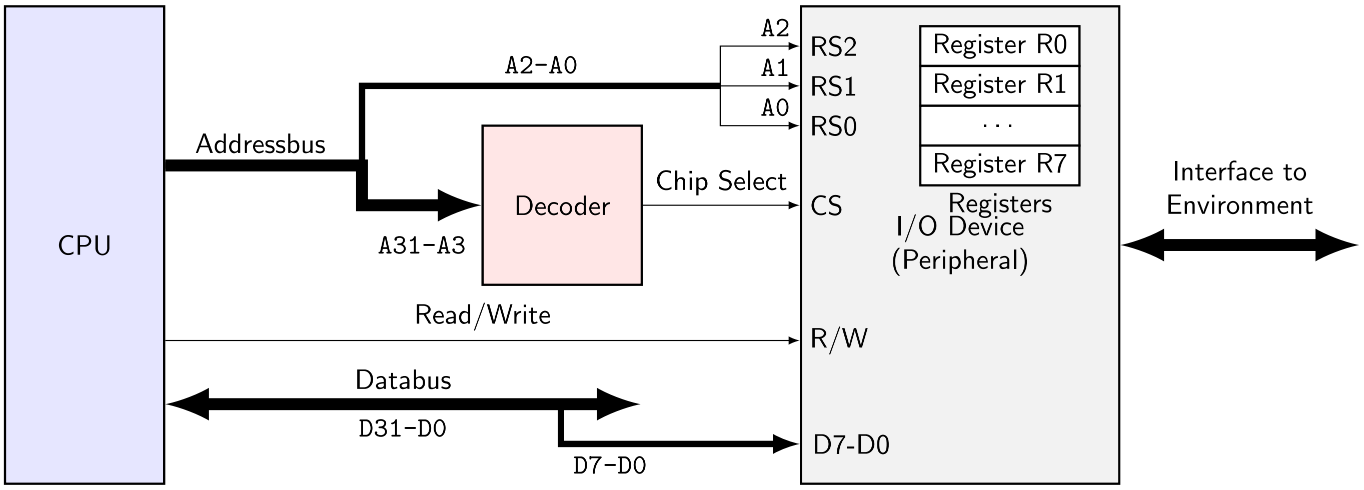
Memory-Mapped I/O

- 'Chip Select' has to be active, when the input of the decoder is a correct address
- Other address bits are used for 'Register Select'
- The decoder can be implemented with a small block of programmable logic or custom hardware (VHDL)



Example: Memory Mapped I/O

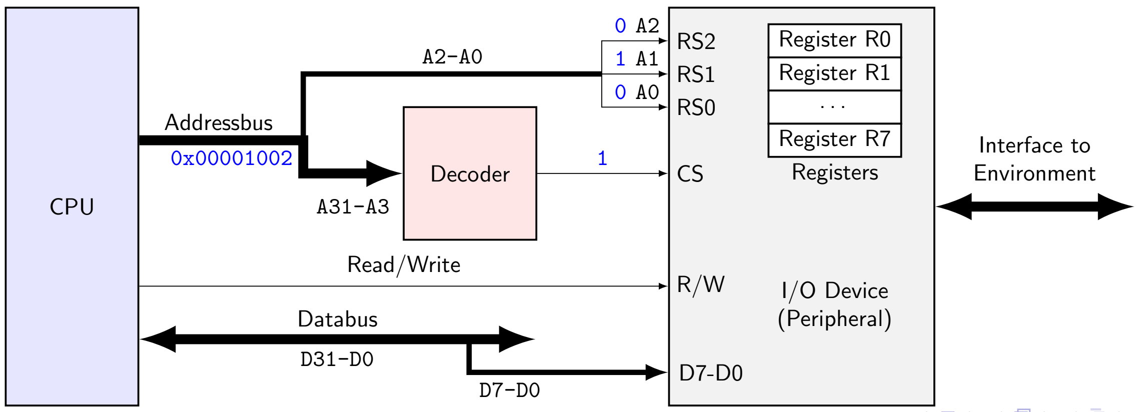
- Register 0 to 7 shall be accessible on addresses 0x00001000 to 0x00001007



Example: Memory Mapped I/O

```
movia r1, 0x1002
movi  r3, 0x08
stb   r3, (r1)
```

sets bit 3 and clears all other bits device register 2



Busy Wait I/O (Polling)

- Busy wait I/O is the most basic way to communicate with an I/O-device
- The processor waits until the I/O-device has completed its current task
- This method is also often called 'polling'
- Example:

```
while (button not pressed) {
  do nothing - just wait;
}
Continue after button has been pressed!
```

Busy Wait I/O (Polling)

- Busy wait I/O is the most basic way to communicate with an I/O-device
- The processor waits until the I/O-device has completed its current task
- This method is also often called 'polling'
- Example:

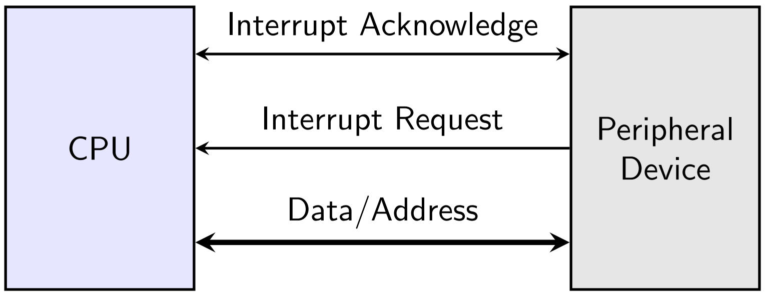
```
while (button not pressed) {
  do nothing - just wait;
}
Continue after button has been pressed!
```

