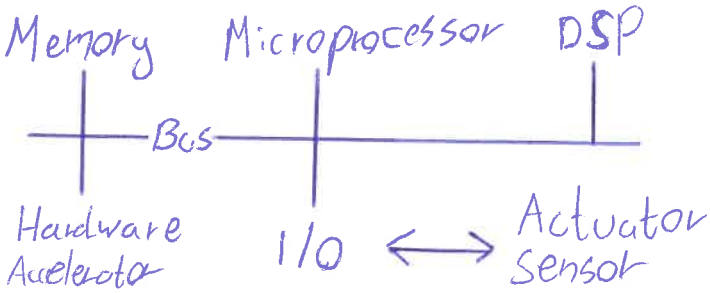
# IL2206 Embedded Systems

## Development

Embedded systems are developed for a single task whose functionality will never change. Criterias:

- Design cost
- Real time
- Saftey
- Power efficiency
- time-to-market
- optimized
- Small overhead




As you lower the abstractions from model to implementation, the design space becomes smaller.

## Microprocessor

Cheap & generalized hardware programmed in C.

## RISC & CISK

Instructions are usually 4 Bytes

### Von Neumann

Single memory for both instructions and data.

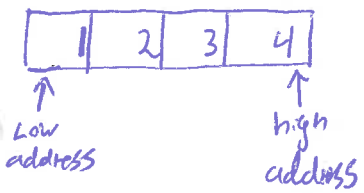
### Harvard

Two different memories, one for data one for instructions. Makes execution more parallel

## Big Endianness

Most significant byte is placed first and least significant placed last.

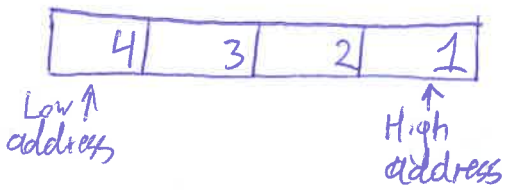
0x01020304




## Small Endianness

Least significant byte is placed first and most significant is placed Last.

0x01020304

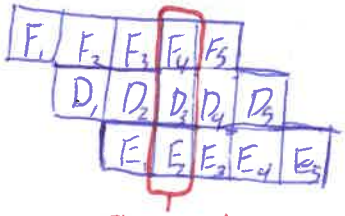



## Processor Pipeline

Since the processor has multiple Steps that are executed sequentially a staggered execution of multiple steps is possible. Using an n-staged pipeline. This requires the entire pipeline to be flushed when a branch is made.

### 3 stage pipeline

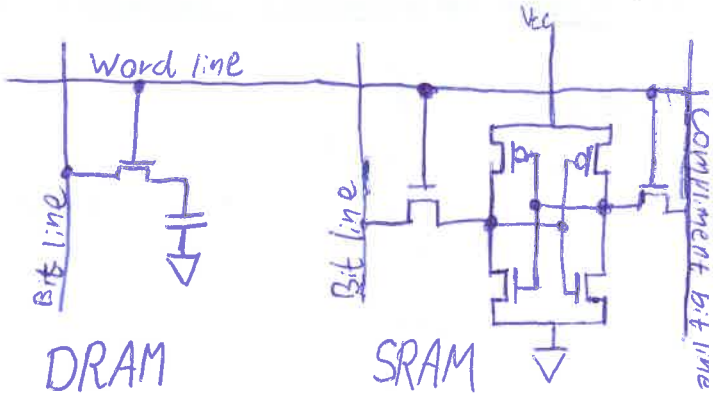
fetch, decode, exequte




Things done at the same time

# SRAM & DRAM

Sram is faster but more expensive
Sram is built with 6 transistors, while Dram is built with a transistor and capacitor and is slower. Dram also needs to be Periodically refreshed.




