Embedded Systems - HS2022
Marvin Steinkellner

version of 30.01.23

Decimal	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

$1KiB = 2^{10}\,Bytes = 2^{13}\,Bits$
$1MiB = 2^{20}\,Bytes = 2^{23}\,Bits$
$1GiB = 2^{30}\,Bytes = 2^{33}\,Bits$

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

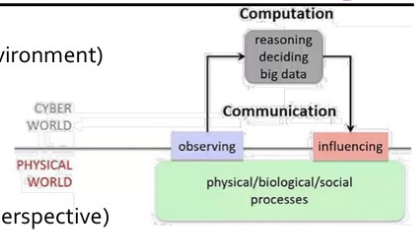
$0xbeef = 15 \cdot 1 + 14 \cdot 16 + 14 \cdot 16^2 + 11 \cdot 16^3 = 48879$
$= 0b1011111011101111$

Embedded Systems

①

Characteristics:

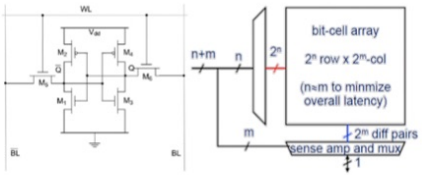
- Often reactive (execution occurs at a pace determined by environment)
- Often must meet hard real-time constraints
- Often specialised to application
- Must be efficient:
  - Cost & weight efficient
  - Energy, memory & runtime efficient (from a worst-case perspective)



Storage

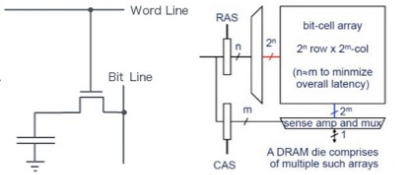
SRAM:

- Very fast volatile memory for low volumes (e.g., registers or cache)
- Read procedure:
  1. Pre-charge all bit-lines to average voltage
  2. Decode address ($n + m$ bits)
  3. Select row of cells using n single-bit word lines (WL)
  4. Selected bit-cells drive all bit-lines BL (2^m pairs)
  5. Sense difference between bit-line pairs and read out
- Write procedure:
  - Select row and overwrite bit-lines using strong signals



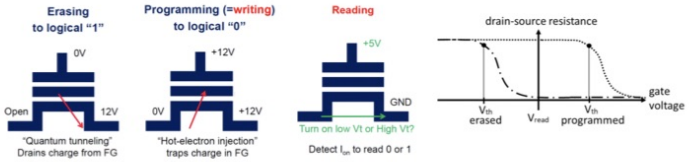
DRAM:

- Higher density than SRAM but with slower access speed
- Capacitors discharge so cells need to be refreshed periodically



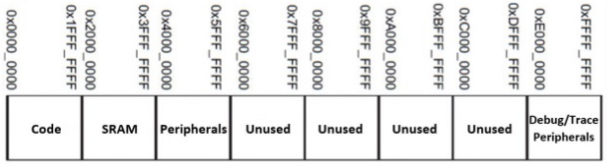
Flash memory:

- Non-volatile electrically programmable storage
- The transistor has a floating gate which can trap electrons, modifying the threshold voltage
- There are 2 common types:
  - NAND: Small cells, high density and low power for mass storage
  - NOR: Fast random access for code storage



Memory map (MSP432):

- The available address space is used to access memories, to address the peripheral units and to access debug and trace information
- The address space is partitioned into zones, each with a dedicated use



Input, Output & Communication

UART protocol: LSB first

- Serial communication protocol via a single signal
- Sender and receiver agree on a symbol rate (bitrate or baud rate)
- Idle state is high
- Structure of a packet:

1 start bit	6-9 data bits	1 parity bit	1-2 stop bits

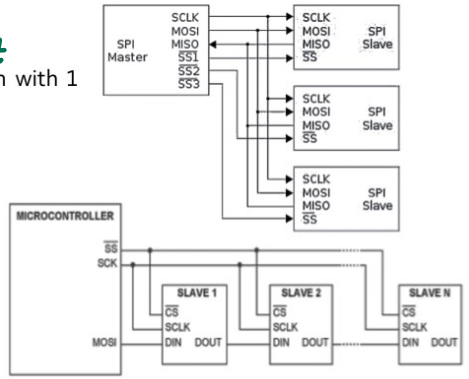
- The receiver runs an internal clock whose frequency is an integer multiple of the chosen bitrate
- The signal is always sampled midway through a bit (detection of start bit facilitates this synchronisation)

UART sender-receiver frequency mismatch: → exercise 2!

- Required clock frequency $f_{req} = \text{Baud rate} \cdot \text{Ticks per bit} = BN_s \approx \frac{f_{source}}{d}$
- Division factor $d = \text{round}\left(\frac{f_{source}}{f_{req}}\right)$
- Actual sender baud rate $r_s = \frac{f_{source,s}}{d \cdot N_s}$
- Actual receiver baud rate $r_r = \frac{f_{source,r}}{d \cdot N_s}$
- The k^{th} symbol is transmitted during $t \in \left(\frac{k-1}{r_s}, \frac{k}{r_s}\right)$ and sampled by the receiver at $t = \frac{k-0.5}{r_r}$
- If the receiver's source clock is faster than the sender's source clock, stop bit 1 must be sampled no earlier than $\frac{k_{s1}-1}{r_s} + \frac{1}{r_r \cdot N_s} \leq \frac{k_{s1}-0.5}{r_r}$. The additive term accounts for signal stability requirements.
- If the receiver's source clock is slower than the sender's source clock, stop bit 2 must be sampled no later than $\frac{k_{s2}}{r_s} - \frac{1}{r_r \cdot N_s} \geq \frac{k_{s2}-0.5}{r_r}$. The subtracted term accounts for signal stability requirements.
- $\frac{N_s(k_{s2}-0.5)+1}{N_s k_{s2}} \leq \frac{f_{source,r}}{f_{source,s}} \leq \frac{N_s(k_{s1}-0.5)-1}{N_s(k_{s1}-1)}$

SPI protocol: see appendix! MSB first

- Full duplex 4 wire synchronised serial communication with 1 master and multiple slave devices
- 4 connections:
  - SCLK: Master clock signal for synchronisation
  - MOSI: Data out to slaves
  - MISO: Data in from slaves
  - $\overline{SS}$/$\overline{CS}$: Chip select
- Ideal for short distance communication from central location

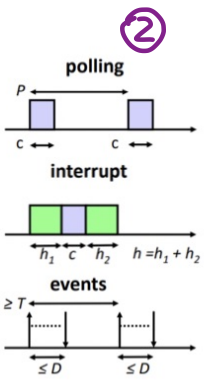


Interrupts:

- A hardware interrupt is an electronic alerting signal sent to the CPU from an internal or external component. The nested vector interrupt controller (NVIC) handles the processing of interrupts.
- The NVIC enables/disables interrupts, allows global masking of interrupts and registers ISRs. It can set the priority of ISRs which is necessary if several interrupts occur or an ISR is interrupted by another one.
- Processing of an interrupt:
  1. An interrupt is generated (e.g., by GPIO or timer) and the corresponding IFG register bit is set
  2. CPU/NVIC saves the return address, masks interrupts globally, determines the interrupt source and calls the corresponding ISR
  3. The ISR saves the context of the system, runs its code, unmasks interrupts and finally restores the context of the system

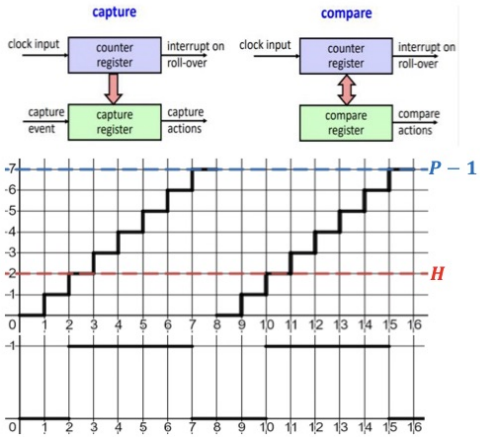
Polling vs interrupts:

- c: Processing time of event
- u: Utilisation of CPU
- h: Overhead handling interrupt
- P: Polling period, $P \geq c$
- T: Minimal time between events
- D: Deadline, $D \leq T$
- For interrupts $u_i = (h + c)/T$ and $h + c \leq D \leq T$
- For polling $u_p = c/P$ and $2c \leq c + P \leq D \leq T$
- Cases:
  1. If $D < c + \min(c, h)$, neither approach is feasible
  2. If $2c \leq D < h + c$, only polling is possible with $u_{opt} = c/(D - c)$
  3. If $h + c \leq D < 2c$, only interrupt is possible with $u_{opt} = (h + c)/T$
  4. If $c + \max(h, c) \leq D$ then both approaches are feasible



Clocks and timers:

- MCUs are usually equipped with many clock sources with different frequencies (for various time granularities), energy consumption and stability
- Watchdog timers are common as they reset the CPU if they themselves are not reset at regular intervals. This prevents the system from getting stuck in an inactive state (e.g., due to deadlocks)
- SysTick (MSP432) is a 24-bit decrementing counter that is part of the NVIC used for periodic interrupts or measuring time
- Measuring time differences, generating interrupts based on counter values or periodic ones and PWM are usually realised using capture and compare registers:
  - The capture register stores the current counter value when a capture event occurs
  - The compare register can be set by the user. Whenever it equals the counter value, an action (e.g., interrupt) can be taken
- For PWM, one compare register (value $P - 1$) is used to set the period and another (value H) is used to set the duty cycle $D = \frac{P-(H+1)}{P}$. The output signal is set to high when H is reached and set to low when $P - 1$ is reached



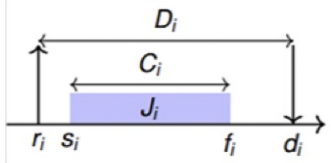
Programming Paradigms

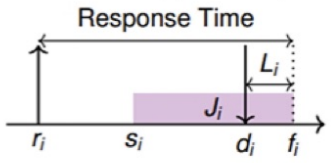
Classification:

- Time triggered:
  - Approaches:
    - Periodic
    - Cyclic executive
    - Generic scheduler
  - Properties:
    - No interrupts except by timers
    - Deterministic schedule is computed offline
    - No problems using shared resources
    - Interaction with environment via polling
- Event triggered:
  - Approaches:
    - Non-preemptive
    - Preemptive – Stack policy
    - Preemptive – Cooperative scheduling
    - Preemptive – Multitasking
  - Properties:
    - Dynamic and adaptive schedules
    - Guarantees can be given online or offline
    - Possible issues with shared resources

General Scheduling Definitions

Γ/J	Task set	T_i	Period of task τ_i
τ_i/J_i	Task	D_i	Relative deadline of task τ_i
$\tau_{i,j}$	j^{th} Instance of task τ_i	C_i	WCET of task τ_i
$d_{i,j}$	Deadline of task $\tau_{i,j}$	Φ_i	Phase of task τ_i
$r_{i,j}$	Release time of task $\tau_{i,j}$	L_i	Lateness $f_i - d_i$ (see below)





Simple Periodic Time-Triggered Scheduler

Method:

- A timer interrupts with period P
- All tasks have the same period

Properties:

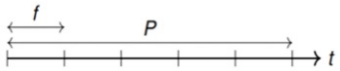
- Later tasks have erratic starting times, but inter-task communication is safe due to static ordering
- $\sum_i C_i \leq P$

Time-Triggered Cyclic-Executive Scheduling

Assumption: Tasks are periodic with period T_i and have a phase $\Phi_i \in \mathbb{R}$. Therefore, release times and deadlines can be expressed as

$$r_{i,j} = \Phi_i + (j-1)T_i \qquad d_{i,j} = \Phi_i + (j-1)T_i + D_i = r_{i,j} + D_i$$

- The period P of the system is divided into frames of length f
- Scheduling is done offline (manually)
- A time interrupts every frame to release the jobs for this frame



Conditions on P and f:

- $\forall \tau_i: f \leq T_i$ — A task executes at most once within a frame
- $f | P$ — P is a multiple of f (usually $\text{lcm}\{T_i\}$ is a good choice)
- $\forall \tau_i: f \geq C_i$ — Tasks begin and end within a single frame
- $\forall \tau_i: 2f - \gcd(T_i, f) \leq D_i$ — The entire frame of execution is between $r_{i,j}$ and $d_{i,j}$ for every task

Correctness of a given schedule:

Let $f_{ij} \in \left\{1, \dots, \frac{P}{f}\right\}$ denote the frame in which task $\tau_{i,j}$ executes. The following conditions should be satisfied:

- P is a multiple of f and a common multiple of all task periods T_i
- Frames are sufficiently long: $\sum_{\{i:\, f_{ij}=k\}} C_i \leq f \qquad \forall k \in \left\{1, \dots, \frac{P}{f}\right\}$
- Release times are respected *or* $\forall \tau_i: \Phi_i = \min_{j \in \{1,\dots,P/T_i\}} \{(f_{ij}-1)f - (j-1)T_i\}$
- Deadlines are respected: $\forall \tau_i,\ \forall j \in \left\{1, \dots, \frac{P}{T_i}\right\}: (j-1)T_i + \Phi_i + D_i \geq f_{ij} f$

Event-Triggered Scheduling

Non-preemptive: Events have a corresponding task. They are inserted in a queue and picked for execution.
Stack: As above but with preemption upon insertion. Tasks are stacked in memory and completed LIFO.
Cooperative multitasking: Threads allow a context switch. The system chooses which thread runs next.
Preemptive multitasking: Threads have a state (run, ready & blocked). OS determines context switches.

Aperiodic EDF Algorithm

Use case: This preemptive algorithm is used when arrival times are arbitrary, and tasks are independent.
Guarantees:

- Minimises the maximum lateness
- If EDF cannot schedule the task set, no other algorithm can

Method: At any given moment, execute the ready task that has the earliest absolute deadline.
Task acceptance test:

- Worst case finishing time of a task J_i at time t: $f_i = t + \sum_{k=1}^{i} c_i(t)$ (tasks are ordered by deadline)
- $f_i \leq d_i\ \forall i \in \{1, \dots, |J|\} \Rightarrow$ Feasible
- Acceptance test algorithm (must be computed whenever a new task arrives):

```
def is_feasible_EDF(J, J_new):
    J' = J ∪ {J_new} # Pending tasks ordered by deadline
    t = current_time() # Arrival time of the new task
    f_0 = t
    for J_i in J':
        f_i = f_{i-1} + c_i(t) # c_i(t) is the remaining WCET of task J_i
        if f_i > d_i: return False
    return True
```

EDD Algorithm

Use case: This non-preemptive algorithm is used when arrival times are equal, and tasks are independent.
Guarantees: Minimises the maximum lateness.
Method: Execute the tasks in non-decreasing order of deadline.

