Embedded Systems and Advanced Computing

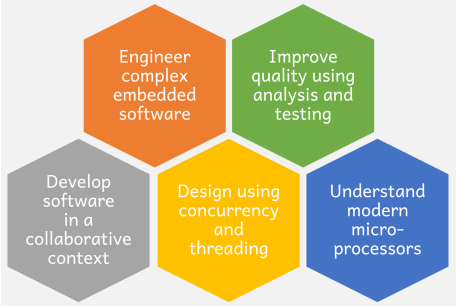
ENCE464

Andy Ming / Quelldateien

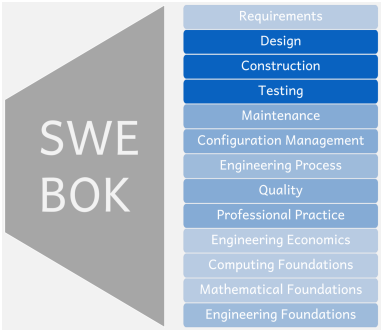
Table of contents

How to work code

Remember that software engineering is 50-70% maintenance. Because modern machines heavily rely on microcontrollers there is great demand.sol

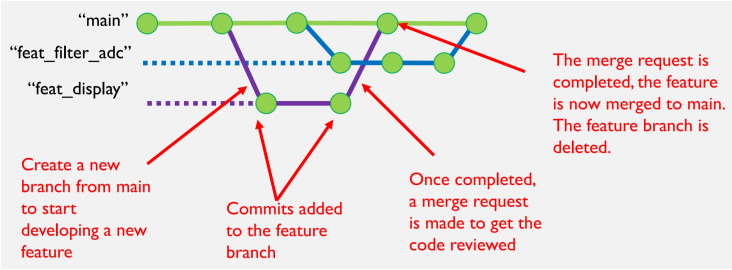


Software engineering has many different aspects (the dark blue ones are focused on here), find out more here.



Feature Branches

To implement different features, use a branch per feature, this guarantees that the main is always in working condition.



Branching Rules

- Feature branches are **temporary** branches for new features, improvements, bug fixes or refactorings.
- Don't push directly to **master/main**.
- Each feature branch is owned by **one** developer.
- Only do merge requests on **complete** changes i.e. don't break main.
- Thoroughly test your change prior to **starting** AND prior to **completing** a merge request.
- Use your commit messages to tell the **story** of your development process.

To minimise integration issues:

- A feature branch should only hold a small increment of change
- If main is updated during feature development, merge the new main into your feature branch **locally**, **before** making a merge request

Clean Code

Smells of Bad Code

- *Rigidity*: Changing a single behaviour requires changes in many places
- *Fragility*: Changing a single behaviour causes malfunctions in unconnected parts
- *Inseparability*: Code can't be reused elsewhere
- *Unreadability*: Original intent can't be derived from code

Reveal Intent

```
// BAD
uint16_t adcAv;  // Average Altitude ADC counts
// GOOD
uint16_t averageAltitudeAdc;
```

Don't Repeat Yourself (DRY)

Avoid duplicate code → Put it into a function. Can you put it in a function? Then you should!

Consistent Abstraction

High-Level ideas shouldn't get lost in **Low-Level** operations.

```
// Bad Example
deviceState.newGoal = readADC() * POT_SCALE_COEFF;
↪  // low-level
deviceState.newGoal = (deviceState.newGoal /
↪  STEP_GOAL_ROUNDING)*STEP_GOAL_ROUNDING;  //
↪  high-level
```

Encapsulation

- *Hide* as much as possible
- *Public* Interface: Header File, only declare what other modules need to know
- *Private* / Inner Workings: Source File
- *Avoid* global variables → Use *getter & setter*

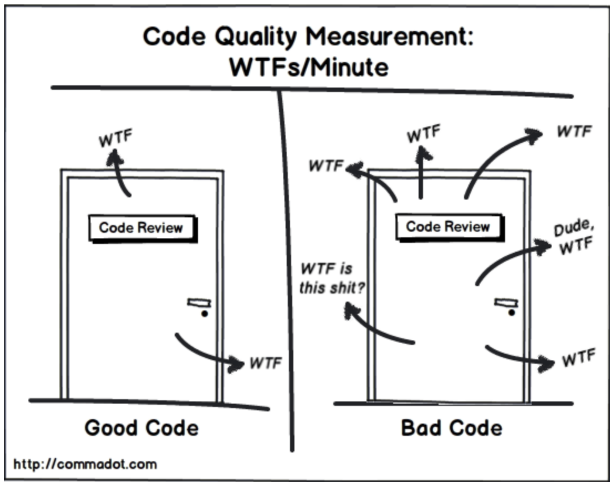
Comments

More comments ≠ better quality. Use comments only to:

1. Reveal intent after you tried everything else
2. Document public APIs - sometimes

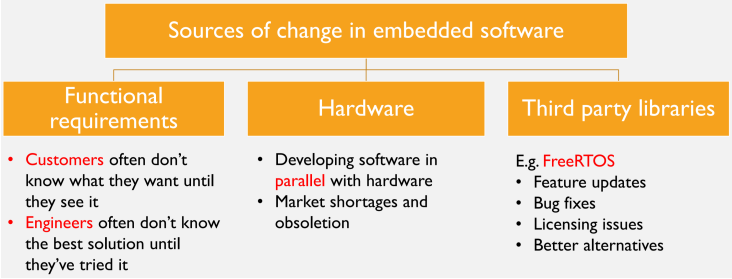
Code Reviews

Use *merge requests*, label feeback as *bug, code, quality, preference*.

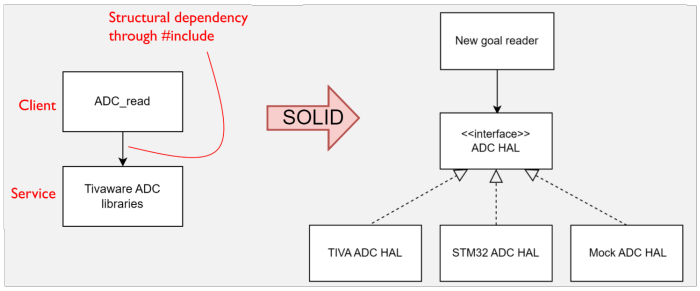


SOLID

How to make designs **flexible**, as requirements change all the time (Agile).



SOLID is all about **managing dependencies**.



Single Responsibility Principle (SRP)

"A module should only have a single responsibility. It should only have one reason to change"

Instead of `ADC_read` which handles the ADC reading, averaging and the setting of the new goal, we breack it up into a module `ADC_HAL` which handles the ADC reading and averaging and the module `new_goal_reader` which just handles the setting of the new goal.

- A module does **one thing** and does it **well**
- Use good names, reveal intent
- Don't access numerous data structures and globals
- If the high level requirement changes, it only affects a single module
- "Hide design decisions"

Open-Closed Principle (OCP)

"Software entities (modules, functions) should be **open to extension, but closed for modification***"*

The `new_goal_reader` doesn't have to be modified in case of a hardware change. But it's functionality can be extended through swapping out the HAL implementation.

- Changes through adding code instead of modifying
- aka USB standard: Plug-n-Play, no hardware change needed
- Abstraction of the connection
- OOP use "interface" or "abstract class"
- C use "header files" or "function pointers"

Liskov Substitution Principle (LSP)

"Subtypes should be substitutable with their base types"

`TIVA ADC HAL` is perfectly substitutable with its base type of the `ADC HAL` interface.

- Exchanged modules behave the same way, no matter of the subtype
- Under the hood might do stuff differently but leads to the same output

Interface Segregation Principle (ISP)

"Clients should not be forced to depend on functions that they do not use"

The interface `ADC HAL` is defined on the needs of `new_goal_reader` (the client). Don't show unneeded functions.

- Write "small" interfaces
- Allows to limit dependencies
- Seperating concerns

Dependecy inversion Principle (DIP)

"High-level modules should not depend upon low-level modules. Both should depend on abstraction"

The interface `ADC HAL` is the abstraction and both, `new_goal_reader` and `TIVA ADC HAL`, are implementing this. `new_goal_reader` doesn't know (and care) which implementation is called.

In OOP this is called **Polymorphism**.

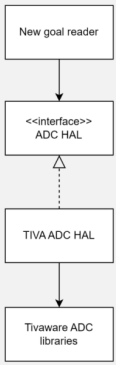
SOLID Models for C

How much design is enough?

- Use simplest flexible design for today's requirements
- Changing requirements → Change **Design**
- Tests ensure functionality is retained

Single Instance Model

- HAL function names same as Interface
- Linker binds interface & implementation
- Choice defined in CMake
- **Single** instance, **Static** binding

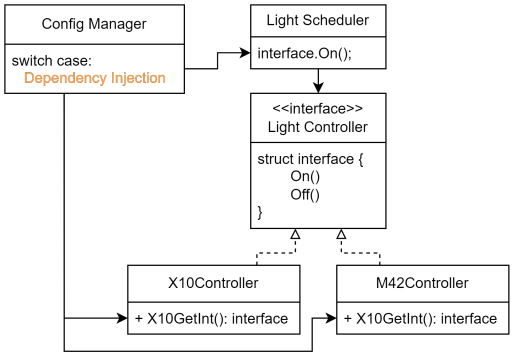


Multiple Instance Model

- Create multiple instances
- Use abstract data type (ADT) → object definitions and details are **encapsulated**
- Public functions to operate on the abstract data type `f(0,x)`
- **Multiple** instance, **Static** binding

```c
// In header file
typedef struc circBuf circBuf_t;

// In source file
struct circBuf {
    uint32_t size;
    uint32_t windex;
    uint32_t rindex;
    int32_t *data;
}
```