DRAM SRAM

| | DRAM | SRAM |
|---|---|---|
| Access time | 50-70 ns | 0.5-5 ns |
| Cost/GB | 100-200 $ | 4000-10 000 $ |

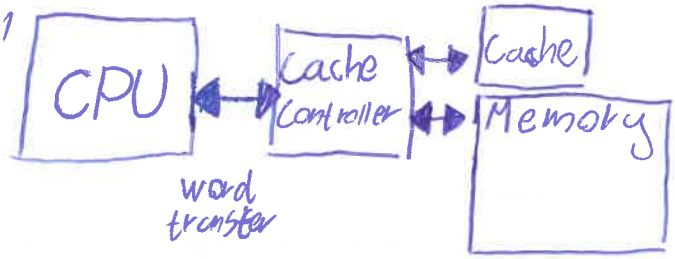
# Cache Memory

A small but fast memory close to the Processor that contains a copy of the content in main memory that is used most frequently. (See spatial ant temporal locality)

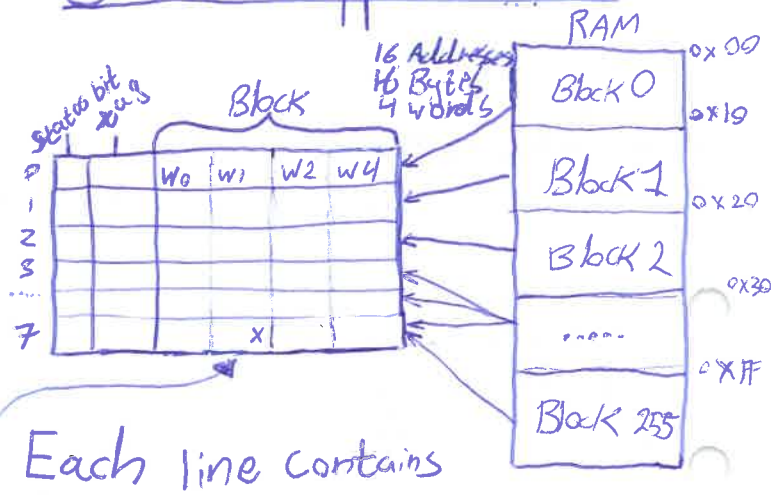
$$\text{Hit rate} := \frac{\text{Hits}}{\text{Requests}}$$

- Cache hit: required memory is in the cache
- Cache miss: required memory is not in the cache
- Working set: set of locations used by program in an interval of time.
- Compulsory miss: location that has never been accessed before.
- Capacity miss: working set is too large.
- Conflict miss: two locations in working set are mapped to the same entry

Harddisk < SSD < DRAM < SRAM < Cache < Register




# Direct Mapped Cache



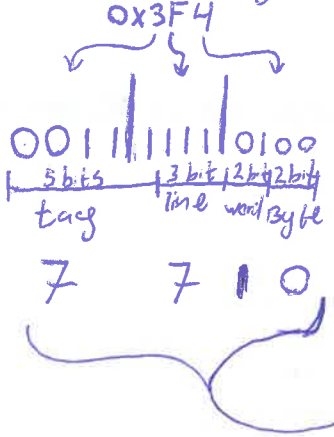

Each line contains status bit, tag and block.

Status Bit: Indicates if a line is valid, Invalid or Dirty.

Tag: Keeps track of exactly which block that is in the cache. The same blocks always gets the same line.

Cache Line = Block number % Number of lines

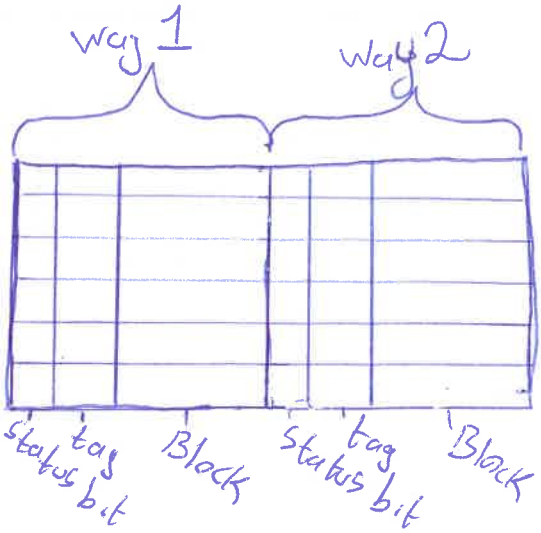
Block: 16 bytes of data copied from main memory.