Busy Wait I/O: Advantage and Disadvantages

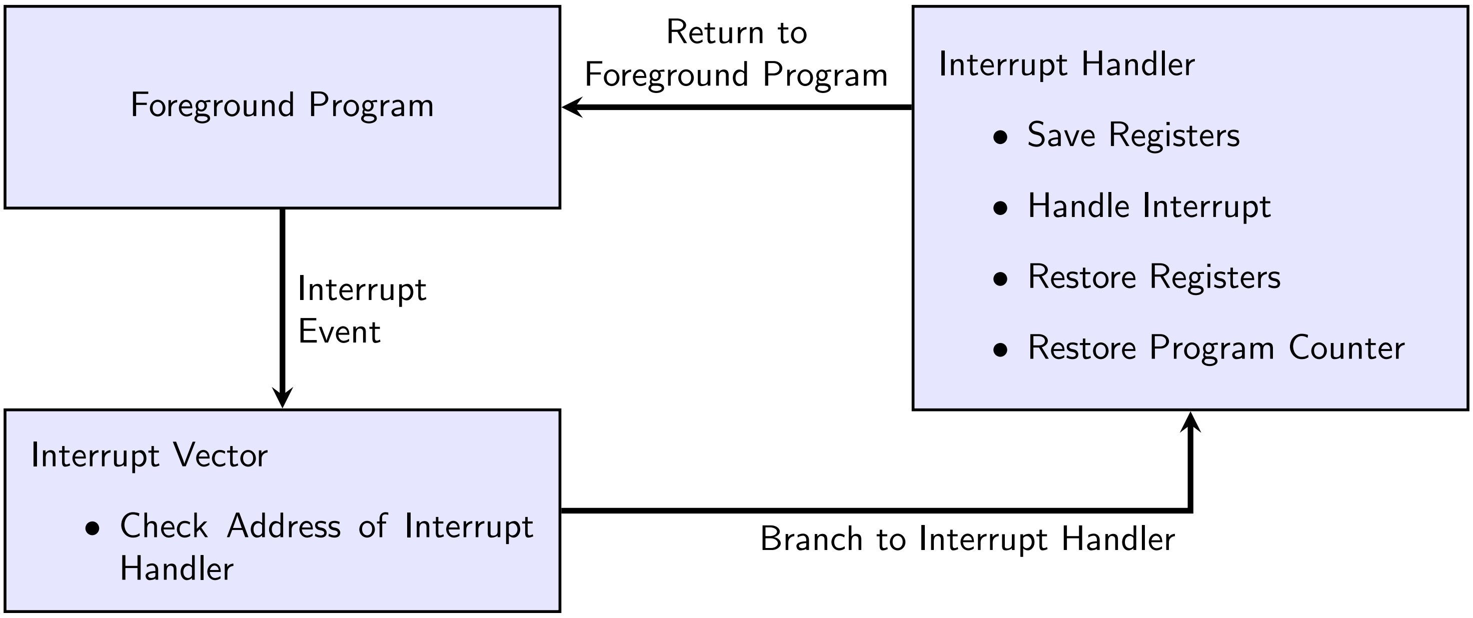
- Advantage: Busy Wait I/O does not require additional hardware resources
- Disadvantage: Processor cannot be used for other tasks during the waiting period

Interrupt I/O

- The idea of the interrupt is that the processor gets notified, when an input event is happening
- No need for the processor to actively wait for input event, but instead processor can do useful work
- Once an IO-event happens, processor is notified by an interrupt request
- If no other interrupt is handled, the processor acknowledges the interrupt, and executes an interrupt service routine, which corresponds to the identity or level of the interrupt.



Interrupt Mechanism



Receive-Send with Busy Wait I/O

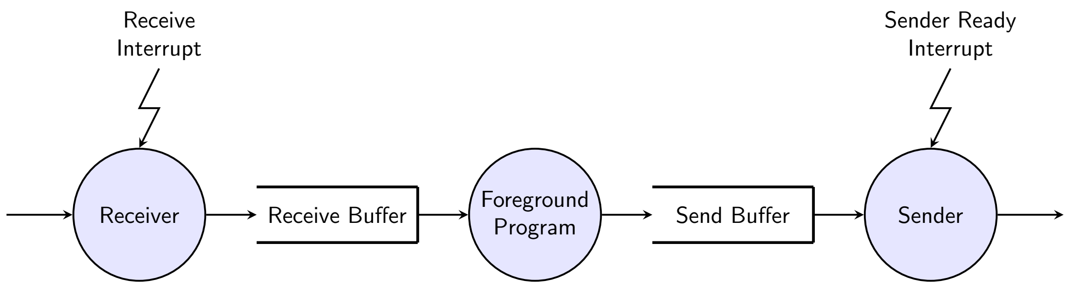
- Assume a program that as part of its duties receives characters and sends them further to another device
- Solution with Busy Wait I/O

```
loop
  Wait for input from receiver;
  Operate on input;
  Send modified input when sender is ready;
end loop;
```

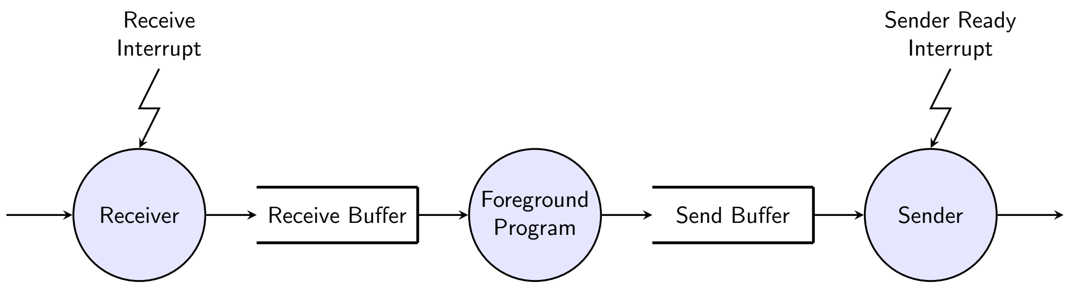
- Program cannot do anything else than wait, if input has not arrived at receiver, or when sender is not ready
- Inefficient use of resources

Receive-Send with Interrupt

- Interrupt allows parallelisation of duties, so that system can do several things at the same time
- Following tasks can be parallelised:
  - Wait for new data from receiver and if received store it in receive buffer (interrupt)
  - Consume data from receive buffer, operate on it and store it in send buffer (foreground program)
  - Wait for the sender to be ready and the availability of data in the send buffer, and the send data (interrupt)



