This document is an exam summary that follows the script of the Embedded Systems lecture at ETH Zurich. The contribution to this is a short summary that includes the most important formulas, algorithms, and proof ideas. This summary was created during the spring semester 2018. Due to updates to the syllabus content, some material may no longer be relevant for future versions of the lecture. This work is published as CC BY-NC-SA.

Embedded Systems

Yannick Merkli
FS 2018

1 Introduction

1.1 Reactivity & Timing (1-20)

Embedded systems are often <u>reactive</u>:

- Reactive systems must react to stimuli from the system environment:

> *„A reactive system is one which is in continual interaction with is environment and executes at a pace determined by that environment" [Bergé, 1995]*

Embedded systems often <u>must meet real-time constraints</u>:

- For hard real-time systems, right answers arriving too late are wrong. All other time-constraints are called soft. A guaranteed system response has to be explained without statistical arguments.

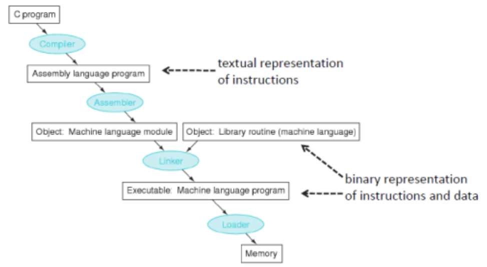
> *„A real-time constraint is called hard, if not meeting that constraint could result in a catastrophe" [Kopetz, 1997].*

1.2 Characteristics of Embedded Systems

- Dependable:
  - Reliability: $R(t)$ probability of system working correctly at t, provided that it worked at t = 0
  - Maintainability: $M(d)$ probability of system working correctly d time units after error
  - Availability: probability of system working at time t
  - Safety: No harm caused
  - Security: Condential and authentic communication
- Efficient with respect to Energy, Code-Size, Data-Memory, Run-Time, Weight and Cost
- Application dedicated
- Real-Time constraints must often be met
- Reactive: system is in continual interaction with environment and executes at a pace determined by that environment

2 Software development

Compilation of a C program to machine language program:

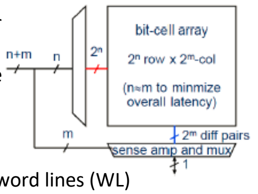

C program
Compiler
Assembly language program
textual representation of instructions
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
binary representation of instructions and data
Loader
Memory

3 Hardware Software Interface

3.1 Storage (3-9)

3.1.1 Static Random Access Memory (SRAM) (3-10)

- Single bit is stored in a bi-stable circuit
- Static Random Access Memory is used for
  - caches
  - register file within the processor core
  - small but fast memories
- Read:
  1. decode address (n+m bits)
  2. select row of cells using n single-bit word lines (WL)
  3. selected bit-cells drive all bit-lines BL (2m pairs)
  4. sense difference between bit-line pairs and read out
  5. Pre-charge all bit-lines to average voltage
- Write:
  - select row and overwrite bit-lines using strong signals



SRAM is volatile.

3.1.2 Dynamic Random Access (DRAM) (3-11)

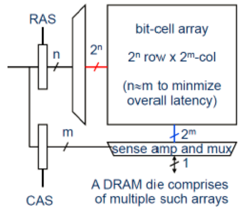
Single bit is stored as a charge in a capacitor

- Bit cell loses charge when read, bit cell drains over time
- Slower access than with SRAM due to (a) small storage capacity in comparison to capacity of bit-line and (b) no differential sensing.
- Higher density than SRAM (1 vs. 6 transistors per bit)

DRAMs require periodic refresh of charge

- Performed by the memory controller
- Refresh interval is tens of ms
- DRAM is unavailable during refresh

(RAS/CAS = row/column address select)


RAS
n
2^n
bit-cell array
2^n row x 2^m-col
(n≈m to minmize overall latency)
m
2^m
sense amp and mux
1
CAS
A DRAM die comprises of multiple such arrays

Typical Access Process:

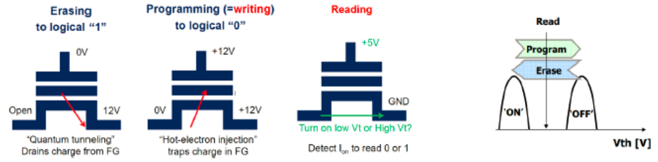
1. Bus transmission
2. Precharge and Row Access
3. Column Access
4. Data transfer and bus transmission

-> Data access on the same row is relatively fast. Random access is quite slow.
DRAM is volatile.

3.1.3 Flash Memory (3-14)

Electrically modifiable, non-volatile storage
Principle of operation:

- Transistor with a second "floating" gate
- Floating gate can trap electrons
- This results in a detectable change in threshold voltage


Erasing to logical "1"
0V
Open
12V
"Quantum tunneling" Drains charge from FG
Programming (=writing) to logical "0"
+12V
0V
+12V
"Hot-electron injection" traps charge in FG
Reading
+5V
GND
Turn on low Vt or High Vt?
Detect I_{on} to read 0 or 1
Read
Program
Erase
'ON'
'OFF'
Vth [V]

3.1.4 Memory map (3-17)

Available memory:
A microprocessor has a certain amount of memory available, split up into different kinds of memory (SRAM, DRAM, Flash, ROM,…)
Address space:
In a microprocessor, we have different units that all need addresses -> we need to subdivide the available address space.
E.g. let's say a processor uses 32 bit addresses:

- $addressable\ memory\ space = 2^{32} Byte = 4\ GByte$
  since each memory location corresponds to 1 Byte
- The address space is used to address the memories (reading and writing), to address the peripheral units, and to have access to debug and trace information (memory mapped microarchitecture).
- The address space is partitioned into zones, each one with a dedicated use.

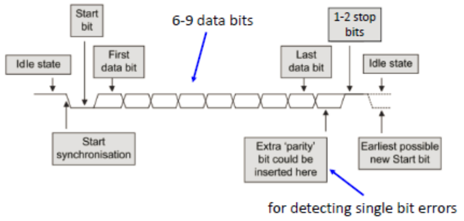
3.2 Input and Output (3-25)

Very often, a processor needs to exchange information with other processors or devices.
Many different communication protocols exist, such as:

- UART (Universal Asynchronous Receiver-Transmitter)
- SPI (Serial Peripheral Interface Bus)
- I2C (Inter-Integrated Circuit)
- USB (Universal Serial Bus)

3.2.1 UART protocol (3-27)

- Serial communication (i.e. we only have "<u>one</u> wire" that carries data) of bits via a single signal, i.e. UART provides parallel-to-serial and serial-to-parallel conversion
- Sender and receiver need to agree on the transmission rate (no common clock!)
- Transmission of a serial packet starts with a start bit, followed by data bits and finalized using a stop bit:


Start bit
6-9 data bits
1-2 stop bits
Idle state
First data bit
Last data bit
Idle state
Start synchronisation
Extra 'parity' bit could be inserted here
Earliest possible new Start bit
for detecting single bit errors

- The receiver runs an internal clock whose frequency is an exact multiple of the expected bit rate.
  -> receiver clock has a higher frequency than the frequency of sent bits
  -> receiver should sample <u>in the middle</u> of the sent bits -> less sensitive to delay of sampling. To know where the middle is, we count the number of cycles of the start bit.
- When a Start bit is detected, a counter begins to count clock cycles e.g. 8 cycles until the midpoint of the anticipated Start bit is reached.
- The clock counter counts a further 16 cycles, to the middle of the first Data bit, and so on until the Stop bit.

3.2.2 Memory mapped device access (3-32)

Configuration of Transmitter and Receiver must match; otherwise, they cannot communicate.
Examples of configuration parameters: transmission rate (baud rate, i.e., symbols/s), LSB or MSB first, number of bits per packet, parity bit, number of stop bits, interrupt-based communication, clock source
Clock subsampling: try to match a large set of frequencies from a single clock source by division
Example: Clock source SMCLK 3MHz, transmission rate 4800 bit/s, 16 clock periods per bit
-> $subsampling\ factor = \frac{3*10^6}{4800*16} = 39.0625$
If we want to write to UART, we write to the **transmit buffer**.
Problem: processor often writes data to transmit buffer much faster than the buffer sends out data -> we'll fill up the buffer -> need to check an **interrupt flag** for buffer overflow!

3.2.3 SPI protocol (3-37)

- Typically communicate across short distances
- Characteristics:
  - 4-wire synchronized (clocked) communications bus
  - supports single master and multiple slaves
  - always full-duplex
  - multiple Mbps transmission speeds can be achieved
  - transfer data in 4 to 16 bit serial packets
- Bus wiring:
  - MOSI (Master Out Slave In) – carries data out of master to slave
  - MISO (Master In Slave Out) – carries data out of slave to master
  - Both MOSI and MISO are active during every transmission
  - SS# (or CS) – unique line to select each slave chip
  - System clock SCLK – produced by master to synchronize transfers

3.3 Interrupts (3-41)

Two ways of doing interrupts:
- Interrupts: when an event occurs, stop and report
- Polling: periodically check with everyone whether everything is ok

3.3.1 Interrupts (3-41)

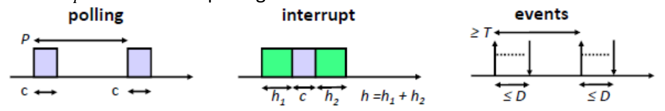
A hardware interrupt is an electronic alerting signal sent to the processor from an external device, either a part of the [device, such as an internal peripheral] or an external peripheral.
Interrupt handling:
- The Nested Vector Interrupt Controller (NVIC) handles the processing of interrupts. It registers interrupt service routines (ISR), sets the priority of interrupts
- Interrupt priorities are relevant if
  - several interrupts happen at the same time
  - the programmer does not mask interrupts in an interrupt service routine (ISR) and therefore, preemption of an ISR by another ISR may happen (interrupt nesting).

Processing of an interrupt:
1. An interrupt occurs (can be generated by peripherals, GPIO pins,…)
2. Interrupt sets a flag bit in a register (IFG register). There's such an IFG register for each interrupt source
3. CPU/ NVIC acknowledges interrupt by:
   - current instruction completes
   - saves return-to location on stack
   - saves 'Status Registers' to the stack
   - mask interrupts globally
   - determines source of interrupt
   - calls interrupt service routine (ISR)
4. Interrupt Service Routine (ISR):
   - save context of system
   - **run your interrupt's code**
   - restore context of system
   - (automatically) un-mask interrupts and
   - continue where it left off

3.3.2 Polling vs. Interrupt (3-52)

Compare polling and interrupt based on the utilization of the CPU.
Definitions:
- utilization u: average percentage, the processor is busy
- computation c: processing time of handling the event
- overhead h: time overhead for handling the interrupt
- period P: polling period
- interarrival time T: minimal time between two events
- deadline D: maximal time between event arrival and finishing event processing

Some relations for interrupt:
$h + c \leq D \qquad u_I \leq (h + c)/T$
where u_I: utilization of interrupts
Some relations for polling:
$c + P \leq D \qquad P \leq T \qquad u_p = c/P \geq c/\min(D - c, T)$
where u_P: utilization of polling



3.4 Clocks and timers (3-55)

3.4.1 Clocks (3-56)

Microcontrollers usually have many different clock sources that have different:
- frequency (relates to precision)
- energy consumption
- stability, e.g., crystal-controlled clock vs. digitally controlled oszillator

From basic clocks, microcontrollers often derive several internally available clock signals, by using clock dividers.