LDF Algorithm

Use case: This non-preemptive algorithm is used when arrival times are equal, and tasks are dependent.
Guarantees: Minimises the maximum lateness.
Method:

- Build a stack by following these steps:
  - Among all tasks with no successors or whose successors have been pushed to the stack, pick the one with the latest deadline
  - Push this task onto the stack and repeat until all tasks are on the stack
- Pop tasks from the stack and execute them

↳ LIFO Stack

EDF-Star Algorithm

Use case: This preemptive algorithm is used when arrival times are arbitrary, and tasks are dependent.
Guarantees: Minimises the maximum lateness.
Method:

- Modify the release times and deadlines by traversing the dependency graph twice:
  - Task J_j may not execute before r_j or before all its predecessors have finished. The new release time is $r_j^* = \max\{r_j, max\{r_i^* + C_i : J_i \to J_j\}\}$. This requires a forward traversal
  - Task J_j may not execute after d_j or after any of its successors must have begun execution. The new deadline is $d_j^* = \min\{d_j, \min\{d_i^* - C_i : J_j \to J_i\}\}$. This requires a backward traversal
- Use the normal EDF algorithm to produce the schedule

Deadline Monotonic Algorithm

Use case: This preemptive algorithm is used when priorities are static.
Method: Execute the ready task with the highest priority (i.e., lowest relative deadline).
Sufficient schedulability test:

- $\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n\left(2^{\frac{1}{n}} - 1\right) \Rightarrow$ Schedulable using DM
- Failing the test does not mean the task set is not schedulable

Necessary and sufficient schedulability test:

- The longest response time of a job is the sum of its computation time and interference: $R_i = C_i + I_i$
- At a given time t, the worst-case interference is $I_i = \sum_{j=1}^{i-1} \lceil t/T_j \rceil C_j$
- The test computes the smallest R_i that satisfies $R_i = C_i + \sum_{j=1}^{i-1} \lceil R_i/T_j \rceil C_j$ for all tasks
- $\forall i \in \{1, \ldots, |\Gamma|\}: R_i \leq D_i \Leftrightarrow$ Schedulable using DM
- Task indices i should be in decreasing order of priority (but iterations are in the opposite order)
- Failing the test does not mean the task set is not schedulable using other algorithms

```
def is_schedulable_DM(Γ):
    for τ_i in reversed(Γ): # Tasks with highest D_i first
        I = 0
        do:
            R = C_i + I
            if R > D_i: return False
            I = Σ_{j=1}^{i-1} ⌈R/T_j⌉ C_j # Sum interference from higher priority tasks
        while R < C_i + I
    return True
```

Notes:

- $\lim_{n\to\infty} a_n = \lim_{n\to\infty} n\left(2^{\frac{1}{n}} - 1\right) = \ln 2 \approx 0.69314718$, a_n is a monotonically decreasing sequence
- A critical instant of a task is the time at which the release of a job will produce the largest response time. This occurs when a job is released simultaneously with all higher priority jobs. If tasks are feasible at their critical instant, they are feasible at any other instant. It can be calculated as $t_{crit,i} = \text{lcm}\{T_j : j \leq i\}$

Rate Monotonic Algorithm

Use case: This preemptive algorithm is used when priorities are static and relative deadlines equal periods.
Guarantees: No other static priority algorithm can schedule a task set that RM cannot.
Method: Execute the ready task with the highest priority (i.e., lowest period).
Sufficient schedulability test:

- $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right) \Rightarrow$ Schedulable using RM
- Failing the test does not mean the task set is not schedulable

Necessary and sufficient schedulability test: Same as that of the DM algorithm

Periodic EDF Algorithm

Use case: This preemptive algorithm is used when priorities can be dynamic.
Guarantees: If EDF cannot schedule the task set, no other algorithm can.
Method: Execute the ready task with the earliest absolute deadline.
Necessary and sufficient schedulability test (for $D_i = T_i$):

- $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \Leftrightarrow$ Schedulable using EDF

Polling Server

Setting:

- An artificial periodic task τ_s is dedicated to serving aperiodic tasks which runs only when aperiodic tasks are pending at the time of its release
- The task has a period T_s and a maximum computation budget of C_s

Sufficient schedulability test in combination with RM:

- $U = \frac{C_s}{T_s} + \sum_{i=1}^{n} \frac{C_i}{T_i} \leq (n+1)\left(2^{\frac{1}{n+1}} - 1\right) \Rightarrow$ Schedulable using RM
- Failing the test does not mean the task set is not schedulable

Sufficient schedulability test for aperiodic requests:

- Assumption: Aperiodic requests finish before new ones arrive
- $\left(1 + \left\lceil \frac{C_a}{C_s} \right\rceil\right) T_s \leq D_a \Rightarrow$ Request schedulable
- Failing the test does not mean the request is not schedulable

Total Bandwidth Server

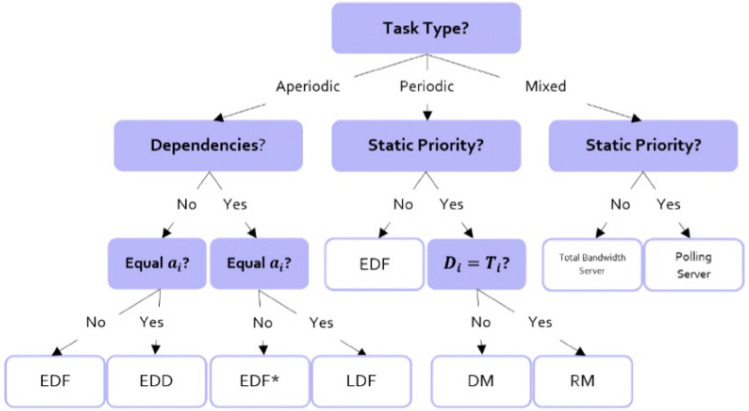
Method:

- When the k^{th} aperiodic request arrives, its deadline is computed as $d_k = \max\{r_k, d_{k-1}\} + \frac{C_k}{U_s}$. $d_0 = 0$ and U_s is the server utilisation.

Necessary and sufficient schedulability test:

- Given a set of n periodic tasks with processor utilisation $U_p = \sum_{i=1}^{n} \frac{C_i}{T_i}$ and a total bandwidth server with utilisation U_s, the following holds: $U_p + U_s \leq 1 \Leftrightarrow$ Schedulable with EDF.

Scheduling Algorithm Decision Tree



Embedded Operating Systems

Unsuitability of desktop OS:

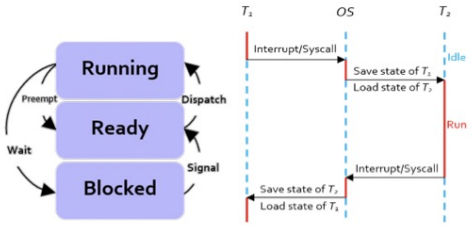
- Desktop kernels offer too many features that take up memory and computation time, are often not modular, fault tolerant or configurable and cannot provide timing guarantees with absolute certainty

Embedded OS and RTOS:

- The timing behaviour on an RTOS is predictable and the OS manages scheduling
- Every task can perform an interrupt (which would be a source of unreliability on a standard OS)
- Protection mechanisms are rarely necessary as the device is built designed for a single purpose
- Task management – the main functionality of RTOS kernels:
  - Execution of quasi-parallel tasks using threads (or processes)
  - CPU scheduling (meeting deadlines, fair resource allocation and minimising waiting times)
  - Inter-task communication via buffering
  - Real time clocks
  - Task synchronisation (critical sections, mutexes, semaphores etc.)

Task states:

- Running: Executing on the CPU, only one task is ever in this state
- Ready: Ready to execute but waiting due to another task which is already running
- Blocked: A task enters this state when it must wait for an event (e.g., a timer)



Shared Resources

Examples of shared resources: Data structures, variables, main memory, files, registers, I/O units etc.
Methods to protect exclusive resources: Disabling interrupts & preemption or using semaphores & mutex.
Semaphores:

- Each resource must be protected by its own semaphore. A task must execute a wait primitive before entering a critical section and execute a signal primitive upon completion
- All tasks blocked on the same resource are kept in a queue associated with the semaphore. When a running task executes a wait on a locked semaphore, it enters a blocked state, until another task signals to unlock the semaphore
- The mutex technique is a type of semaphore. Each resource has a "token" indicating its usage state

Priority inversion:

- Medium priority tasks can run if a lower priority one in a critical section blocks a higher priority one
- Solutions: Disabling preemption (downside: unrelated tasks get blocked) or PIP for static priorities
- PIP: J_{low} inherits the priority of J_{high}, if it blocks J_{high}, for the remainder of the critical section only
- PIP schedules are still prone to deadlocks

Timing anomalies: Faster/more CPUs, reduced dependencies or computation time ⇏ faster execution
Inter-task communication:

- Synchronous: 2 tasks wait on each other to exchange data (downside: estimating blocking time is hard)
- Asynchronous:
  - Mailbox: Sender deposits messages into a FIFO queue (shared memory buffer) for receiver (downside: overflow of buffer causes sender to be blocked)
  - Cyclical asynchronous buffer: As above but older messages get overwritten. Blocking does not occur

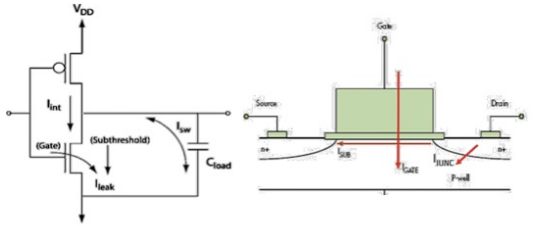
Power & Energy

5

Minimising power vs minimising energy:

- Minimising power is important for:
  - Power supply and voltage regulator design
  - The dimensioning of interconnects
  - Cooling (costs money and space)
- Minimising energy is important for:
  - Limited energy availability or battery capacity
  - High cost of energy
  - Longer lifetime

Power consumption of CMOS technology:

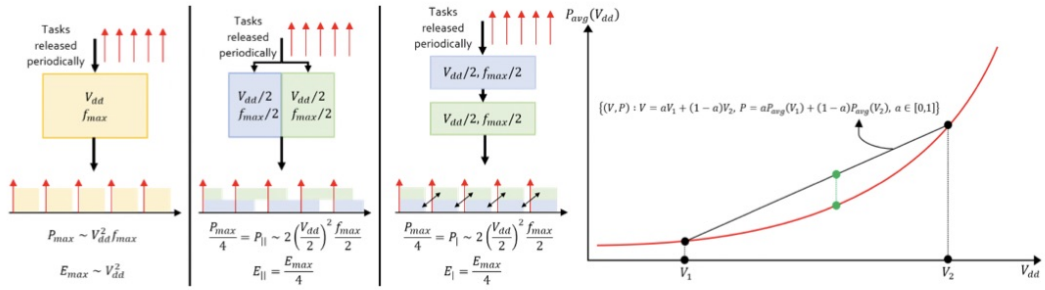


- Dynamic power: Capacitor transients and temporary shorts between supply rails during switching. There are many ways to reduce this power
- Static power: Gate-oxide, subthreshold and junction leakage. A common way to reduce this is by cutting components off from the power supply (power gating)
- Delay, power and energy consumption (no leakage): $P_{avg} \propto \alpha C_L V_{dd}^2 f$, where α is the switching activity. $E = \frac{N_{cycles}}{f} P_{avg}$. $\tau \propto \frac{C_L V_{dd}}{(V_{dd}-V_t)^2}$ → Implies that reducing V_{dd} means reducing f by (about) the same factor

↳ delay $\propto \frac{1}{V_{dd}}$

Reducing dynamic power:

- Parallelism: Tasks are split between multiple processors running at a lower voltage (and frequency)
- Pipelining: Tasks are split into sequential "chunks" and handled by a pipeline running at a lower voltage (and frequency)
- VLIW architectures: These offer a high degree of parallelism if combined with an appropriate parallel instruction set and compiler which can parallelise the code written by the user
- DVFS optimality rule: If P_{avg} is a convex function of voltage (or frequency), it is always worse to run at V_1 for aT then at V_2 for $(1-a)T$ than to run at $V_{avg} = aV_1 + (1-a)V_2$ for time T. This is the basis for the YDS algorithm



Dynamic power management:

- DPM tries to assign optimal power saving states (e.g., run, sleep or idle) during program execution
- Switching states usually incurs some overhead so the system must determine if it is worth switching
- The breakeven time is defined as the minimum waiting time to compensate the cost of switching
- $t_{breakeven} = \max\left\{2\frac{E_{transition} - t_{transition}P_{LPM}}{P_{HPM}-P_{LPM}}, 2t_{transition}\right\}$ (assuming transition energies and times are the same both ways)

Breakeven → exam spring 2020, 2.2b)

YDS Optimal DVFS Offline Algorithm ($O(|J|^3)$)

1. Determine the workload $c(\tau)$ of each task independent of frequency (in CPU cycles)
2. Plot the arrival-deadline timeline cutting out assigned critical intervals (see note 1 and image below).
3. For each arrival-deadline interval $[a, d]$ (of which there are at most $|J|^2$), compute the intensity defined as $G(a,d) = \frac{1}{d-a}\sum_{\tau \in V(a,d)} c(\tau)$ where $V(a,d)$ is the set of tasks fully contained within $[a,d]$. Not all $|J|^2$ intervals need to be considered. Skip an interval if any of the following apply:
   - $d < a$
   - $V(a,d)$ is empty (i.e., there are no tasks fully contained within the interval)
   - Shrinking $[a,d]$ without changing $V(a,d)$ is possible
4. Pick the interval $[a_c, d_c]$ with the highest intensity. This is the critical interval for this iteration. Assign the tasks in $V(a_c, d_c)$ in the order given by EDF to the critical interval. The CPU frequency is set to $G(a_c, d_c)$ for these tasks
5. To omit occupied times, all arrivals and deadlines are updated as follows:

$$a_{new} = \begin{cases} a, & a < a_c \\ a_c, & a \in [a_c, d_c] \\ a - (d_c - a_c), & a > d_c \end{cases} \qquad d_{new} = \begin{cases} d, & d < a_c \\ a_c, & d \in [a_c, d_c] \\ d - (d_c - a_c), & d > d_c \end{cases}$$

6. Repeat steps 2-5 until all tasks are assigned to a time interval
7. Assemble the overall schedule in decreasing order of intensity (i.e., the tasks in earlier iterations are assembled first). If the current critical interval overlaps with a previously scheduled critical interval, split the current one to fit around the other one bearing in mind the original arrival and deadline restrictions

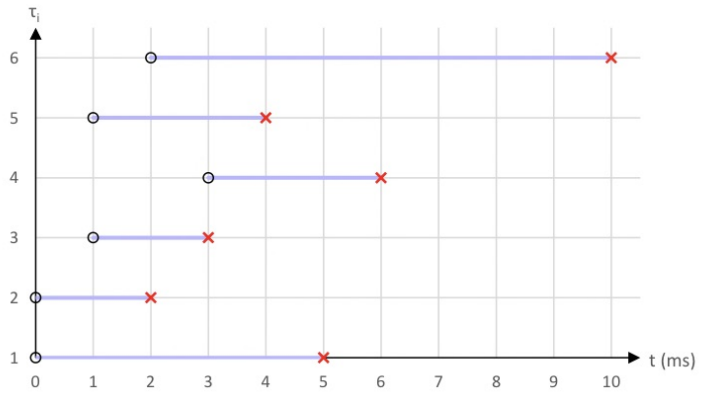
Notes:

1. To make step 7 easier, use a secondary time axis in each timeline which translates the primary axis times to "real" time
2. This algorithm guarantees finding the schedule with no deadline misses which uses the least energy

YDS Optimal DVFS Online Algorithm ($r = 27$)

Whenever a new task arrives or a pending critical interval ends (say at time t_{curr}):

1. Determine the *remaining* workload $c(\tau)$ of each pending task independent of frequency (in CPU cycles)
2. Plot the arrival-deadline timeline starting at t_{curr}. Arrival times before t_{curr} get clipped
3. See steps 3 and 4 of the offline version



Example of an arrival-deadline timeline

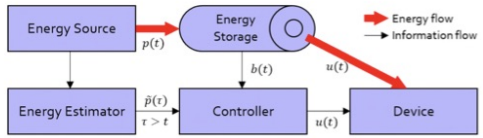
Battery Operated Systems and Energy Harvesting