# 2-Way Set-Associative Cache

Instead of directly mapped cache, the cache is expanded horizontally in order to prevent excessive conflict misses. This is more flexible but comes at the cost of having to look in both locations for the data.

Every time data is loaded both ways need to be checked. However it can be done in parallel.




# N-way Set associative Cache

Increasing the number of ways in the cache can progressively be done until eventually the cache is fully associative. At which point the exact data needed can be put in the cache without conflict misses. for this to work effectively lots of hardware is required so it can only be justified for small caches.

# Block Replacement

When a conflict occurs blocks can be replaced based on a few factors. Most used are:

Random: replace random block

Least Recently used: Replace oldest

# Write-through & Write back

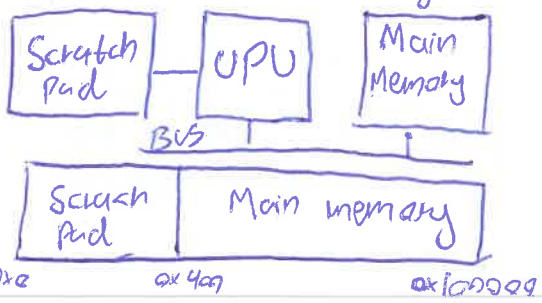
Write-through always writes back to main memory when the cache is modified. This is slow, and requires a Valid (V) and invalid (I) Bit for each line. All caches are invalid at system start.

Write back only writes the cache back to main memory when the block is removed from the cache. This requires a modified (M) (dirty) status bit.

# Scratchpad

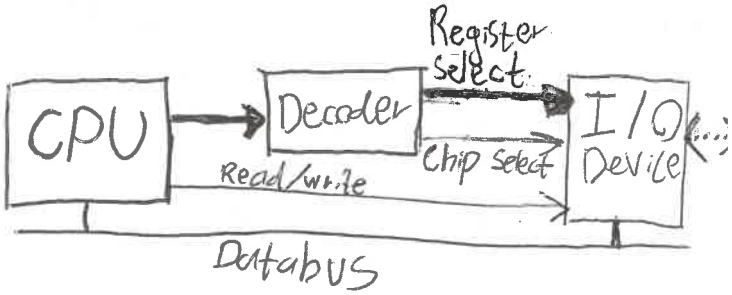
Caches give bad estimate for worst case execution time (wcet). In this way a scratchpad memory gives worse average performance but better wcet.

Scratchpad works by dividing the address space between fast and main memory and the designer must make sure to put frequently used addess in the fast memory.



# Memory Mapped I/O

A system needs to communicate with the environment. This is done with I/O. Memory mapped I/O have certain addresses that interact with pheripheral devices.




## Busy-Wait & Interrupts

Continiously checks in software if the I/O state changes.

```
while (Buton not pressed)
  {
    do nothing
}
/* do stuff */
```

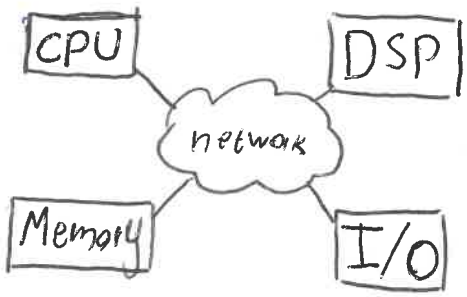
The processor is wasting time while button is not pressed

The processor gets notified when a state change occurs. The Processor can do whatever until an interrupt occurs and the interrupt service Routine occurs and takes over. Afterwards the original program + program counter are restored.