3.4.2 Watchdog Timer (3-61)

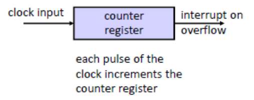
Watchdog Timers provide system fail-safety:
- If their counter ever rolls over (back to zero), they reset the processor. The goal here is to prevent your system from being inactive (deadlock) due to some unexpected fault.
- To prevent your system from continuously resetting itself, the counter should be reset at appropriate intervals.

Example: We've made some mistake in software or memory is completely full -> embedded system completely stops and does nothing -> watchdog timer can completely reset the system

3.4.3 System tick (3-63)

SysTick is a simple decrementing 24 bit counter that is part of the NVIC controller (Nested Vector Interrupt Controller). Its clock source is MCLK and it reloads to period-1 after reaching 0.
It's a very simple timer, mainly used for periodic interrupts or measuring time (i.e. very simple counting tasks)

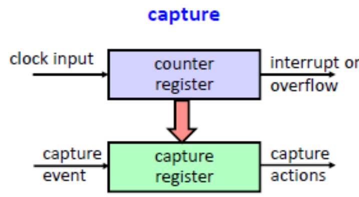
3.4.4 Timer (3-66)



Usually, embedded microprocessors have several elaborate timers that allow to time lots of different tasks, such as: capture the current time, generate interrupt when counter overflows, generate periodic interrupt, etc.
Typically, the mentioned functions are realized via capture and compare registers.
How do we create an analogue measure digitally? -> PWM
Pulse width modulation (PWM) can be used to change the average power of a singal. The use case could be to change the speed of a motor or to modulate the light intensity of a LED.

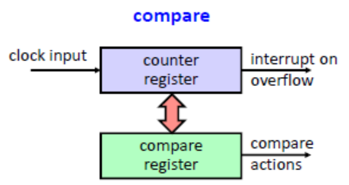
3.4.4.1 Capture registers (3-68)

- the value of counter register is stored in capture register at the time of the capture event (input signals, software)
- the value can be read by software
- at the time of the capture, further actions can be triggered (interrupt, signal)



3.4.4.2 Compare registers (3-68)

- the value of the compare register can be set by software
- as soon as the values of the counter and compare register are equal, compare actions can be taken such as interrupt, signaling peripherals, changing pin values, resetting the counter register



4 Programming paradigms

4.1 Timing guarantees (4-3)

Hard real-time systems can be often found in **safety-critical applications**. They need to provide the result of a computation within a fixed time bound.

4.1.1 Real-time systems (4-10)

- ES are expected to **finish tasks reliably within time bounds**.
- Essential for the design and analysis of real-time systems: Upper bounds on the execution times of all tasks statically known, commonly called **Worst-Case Execution Time (WCET)**
  Analogously, one can define the lower bound on the execution time, the Best-Case Execution Time BCET

Modern processors increase the average performance (execution of tasks) by using caches, pipelines, branch prediction, and speculation techniques, for example.
These features make the computation of the WCET very difficult: The execution times of single instructions vary widely.

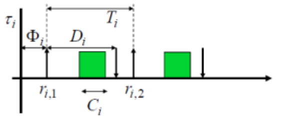
4.1.1.1 Determine execution times (4-15)

- Measurements: determine execution times directly by observing the execution or a simulation on a set of inputs.
  -> Does not guarantee an upper bound!
- Simulation
- Compute upper bounds along the structure of the program

4.2 Different Programming Paradigms (4-17)

Notations:

- Γ: denotes the set of all periodic tasks
- τ_i: denotes a periodic task
- $\tau_{i,j}$: denotes the jth instance of task i
- $r_{i,j}, d_{i,j}$: denote the release time and absolute deadline of the jth instance of task i
- ϕ_i: phase of task i (release time of its first instance)
- D_i: relative deadline of task i
- d_i: absolute deadline of task i
- $f_{i,j}$: number of the frame in which instance j of task i executes



4.2.1 Time-Triggered Systems (4-21)

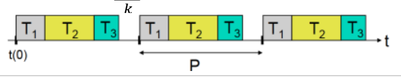
Pure time-triggered model:

- Periodic, cyclic executive
- generic time-triggered scheduler
- no interrupts allowed, except by timers
- schedule computed offline -> can use complex scheduling algorithms
- deterministic at runtime
- interaction with environment through polling
- Possible Extensions:
  - Allow interrupts
  - Allow preemptable background processes
  - Check for task overruns using a watchdog timer

4.2.1.1 Simple Periodic TT Scheduler (4-22)

- Timer interrupts with period P which is the same for all processes
- Later processes T2; T3 have unpredictable starting times
- Interprocess-communication or use of common resources is safe since there is a static ordering of tasks
- Necessary condition: sum of WCETs of all tasks within period is bounded by the period P:

$$\sum_k WCET(T_k) < P$$

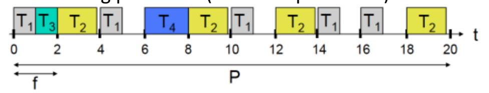


```
main:
    for (k=0,1,...,m−1) {determine table of
        processes (k, T(k))};
    i=0;
    set the timer to expire at initial phase t(0);
    while (true) sleep();

Timer Interrupt:
    i=i+1;
    set the timer to expire at t(0)+i*P;
    for (k=0,...,m−1) {execute process T(k)};
    return;
```

4.2.1.2 Time-Triggered Cyclic Executive Scheduler (4-24)

Tasks start and complete within a single frame

- Processes may have different periods
- Period P is partitioned into frames of length f
- Problematic for long processes (must be split -> bad)



Assumptions on the tasks:

- The instances of a periodic task are regularly activated at a constant Interval T_i:
  $$r_{i,j} = \phi_i + (j-1)T_i$$
- All instances of a periodic task have the same WCET C_i
- All instances of a periodic task have the same relative deadline D_i. Therefore, the absolute deadline:
  $$d_{i,j} = \phi_i + (j-1)T_i + D_i$$

Conditions for period P and frame length f:

- A task executes at most once within a frame:
  $$f \leq T_i \; \forall \; tasks \; \tau_i$$
- P is a multiple of f.
- Period P is least common multiple of all periods T_k.
- Tasks start and complete within a single frame:
  $$f \geq C_i \; \forall \; tasks \; \tau_i$$
- Between release time and deadline of every task there is at least one full frame:
  $$2f - \gcd(T_i, f) \leq D_i \; \forall \; tasks \; \tau_i$$

Checking for correctness of schedule:

- Is P a common multiple of all periods T_i?
- Is P a multiple of f?
- Is the frame sufficiently long?
  $$\sum_{\{i|f_{i,j}=k\}} C_i \leq f \;\; \forall 1 \leq k \leq \frac{P}{f}$$
- Determine offsets such that instances of tasks start after their release time:
  $$\phi_i = \min_{1 \leq j \leq \frac{P}{T_i}} \{(f_{ij}-1)f - (j-1)T_i\}$$
- Are deadlines respected?
  $$(j-1)T_i + \phi_i + D_i \geq f_{ij}f \;\; \forall \; tasks \; \tau_i, 1 \leq j \leq \frac{P}{T_i}$$

4.2.1.3 Generic Time-Triggered Scheduler

Establish a-priori control structure of tasks offline. Encode it in **Task-Descriptor List (TDL)** that contains cyclic schedule for all activities of the node. All constraints must be considered. A dispatcher is activated by synchronized clock and performs the actions as planned in the TDL.

4.2.1.4 Simplified Time-Triggered Scheduler

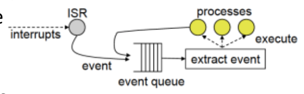
```
main:
    for (k=0,1,...,n−1) {determine static schedule
        (t(k),T(k))};
    determine period of schedule =: P;
    set i=k=0 initially; set the timer to expire at
         t(0);
    while (true) sleep();

Timer Interrupt:
    k_old = k;
    i := i+1;
    k := i mod n;
    set the timer to expire at t(k)+floor(i/n)*P;
    execute process T(k_old);
    return;
```

4.2.2 Event Triggered Systems (4-36)

Dynamic and adaptive, possible problems with timing, shared resources, buffer over-/underflow. Guarantees can be given offline or during run-time.

4.2.2.1 Non-Preemptive ET Scheduling (4-37)

- To each event is attributed a corresponding process to be executed
- Events are emitted by external interrupts or tasks
- Events are collected in a single queue
- Tasks cannot be preempted
- Extensions.
  - Preemptable background process
  - Timed events can be put into the queue (e.g. periodically)



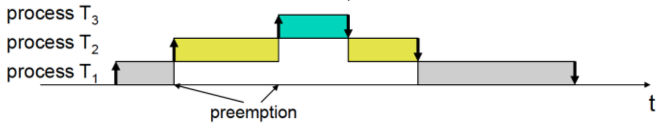
Properties:

- Interprocess communication is simple, interrupts can cause problems with shared resources
- Buffer overflow in case of too many events
- Long running tasks prevent others from running and may cause buffer overflow

```
main:
    while (true) {
        if (event queue is empty) {
            sleep();
        } else {
                             extract event from event
                                 queue;
                       execute process corresponding to
                             event;
        }
}

Interrupt:
    put event into event queue;
    return;
```

4.2.2.2 Preemptive ET Scheduling (4-40)

- Like Non-preemptive, but processes can be preempted by others.
- Stack-based context mechanism of process calls



- Tasks must finish in LIFO order of their instatiation (restricts flexibility)
- Shared resources (communication between tasks!) must be protected

```
main:
    while (true) {
        if (event queue is empty) {
            sleep();
        } else {
                         select event frome event
                             queue;
                  execute selected process;
                  remove selected event from queue;
        }
}

InsertEvent:
    put new event into queue;
    select event from event queue;
    if (selected process != running process) {
        execute selected process;
        remove selected event from queue;
    }
return;
```

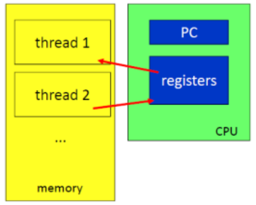
4.2.3 Processes and CPU

- A process is a unique execution of a program and has its own state. In case of a thread, this state consists mainly of register values and memory stack.
- Activation record: Copy of process state, includes registers and local data structures
- Context Switch: Current CPU context goes out, new CPU context goes in

4.2.4 Thread (4-43)

- A thread is a unique execution of a program.
  - Several copies of such a “program” may run simultaneously or at different times.
  - Threads share the same processor and its peripherals.
- A thread has its own local state. This state consists mainly of:
  - register values;
  - memory stack (local variables);
  - program counter;
- Several threads may have a shared state consisting of global variables.

Threads and Memory Organization



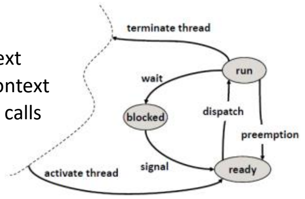
- Activation record (also denoted as the thread context) contains the thread local state which includes registers and local data structures.
- Context switch
  - Current CPU context goes out
  - New CPU context goes in

4.2.5 Co-operative Multitasking (4-45)

- Each process allows a context switch at cswitch() call, scheduler chooses which process runs next.
- Pros:
  - Predictable, where context switches can occur
  - Less errors with use of shared resources
- Cons:
  - Processes may never give up CPU
  - Real-time behavior threatened if process keeps CPU too long

4.2.6 Preemptive Multitasking (4-47)

- Most powerful form of multitasking
- OS controls when context switches
- OS determines which process runs next
- Use of timers to call OS and switch context
- Use of HW or SW interrupts or direct calls to OS routines to switch context



5 Operating Systems

5.1 Embedded Operating Systems (5-2)

Why an OS at all?