Receive-Send with Interrupt



- Buffers enable to decouple activities, so that concurrent execution is possible
- System can do other thing while waiting for receiver or sender
- Buffer is needed to store elements
- Size of buffer must be chosen carefully
  - too small ⇒ buffer overflow
  - too large ⇒ too expensive design

Typical Embedded Design Problems

- Embedded systems are inherently parallel (concurrent), since they interact with heterogeneous environment
  - Parallelisation allows for a faster processing, since work can be done in parallel
  - Waiting times can be avoided
- The need for buffers is a logical consequence of parallelisation

Design Challenge

System designer needs to find the right amount of parallelisation and the right buffer size

Interrupts and Debugging

- Difficult to debug programs, which use interrupts
- Foreground program can be interrupted at any time ⇒ Very difficult or even impossible to reproduce the exact situation, where the error occurred
- Hard to repeat errors and debug program

Interrupts and Debugging

- Difficult to debug programs, which use interrupts
- Foreground program can be interrupted at any time ⇒ Very difficult or even impossible to reproduce the exact situation, where the error occurred
- Hard to repeat errors and debug program

Very careful design discipline is required when dealing with interrupts!

Prioritised Interrupts

- Some CPUs (as Nios II) support several interrupt levels by their hardware
- Otherwise extra hardware (priority decoder) can be used to create several levels of interrupt
- Interrupt with priority lower than current priority is not recognised until the pending interrupt is served (Masking)
- The highest-priority interrupt is called non-maskable interrupt and never masked

Sources of Interrupt Overhead

- Handler execution time
- Interrupt mechanism overhead
- Register save/restore
- Pipeline-related penalties
- Cache-related penalties

Sources of Interrupt Overhead

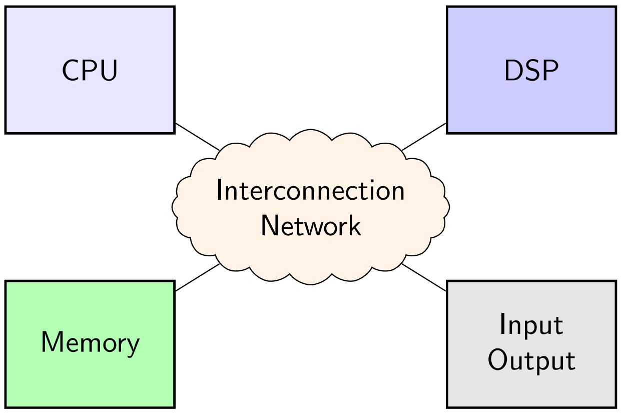
- Handler execution time
- Interrupt mechanism overhead
- Register save/restore
- Pipeline-related penalties
- Cache-related penalties

Interrupt vs Busy Wait I/O

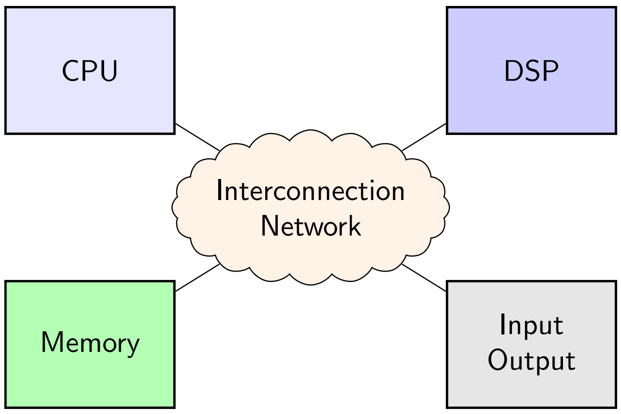
Interrupt is not always better than polling

Outline

Interconnection Networks



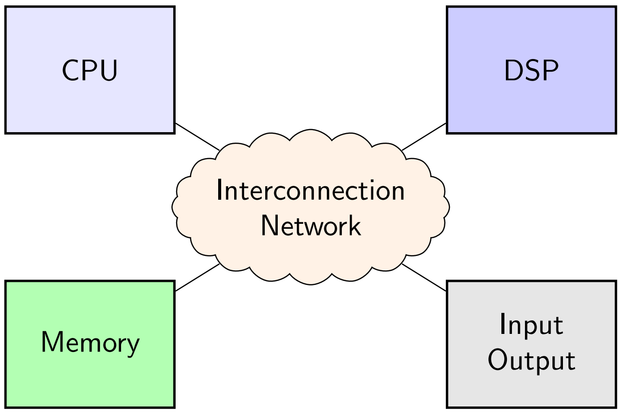
Interconnection Networks



Ideally the interconnection network provides

- full connectivity between components
- negligible delay;
- non-blocking connections;
- unlimited bandwidth; and
- scalability.

Interconnection Networks

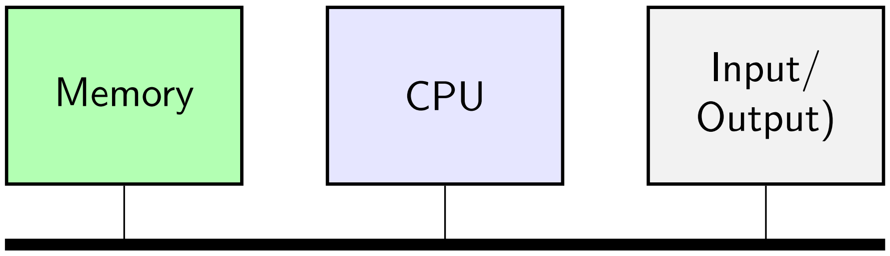


Ideally the interconnection network provides

- full connectivity between components
- negligible delay;
- non-blocking connections;
- unlimited bandwidth; and
- scalability.