Pheripheral has registers that can be read and written to. They are selected based on the address and chip select signal The data is transferred over the regular bus.

# Network on Chip

Each device is considered as a seperate device and can communicate in parallell using a network like protocol.




# The BUS

A shared communication link that is easily implemented.

Communication uses a serialised format, where only one device can write to the bus at the same time, while multiple devices can read at the same time.

## Synchronious Bus

A Bus that is synchronious requires a clock signal for data to be sent. Requires little extra logic and can run fast. However devices must share the same clock. This can result in clock-skew for large systems. Has CLK, READ, ADR, DATA

## Asynchronious Bus

Does not use a clock but a handshake is performed instead.

a) controller writes addres and activates READ
b) target activates DATA and ACK
c) Controller deactivates READ when done reading
d) target deactivates ACK

has READ, ADR, DATA, ACK

# Bus Arbitration

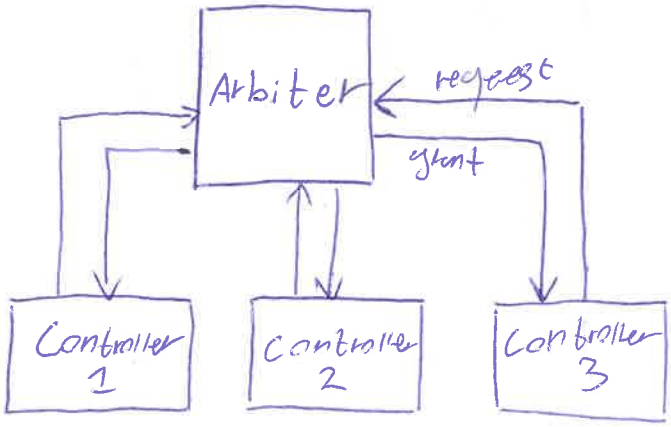
In order to prevent multiple messages being sent at the same time. A bus controller desices which targets may access the bus.

This balances two factors, Priority & fairness.

Weak fairness: Any request will eventually be serived.

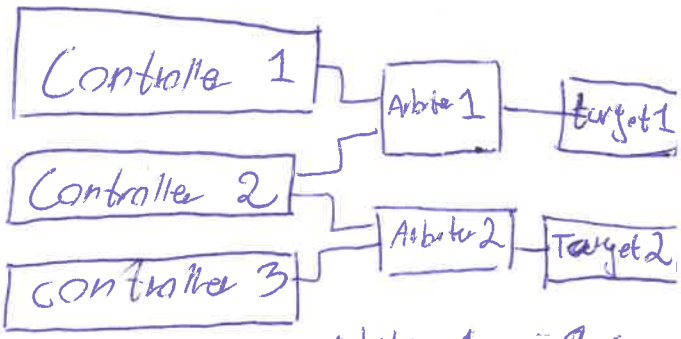
strong fairness: everything shares the bus equally.

weighted strong fairness: Each device has a weight and gets access proportionally to it.




# Target-Side Arbitration

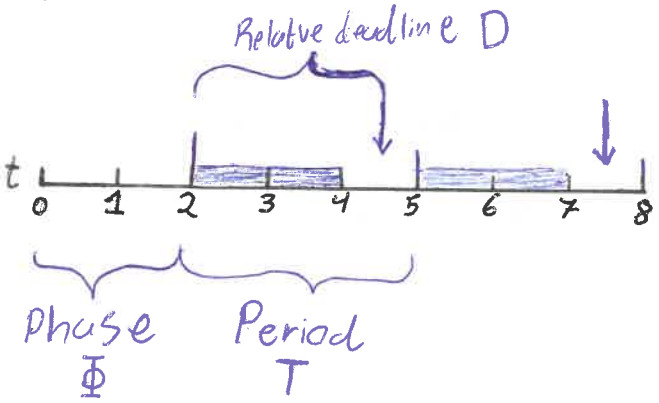
Use multiple Arbiters when you have many targets. This allows for simultaneous access to different targets, greatly improving performance.