Advantages	Disadvantages
With an OS we can use lots of drivers and already programmed libraries	OS makes the system slower -> bad for critical applications

Embedded OS

- Desktop OS take lots of memory and are timing uncertain, not modular
- No single OS will fit all needs -> lots of different embedded OSes
- Embedded OS: remove unused functions/ libraries

5.1.1 Real-time Operating Systems (5-6)

Why is a desktop OS not suited?

- Monolithic kernel is too feature rich, is not modular, fault tolerant, configurable. Is too resource hungry (memory, computation time)
- Not designed for mission-critical applications (large timing uncertainty)

Advantages of Embedded OS

- OS can be fitted to individual need: Remove unused functions, conditional compilation depending on hardware, replace dynamic data by static data
- Improved predictability because of scheduler
- Interrupts can be employed by any process

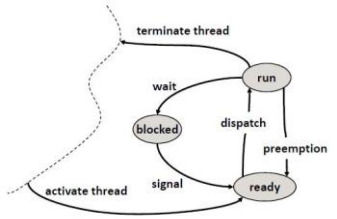
5.1.1.1 Requirements for RTOS (5-6)

- The timing behavior of the OS must be predictable
  - For all services: upper bound on execution time!
  - RTOS must be deterministic (unlike Java)
- OS must manage the timing and scheduling
  - OS may have to be aware of deadlines
  - OS must provide precise time services
- OS must be fast

5.1.2 Main functionality of RTOS-Kernels

- **Task management**: Execution of quasi-parallel tasks on a processor using processes or threads by maintaining process states, process queueing; allowing for preemptive tasks and quick interrupt handling
- **CPU scheduling**: guaranteeing deadlines, minimizing waiting times, fairness in granting resources
- **Task synchronization**: critical sections, semaphores, monitors, mutual exclusion
- **Inter-task communication**: buffering
- **Real-time clock**: as an internal time reference

5.1.3 Task states (5-12)



- **Run**: A task enters this state when it starts executing on the processor
- **Ready**: State of tasks that are ready to execute but cannot be executed because processor is assigned to another task
- **Blocked**: A task enters this state when it executes a synchronization primitive to wait for an event, e.g. a wait primitive on a semaphore or timer. In this case, the task is inserted in a queue associated with the semaphore.
- **Idle**: A periodic job enters this state when it completes its execution and has to wait for the beginning of the next period.

5.1.4 Threads (5-15)

Definition Thread:
A thread is the smallest sequence of programmed instructions that can be managed independently by a scheduler (a thread is a basic unit of CPU utilization). Multiple threads can exist within the same process and share resources such as memory, while different processes do not shore these resources.

- Typically shared by threads: memory
- Typically owned by threads: registers, stacks
- Threads are fast to switch between.
- The **Thread Control Block (TCB)** stores information needed to manage and schedule a thread.

The operating system maintains for each thread a data structure (TCB-thread control block) that contains its current status such as program counter, priority, state, scheduling information, thread name.

5.1.5 Classes of Operating Systems (5-18)

- **Class 1: Fast Proprietary Kernels**
  - Are fast, don't use a lot of resources but they are not especially predictable -> usually they're fast but we have no bounds -> not suited for hard real-time systems.
- **Class 2: Extension to Standard OS**
  - Attempt to exploit existing and comfortable main stream operating systems.
  - A real-time kernel runs all real-time tasks.
  - The standard-OS is executed as one task.

  Pros: Crash of standard-OS doesn't affect RT-tasks
  Cons: RT-tasks cannot use standard-OS services
- **Class 3: Research Systems**
  - try to avoid limitations of existing real-time and embedded operating systems.

6 Aperiodic and Periodic Scheduling

6.1 Basic terms (6-3)

Hard/ Soft RT-Models

- **Hard:** A real-time task is said to be hard, if missing its deadline may cause catastrophic consequences on the environment under control. Examples are sensory data acquisition, detection of critical conditions, actuator serving.
- **Soft**: A real-time task is called soft, if meeting its deadline is desirable for performance reasons, but missing its deadline is not catastrophic. Examples are command interpreter of the user interface, displaying messages on the screen.

Schedule
A schedule is an assignment of tasks $J = \{J_1, J_2, \dots\}$ to the processor, such that each task is executed until completion. It can be defined as a function

$$\sigma: \mathbb{R} \longrightarrow \mathbb{N}, t \longmapsto \sigma(t)$$

Where $\sigma(t)$ denotes the task, which is executed at time t. If $\sigma(t) = 0$, the processor is called idle.
If σ changes its value, the processor performs a context switch.
Each interval in which σ is constant is called a time slice.

A preemptive schedule is a schedule in which the running task can be suspended at any time.

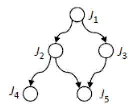
6.1.1 Schedule and Timing

A schedule is said to be feasible, if all task can be completed according to a set of specified constraints. A set of tasks is said to be schedulable, if there exists at least one algorithm that can produce a feasible schedule.
Terms:

- J_i: Task i
- τ_i: Periodic task i -> infinite sequence of identical activities, called instances or jobs, that are regularly activated at period T_i.
  - T_i: Period
  - ϕ_i: Phase (start time of first instance of periodic task i)
- a_i, r_i: Arrival time (or release time), the time at which a task becomes ready for execution
- C_i: Computation time, time necessary to the processor for executing the task without interruption
- d_i/D_i : Absolute/ relative Deadline
- s_i: Start time, the time at which a task starts its execution
- f_i: Finishing time, the time at which a task finishes its execution
- $L_i = f_i - d_i$: Lateness (negative if on early)
- $E_i = \max\{0, L_i\}$: Tardiness/ Exceeding time -> the time a tasks stays active after its deadline
- $X_i = d_i - a_i - C_i$: Laxity/ Slack time -> maximum time a task can be delayed

6.1.1.1 Precedence constraints

Precedence relations between tasks can be described through an acyclic directed graph G where tasks are represented by nodes and precedence relations by arrows. G induces a partial order on the task set.
We use this interpretation: All successors of a task are activated (concurrent task execution).



6.1.1.2 Scheduling Algorithms

- **Preemptive Algorithms**: The running task can be interrupted at any time to assign the processor another active task
- **Non-preemptive Algorithms**: A task, once started, is executed by the processor until completion
- **Static Algorithms**: Scheduling decisions are based on fixed parameters, assigned to tasks before their activation (constant priorities)
- **Dynamic Algorithms**: Scheduling decisions based on dynamic parameters that may vary during system execution
- An algorithm is called **optimal**, if it minimizes some given cost function defined over the task set
- An algorithm is called **heuristic**, if it tends to but does not guarantee to and an optimal schedule
- **Acceptance test:** The runtime system decides whenever a task is added to the system, whether it can schedule the whole task set without deadline violations

6.1.2 Metrics

- Average response time:
$$\bar{t_r} = \frac{1}{n}\sum_{i=1}^{n}(f_i - r_i)$$
- Total completion time:
$$t_c = \max_i(f_i) - \min_i(r_i)$$
- Weighted sum of response time:
$$t_w = \frac{\sum_{i=1}^{n} w_i(f_i - r_i)}{\sum_{i=1}^{n} w_i}$$
- Maximum lateness:
$$L_{max} = \max_i(f_i - d_i)$$
- Number of late tasks:
$$N_{late} = \sum_{i=1}^{n} miss(f_i) \; , miss(f_i) = \begin{cases} 0, if\ f_i \le d_i \\ 1, else \end{cases}$$

6.2 Aperiodic Tasks (6-16)

Aperiodic tasks	Equal arrival times **non-preemptive**	Arbitrary arrival times **preemptive**
Independent tasks	EDD (Jackson's Rule)	EDF (Horn's Rule)
Dependent tasks	LDF (Lawler's Rule)	EDF* (Chetto's Rule)

6.2.1 Earliest Deadline Due (Jackson's Rule) (6-18)

Algorithm: Task with earliest deadline is processed first. (Arrival times are equal for all tasks, Scheduling is non-preemptive.)
Jackson's Rule: Given a set of n independent tasks. Processing in order of non-decreasing deadlines is optimal with respect to minimizing the maximum lateness.

6.2.2 Earliest Deadline First (Horn's Rule) (6-22)

Algorithm: Task with earliest deadline is processed first. If new task with earlier deadline arrives, current task is preempted.
Horn's Rule: Given a set of n independent tasks with arbitrary arrival times, any algorithm that at any instant executes the task with the earliest absolute deadline among the ready tasks is optimal with respect to minimizing the maximum lateness.
Used quantities:

- $\sigma(t)$ identifies the task executing in the slice $[t, t+1)$
- $E(t)$ identifies the ready task that, at time t, has the earliest deadline
- $t_E(t)$ is the time $(\geq t)$ at which the next slice of task $E(t)$ begins its execution in the current schedule

Guarantee:

- Worst case finishing time of task i:
$$f_i = t + \sum_{k=1}^{i} c_k(t)$$
where $c_k(t)$ is the remaining worst-case execution time of task k and tasks are sorted in order of decreasing priorities.

- EDF guarantee condition:

$$\forall i = 1, \dots, n: f_i = t + \sum_{k=1}^{i} c_k(t) \le d_i$$

- Algorithm:

```
Algorithm: EDF_guarantee (J, $J_{new}$)
{       J'=J∪{$J_{new}$};  /* ordered by deadline */
        t = current_time();
        $f_0$ = t;
        for (each $J_i$∈J') {
                $f_i$ = $f_{i-1}$ + $c_i$(t);
                if ($f_i$ > $d_i$) return(INFEASIBLE);
        }
        return(FEASIBLE);
}
```

A new task is accepted if the schedule remains feasible.

6.2.3 EDF* (6-27)

The EDF* algorithm determines a feasible schedule in the case of tasks with precedence constraints if there exists one.
Algorithm: Modify release times and deadlines. Then use EDF.

Modification of release times:
1) For any initial node of the precedence graph set $r_i^* = r_i$
2) Select a task j such that its release time has not been modified but the release times of all its immediate predecessors i have been modified.
 If no such task exists, exit.
3) Set $r_j^* = \max(r_j, \max(r_i^* + C_i : J_i \to J_j))$
4) Return to step 2

Modification of deadlines:
1) For any terminal node of the precedence graph set $d_i^* = d_i$
2) Select a task i such that its deadline has not been modified but the deadlines of all its immediate successors j have been modified.
 If no such task exists, exit.
3) Set $d_i^* = \min(d_i, \min(d_j^* - C_j : J_i \to J_j))$
4) Return to step 2

If under the new release times and deadlines, we can schedule with EDF, then the task set with original release times and deadlines is also schedulable.
Note: When calculating metrics such as the average response time, use the original release times and deadlines!

6.2.4 Latest Deadline First (Lawler's Rule)

Non-preemptive scheduling for precedence constraints.
Algorithm:
1) A precedence graph is constructed
2) From leaves to roots: Select task with latest deadline among all available tasks to be scheduled last.
 Repeat procedure until all tasks in the set are selected.
3) At runtime: tasks are extracted from the head of the queue: first task inserted into queue will be executed last (FILO).