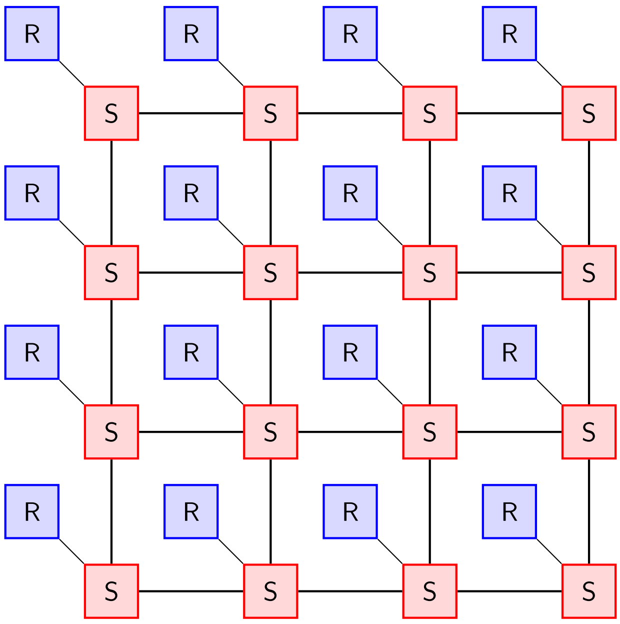
But in practice, it will not be possible to achieve such a network at a reasonable cost.

Bus Architectures



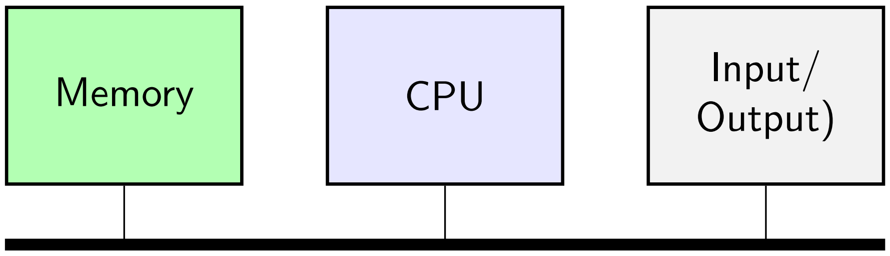
- Bus provides logical communication link which all processing elements can access
- Only one bus master can take control of bus at each instance of time

Network-on-Chip Architectures



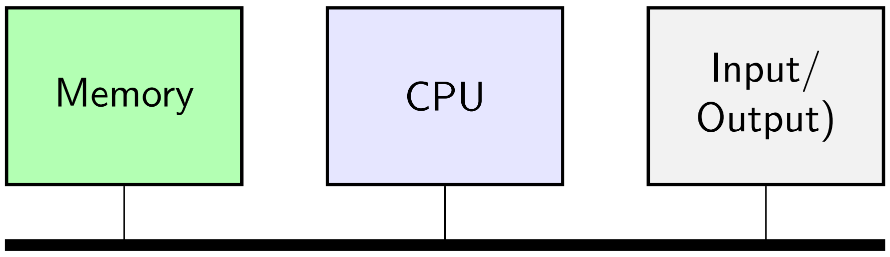
- Resources (R) contain computing resources
    - Typical resources would are bus-based processor memory architectures, which connect to NoC via network interface (NI)
- Packets are sent from the resources using the NI to other resources via communication links and switches (S)
- The topology of the network and the capability of the switches and communication channels determines the capacity of the network

Bus Architecture



- Dominating architecture for embedded system interconnection networks
- Devices are connected via a single shared communication channel

Bus Properties

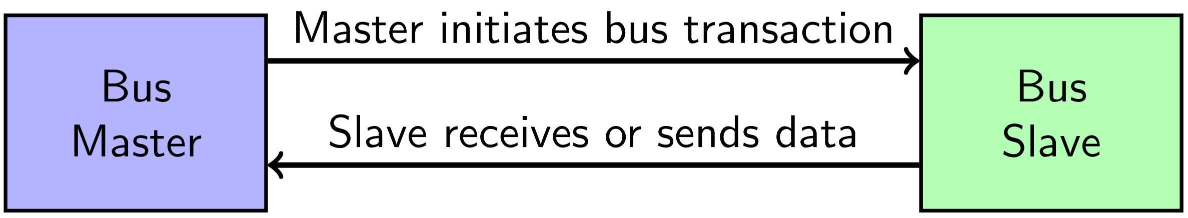


- Serialisation
  - Only one component can send a message at any given time
  - Total order of messages
- Broadcast
  - Bus master can send message to several other components without an extra cost
  - One Sender - Many Receivers

Buses: Transactions, Messages and Cycles

- Bus transaction consists of a sequence of messages which together form a transaction
- Bus message is formed by its logical unit of information, e.g. a read message contains an address and control signals for read access
- Message requires a number of cycles to be transferred from sender to receiver over the bus

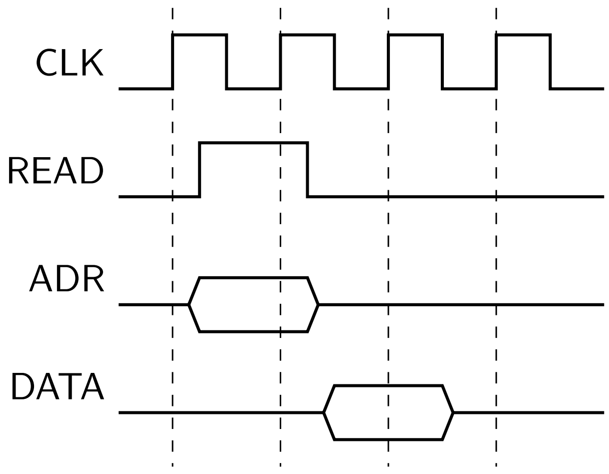
Bus Master and Bus Slave


Bus Master
Master initiates bus transaction
Slave receives or sends data
Bus Slave

- Bus transaction is divided into two phases:
  - Bus master issues a request for a bus transaction by sending a message by assigning an address and the corresponding control signals
  - Bus slave reacts by either sending data to or receiving data from the bus master