- Batteries are used if no continuous source of power is available or if the device is mobile/autonomous
- Energy harvesting can provide infinite lifetime if rechargeable batteries (or supercapacitors) are used
- Sources can have a variable output and nonlinear I-V-characteristics based on the environment, so the ideal operating point is not trivial. Maximum power point tracking can be used in this case:

```
# Returns the voltage setting V_{k+1} for the next period
# k ≥ 1 is the current time index
def simple_MPPT_iteration(V_k, V_{k-1}, P_k, P_{k-1}, ΔV):
    if P_k > P_{k-1}:
        return V_k + ΔV if V_k > V_{k-1} else V_k − ΔV
    else:
        return V_k − ΔV if V_k > V_{k-1} else V_k + ΔV
```

Application control:

- Discrete time model of adaptive controller with the aim to never run out of energy:
  - Harvested energy (prediction) in $[t, t+1)$: $p(t)$, $(\tilde{p}(t))$
  - Used energy in $[t, t+1)$: $u(t)$
  - Battery model: $b(t+1) = \min\{B, b(t) + p(t) - u(t)\}$
  - Failure state: $b(t) + p(t) - u(t) < 0$
  - Utility: $U(t_1, t_2) = \sum_{t_1 \le \tau < t_2} \mu\big(u(\tau)\big)$, where μ is strictly concave (diminishing marginal utility)



- The optimal control $u^*(t)$ for the time interval $[t, t+1)$ satisfies the following for all $t \in [0, T)$:
  - $\forall t \in [0,T) : b^*(t) + p(t) - u^*(t) \ge 0$ — The system never enters the failure state
  - $\forall u : \min_{t\in[0,T)}\{u(t)\} \le \min_{t\in[0,T)}\{u^*(t)\}$ — The minimal use is maximal over all feasible u
  - $\forall \tilde{u} : U(0,T)|_{u=\tilde{u}} \le U(0,T)|_{u=u^*}$ — The use function maximises the utility $U(0,T)$
  - $b^*(T) \ge b^*(0)$ — The battery state has not degraded by the end

- Given a use function $u^*(t)$ that satisfies the first 2 conditions and maximises $U(t,T)$ for all $t \in [0,T)$, the following hold for all $\tau \in [0,T)$:
  - $u^*(\tau) > u^*(\tau-1) \Rightarrow b^*(t) = 0$ and $u^*(\tau) < u^*(\tau-1) \Rightarrow b^*(t) = B$
  - The contrapositives of these yield $\forall \tau \in (s,t] : b^*(\tau) \in (0,B) \Rightarrow \forall \tau \in [s,t] : u^*(\tau) = u^*(\tau-1)$
  - In plain English, if the battery is neither full nor empty, the optimal use function stays constant

- The problem of finding $u^*(t)$ can be solved by converting the conditions to a linear program. This can also be done by hand if the predicted energy input is given and correct i.e., $p(t) = \tilde{p}(t)$ for all $t \in [0,T)$:
  1. Determine the highest constant use $u^*(t) = \bar{u}$ which is still feasible
  2. Identify time points where $u^*(t)$ can be increased without violating constraints
  3. Repeat from step 1 but with the new time points only

- Predicting the future without error is impossible so finite horizon control becomes a more viable option:
  1. At time t, compute the optimal control for $\tau \in [t, t+T)$ with predictions $\tilde{p}(\tau)$ and $b(t+T) = b(t)$
  2. Use only the first value of the computed function
  3. Repeat from step 1 for the next time step with new predictions and battery state

Architecture Synthesis – Setup

Architecture synthesis as an optimisation problem:

- Synthesis means to determine the necessary hardware resources (allocation), schedules and relations of individual operations to hardware (binding) via exact or heuristic methods for a given algorithm
- This is a multi-objective optimisation problem with the goal to minimise the hardware cost, the algorithm's latency, power and energy consumptions
- The notion of Pareto efficiency is vital for the comparison of solutions. A solution is Pareto dominated if there exists another solution which is (strictly) better in every objective. A solution is Pareto efficient if it is not Pareto dominated

Dependency graphs (DG) as a model for computation:

- A DG is a DAG $G = (V, A)$ where vertices V represent operations of the algorithm and arcs such as $(v_i, v_j) \in A$ represent dependencies between operations (v_j executes after v_i)
- The tail of an arc is called the direct predecessor of the head which is its direct successor. If a path exists from v_i to v_j, then v_j is a successor of v_i and v_i is a predecessor of v_j
- A variable can only be assigned once – introduce a new variable instead to remedy this

Marked graphs:

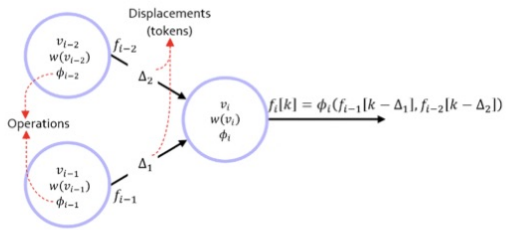
- In this course marked graphs are simplified Petri nets $G = (V, A, M)$ (without transitions) where vertices and arcs retain their meanings from DGs and the function $M: A \to \mathbb{N}_0$ represents the marking (number of tokens) of an arc. The marking of the entire graph is often represented as a vector. Vertices are called actors and can fire if activated (each incoming arc has at least 1 token) to produce 1 token on each outgoing arc. Tokens are thought of as data and can have a value associated with them

Model for architecture synthesis:

- A sequence graph $G_S = (V_S, A_S)$ which is an extension of DGs with one start and end vertex (v_0 and v_n) with no incoming and outgoing arcs respectively
- A resource graph $G_R = (V_R, A_R)$ which is a bipartite graph with $V_R = V_S \sqcup V_T$ where V_T denotes the resource types available. A weighted arc $(v_s, v_t) \in A_R$ denotes the availability of a resource type v_t for an operation v_s. The weight $w: A_R \to \mathbb{N}_0$ is the execution time of the operation on the resource
- A cost function $c: V_T \to \mathbb{Z}$ which quantifies a certain quality of the chosen resource types
- An allocation is a function $\alpha: V_T \to \mathbb{N}_0$ which denotes the number of available instances of $v_t \in V_T$
- A binding is defined by the functions $\beta: V_S \to V_T$ and $\gamma: V_S \to \mathbb{N}_0$. $\beta(v_s) = v_t$ and $\gamma(v_s) = k$ means the operation v_s will be executed on the k^{th} instance of v_t.
- A schedule is a function $\tau: V_S \to \mathbb{N}_0$ that determines the starting times of operations. A schedule is feasible if $\forall (v_i, v_j) \in A_S: \tau(v_j) \geq \tau(v_i) + w(v_i, \beta(v_i))$. The latency L is defined as $\tau(v_n) - \tau(v_0)$

Extended sequence graphs:

- For iterative algorithms, dependencies between iterations are modeled using extended sequence graphs $G_S = (V_S, A_S, d)$. Each arc $(v_i, v_j) \in A_S$ has a weight d_{ij} (displacement) which models the situation where variable v_j in the k^{th} iteration depends on variable v_i in the $(k - d_{ij})^{th}$ iteration
- With such graphs, a pipelined implementation with iteration interval P and throughput $\frac{1}{P}$ can be found



Architecture Synthesis – Algorithms & ILP

Scheduling without resource constraints:

```
# Schedule τ using ASAP
# vwpp: v ∈ V_S w/ planned predecessors
def ASAP_schedule(G_S, w):
  τ(v_0) = 0
  while not is_planned(v_n):
    v_i = vwpp(V_S)
    τ(v_i) = max{τ(v_j) + w(v_j) ∀(v_j, v_i) ∈ A_S}
  return τ
```

```
# Schedule τ using ALAP
# vwps: v ∈ V_S w/ planned successors
def ALAP_schedule(G_S, w, L_max):
  τ(v_n) = L_max
  while not is_planned(v_0):
    v_i = vwps(V_S)
    τ(v_i) = min{τ(v_j) ∀(v_i, v_j) ∈ A_S} − w(v_i)
  return τ
```