6.3 Periodic Tasks (6-32)

Definitions:
- Γ: a set of periodic tasks
- τ_i: a generic periodic task
- $\tau_{i,j}$: the j-th instance of task i
- $r_{i,j}, s_{i,j}, f_{i,j}, d_{i,j}$: release time, start time, finishing time and absolute deadline of task $\tau_{i,j}$
- ϕ_i: phase of task i (release time of its first instance)
- D_i: relative deadline of task i

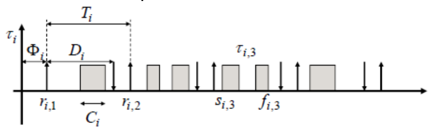
Hypotheses on periodic tasks:
- Tasks are regularly activated at a constant rate

$$r_{i,j} = \phi_i + (j-1)T_i$$

- All instances have the same worst-case execution time C_i and the same relative deadline D_i. Therefore, the absolute deadline $d_{i,j}$ satisfies

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

- Often, the relative deadline equals the period $D_i = T_i$ (implicit deadline) and therefore: $d_{i,j} = \phi_i + jT_i$
- All periodic tasks are independent
- Example:



Periodic tasks	Deadline = Period $\boldsymbol{D_i = T_i}$	Deadline < Period $\boldsymbol{D_i < T_i}$
Static priority	RM (rate-monotonic)	DM (deadline-monotonic)
Dynamic priority	EDF	EDF*

6.3.1 Rate Monotonic Scheduling (6-39)

- Task priorities are assigned to tasks before execution and do not change over time (static priority assignment).
- RM is intrinsically preemptive: the currently executing task is preempted by a task with higher priority.
- Deadlines equal the periods $D_i = T_i$

Algorithm: Each task is assigned a priority. Tasks with higher request rates (that is with shorter periods) will have higher priorities. Tasks with higher priority interrupt tasks with lower priority.
Schedulability condition: (sufficient but not necessary)

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \le n(2^{1/n} - 1)$$

where U is the processor utilization factor.
Optimality: RM is optimal among all fixed-priority assignments in the sense that no other fixed-priority algorithm can schedule a task set that cannot be scheduled by RM.
Definition: Critical instant
A critical instant of a task is the time at which the release of a task will produce the largest response time.
Lemma: For any task, the critical instant occurs if that task is simultaneously released with all higher priority tasks.

RM sufficient & necessary schedulability test: (same as for DM)

```
Algorithm: DM_guarantee (Γ)
{       for (each $\tau_i$∈Γ){
                I = 0;
                do {
                        R = I + $C_i$;
                        if (R > $D_i$) return(UNSCHEDULABLE);
                        I = $\sum^{j=1,\dots,(i-1)} \lceil R/T_j \rceil C_j$;
                } while (I + $C_i$ > R);
        }
        return(SCHEDULABLE);
}
```

Order tasks by increasing period $(i.e.\ \tau_1, \dots, \tau_n : T_1 < \dots < T_n)$!!

6.3.2 Deadline Monotonic Scheduling (6-49)

- Task priorities are assigned to tasks before execution and do not change over time (static priority assignment).
- DM is intrinsically preemptive: the currently executing task is preempted by a task with higher priority.
- Deadlines may be smaller than the period: $C_i \le D_i \le T_i$

Algorithm: Each task is assigned a priority. Tasks with smaller relative deadlines will have higher priorities. Tasks with higher priority interrupt tasks with lower priority.
Schedulability condition: (sufficient but not necessary)

$$U = \sum_{i=1}^{n} \frac{C_i}{D_i} \le n(2^{1/n} - 1)$$

Schedulability condition: (sufficient and necessary)
- The worst-case processor demand occurs when all tasks are released simultaneously (! critical instances).
- For each task i, the sum of its processing time and the interference imposed by higher priority tasks must be less than or equal to D_i
- Order tasks by increasing relative deadlines D_i, $(i.e.\ \tau_1, \dots, \tau_n : D_1 < \dots < D_n)$, then the worst-case interference for task i is

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j$$

- The longest response R_i time of a periodic task i is computed, at the critical instant, as the sum of its computation time and the interference due to preemption by higher priority tasks:

$$R_i = C_i + I_i$$

- Thus, schedulability test needs to compute the smallest R_i that satisfies

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j$$

 for all tasks i. Then, $R_i \le D_i$ must hold for all tasks $i = 1, \dots, n$

The longest response time R_i of τ_i can be computed iteratively by:

```
Algorithm: DM_guarantee (Γ)
{       for (each $\tau_i$∈Γ){
                I = 0;
                do {
                        R = I + $C_i$;
                        if (R > $D_i$) return(UNSCHEDULABLE);
                        I = $\sum^{j=1,\dots,(i-1)} \lceil R/T_j \rceil C_j$;
                } while (I + $C_i$ > R);
        }
        return(SCHEDULABLE);
}
```

6.3.3 EDF Scheduling (6-56)

Dynamic priority assignment, intrinsically preemptive, $D_i \le T_i$.
Algorithm: The currently executing task is preempted whenever another periodic instance with earlier deadline becomes active

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

Optimality: EDF is optimal w.r.t. schedulability (no set of periodic tasks can be scheduled if it can't be scheduled by EDF).
EDF schedulability test:

$D_i = T_i$	$D_i < T_i$
$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \le 1$ **Necessary & Sufficient**	$\sum_{i=1}^{n} \frac{C_i}{D_i} \le 1$ **Sufficient (but not necessary)**

Average processor utilization: $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$

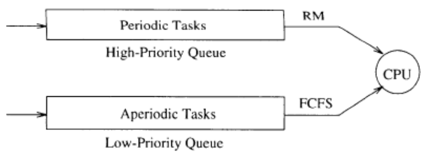
6.4 Mixed Task Set (6-63)

In many applications, there are aperiodic as well as periodic tasks.

- **Periodic tasks**: time-driven, execute critical control activities with hard timing constraints aimed at guaranteeing regular activation rates.
- **Aperiodic tasks**: event-driven, may have hard, soft, non-real-time requirements depending on the specific application.
- **Sporadic tasks**: Offline guarantee of event-driven aperiodic tasks with critical timing constraints can be done only by making proper assumptions on the environment; that is by assuming a maximum arrival rate for each critical event. Aperiodic tasks characterized by a minimum interarrival time are called sporadic.

6.4.1 Background Scheduling (6-65)

- Schedule periodic tasks with RM and EDF
- Processing of aperiodic tasks in the background, i.e. execute if there are no pending periodic requests.
- Good: Periodic tasks are not affected.
- Bad: Response of aperiodic tasks may be prohibitively long and there is no possibility to assign a higher priority to them.
- Example:



6.4.2 Rate-Monotonic Polling Server (6-67)

- Idea: Introduce an artificial periodic task whose purpose is to service aperiodic requests as soon as possible (therefore, "server").
- Function of polling server (PS)
  - At regular intervals equal to T_s, a PS task is instantiated. When it has the highest current priority, it serves any pending aperiodic requests within the limit of its capacity.
  - If no aperiodic requests are pending, PS suspends itself until the beginning of the next period and the time originally allocated for aperiodic service is not preserved for aperiodic execution.
  - Polling server only executes, if at the beginning of its period an aperiodic task is pending! If not, it suspends itself for the whole period.
  - If the polling server has to execute for less time than its given computation time it only executes for the needed time.
  - Its priority (period!) can be chosen to match the response time requirement for the aperiodic tasks.
- Disadvantage: If an aperiodic request arrives just after the server has suspended, it must wait until the beginning of the next polling period.

Schedulability condition: (sufficient but not necessary)

$$\frac{C_s}{T_s} + \sum_{i=1}^{n} \frac{C_i}{T_i} \le (n+1)\left(2^{1/(n+1)} - 1\right)$$

Aperiodic guarantee: (Sufficient schedulability)
Assumption: An aperiodic task is finished before a new one arrives.

- Computation time C_a, deadline D_a:
- Sufficient schedulability test:

$$\left(1 + \left\lceil \frac{C_a}{C_s} \right\rceil\right) T_s \le D_a$$

6.4.3 EDF Total Bandwidth Server (6-71)

Total Bandwidth Server:

- When the kth aperiodic request arrives at time $t = r_k$, it receives a deadline

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_s}$$

  ➔ k -> order aperiodic requests by release times, i.e. k=1 for task that arrives first.

  where C_k is the execution time of the request and $U_s (\le 1 - U_p)$ is the server utilization factor (that is, its bandwidth).
  By definition, $d_0 = 0$.
- Once a deadline is assigned, the request is inserted into the ready queue of the system as any other periodic instance.

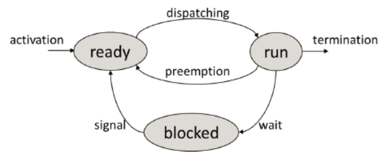
Schedulability condition: (sufficient and necessary)
Given a set of n periodic tasks with processor utilization U_p and a total bandwidth server with utilization U_s, the whole set is schedulable by EDF if and only if

$$U_p + U_s \le 1$$

7 Shared Resources

Examples of shared resources: data structures, variables, main memory area, file, set of registers, I/O unit, …
Many shared resources do not allow simultaneous accesses but require mutual exclusion (exclusive resources). These resources are called exclusive resources. In this case, no two threads are allowed to operate on the resource at the same time.
Definition: Critical Section
A piece of code executed under mutual exclusion constraints.

- A task waiting for an exclusive resource is said to be blocked
- Else, it enters the critical section and holds the resource.
- When a task leaves a critical section, the associated resource becomes free.

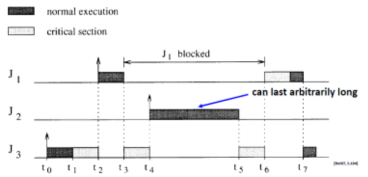


7.1 Semaphore (7-4)

Each exclusive resource R_i must be protected by a different semaphore S_i and each critical section operating on a resource must begin with a $wait(S_i)$ primitive and end with a $signal(S_i)$ primitive.
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 waiting state until another task executes a signal primitive that unlocks the semaphore.

7.2 Priority Inversion (7-9)

Example: J_3 (low priority) holds an exclusive resource that J_1 needs -> J_1 is blocked



A low priority task holds an exclusive resource and prevents a high priority task from running. Meanwhile, a medium priority task can preempt the low priority task and run while high priority task stays blocked.
Solution: Disallow preemption during the execution of critical sections. But this may block unrelated tasks with higher priority unnecessarily.

7.3 Priority Inheritance Protocol (PIP) (7-14)

Assumptions:

n tasks which cooperate through m shared resources. Fixed priorities, all critical sections on a resource begin with a $wait(S_i)$ and end with a $signal(S_i)$ operation.

Basic idea:

When a task J_i blocks one or more higher priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks.

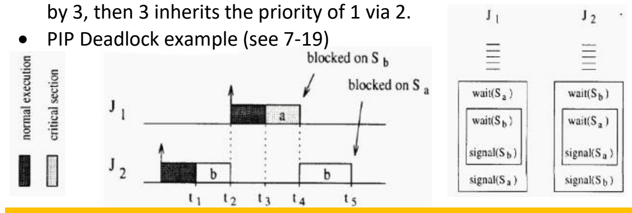
Terms:

Nominal priority P_i and active priority $p_i \leq P_i$. Jobs $J_1, \dots, J_n$ are ordered w.r.t. nominal priority where J_1 has highest priority. Jobs do not suspend themselves.

- Direct Blocking: higher-priority job tries to acquire a resource held by a lower-priority job and is blocked by it.
- Push-through Blocking: medium-priority job is blocked by a lower-priority job that has inherited a higher priority from a job it directly blocks.