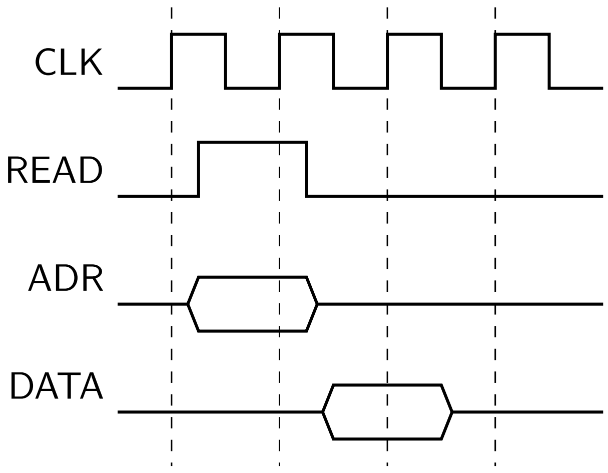
Synchronous Bus

- requires a clock
- enables a protocol that expects that data is available after certain clock cycles



Synchronous Bus

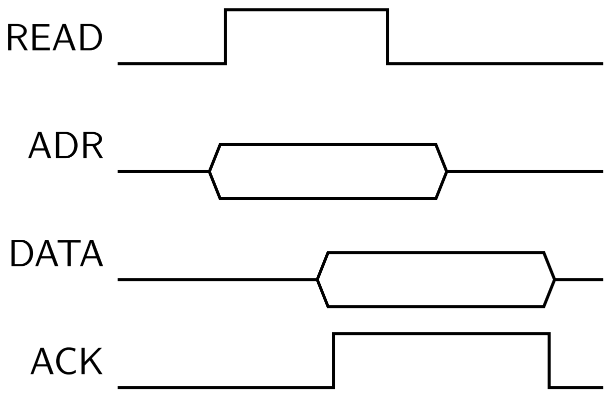
- requires a clock
- enables a protocol that expects that data is available after certain clock cycles



- Pros
  - requires very little extra logic and can run very fast
- Cons
  - devices are required to share the same clock
  - potential problem of clock skew: clock edge will not arrive at the same time at all components due to different physical distances from clock source

Asynchronous Bus

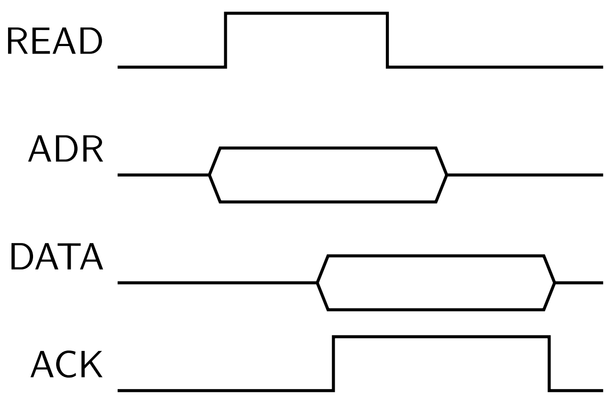
- does not require a clock
- many different devices can be connected



1. master assigns address on bus and activates READ when address is stable
2. slave activates data signals with requested data and activated ACK if the data is stable
3. master deactivates READ when data is read
4. slave deactivates ACK

Asynchronous Bus

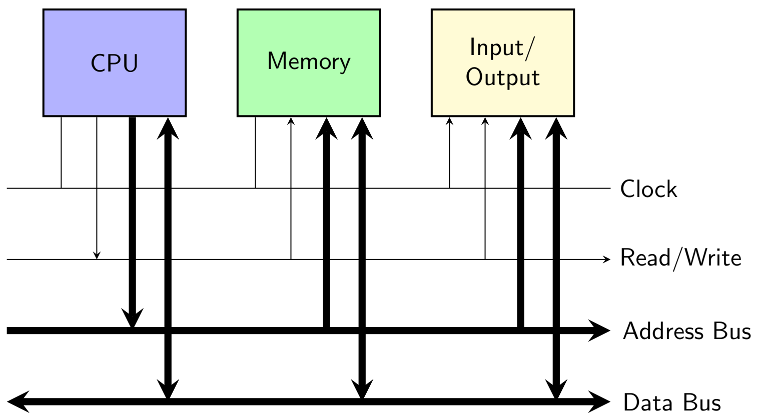
- does not require a clock
- many different devices can be connected



- Pros
  - potential problem of clock skew is avoided
  - components that experience data-dependent delay can be connected
- Cons
  - requires handshake protocol
  - difficult to design fast asynchronous buses

Physical View of Bus Architecture

- On more abstract level, bus can be seen as single logical bidirectional communication link
- On a more detailed level, bus is composed of a unidirectional address bus, a bidirectional data bus and several control signals



Bus Arbitration

- In a simple embedded system with only one processor, one memory, and a simple input/output device, the processor is the only bus master
- But what happens, if there is more than on bus master?

Bus Arbitration

- In a simple embedded system with only one processor, one memory, and a simple input/output device, the processor is the only bus master
- But what happens, if there is more than on bus master?

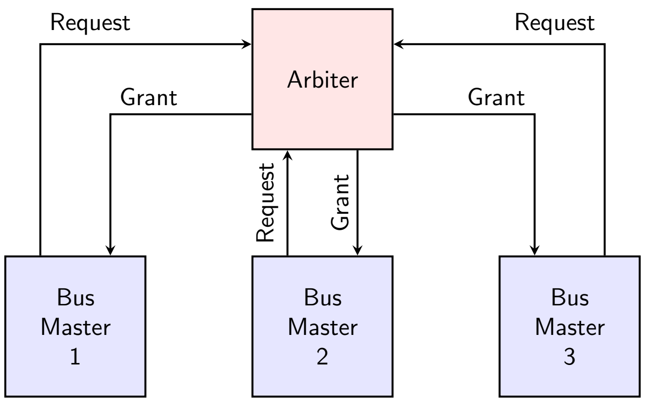
Need for Arbitration

If there is more than one bus master, an arbitration protocol is required, which secures that only one bus master accesses the bus at each time instance!