Arbiter 1 and 2 can work in parallel.

# Real Time Systems

A system that works on a timely basis is a realtime System. This can be modeled With schedules




Phase Φ: Time from start of system until first task is launched.

Period T: How often a task is launched.

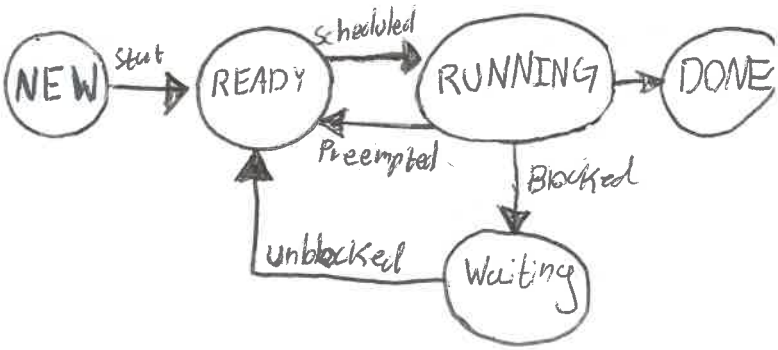
Deadline D: Time until a task is required to be finished

- Hard: Missed deadline has catastrophic consequenses
- Firm: Results produced after deadline are useless.
- Soft: Late results are useful but cause performance degregation

# Concurrency & Parallelism

Concurrency is what appears as parallel execution, but in reality a single execution is consistently swapped between different Jobs.

Parallelism requires extra hardware and actually carries out multiple calculations at the same time.




# Semaphore

Has two values V and L. V is a natural number that is initialized to $\mathbb{R} \geq 0$. L is a list of blocked tasks.

## Pend

```
if S.V > 0 {
    S.V --;
} else {
    S.L += p   // process
    P.State = Blocked
}
```

## Post

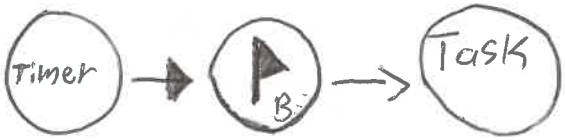
```
if S.L = ∅ {
    S.V ++;
} else {
    q = S.L.popO();
    q.state = Ready
}
```

# Message Queues

A task can wait for a message to be sent to a specific que and in that time yield its execution. and then be scheduled once the message is received.

# Soft Timer

Two types, Periodic & One shot.
Combination of soft- and hardware
Usually is used to signal
semaphores periodically.




# Real Time System Model

Three parts, workload, Resources and scheduler.

Important terminology:

- task: $T_i$
- Job: $k$ $T_{i,k}$
- Period: $T_i$
- computation time/ execution time: $C_i$
- Phase: $\phi_i$
- release/ arrival time: $r_i / a_i$
- Absolute deadline: $d_i$ $\quad d_i = \phi + r + D$
- Relative deadline: $D_i$
- Start time: $s_i$
- Finish time: $f_{i,k}$
- Response time: $R_{i,k}$ $\quad R_{i,k} = f_{i,k} - r_{i,k}$
- Lateness: $L_{i,k}$ $\quad L_{i,k} = f_{i,k} - d_{i,k}$
- Tardiness: $E_{i,k}$ $\quad \max(L_{i,k}, 0)$

Periodic task $T_i$

$T_i = (\phi_i, T_i, C_i, D_i)$

$\phi$ = Phase (default = 0)

T = Period

C = execution time

D = Deadline (default = T)

# Scheduling Algorithms

Desides when a task runs,
Static Scheduling occurs at compiletime
while dynamic scheduling occurs at
Runtime. A fesible schedule is a schedule
where all constraints are met. An Optimal
scheduler will always produce a fesible
schedule if it exists.