Algorithm:

- Jobs are scheduled based on their active priorities. Jobs with the same priority are executed in a FCFS discipline.
- When a job J_i tries to enter a critical section and the resource is blocked by a lower priority job, the job J_i is blocked.
  Otherwise it enters the critical section.
- When a job J_i is blocked, it transmits its active priority to the job J_k that holds the semaphore. J_k resumes and executes the rest of its critical section with a priority $p_k = p_i$ (it inherits the priority of the highest priority of the jobs blocked by it).
- When J_k exits a critical section, it unlocks the semaphore and the highest priority job blocked on that semaphore is awakened. If no other jobs are blocked by J_k, then p_k is set to P_k, otherwise it is set to the highest priority of the jobs blocked by J_k.
- Priority inheritance is transitive, i.e. if 1 is blocked by 2 and 2 is blocked by 3, then 3 inherits the priority of 1 via 2.
- PIP Deadlock example (see 7-19)



7.4 Timing Anomalies (7-25)

Suppose, a real-time system works correctly with a given processor architecture. Now, you replace the processor with a faster one.
Are real-time constraints still satisfied?
Unfortunately, this is not true in general. Monotonicity does not hold in general, i.e., making a part of the system operate faster does not lead to a faster system execution. In other words, many software and systems architectures are fragile.
There are usually many timing anomalies in a
system, starting from the microarchitecture (caches, pipelines, speculation) via single processor scheduling to multiprocessor scheduling.
(Example: see 7-27 following)

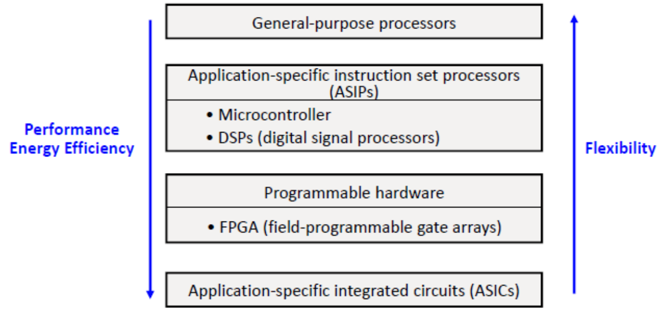
7.5 Communication and Synchronization (7-32)

Problem: the use of shared memory for implementing communication between tasks may cause priority inversion and blocking. Therefore, either the implementation of the shared medium is "thread safe" or the data exchange must be protected by critical sections.

7.5.1 Communication Mechanisms (7-34)

- Synchronous communication:
  - Whenever two tasks want to communicate they must be synchronized for a message transfer to take place (rendez-vous).
  - They have to wait for each other, i.e. both must be at the same time ready to do the data exchange.
  - Problem:
    - In case of dynamic real-time systems, estimating the maximum blocking time for a process rendez-vous is difficult.
    - Communication always needs synchronization. Therefore, the timing of the communication partners is closely linked.
- Asynchronous communication:
  - Tasks do not necessarily have to wait for each other.
  - The sender just deposits its message into a channel and continues its execution; similarly the receiver can directly access the message if at least a message has been deposited into the channel.
  - More suited for RT systems than synchronous communication
  - Mailbox: Shared memory buffer, FIFO-queue, basic operations are send and receive, usually has a fixed capacity.
  - Problem:
    - Blocking behavior if the channel is full or empty; alternative approach is provided by cyclical asynchronous buffers or double buffering.

8 Hardware components


General-purpose processors
Application-specific instruction set processors (ASIPs)
• Microcontroller
• DSPs (digital signal processors)
Programmable hardware
• FPGA (field-programmable gate arrays)
Application-specific integrated circuits (ASICs)
Performance
Energy Efficiency
Flexibility

8.1 General-Purpose Processors (8-9)

- **High performance**: Highly optimized, use of parallelism, complex memory hierarchy
- Not suited for real-time applications: Execution times unpredictable because of intensive resource sharing and dynamic decisions
- **Properties**: Good average performance for large application mix, high power consumption
- **Multicore processors**: Useful in high-performance embedded systems, disadvantages and problems:
  - Increased interference on shared resources (e.g. buses)
  - Increased timing uncertainty
  - Often only limited parallelism in embedded applications

8.2 System specialization (8-15)

The main difference between general purpose highest volume microprocessors and embedded systems is specialization.
Specialization should respect flexibility:

- Application domain specic systems shall cover a class of applications
- Some flexibility is required to account for late changes and debugging

System analysis required:

- Identication of application properties which can be used for specialization
- Quantification of individual specialization effects

Example:

- RISC (Reduced Instruction Set Computers) machines designed for run-time-, not for code-size-efficiency.
- Multimedia-Instructions exploit that many registers, adders etc. are quite wide (32/64 bit), whereas most multimedia data types are narrow (e.g. 8 bit per color, 16 bit per audio sample per channel).
  Idea: Several values can be stored per register and added in parallel.
- Heterogeneous Processor Registers: Different functionality of registers
- Multiple Memory Banks
- Address Generation Units

8.3 Application Specific Instruction Sets (8-21)

8.3.1 Control Dominated Systems / Microcontrollers (8-22)

Reactive systems with event driven behavior. System description typically Finite State Machines or Petri Nets.
Microcontrollers are for control-dominant applications.

- Control-dominant applications:
  - Support process scheduling, synchronization, preemption and context switch
  - Short latency times
- Low power consumption
- Peripheral units often integrated
- Suited for real-time applications

8.3.2 Digital Signal Processor (8-24)

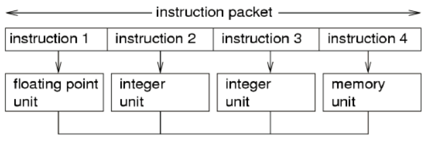
Streaming oriented systems with mostly periodic behaviour. System description typically flow graph. DSPs are for computation.

- Optimized for data-flow applications, only simple control-flow
- Parallel hardware units (VLIW)
- Specialized instruction set, specialized memory
- High data throughput, zero-overhead loops
- Suited for real-time applications

DSPs don't have lots of conditionals and context switches -> pipelining and branch prediction work very well -> DSPs are highly pipelined processors

Definition: Very Long Instruction Word (VLIW)

- Key idea: detection of possible parallelism to be done by compiler, not by hardware at run-time (inefficient).
- VLIW: parallel operations (instructions) encoded in one long word (instruction packet), each instruction controlling one functional unit.



8.4 Programmable Hardware / FPGA (8-31)

Classication of FPGAs:

- Granularity of logic units: Gate, tables, memory, functional blocks (ALU, control, data path, processor)
- Communication network: Crossbar, hierarchical mesh, tree
- Reconfiguration: Fixed at production time, once at design time, dynamic during run-time

8.5 Application Specific Circuits (ASICS) (8-38)

Custom designed circuits are necessary if ultimate speed or ultimate energy efficiency is the goal and large numbers can be sold. Drawbacks are long design times, lack of flexibility and high costs.

8.6 Heterogeneous Architecture (8-40)

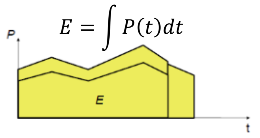
Modern chips have Microcontrollers AND DSPs on chip and other, even more specialized units (e.g. Audio unit, Cryptography unit,…)

9 Power and Energy

„Power is considered as the most important constraint in embedded systems."

Energy efficiency: It is necessary to optimize hardware and software. Use heterogeneous architectures in order to adapt to required performance and to class of application.

9.1 Power and Energy (9-7)



9.1.1 Low Power vs. Low Energy (9-9)

Minimizing the power consumption is important for

- the design of the power supply
- the design of voltage regulators
- the dimensioning of interconnect
- cooling (short term cooling)
- high cost & limited space

Minimizing the energy consumption is important due to

- restricted availability of energy (mobile systems)
- limited battery capacities (only slowly improving)
- very high costs of energy (solar panels, in space)
- long lifetimes, low temperatures

9.1.2 Power Consumption of a CMOS Processor (9-11)

Main sources:

- Dynamic power consumption:
  - charging and discharging capacitors
  - Short circuit power consumption: short circuit path between supply rails during switching
- Leakage and static power:
  - gate-oxide/subthreshold/junction leakage
  - becomes one of the major factors due to shrinking feature sizes in semiconductor technology (leakage current stays constant with smaller features)

- **Power:** $P \sim \alpha C_L V_{dd}^2 f$
- **Energy:** $E \sim \alpha C_L V_{dd}^2 f t = \alpha C_L V_{dd} \cdot (\#cycles)$
- **Delay:** $\tau \sim C_L \frac{V_{dd}}{(V_{dd}-V_T)^2}$
- **Max. Frequency:** $f_{MAX} \sim V_{dd}$

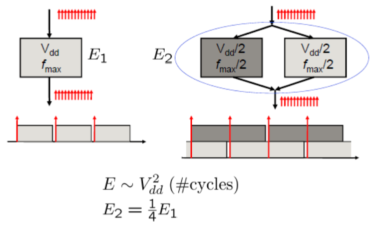
V_{dd}:	Supply voltage
α:	Switching Activity (how often the output changes on average)
C_L:	Load Capacity
f:	Clock Frequency
V_T:	Threshold Voltage $V_T \ll V_{dd}$

Reducing Static Power – Power Supply Gating

Power gating is one of the most effective ways of minimizing static power consumption (leakage). Cut-off power supply to inactive units/components.

9.2 Techniques to Reduce Dynamic Power (9-15)

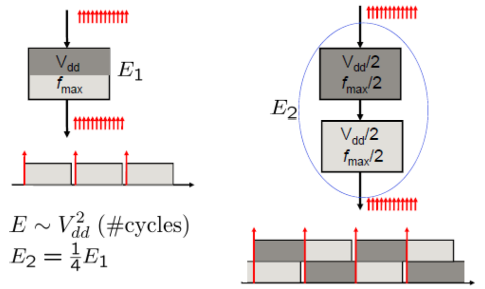
9.2.1 Parallelism (9-16)



Why $E_2 = \frac{1}{4}E_1$ and not $E_2 = \frac{1}{4}E_1 + \frac{1}{4}E_1 = \frac{1}{2}E_1$? Since we have half the frequency, execution takes twice as long -> to compare the two energies, we need to consider the energy over half the time interval. We thus have a factor 1/2 from the supply voltage and a factor 1/2 from the time interval:

➔ $E_2 = \frac{1}{2} \cdot \frac{1}{2} \cdot E_1 = \frac{1}{4}E_1$

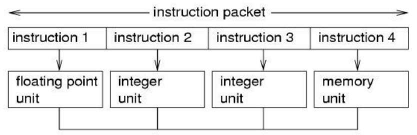
9.2.2 Pipelining (9-17)



Parallelism & Pipelining make at the same time execution time faster and more efficient at lower voltage.

9.2.3 VLIW (Very Long Instruction Word) Architectures (9-18)

- Large degree of parallelism
  - many parallel computational units, (deeply) pipelined
- Simple hardware architecture
  - explicit parallelism (parallel instruction set)
  - parallelization is done offline (compiler)



➔ One instruction word includes 4 instructions

9.2.4 Dynamic Voltage and Frequency Scaling (DVFS) (9-20)

-> Dynamic Voltage Scaling (DVS) see 9-13
Idea: Adapt voltage and frequency to save energy.
Optimal strategy: If possible, running constantly at lowest possible voltage and frequency minimize energy consumption for DVS.