Bus Arbitration

- Only one bus master can use bus at a given time
- Policy and protocol required to decides, which bus master is granted right to control bus
- Arbiter implements this protocol
- Additional connections required for Bus Request and Bus Grant
- Only a bus master that has received a grant from the bus master is allowed to take control of the bus

Bus Arbitration



1. Bus master issues a request to the arbiter by activating its Request signal
2. Arbiter selects one bus master by activating the corresponding Grant signal
3. When bus transaction is finished, bus master deactivates its Request signal

Bus Arbitration: Priority vs Fairness

Arbitration scheme tries to balance two important factors.

- Priority: There might be bus masters, which control functions that are more important. In particular, in safety-critical systems, it is of extreme importance that these devices can execute their functions in time.
- Fairness: All devices should get access to the bus. The situation that a device is totally locked out from a bus shall be avoided.

Fairness

Different degrees of fairness

- No fairness: There are devices in the system, which requests might never be served
- Weak fairness: A request for a device in a system will be eventually served
- Strong fairness: Requests from different devices will be served equally often
- Weighted strong fairness: Each device has a weight, and the number of times a device is served is corresponding to its weight
- FIFO fairness: Requests of different devices are served in the timely order the requests have been made

Fairness

Different degrees of fairness

- No fairness: There are devices in the system, which requests might never be served
- Weak fairness: A request for a device in a system will be eventually served
- Strong fairness: Requests from different devices will be served equally often
- Weighted strong fairness: Each device has a weight, and the number of times a device is served is corresponding to its weight
- FIFO fairness: Requests of different devices are served in the timely order the requests have been made

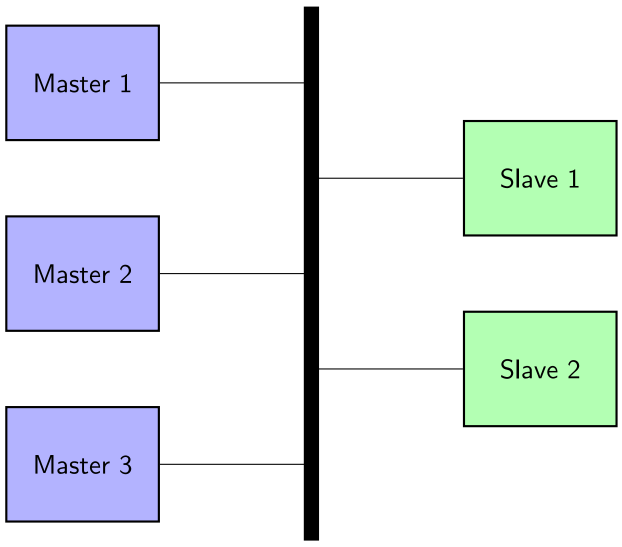
There is no best fairness! The right amount of fairness depends on the application and its design constraints!

Original Bus Summary

- Very versatile interconnection network that enables to easily add new devices as long as they follow the same bus standard
- Important properties: serialisation and broadcast
- Low cost implementation using only a single shared communication link
- Limited scalability ⇒ bus can easily become bottleneck in embedded system

Original Bus is Bottleneck!

- Only one master can take control of the bus
- If one master accesses a slave, no other master can access any other slave

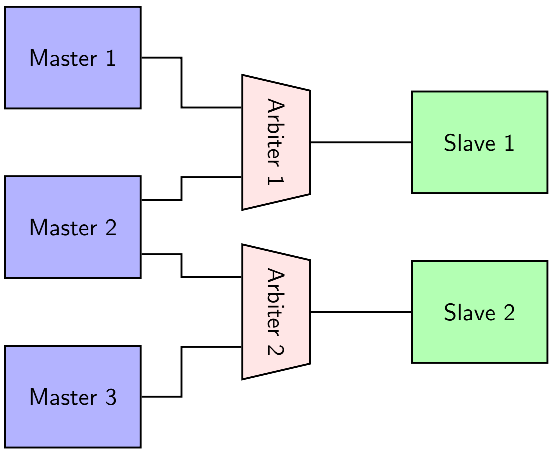


Slave-Side Arbitration

- Slave-side arbitration removes bottleneck of original bus by using dedicated arbiter for each slave
- Enables large communication bandwidth

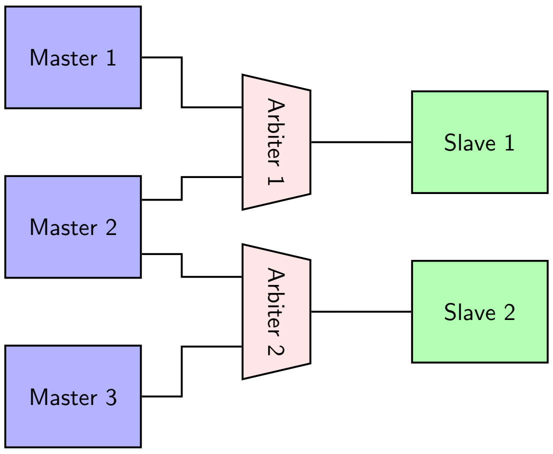
Slave-Side Arbitration

- Slave-side arbitration removes bottleneck of original bus by using dedicated arbiter for each slave
- Enables large communication bandwidth



Slave-Side Arbitration

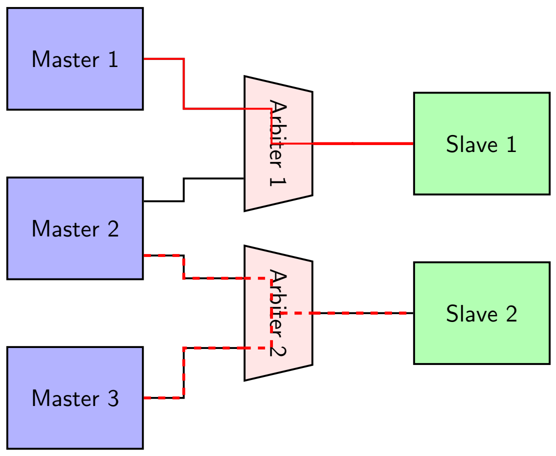
- Slave-side arbitration removes bottleneck of original bus by using dedicated arbiter for each slave
- Enables large communication bandwidth