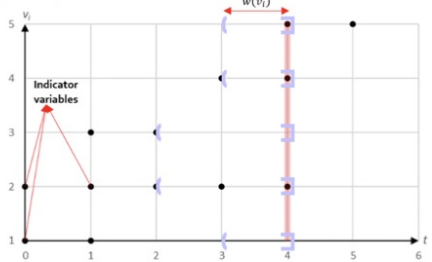
- The lecture's convention is to set $\tau(v_0)$ and $\tau(v_n)$ to $t = 1$ and $t = L_{max} + 1$ respectively

Scheduling with resource constraints:

- List scheduling heuristic:

```
# Priorities P of operations are determined e.g. by length of the longest
# path to the end node or according to the "mobility" of the task
def list_schedule(G_S, G_R, α, β, P):
    t = 0
    while not is_planned(v_n):
        for v_resource in V_T:
            U = Set of executable operations with β(v_i) = v_resource
            T = Set of operations running on v_resource
            S = Subset of U with highest priorities and |S ∪ T| ≤ α(v_resource)
            τ(v_i) = t for v_i in S
        t += 1
    return τ
```

- Converting the problem to an integer linear program and solving this would yield an optimal solution. The ILP formulation given a non-iterative algorithm and a binding is as follows:
  - Minimise $\tau(v_n) - \tau(v_0)$, subject to:
  1. $x_{i,t} \in \{0,1\}$ $\quad \forall v_i \in V_S, \forall t: l_i \leq t \leq h_i$ ● Starting time indicator variables with ASAP & ALAP bounds
  2. $\sum_{t=l_i}^{h_i} x_{i,t} = 1$ $\quad \forall v_i \in V_S$ ● Each task has exactly one starting time
  3. $\sum_{t=l_i}^{h_i} t x_{i,t} = \tau(v_i)$ $\quad \forall v_i \in V_S$ ● Task starting times in terms of indicator variables
  4. $\tau(v_j) \geq \tau(v_i) + w(v_i, \beta(v_i))$ $\quad \forall (v_i, v_j) \in A_S$ ● Precedence constraints
  5. $\sum_{\{i:\,(v_i,v_k)\in A_R\}} \sum_{p'=\max\{0,t-h_i\}}^{\min\{w(v_i)-1,\,t-l_i\}} x_{i,t-p'} \leq \alpha(v_k)$ $\quad \forall v_k \in V_T, \forall t \in \{0, \dots, \max\{h_i: v_i \in V_S\}\}$ ● Resource constraints



The "fishing net" method for condition 5

- For an iterative algorithm, the following conditions change:
  1. For ASAP & ALAP bounds, use arcs with $d_{ij} = 0$ (i.e., ignore dependencies across iterations)
  4. $\tau(v_j) \geq \tau(v_i) + w(v_i, \beta(v_i)) - d_{ij}P$ $\quad \forall (v_i, v_j) \in A_S$
  5. $\sum_{\{i:\,(v_i,v_k)\in A_R\}} \sum_{p'=\max\{0,t-h_i\}}^{\min\{w(v_i)-1,\,t-l_i\}} \sum_{\{p:\,l_i\leq t-p'+pP\leq h_i\}} x_{i,t-p'+pP} \leq \alpha(v_k)$ $\quad \forall v_k \in V_T, \forall t \in \{0, \dots, P-1\}$

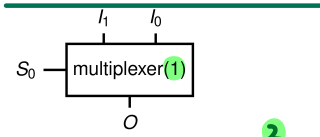
Appendix

Multiplexers

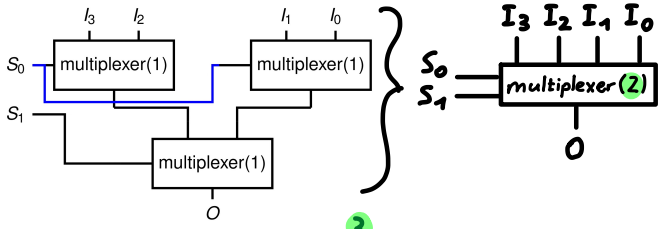
Convert $n = 2^L$ inputs to 1 output

- **general: 2^L-to-1 mux**
  - needs
    - L 'control lines' $(S_0, \dots, S_{L-1})$
    - $2 \times$ 2^{L-1}-to-1 mux
    - $1 \times$ 2^1-to-1 mux

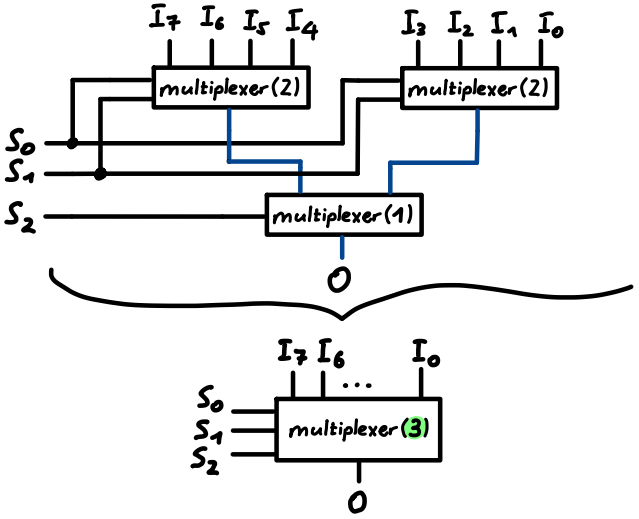
- **2^1-to-1 mux = 2^1-to-1 mux:**



- **2^2-to-1 mux = 2^2-to-1 mux:**



- **2^3-to-1 mux = 2^3-to-1 mux:**



$A(k)$: Area of 2^k-to-1 mux

$$\Rightarrow A(k) = \begin{cases} A_1 & , k = 1 \\ 2A(k-1) + A_1 & , \text{else} \end{cases}$$

Decoders

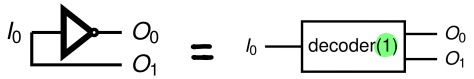
converts k inputs to 2^k outputs.
at all times, **exactly one** output is high

- **general: k-to-2^k „k bit decoder"**

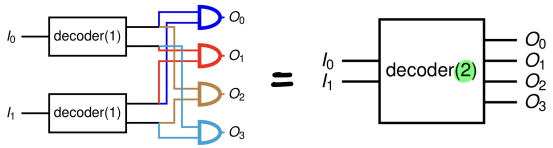
needs
- $2\times$ $\frac{k}{2}$ bit decoder , k even
- $\frac{k-1}{2}$ bit dec. & $\frac{k+1}{2}$ bit dec. , k odd
- 2^k 2-input AND gates
  - in addition to the smaller decoders

$$D(k) = \begin{cases} A_{NOT} & \text{if } k = 1 \\ 2 \cdot D(\frac{k}{2}) + A_{AND} \cdot 2^k & \text{if } k > 1 \text{ and } k \text{ is even} \\ D(\frac{k-1}{2}) + D(\frac{k+1}{2}) + A_{AND} \cdot 2^k & \text{if } k > 1 \text{ and } k \text{ is odd} \end{cases}$$

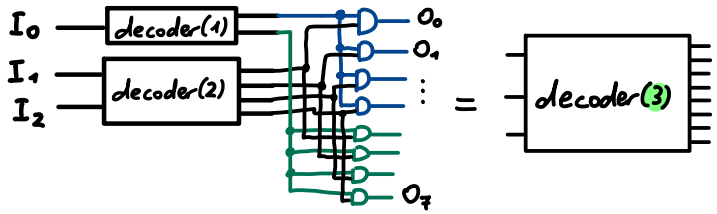
- **1-to-2 decoder**



- **2-to-4 decoder**

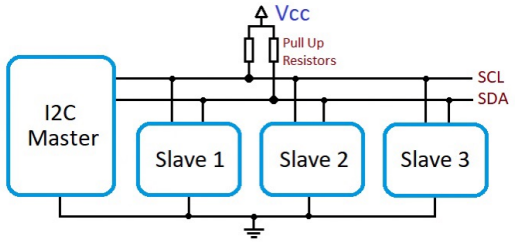


- **3-to-8 decoder**

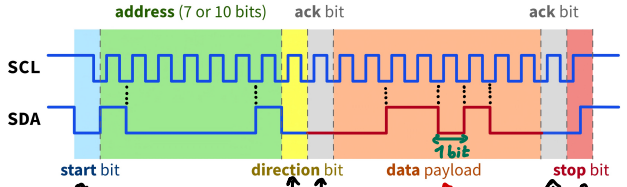


I2C Protocoll

(8)