- **Power:** $P \sim \alpha C_L V_{dd}^2 f$
- **Energy:** $E \sim \alpha C_L V_{dd}^2 f t = \alpha C_L V_{dd} \cdot (\#cycles)$
- **Delay:** $\tau \sim C_L \frac{V_{dd}}{(V_{dd}-V_T)^2}$

V_{dd}:	Supply voltage
α:	Switching Activity (how often the output changes on average)
C_L:	Load Capacity
f:	Clock Frequency
V_T:	Threshold Voltage $V_T \ll V_{dd}$

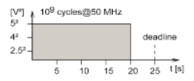
- Decreasing V_{dd} reduces P quadratically (f constant).
  -> Problem: reducing V_{dd} leads to slow performance since it takes longer to charge capacitors etc.
- The gate delay increases reciprocally with decreasing V_{dd}.
- Maximal frequency f_{max} decreases linearly with decreasing V_{dd}.
  ➔ $f_{max} \sim \tau \sim V_{dd}$

Saving energy for a given task: $E \sim \alpha C_L V_{dd}^2 f t = \alpha C_L V_{dd} \cdot (\#cycles)$
Only reducing the frequency f doesn't make any sense -> doesn't do anything to Energy. Processor just operates slower and more leakage current -> need to decrease f and V_{dd}

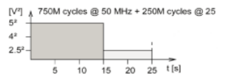
- reduce the supply voltage V_{dd}
- reduce switching activity α
- reduce the load capacitance C_L
- reduce the number of cycles $\#cycles$

Example:

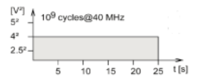
- Complete task as early as possible:



- Use two voltages:

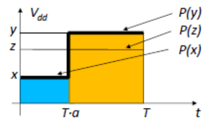


- Use one voltage:



9.2.4.1 DVFS: Optimal Strategy (9-27)

Case A: execute at voltage x for $T \cdot a$ time units and at voltage y for $(1\text{-}a) \cdot T$ time units; energy consumption $T \cdot (P(x) \cdot a + P(y) \cdot (1\text{-}a))$
Case B: execute at voltage $z = a \cdot x + (1\text{-}a) \cdot y$ for T time units; energy consumption $T \cdot P(z)$



Theorem: If possible, running at a constant frequency (voltage) minimizes the energy consumption for dynamic voltage scaling:
case A is always worse if the power consumption is a convex function of the supply voltage

9.3 YDS Algorithm

9.3.1 YDS Algorithm for Offline Scheduling (9-30)

Definition: Intensity G
The intensity $G([z,z'])$ in the interval $[z,z']$ is the average accumulated execution time of all tasks that have arrival and deadline in $[z,z']$ normalized to the length of the interval $z'-z$:

$$V'([z,z']) = \{v_i \in V : z \leq a_i < d_i \leq z'\}$$

$$G([z,z']) = \sum_{v_i \in V'([z,z'])} \frac{c_i}{z'-z}$$

Algorithm:

1) Find critical interval (i.e. interval with highest density) and schedule its task with EDF. Run the task at the intensity as frequency.
   The frequency is G times the nominal frequency (i.e. the lowest possible frequency).
2) Adjust arrival times and deadlines by excluding the critical interval.
3) Run the algorithm for the revised input again.
4) Put pieces together.

Remark:
Offline: The algorithm guarantees minimal energy consumption while satisfying the timing constraints. Time complexity is $O(N^3)$.

9.3.2 YDS Algorithm for Online Scheduling (9-36)

Algorithm:

1) Start at $t = 0$ and calculate the intensity for all task that have already arrived (intensity up until deadlines of arrived tasks). Then execute tasks, scheduled using EDF, in critical interval at intensity as frequency.
2) As soon as next task arrives: recompute all intensities with the current time as lower bound on the interval. Find the critical interval, schedule tasks with EDF, and execute at intensity as frequency.
3) Wait until new task arrives: go back to 2)

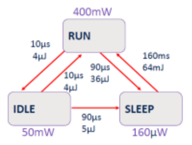
Remark:
Online: Compared to the optimal offline solution, online schedule uses at most 27 times of the minimal energy consumption.

9.4 Dynamic Power Management (DPM) (9-38)

DPM tries to assign optimal power saving states.

- RUN: operational
- IDLE: a SW routine may stop the CPU when not in use, while monitoring interrupts
- SLEEP: shutdown of on-chip activity

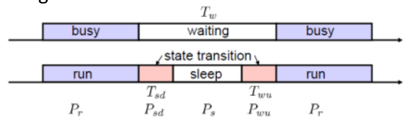
Desired: Shutdown only during long idle times!
Tradeoff between savings and overhead.



Definition: DVS Critical Frequency (Voltage)
Running at any frequency (voltage) lower than this frequency (voltage) is not worthwhile for execution.

9.4.1 Break-Even Time (9-41)

Definition: The minimum idle time required to compensate the cost of entering an inactive (sleep) state. Enter an inactive state is beneficial only if the idle time is longer than the break-even time.



- No transition: $E_1 = T_w P_w$
- State transition: $E_2 = T_{sd} P_{sd} + T_{wu} P_{wu} + (T_w - T_{sd} - T_{wu}) P_{sleep}$

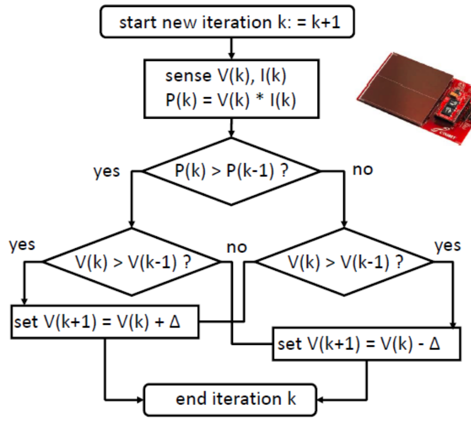
Break-even time: Limit for T_w s.t. $E_2 \leq E_1$
Time constraint: $T_w \geq T_{sd} + T_{wu}$

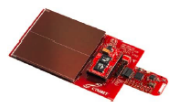
9.5 Battery-Operated Systems & Energy Harvesting (9-45)

Battery: -> no continuous power source available
Energy harvesting: -> prolong lifetime of battery-operated systems, autonomous operation

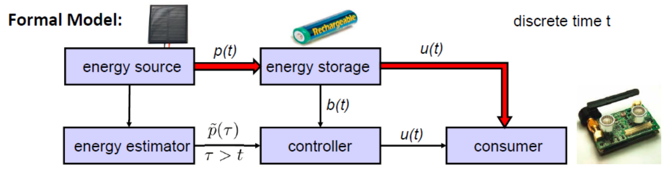
9.5.1 Power Point Tracking

simple tracking algorithm (assume constant illumination) :





9.5.2 Application Control (9-50)



- harvested and used energy in $[t, t+1)$: $p(t), u(t)$
- battery model: $b(t+1) = \min\{b(t) + p(t) - u(t), B\}$
- failure state: $b(t) + p(t) - u(t) < 0$
- utility:

$$U(t_1, t_2) = \sum_{t_1 \le \tau < t_2} \mu(u(\tau))$$

μ is a strictly concave function; higher used energy gives a reduced reward for the overall utility.

Theorem: Given a use function u*(t) such that the system never enters a failure state. If the following relations hold for all $\tau \in (t, T)$

$$u^*(\tau - 1) < u^*(\tau) \Longrightarrow b^*(\tau) = 0$$

empty battery

$$u^*(\tau - 1) > u^*(\tau) \Longrightarrow b^*(\tau) = B$$

full battery

then u*(t) is optimal with respect to maximizing the minimal used energy among all use functions and unique.

Finite Horizon Application Control: see (9-60 ff.)

10 Architecture Synthesis

Determine a hardware architecture that efficiently executes a given algorithm.

Major tasks of architecture synthesis:

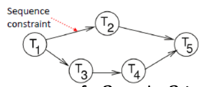
- allocation (determine the necessary hardware resources)
- scheduling (determine the timing of individual operations)
- binding (determine relation between individual operations of the algorithm and hardware resources)

10.1 Specification Models (10-3)

Models for architecture level (hardware) synthesis.

10.1.1 Task Graph / Dependence Graph (DG) (10-5)

A **dependence graph** is a directed graph $G = (V, E)$ in which $E \subset V \times V$ is a partial order.

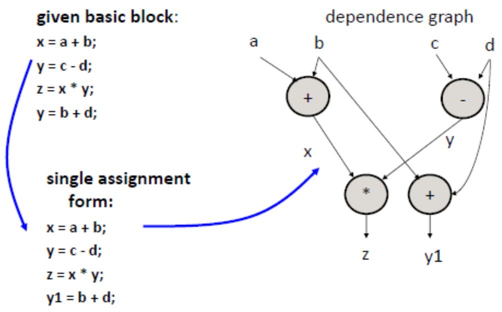


If $(v1, v2) \in E$, then $v1$ is called an **immediate predecessor** of $v2$ and $v2$ is called an **immediate successor** of $v1$.

Suppose E^* is the transitive closure of E. If $(v1, v2) \in E^*$, then $v1$ is called a **predecessor** of $v2$ and $v2$ is called a **successor** of $v1$.

- A DG describes order relations for the execution of single operations or tasks. Nodes correspond to tasks or operations, edges correspond to relations („executed after“).
- Represents parallelism in a program, but no branches in control flow.
- A Dependence Graph is acyclic.

Example:



given basic block:

```
x = a + b;
y = c - d;
z = x * y;
y = b + d;
```

single assignment form:

```
x = a + b;
y = c - d;
z = x * y;
y1 = b + d;
```

10.1.2 Control-Data Flow Graph (CDFG)

Description of control structures (e.g. branches) and data dependencies. Combines control flow (FSM) and dependence representation.

Control Flow Graph:

- Corresponds to a nite state machine, represents sequential control flow in a program.
- Branch conditioins often associated to the outgoing edges of a node
- Operations to be executed are associated in form of a dependence graph

Dependence Graph (Data Flow Graph DFG):

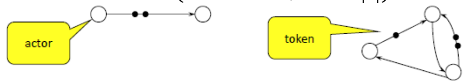
- NOP operations represent the start and end point of the execution (polar graph).

10.1.3 Marked Graph (10-9)

A marked graph $G = (V, A, del)$ consists of:

- nodes (actors) $v \in V$
- edges $a = (v_i, v_j) \in A, A \subseteq V \times V$
- number of initial tokens (or marking) on edges $del: A \to \mathbb{Z}^{\ge 0}$
- The marking (distribution of tokens) is often represented in form of a vector:

$$del = (del_1, \dots, del_i, \dots, del_{|A|})$$



- The token on the edges correspond to data that are stored in FIFO queues.
- A node (actor) is called activated if on every input edge there is at least one token.
- A node (actor) can fire if it is activated.
- The firing of a node v_i (actor operates on the first tokens in the input queues) removes from each input edge a token and adds a token to each output edge. The output token corresponds to the processed data.
- Marked graphs are mainly used for modeling regular computations, for example signal flow graphs.

Implementations of Marked Graphs (10-12 ff):

- Hardware implementation as synchronous digital circuits:
  Actors are combinatorial circuits, Edges are synchronously clocked shift registers.
- Hardware implementation as self-timed asynchronous circuit:
  Actors and FIFO registers independently implemented, Coordination and Synchronization of rings by handshake protocol
  (-> delay insensitive implementation of the semantics)
- Software implementation with static scheduling: At first, a feasible sequence of actor rings is determined which ends in the starting state (initial token distribution). This sequence is implemented in software.
- Software implementation with dynamic scheduling: Scheduling is done by RTOS. Actors correspond to threads. After ring, the thread is put into wait state. It is put into ready state when all necessary input data are present.