- Master can access a slave, if no other master accesses this slave

Slave-Side Arbitration

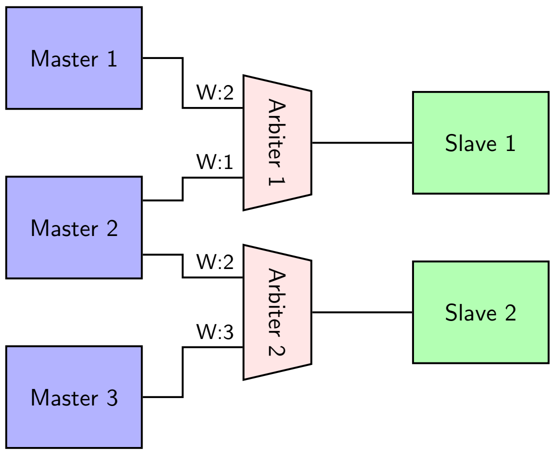
- Slave-side arbitration removes bottleneck of original bus by using dedicated arbiter for each slave
- Enables large communication bandwidth



- When master 1 accesses slave 1, either master 2 or master 3 can still access slave 2

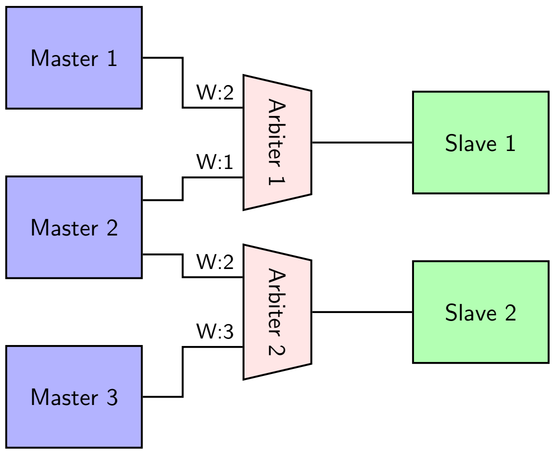
Slave-Side Arbitration with Weighted Round-Robin

- Altera Switch Fabric uses slave-side-arbitration with weighted round-robin
- Each master is assigned a weight (integer number)
- Master gets at least the number of accesses corresponding to its weight
- If a master does not use it shares another master might be able to use these shares (no general rule, depends on implementation)



Slave-Side Arbitration with Weighted Round-Robin

- Slave 1
  - Weight Master 1: 2
  - Weight Master 2: 1
  - Possible Rounds: 1-1-2, 2-1-1, 1-2-1, ...
- Slave 2
  - Weight Master 2: 2
  - Weight Master 3: 3
  - Possible Rounds: 2-2-3-3-3, 3-2-2-3-3, 3-3-3-2-2, ...



Slave-Side Arbitration Summary

- can have significantly higher performance than original bus
- interconnection network can grow very quickly and may result in very high hardware costs for larger systems
- very promising solution for high-performance systems

Slave-Side Arbitration Summary

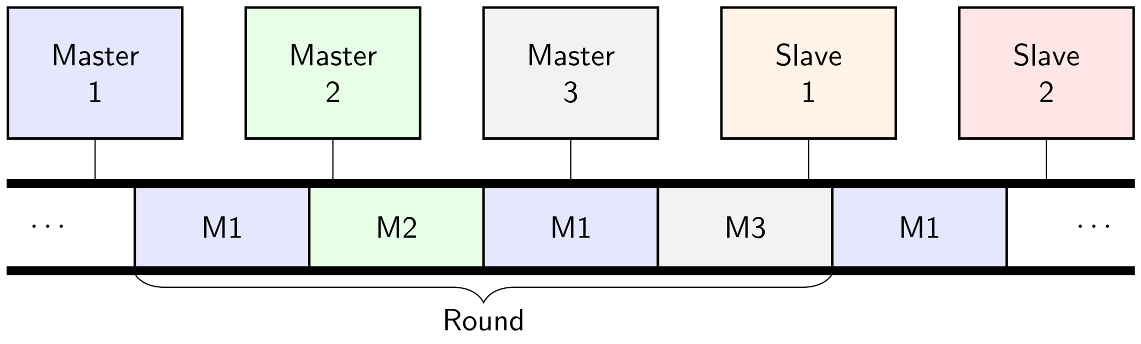
- can have significantly higher performance than original bus
- interconnection network can grow very quickly and may result in very high hardware costs for larger systems
- very promising solution for high-performance systems

Original Bus and Slave-Side Arbitration

- Combination of slave-side arbitration with original buses can be well-motivated
- Slave-Side Arbitration: High-performance part of system
- Original bus: Low-performance part (e.g. Input/Output)
- Bus bridge to connect both buses

Time-Division Multiplex Bus

- Bus access is divided into time slots
- Time slots are statically assigned to different bus masters, which only can send or receive during these time slots
- Guaranteed quality-of-service for all bus masters due to reserved resources
- No need for additional arbiter



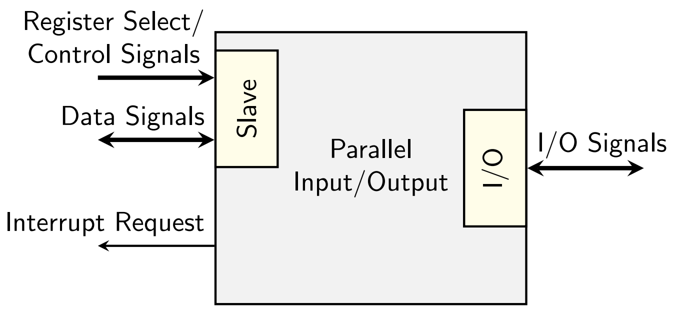
Outline

Peripheral Components

- Input/output devices are just a special case of peripheral components
- Peripheral components
  - are used to off-load the processor and to execute specialised functionality
  - are connected to the processor using memory mapped-I/O
  - always have a slave port, but can also have master ports