Basic realtime model:

$T_i = (T_i, C_i)$

Tasks are released periodically
$T_i = D_i$ $\quad \phi = 0$
All jobs must complete
before their deadline.
Tasks can always be preempted
No Dependency between tasks
0 context switch time
Single thread processor

# Rate-Monotonic Scheduler

Higher Priority based on how short your period is. the Task that is ready with the highest priority will always be scheduled. RM scheduling sometimes fails to find feasible schedules, and is thus not an optimal scheduler.

## Hyperperiod

Hyperperiod is the lowest common multiple of all periods.

$$H = LCM(T_1, T_2, \dots T_N)$$

If a schedule can be made for one entire hyperperiod, the schedule is feasible.

# Earliest Deadline First

Instead of having static priorities EDF gives the highest priority to the task with the earliest deadline at any given time. The scheduler itself reqires more computation and information about the tasks.

## optimality of EDF

EDF is optimal if preemption is allowed, jobs do not compete for resources,

# Schedulable Utilization

Utilization of a task set is the sum of the utilization of each task. A scheduler can schedule any task set if the utilization is less or equal to the schedulable utilization $U_{alg.}$ of that algorithm.

If the density of a set of tasks is less than 1 there is a feasible schedule for the tasks.

For RMS if utilization is less than or equal to

$$U_{RM}(n) = n(2^{\frac{1}{n}} - 1)$$

$$n \to \infty \Rightarrow U_{RM} = \ln(2) = 0.693$$

$$Utilization(T_i) = u_i = \frac{C_i}{T_i}$$

$$Utilization = U = \sum_{i=1}^{n} u_i$$

$$U \geqslant 1 \Rightarrow infeasible$$

$$Density(T_i) = \delta = \frac{C_i}{\min(D_i, T_i)}$$

← Use if there is a deadline that is less than the period

$$\Delta = \sum \delta_i$$

## Critical Instance

For a fixed priority system the critical instance of a task $T_i$ occurs when one of its jobs is released at the same time as a job in every higher Priority task

# Time Demand Analysis

Analyses the time demand for all tasks at their critical instance. If $w_i(t)$ is less than $t$ for any $t \leq D_i$ all Jobs of $T_i$ can complete before its deadline. If the opposite is true that Job cannot be completed. If all tasks meet their time demand at ther critical instance the taskset $\Gamma$ is schedulable.

$$w_i(t) = C_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{T_k} \right\rceil C_k \quad \text{for } t < T$$

# Response Time Analysis

Computed based on response time at the critical instance. Also has an iterative solution to the response time. Neither of these ways account for:

- Low prio Jobs blocking high prio Jobs
- Preemption is always possible
- Jobs never suspend themselves
- context switch is zero
- Priorities are constant

$$R_i = C_i + I_i \quad \text{(interference from other tasks)}$$

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

# Priority Inversion

When a low priority task uses a resource that a high priority task wants to use, a deadlock or missed deadlines can happen. This can also cause timing anomalies where a task executing faster may result in worse performance.

# Priority Inheritance

When a tasks blocks anotner tast it inherits the priority of the blocked task to allow for the blocket task to be unblocked quicker.