10.2 Models for Architecture Synthesis (10-16)

Determines a hardware architecture that efficiently executes a given algorithm. Tasks are

- Allocation: Determine necessary hardware resources
- Scheduling: Determine timing of individual operations
- Binding: Determine relation between individual operations and HW resources

Used sets:

- V_S: operations of the algorithm
- E_S: dependence relations of operations
- $V_R = V_S \cup V_T$: resources and bindings
- V_T: resource type

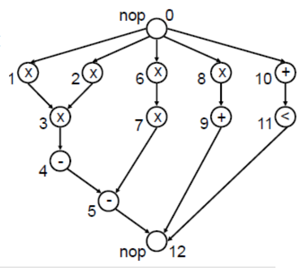
10.2.1 Sequence graph (SG) (10-16)

A sequence graph $G_S = (V_S, E_S)$ is a dependence graph with a single start node (no incoming edges) and a single end node (no outgoing edges).
V_S denotes the operations of the algorithm and E_S denotes the dependence relations.
A sequence graph is a hierarchy of directed graphs. A generic element of the graph is the dependence graph with the following properties:

- It contains two kinds of nodes: operations / tasks hierarchy nodes
- Each graph is acyclic and polar with start and end node (NOP)
- There are the following hierarchy nodes:
    - module call (CALL)
    - branch (BR)
    - iteration (LOOP)

Example: Algorithm to Sequence graph synt

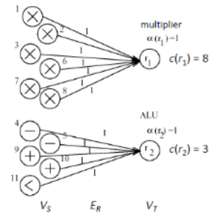
```
int diffeq(int x, int y, int u, int dx, int a) {
  int x1, u1, y1;
  while ( x < a ) {
    x1 = x + dx;
    u1 = u - (3 * x * u * dx) - (3 * y * dx);
    y1 = y + u * dx;
    x = x1;
    u = u1;
    y = y1;
  }
  return y;
}
```



10.2.2 Resource graph (10-16)

A resource graph $G_R = (V_R, E_R), V_R = V_S \cup V_T$ models resources and bindings. V_T denote the resource types of the architecture and G_R is a bipartite graph. An edge $(v_s, v_t) \in E_R$ represents the availability of a resource type v_t for an operation v_s.
Edge $(v_s, v_t) \in E_R$ has weight $w(v_s)$ (execution time of v_s)

Example:



10.2.3 Cost function (10-16)

Defines for each resource type $v_t \in V_T$ an associated cost:

$$c: V_T \to \mathbb{Z}$$

10.2.4 Execution times (10-16)

$w: E_R \to \mathbb{Z}^{\geq 0}$ are assigned to each edge $(v_s, v_t) \in E_R$ and denote the execution time of each operation $v_t \in V_T$ on resource type $v_s \in V_S$

10.2.5 Allocation (10-20)

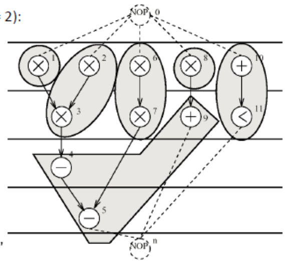
An allocation is a function $\alpha: V_T \to \mathbb{Z}^{\geq 0}$ that assigns to each resource type $v_t \in V_T$ the number $\alpha(v_t)$ of available instances.

10.2.6 Binding (10-20)

A binding is defined by functions $\beta: V_S \to V_T$ and $\gamma: V_S \to \mathbb{Z}^{\geq 0}$.
Here, $\beta(v_s) = v_t$ and $\gamma(v_s) = r$ denote that operation $v_s \in V_S$ is implemented on the rth instance of resource type $v_t \in V_T$.
Example binding:

Example binding ($\alpha(r_1) = 4, \alpha(r_2) = 2$):

$\beta(v_1) = r_1, \gamma(v_1) = 1,$
$\beta(v_2) = r_1, \gamma(v_2) = 2,$
$\beta(v_3) = r_1, \gamma(v_3) = 2,$
$\beta(v_4) = r_2, \gamma(v_4) = 1,$
$\beta(v_5) = r_2, \gamma(v_5) = 1,$
$\beta(v_6) = r_1, \gamma(v_6) = 3,$
$\beta(v_7) = r_1, \gamma(v_7) = 3,$
$\beta(v_8) = r_1, \gamma(v_8) = 4,$
$\beta(v_9) = r_2, \gamma(v_9) = 1,$
$\beta(v_{10}) = r_2, \gamma(v_{10}) = 2,$
$\beta(v_{11}) = r_2, \gamma(v_{11}) = 2$



10.2.7 Schedule (10-23)

A schedule is a function $\tau: V_S \to \mathbb{Z}^{\geq 0}$ that determines the starting times of operations. A schedule is feasible if the conditions

$$\tau(v_j) - \tau(v_i) \geq w(v_i) \quad \forall (v_i, v_j) \in E_S$$

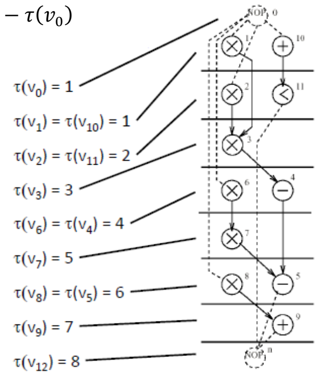
are satisfied.
$w(v_i) = w(v_i, \beta(v_i))$ denotes the execution time of operation v_i.

10.2.8 Latency (10-23)

The latency L of a schedule is the time difference between start node v_0 and end node v_n:

$$L = \tau(v_n) - \tau(v_0)$$

Example:
Latency here:
$L = \tau(v_{12}) - \tau(v_0) = 8 - 1 = 7$


$\tau(v_0) = 1$
$\tau(v_1) = \tau(v_{10}) = 1$
$\tau(v_2) = \tau(v_{11}) = 2$
$\tau(v_3) = 3$
$\tau(v_6) = \tau(v_4) = 4$
$\tau(v_7) = 5$
$\tau(v_8) = \tau(v_5) = 6$
$\tau(v_9) = 7$
$\tau(v_{12}) = 8$
NOP 0
NOP n

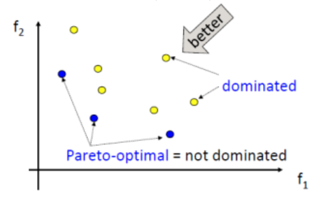
10.3 Multiobjective Optimization (10-25)

Architecture Synthesis is an optimization problem with more than one objective:

- Latency
- Hardware cost (memory, communication, computing units, control)
- Power and energy consumption

10.3.1 Pareto-Dominance (10-29)

Definition: A solution $a \in X$ weakly Pareto-dominates a solution $b \in X$, denoted as $a \preceq b$, if it is as least as good in all objectives, i.e. $f_i(a) \leq f_i(b)$ for all $1 \leq i \leq n$.
Solution a is better than b, denoted as $a \prec b$, iff $(a \preceq b) \wedge (b \npreceq a)$.
A solution is Pareto-optimal, if it is not Pareto-dominated by any other solution.
The set of all Pareto-optimal solutions is the Pareto-optimal front.



10.3.2 Classification of Scheduling Algorithm (10-32)

Classification

- unlimited resources: no constraints in terms of the available resources are defined.
- limited resources: constrains are given in terms of the number and type of available resources

Classes of synthesis algorithms

- iterative algorithms: an initial solution to the architecture synthesis is improved step by step.
- constructive algorithms: the synthesis problem is solved in one step.
- transformative algorithms: the initial problem formulation is converted into a (classical) optimization problem.

10.4 Synthesis without Resource Constraints (10-31)

The corresponding scheduling method can be used

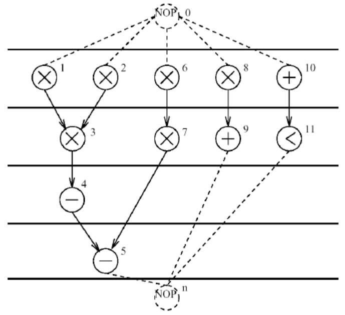
- as a preparatory step for the general synthesis problem
- to determine bounds on feasible schedules in the general case
- if there is a dedicated resource for each operation.

Given is a sequence graph $G_S = (V_S, E_S)$ and a resource graph $G_R = (V_R, E_R)$. Then the latency minimization without resource constraints is defined as

$$L = \min\{\tau(v_n) : \tau(v_j) - \tau(v_i) \geq w(v_i) \ \ \forall (v_i, v_j) \in E_S\}$$

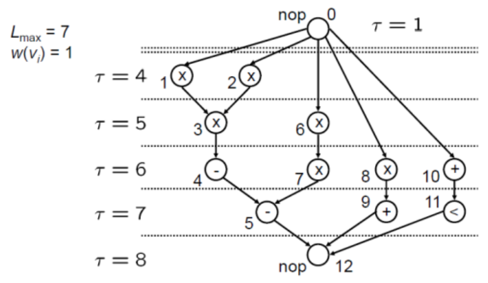
10.4.1 ASAP algorithm (10-34)

```
ASAP($G_S(V_S, E_S), w$) {
    $\tau(v_0) = 1$;
    REPEAT {
        Determine $v_i$ whose predec. are planed;
        $\tau(v_i) = \max\{\tau(v_j) + w(v_j)\ \forall (v_j, v_i) \in E_S\}$
    } UNTIL ($v_n$ is planned);
    RETURN ($\tau$);
}
```



10.4.2 ALAP algorithm (10-36)

```
ALAP($G_S(V_S, E_S), w, L_{max}$) {
    $\tau(v_n) = L_{max} + 1$;
    REPEAT {
        Determine $v_i$ whose succ. are planed;
        $\tau(v_i) = \min\{\tau(v_j)\ \forall (v_i, v_j) \in E_S\} - w(v_i)$
    } UNTIL ($v_0$ is planned);
    RETURN ($\tau$);
}
```



10.5 Scheduling with Timing Constraints (10-39)

- Deadlines: Latest finishing time, e.g. $\tau(v_2) + w(v_2) \leq 5$
- Release Times: Earliest starting time, e.g. $\tau(v_3) \geq 4$
- Relative Constraints: (differences between starting times of a pair of operations), e.g. $\tau(v_6) - \tau(v_7) \geq 4, \tau(v_4) - \tau(v_1) \leq 2$

We model all timing constraints using relative constraints. Deadlines and release times are defined relative to the start node v_0. Minimum, maximum and equality constraints can be converted into each other:

- Minimum constraints:
$$\tau(v_j) \geq \tau(v_i) + l_{ij} \iff \tau(v_j) - \tau(v_i) \geq l_{ij}$$
- Maximum constraints:
$$\tau(v_j) \leq \tau(v_i) + l_{ij} \iff \tau(v_i) - \tau(v_j) \leq -l_{ij}$$
- Equality constraints:
$$\tau(v_j) = \tau(v_i) + l_{ij} \iff \tau(v_j) - \tau(v_i) \leq l_{ij} \wedge \tau(v_j) - \tau(v_i) \geq l_{ij}$$

Examples:

- v_2 should start no earlier than 5 time units after the start of v_3:
$$\tau(v_2) \geq \tau(v_3) + 5 \iff \tau(v_2) - \tau(v_3) \geq 5$$
- v_4 should finish no earlier than 3 time units after the start of v_5:
$$\tau(v_4) + \mathrm{w}(v_4) \geq \tau(v_5) + 3 \iff \tau(v_4) - \tau(v_5) \geq 3 - \mathrm{w}(v_4)$$
- v_9 should start no later than 8 time units after the finish of v_5:
$$\tau(v_9) \leq \tau(v_5) + \mathrm{w}(v_5) + 8 \iff \tau(v_5) - \tau(v_9) \geq -\mathrm{w}(v_5) - 8$$
- v_{10} should start exactly 3 time units after the start of v_{11}:
$$\tau(v_{10}) = \tau(v_{11}) + 3 \iff \tau(v_{10}) - \tau(v_{11}) \geq 3 \wedge \tau(v_{11}) - \tau(v_{10}) \geq -3$$
- v_{10} should finish no later than 14 time units after the finish of v_3:
$$\tau(v_{10}) + w(v_{10}) \leq \tau(v_3) + w(v_3) + 14$$
$$\iff \tau(v_3) - \tau(v_{10}) \geq w(v_{10}) - w(v_3) - 14$$

10.5.1 Weighted Constraint Graph (10-40)

A weighted constraint graph $G_C = (V_C, E_C, d)$ related to a sequence graph $G_S = (V_S, E_S)$ contains nodes $V_S = V_C$ and a weighted edge for each timing constraint. An edge $(v_i, v_j) \in E_C$ with weight $d(v_i, v_j)$ denotes the constraint $\tau(v_j) - \tau(v_i) \geq d(v_i, v_j)$.