- **half-duplex** (both directions but not simultaneously)
- **speeds** 100kHz, 400kHz, 1MHz, 5MHz
  - fixed!
  - 100kHz, 400kHz: standard
  - 1MHz: fast
  - 5MHz: ultra fast
  - 1MHz, 5MHz: not always supported



Master sets SDA 1→0 while SCL is still 1 then SCL „begins" (driven by Master)

8 bits

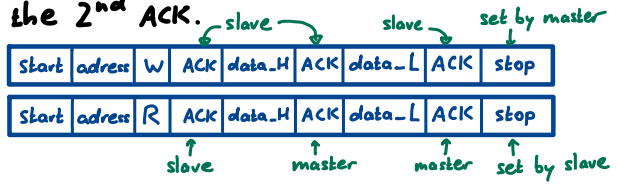
direction 0 ⇔ write ⇔ master → slave
direction 1 ⇔ read ⇔ slave → master

1st ACK/NACK: set by slave

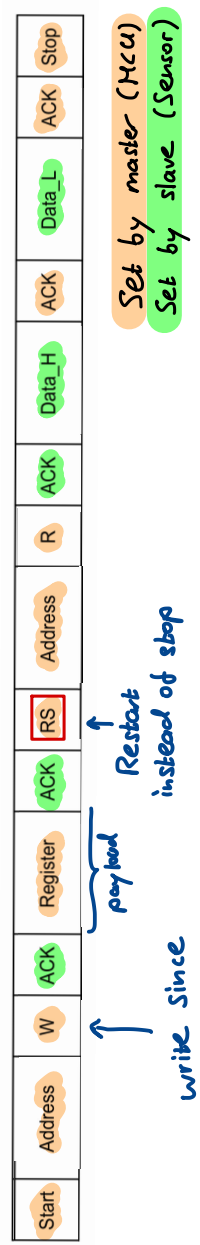
2nd ACK/NACK: set by master (Read) or slave (Write)

At the end of the transfer, the **master** transmits a **stop bit**:
- first, it sets **SDA** to **0**
- then it releases **SCL** (i.e. it lets it go to **1**)
- finally, it releases **SDA** which also goes to **1**

- except for start, stop & restart: transitions of SDA happen while SCL=0
- if we want to send/receive >8 bits we can just send another payload after the 2nd ACK.



- if you need to Write & Read to/from the same slave directly after another, you can use **Restart** instead of stop in the middle

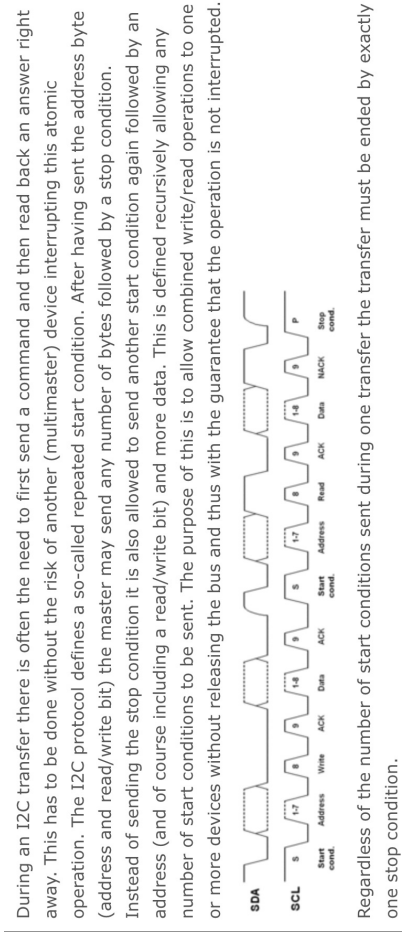


Set by master (MCU)
Set by slave (Sensor)

↑ Restart instead of stop

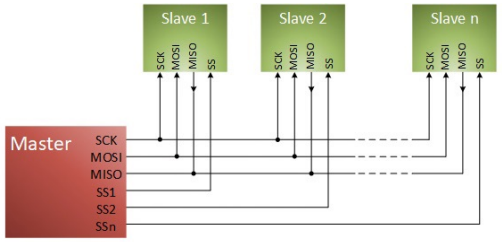
payload

↑ write since MCU needs to tell Sensor which register to send on the bus



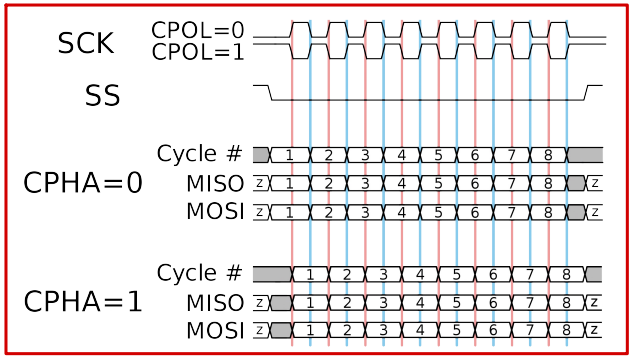
- latency = $\frac{\#\text{bits (incl. overhead)}}{f_{CLK}}$ → $f_{CLK} \in \{100kHz, 400kHz, \dots\}$

SPI Protocol



- full duplex
- 3+n wires at Master, n = #slaves
  - MISO (Master-in, Slave out) = SDI (SPI Data in)
  - MOSI (Master-out, Slave-in) = SDO (SPI Data Out)
  - SCK = SCLK = CLK = SPC = Clock
  - CSN = CS = SS = SSN = Chip Select (active low)
- CPOL = Clock Polarity $\in \{0,1\}$
  - CPOL = 0 ⇒ Clock idles at 0
  - CPOL = 1 ⇒ Clock idles at 1
- CPHA = Clock phase $\in \{0,1\}$

  CPHA determines phase at which MOSI is switched & MISO gets sampled



Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

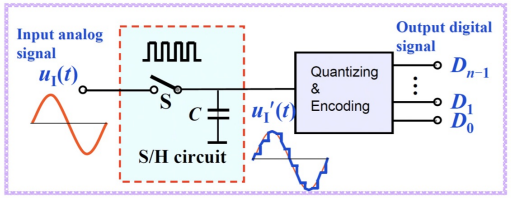
given (CPOL, CPHA) we can define modes

Note that a master can for example transmit in mode 0 and receive in mode 1 as long as CPOL is constant

naive implementation

```
/*
 * Simultaneously transmit and receive a byte on the SPI.
 * Polarity and phase are assumed to be both 0, i.e.:
 *   - input data is captured on rising edge of SCLK.
 *   - output data is propagated on falling edge of SCLK.
 * Returns the received byte.
 */
uint8_t SPI_transfer_byte(uint8_t byte_out)
{
    uint8_t byte_in = 0;
    uint8_t bit;

    //0x80 == 0b1000_0000 and 0x80 >> 1 == 0b0100_000 etc.
    for (bit = 0x80; bit != 0; bit = bit >> 1) {
        // Shift-out a bit to the MOSI line
        write_MOSI(byte_out & bit); //bitwise and

        // Delay for at least the peer's setup time
        delay(SPI_SCLK_LOW_TIME);

        // Pull the clock line high
        write_SCLK(1);

        // Shift-in a bit from the MISO line
        // read_MISO returns 1 or 0
        // --> a bit in byte_in gets set to 1 of MISO is 1
        if (read_MISO()) byte_in |= bit;

        // Delay for at least the peer's hold time
        delay(SPI_SCLK_HIGH_TIME);

        // Pull the clock line low
        write_SCLK(0);
    }
    return byte_in;
}
```