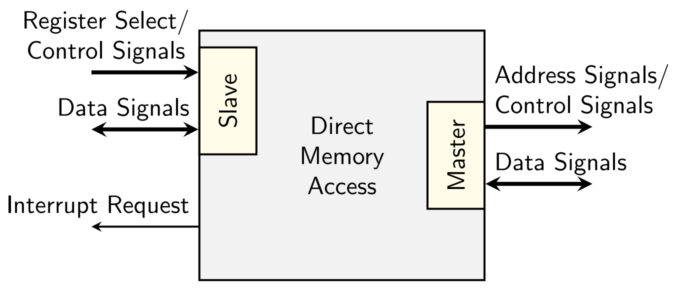
Parallel Input/Output Peripheral (PIO)

- provides several parallel input/output ports
- ports can be programmed as unidirectional input, output or bidirectional ports
- has a slave port
- can be configured to request an interrupt on activity at input port



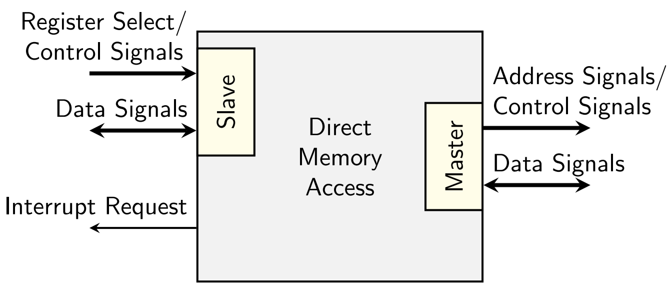
Direct-Memory Access (DMA)

- offloads processor by handling data transfer from I/O-device to memory, so that processor can do other useful work
- has slave port, which is used to set up the DMA for the transactions
- has master port, which DMA uses to conduct the transactions
- has interrupt request output to inform processor that transfer is finished



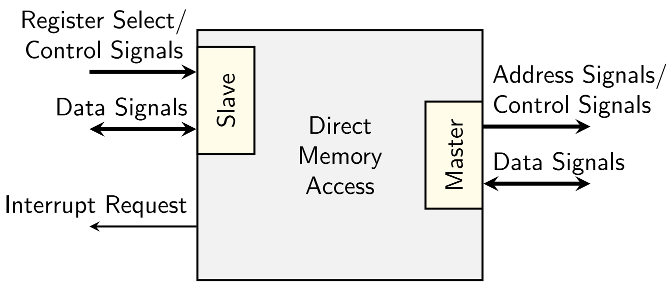
Direct-Memory Access (DMA)

1. CPU configures DMA for I/O transfer by writing to slave port of DMA
   - source address, destination address, block length corresponding to data transfer
2. DMA conducts I/O transfer using its master port
3. DMA issues interrupt request after I/O transaction



Direct-Memory Access (DMA)

1. CPU configures DMA for I/O transfer by writing to slave port of DMA
   - source address, destination address, block length corresponding to data transfer
2. DMA conducts I/O transfer using its master port
3. DMA issues interrupt request after I/O transaction

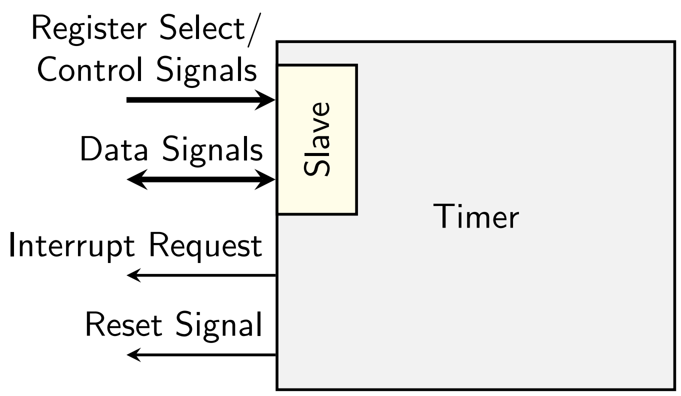


DMA will compete on bus resources!

- DMA acts as bus master and can block CPU and other masters from accessing the bus!
- I/O-transfer might consume significant bus bandwidth for some time, and can affect responsiveness of processor, if not carefully designed!

Timer Peripheral

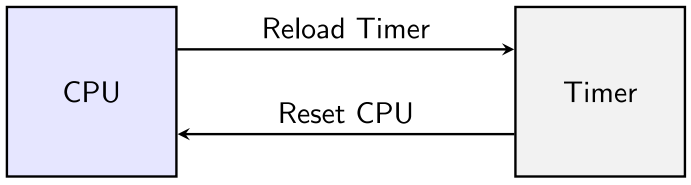
- A timer is used to be able to measure time in an embedded system
- Implemented as counter that can be loaded with a number and counts downwards (or upwards)
- Optional interrupt request when counter reaches zero
- Optional reset request when counter reaches (for watchdog timer)



Timer is key prerequisite for real-time operating systems

Watchdog Timer

- Watchdog timer is used to check periodically, if system is still working
- Timer is started with preloaded value and counts downwards
- If timer reaches zero, a reset request is issued (shall never happen)
- Processor must periodically (re)set the timer
- If CPU gets deadlocked and fails to reset the timer, the system is reset or put in safe state



Summary

- Embedded computing hardware platform consists of many different parts, which need to interact
  - Processor, memory system, interconnection network, peripheral components
- Efficient system requires a suitable composition of components
- Components needs to be well-balanced, otherwise bottleneck can be easily created
- Platforms for high average case performance and platforms for hard real-time systems will be different

Summary

- Embedded computing hardware platform consists of many different parts, which need to interact
  - Processor, memory system, interconnection network, peripheral components
- Efficient system requires a suitable composition of components
- Components needs to be well-balanced, otherwise bottleneck can be easily created
- Platforms for high average case performance and platforms for hard real-time systems will be different

Embedded computing platform contains also software!

- Software in form of hardware drivers and operating systems are part of the embedded computing platform
- Hardware platform and software platform need to operate together!