In order to represent a feasible schedule, we have one edge corresponding to each precedence constraint with
$$d(v_i, v_j) = w(v_i)$$
where $w(v_i)$ denotes the execution time of v_i.
-> i.e. to convert a sequence graph to a weighted constraint graph, we add weight $w(v_i)$ to each edge (v_i, v_j).

Adding an edge:
Constraint $\tau(v_j) - \tau(v_i) \geq l_{ij}$ corresponds to an edge from v_i to v_j with weight l_{ij}.

A consistent assignment of starting times $\tau(v_i)$ to all operations can be done by solving a single source longest path problem.
A possible algorithm (Bellman-Ford) has complexity $\mathcal{O}(|V_C|\ |E_C|)$:

- Start at $\tau(v_0) = 1$
- Iteratively set:
$$\tau(v_j) = \max\{\tau(v_j), \tau(v_i) + d(v_i, v_j) : (v_i, v_j) \in E_C\}$$
for all $v_i \in V_C$ starting from
$$\forall v_i \in V_C \setminus \{v_0\}: \ \tau(v_i) = -\infty$$

10.6 Synthesis with Resource Constraints (10-43)

Given is a sequence graph $G_S = (V_S, E_S)$, a resource graph $G_R = (V_R, E_R)$ and an associated allocation α and binding β.
Then the minimal latency is defined as
$L = \min\{\tau(v_n) :$
$(\tau(v_j) - \tau(v_i) \geq w(v_i, \beta(v_i))\ \forall (v_i, v_j) \in E_S) \ \wedge$
$(|\{v_s\!: \ \beta(v_s) = v_t \wedge \tau(v_s) \leq t < \tau(v_s) + w(v_s, v_t)\}| \leq \alpha(v_t)$
$\forall v_t \in V_T, \forall 1 \leq t \leq L_{max})\}$

- dependencies are respected
- there are not more than the available resources in use at any moment in time and for any resource type

where L_{max} denotes an upper bound on the latency.

10.6.1 List scheduling (10-45)

Algorithm for scheduling under resource constraints. Principles:

- Each operation has a static priority (i.e. determined before the List scheduling). Priority denotes the urgency of being scheduled.
- Algorithm schedules one time step after the other.
- $U_{t,k}$ denotes the set of operations that are ready to be executed (i.e. their predecessors are finished) on resource v_k at time t
- $T_{t,k}$ denotes the set of running operations on resource v_k at time t
- $S_{t,k}$ denotes the set of operations that are to be executed on v_k in the next time step $t + 1$

```
LIST($G_S(V_S, E_S)$,$G_R(V_R, E_R)$,$\alpha$,$\beta$,$priorities$){
    $t = 1$;                              resource types
    REPEAT {
        FORALL $v_k \in V_T$ {             $v \in V_S$ with $\beta(v) = v_k$
            determine candidates to be scheduled $U_k$;
            determine running operations $T_k$;
            choose $S_k \subseteq U_k$ with maximal priority
              and $|S_k| + |T_k| \leq \alpha(v_k)$;
            $\tau(v_i) = t\ \ \forall v_i \in S_k$;   }
        $t = t + 1$;
    } UNTIL ($v_n$ planned);
    RETURN ($\tau$);   }
```

List scheduling is a **heuristic** algorithm! It is thus **not** guaranteed to find an optimal solution (i.e. minimal latency).

Example: List schedule by hand -> go through table step for step

t	k	$U_{t,k}$	$T_{t,k}$	$S_{t,k}$
0	r_1, r_2	ν_1, ν_4, ν_5	—	ν_1, ν_5
	r_3	ν_2, ν_3	—	ν_2
1	r_1, r_2	ν_4, ν_9	—	ν_4, ν_9
	r_3	ν_3	ν_2	—
2	r_1, r_2	ν_{11}	—	ν_{11}
	r_3	ν_3, ν_6	—	ν_6

10.6.2 Integer Linear Programming (ILP) (10-50)

- Yields optimal solution to synthesis problems as it is based on an exact mathematical description of the problem.
- Solves scheduling, binding and allocation simultaneously.
- Note: solving linear programming problems is very easy and efficient. However, solving **integer** linear programming (some variables are forced to be integers) is much harder.

For the following example, we assume:

- Binding is already fixed (execution times $w(v_i)$ known)
- Earliest/latest starting times of operations v_i are l_i, h_i

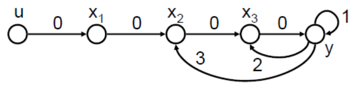
1) Minimize
$$L = \tau(v_n) - \tau(v_0)$$
subject to the constraints 2) – 6)
2) Decision variables binary:
$$\forall v_i \in V_S:\ \forall t: l_i \le t \le h_i: \quad x_{i,t} \in \{0,1\}$$
$x_{i,t} = 1$ means that operations v_i starts at time t
3) Only one variable is non-zero (i.e. each v_i can only execute once):
$$\boxed{\forall v_i \in V_S: \sum_{t=l_i}^{h_i} x_{i,t} = 1}$$
4) Relation between variables x and starting times of operations τ:
$$\boxed{\forall v_i \in V_S: \sum_{t=l_i}^{h_i} t \cdot x_{i,t} = \tau(v_i)}$$
5) Guarantee, that all precedence constraints are satisfied:
$$\boxed{\tau(v_j) - \tau(v_i) \ge w(v_i) \quad \forall (v_i, v_j) \in E_S}$$
(don't forget conditions for $(.., \tau_n)$ edges)!!
6) Guarantee of resource constraints:
$$\forall v_k \in V_T: \ \forall t: 1 \le t \le \max\{h_i : v_i \in V_S\}$$
$$\boxed{\sum_{\forall i:(v_i,v_k)\in E_R} \ \sum_{p'=\max\{0,t-h_i\}}^{\min\{w(v_i)-1,t-l_i\}} x_{i,t-p'} \le \alpha(v_k)}$$
Explanation: For each time step t, for each resource v_k, determine which operations could be starting *or* still be executing at time t.
Example: Operations:
   - $v_1: l_1 = 1, h_1 = 3$, can executed on resource $r_1, w(r_1) = 1$
   - $v_2: l_2 = 2, h_2 = 3$, can executed on resource $r_1, w(r_1) = 1$
   - $v_3: l_3 = 1, h_3 = 3$, can executed on resource $r_2, w(r_2) = 2$
   - $v_4: l_4 = 2, h_4 = 3$, can executed on resource $r_2, w(r_2) = 2$

   The resource constraints then are:
   - $t = 1$:
     - $r_1: x_{1,1} + x_{2,1} \le \alpha(r_1)$
     - $r_2: x_{3,1} + x_{4,1} \le \alpha(r_2)$
   - $t = 2$:
     - $r_1: x_{1,2} + x_{2,2} \le \alpha(r_1)$
     - $r_2: x_{3,1} + x_{3,2} + x_{4,1} + x_{4,2} \le \alpha(r_2)$
   - $t = 3$:
     - $r_1: x_{1,3} + x_{2,3} \le \alpha(r_1)$
     - $r_2: x_{3,2} + x_{3,3} + x_{4,2} + x_{4,3} \le \alpha(r_2)$

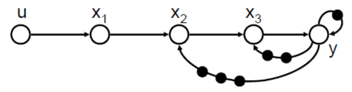
10.7 Synthesis for Iterative Algorithms (10-56)

Iterative Algorithms consist of a set of indexed equations that are evaluated for all values of an index variable I:
$$x_i[l] = \boldsymbol{F}_i[\dots, x_j[l - d_{ji}], \dots] \quad \forall l\, \forall i \in I$$
Here, x_i denote a set of indexed variables, $\boldsymbol{F}_i$ denote arbitrary functions and d_{ji} are constant index displacements.
Examples of well-known representations are signal flow graphs, marked graphs.
Multiple representations are possible:

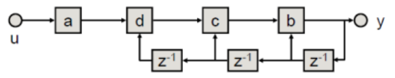
- One indexed equation with constant index dependencies:
$$y[l] = au[l] + by[l-1] + cy[l-2] + dy[l-3] \ \ \forall l$$
- Equivalent set of indexed equation:
$$x_1[l] = au[l] \ \ \forall l$$
$$x_2[l] = x_1[l] + dy[l-3] \ \ \forall l$$
$$x_3[l] = x_2[l] + cy[l-2] \ \ \forall l$$
$$y[l] = x_3[l] + by[l-1] \ \ \forall l$$
- Extended sequence graph $G_S = (V_S, E_S, d)$: To each $(v_i, v_j) \in E_S$ there is associated the index displacement d_{ij}. An edge $(v_i, v_j) \in E_S$ denotes that the variable corresponding to v_j depends on the variable corresponding to v_i with displacement d_{ij}



- Equivalent marked graph:



- Equivalent signal flow graph:



- Equivalent loop program:

```
while(true) {
        t1 = read(u);
        t5 = a*t1 + d*t2 + c*t3 + b*t4;
        t2 = t3;
        t3 = t4;
        t4 = t5;
        write(y, t5);}
```

Definitions:

- An iteration is the set of all operations necessary to compute all variables $x_i[l]$ for a fixed index l.
- The iteration interval P is the time distance between two successive iterations of an iterative algorithm.
- $1/P$ is the throughput of the implementation
- The latency L is the maximal time distance between the starting and the finishing times of operations belonging to one iteration.
- In functional pipelining, there exist time instances where the operations of different iterations l are executed simultaneously.
- In case of loop folding, starting and finishing times of an operation are in different physical iterations.

10.7.1 Solving Synthesis Problem using ILP (10-64)

1) Start with ILP formulation using simple sequence graph
2) Use extended sequence graph, including displacements d_{ij}
3) ASAP and ALAP scheduling for upper and lower bounds h_i, l_i. Use only edges with $d_{ij} = 0$
4) Guess a suitable iteration interval P. If this is not feasible, increase P.
5) Replace equation 5 with
$$\forall (v_i, v_j) \in E_S: \tau(v_j) - \tau(v_i) \ge w(v_i) - d_{ij} \cdot P$$
6) Replace equation 6 with
$$\sum_{\forall i:(v_i,v_k)\in E_R} \ \sum_{p'=0}^{w(v_i)-1} \ \sum_{\forall p:\, l_i \le t-p'+pP \le h_i} x_{i,t-p'+pP} \le \alpha(v_k)$$

10.8 Dynamic Voltage Scaling (DVS) with ILP (10-67)

Optimize energy in case of DVS using ILP:

- $|K|$ different voltage levels
- A task $v_i \in V_S$ can use one of the execution times $\forall k \in K$: $w_k(v_i)$ and corresponding energy $e_k(v_i)$
- Deadlines $d(v_i)$ for each operation
- no resource constraints

1) Minimize
$$\sum_{k\in K} \sum_{v_i \in V_S} y_{ik} \cdot e_k(v_i)$$
subject to the constraints 2) – 5)
2) Binary variables:
$$\forall\, v_i \in V_S, k \in K: \quad y_{ik} \in \{0,1\}$$
3) One voltage $k \in K$ chosen for each operation:
$$\forall v_i \in V_S: \sum_{k\in K} y_{ik} = 1$$
4) Precedence constraints:
$$\forall (v_i, v_j) \in E_S: \quad \tau(v_j) - \tau(v_i) \ge \sum_{k\in K} y_{ik} \cdot w_k(v_i)$$
5) Guarantee of deadlines:
$$\forall v_i \in V_S: \quad \tau(v_i) + \sum_{k\in K} y_{ik} \cdot w_k(v_i) \le d(v_i)$$