# Hardware/Software Co-Design

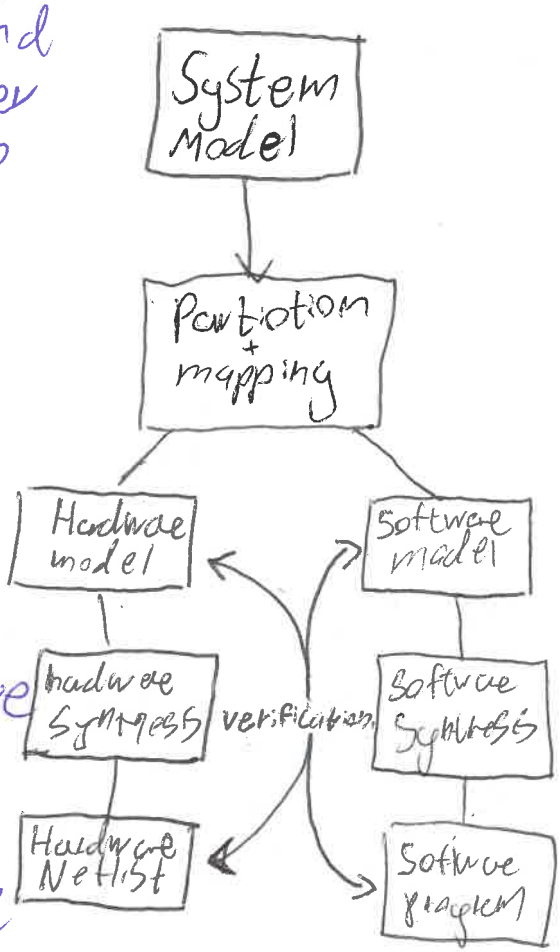
Hardware is significantly faster and more efficient than software, however it is less flexible. Combining the two in the following:

Co-Specification: How to write specification that addresses both soft & hardware?

Co-Synthesis: How to generate both hardware & software?

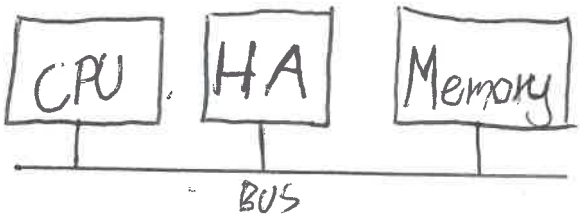
Co-Simulation: How can a system be simulated on multiple layers of abstraction?

Co-Verification: How can a system of both hard- and software be verified?




# Hardware Accelerator

Hardware is simply faster than software. Critical parts of a system can be speed up by hardware. The memory is often shared between cpu and HA and thus communication overhead needs to be accounted for. From the equations it is evident that in order to achive a high speedup, both f and s must be maximized. The same logic also applies on parallelization.




## Amdahl's Law

Speedup is defined by

$$S = \frac{C_{old}}{C_{new}}$$

theoretical speedup

$$S_{theo} = \frac{1}{(1-f) + \frac{f}{S}}$$

f = fraction of system that benefits from speedup
S = speedup factor of affected sections

# Parallelism

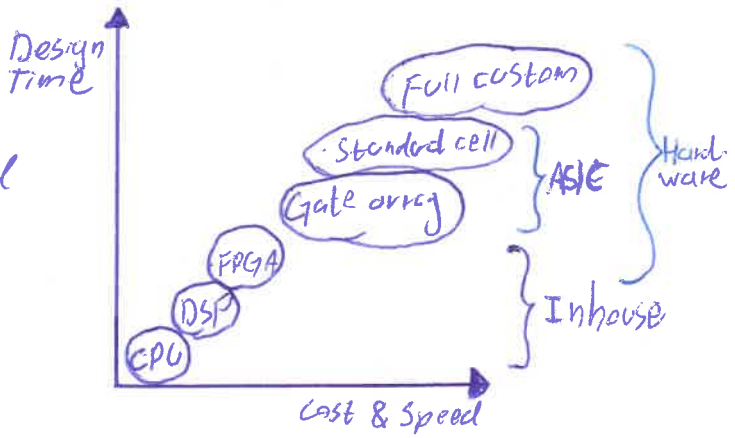
Functions can be executed in parallel if there is no dependency between them, I/O can be overlapped with computation, buffers can be used.

# Caches

Parallel Systems make cache coherence much difficult.
one strategy is Bus Snooping since all bus transactions can be read and update the cache that way

# Digital Signal Processor

Processes computation intensive applications, in particular multiply and accumulate. comes in fixed and floating point calculations.




# Fixed point

Twos compliment with a decimal point. First bit is the sign and the following are the powers of $2^{-n}$. Allowing for numbers between $-1$ and $1-2^{(n-1)}$ for $n=8$ $[-1, 0.9921875]$
There is no overflow, just precision loss.