ADC - Analog to Digital Conversion



2 steps
- Sampling and Holding (S/H)
- Quantizing and Encoding (Q/E)

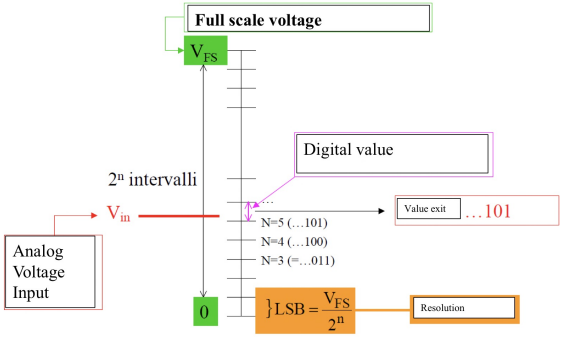
Sampling Theorem / Nyquist-Shannon Theorem

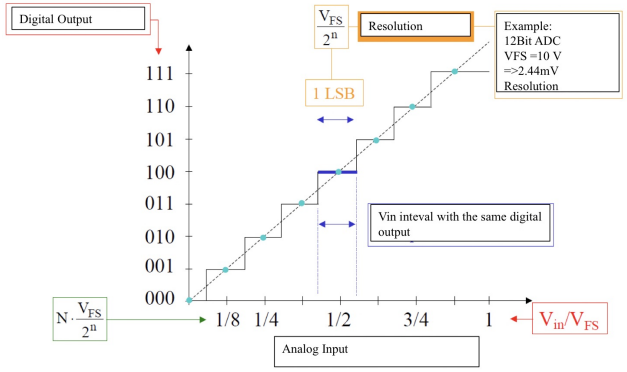
T_s : sampling Period

$$T_s \overset{!}{\leq} \frac{1}{2B} \Leftrightarrow f_s \overset{!}{\geq} 2B$$

for perfect reconstruction (no aliasing)

Quantizing & encoding





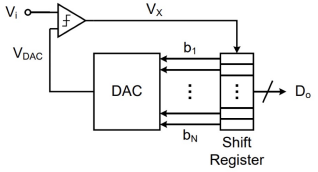
Resolution, Accuracy & Speed

- Resolution, *R*:
  - The resolution specifies the width of the digital output word;
    - 10, 12, 16 Bit ADC
  - The width of the word implies the smallest change to the analogue voltage that can be converted into a digital code;
  - The Least Significant Bit (LSB):

$$V_{LSB} = \frac{V_{ref}}{2^n}$$

- Accuracy:
  - Degree of conformity of a digital code representing the analogue voltage.
- Speed:
  - Maximum output data rate expressed in sample per second (sps)

SAR-ADC (Successive approximation)



Successive approximation ADC implements the **binary search algorithm**
- Is built out of a comparator, a DAC and a shift register to store the conversion progress.
- Analog voltage Vi is successively approximated; conversion needs several comparisons.
- SAR-ADC needs a very high clock frequency for fast conversions (oversampling)

- beginning: $V_{DAC} = \frac{V_{FS}}{2}$
- $V_x = \begin{cases} high, & V_i > V_{DAC} \\ low, & V_i < V_{DAC} \end{cases}$

⇒ write 1 or 0 in the shift register
(if V_x = high / if V_x = low)

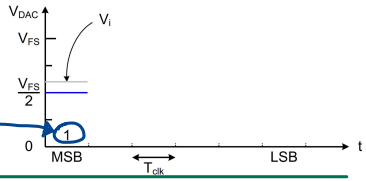
- next step: set $V_{DAC,new} = V_{DAC,old} \pm \frac{V_{FS}}{4}$ (+ if V_x was high, − if V_x was low)
- calculate new V_x etc.

→ $V_{DAC,end} = \frac{V_{FS}}{2} \pm \frac{V_{FS}}{4} \pm \frac{V_{FS}}{8} \pm \ldots \pm \frac{V_{FS}}{2^n}$ where n = #bits

example

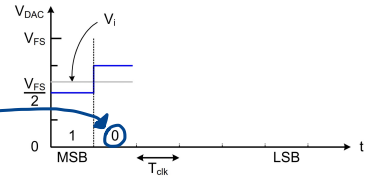
- beginning: $V_{DAC} = \frac{V_{FS}}{2}$
- $V_i > V_{DAC} \Rightarrow V_x$ = high

⇒ MSB = 1



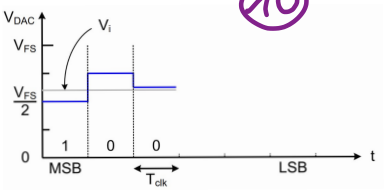
- $V_{DAC} = V_{DAC,old} + \frac{V_{FS}}{4} = \frac{V_{FS}}{2} + \frac{V_{FS}}{4} = \frac{3V_{FS}}{4}$
- $V_i < V_{DAC} \Rightarrow V_x$ = low

⇒ set bit to 0



- $V_{DAC} = V_{DAC,old} - \frac{V_{FS}}{8} = \frac{5V_{FS}}{8}$ (2nd bit was 0)
- $V_i < V_{DAC} \Rightarrow V_x$ = high

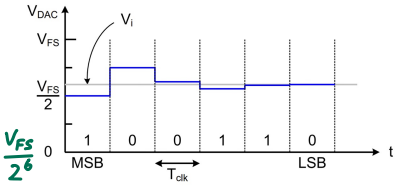
⇒ 3rd bit is 1



(10)

final value is $0b100110 = 38_{10}$

$\Rightarrow V_{sampled} = 38\,V_{LSB} = 38\,\frac{V_{FS}}{2^6}$



Convertion time → see Exercise X

$T_{conversion} = \underbrace{T_{sample\,\&\,hold}}_{\text{sometimes „sample time"}} + T_{encode}$

where $T_{encode} = \frac{\#bits\ (resolution)}{f_{CLK\_ADC}}$

Random Stuff

Dynamic Voltage Scaling without YDS

↳ exam autumn 2021, 2.2b)

given:

Task	T_1	T_2	T_3
Period (ms)	T_1	T_2	T_3
Cycles ($\cdot 10^3$)	$\tilde{c}_1$	$\tilde{c}_2$	$\tilde{c}_3$

and $P = g(f^3)$

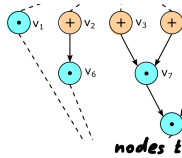
find a schedule (EDF) that minimises average power

$T = Lcm(T_1, T_2, T_3)$

$f = \frac{1}{T}\left(\frac{T}{T_1}\cdot\tilde{c}_1 + \frac{T}{T_2}\cdot\tilde{c}_2 + \frac{T}{T_3}\tilde{c}_3\right)$

⇒ f is just fast enough such that the utilisation is 100%

ILP - Resource constraints



given 2 adders and 3 multipliers

find all inequalities formulating resource-constraints that contain $x_{i,3}$ for some i

⊕ takes 1 time unit, ⊙ takes 2 time units

nodes that do addition

adders: $x_{2,3} + x_{3,3} + x_{4,3} + x_{5,3} \leq 2$ „all ongoing additions at t=3"
(starting at t=3)

multipliers: $\{x_{1,2} + x_{6,2} + x_{7,2} + x_{8,2} + x_{1,3} + x_{6,3} + x_{7,3} + x_{8,3} \leq 3$ „ongoing ⊙ at t=3"
$x_{1,3} + x_{6,3} + x_{7,3} + x_{8,3} + x_{1,4} + x_{6,4} + x_{7,4} + x_{8,4} \leq 3$ „ongoing ⊙ at t=4"

↓ 2 eg. since it takes 2 time-units
started at t=3 (not finished at t=4)
started at t=4 because mult. takes 2 time units