$Q_4$

| 1 | . | 1 | 0 | 1 | 0 |
|---|---|---|---|---|---|
| -1 | | 0.5 | 0.25 | 0.125 | 0.0625 |

$-1+0.5+.125=$

$-0.375$

# Floating Point

Consists of:
- 1 sign bit s
- $n_e$ exponent bits
- $n_m$ mantissa bits
- bias which is predefined

|S| exponent| mantissa|

IEEE-754 defines 32 bit float as:
$e=8$ $m=23$ bias=127

$$D_{10} = -1^S \cdot 1.m \cdot 2^{e-bias}$$

$e=2$ $m=1$ bias =1

| s | e | m |
|---|---|---|
| 1 | 1 | 0 1 |

$= -1 \cdot 1.5 \cdot 2^{2-1}$
$= -1.5$

IEEE 754 special cases:

| e | m | value |
|---|---|---|
| 0 | 0 | $\pm 0$ |
| 255 | 0 | $\pm \infty$ |
| 0 | not 0 | Denormalized |
| 255 | not 0 | NaN |

# Float Point Multiplication

1) zero checks: multiplication with zero is zero ($e=0, m=0$)
2) add exponents: with $e = e_a + e_b + \text{bias}$
3) multiplication: Multiply Significands while taking sign into account.
4) normalize and round: The Significand must be shifted so leftmost bit is one. Significant might need to be rounded.

# Floating Point Addition

1) zero check
2) Align Significands the exponents are equal.
3) add Significand, taking sign into account.
4) normalize resulting significand so that the leftmost bit is one.

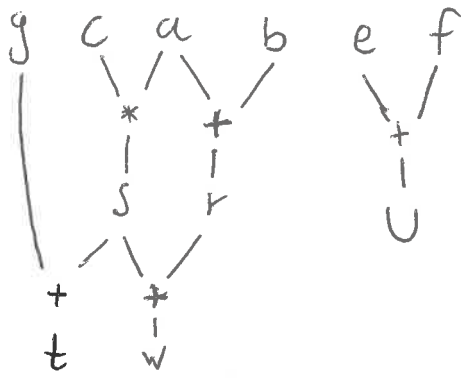
# Float vs Fixed

Float has a larger range but also has rounding errors

# Program Analysis

Data flow & control graph extracts how data depents on each other and visualizes how execution can be handeled

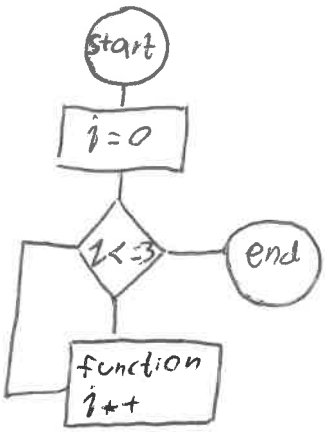
```
r = a + b
u = e + f
s = a * c
t = g + s
w = r + s
```




## Compiler optimization

Looking at the controlflow the compiler can make the code faster, or smaller depending on its settings.

```
i = 0
while i <= 3
    function();
    i++
```




## Inlining

Instead of using a function call, the code of the function can be directy inserted

```
Fn(x, y, z){
  return x+y+z
}
```

## Loop Unrolling

instead of a loop the code can just be repeated.

```
for(x=0; x<3; x++){
    fnc();
}
```

## loop fusion

Combining two loops into one.

```
for x in 0...3
    fn1()
for y in 0...3
    fn2()
```

## Registers

Reorder code so that register usage is smaller.

## Platform optimization

Some equivalent functions are faster on some platforms.

```
a*8 = a<<3
```

## GNU

has the following Compiler flags

- -O optimise Lvl 1
- -O2 optimize LVL 2 includes space tradeoff
- -O3 optimizes LVL 3
- -O0 make debugging easer
- -Os optimize for size