Dynamic Interface



- Configuration determined in runtime
- Interface is a *public struct of function pointers*
- **Single** instance, **Dynamic** binding

Pre-Type Dynamic Interface

- Support any number & combination of drivers
- Each type has its own constructor
- Objects cast to abstract interface
- Have a array of abstract instances
- **Multiple** instance, **Dynamic** binding

Design Patterns

23 patterns introduced by the **G**ang **o**f **F**our (GoF), for **Dependency Management**. There are 3 types:

- **Creational**: Create instances of objects
- **Structural**: Set communication pathways
- **Behavioural**: Distribute responsibilities

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

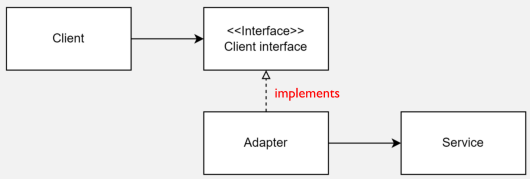
Recommendations

- Don't overcomplicate, design for todays requirements
- Use pattern if beneficial, maybe simple code is sufficient

- Customise patterns to application

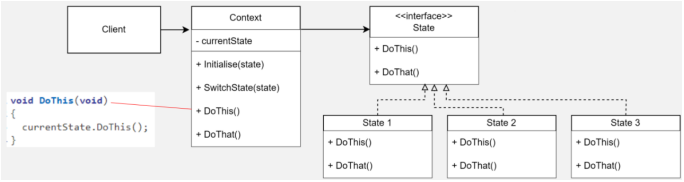
Adapter Pattern

- What: Wrap adapter around another module to give it a more desirable interface
- Hiding ugly interfaces of a 3rd-party service
- Hiding data conversions
- Make incompatible modules compatible
- Fulfills SRP, ISP, LSP



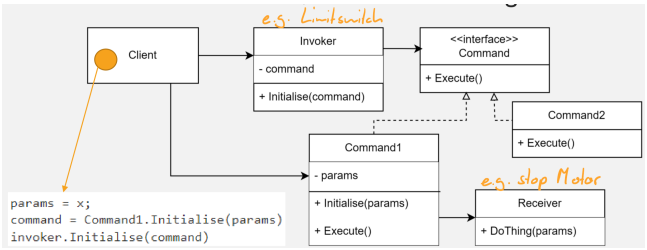
State Pattern

- What: Module will behave differently depending on internal states
- Allows implementation of FSM without several lengthy conditional statements
- State is saved in the private `currentState` variable
- There is a module implementation per state
- The client initialises the context with a state
- Fulfills SRP, OCP



Command Pattern

- What: Turns request into a stand-alone object
- Invoker calls receiver through command object
- Client attaches invoker with its commands
- Multiple things can execute one command
- Command is placed in queue until receiver is ready
- Triggers can invoke series of commands
- Fulfills SRP, OCP



Legacy / Vererbt / Veralteter Code

If handed bad quality (no tests, no encapsulation, no good practices, . . .) and you have to add features you can either (1) *add to the mess by hacking in new features* or (2) *rewrite code from scratch*.

The issues are (1) will *reduce productivity* and it's *easy to introduce bugs* but (2) is very *time consuming*, it's *difficult to maintain two versions* (there might still be old versions in the field which have to be maintained) and there will be a *new set of bugs* to deal with.

So we try to refactor until it's easy to make changes. To preserve functionality we iterate *in small increments* with **Trageted Tests** *(allows changes and new features to happen at the same time)*:

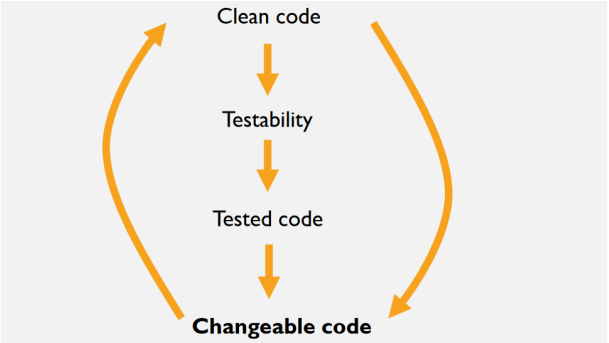
1. **Write tests** of the code you need to change
2. **Test drive** changes to legacy code
3. **Test drive** new code
4. **Refactor** tested area

Also add **Characterisation Tests** where understanding is required. *Tests are a living documentation.*

To make sure *key functionality* isn't altered, add **Strategic Tests**. (e.g. control algorithm, safety-critical error detection, . . .)

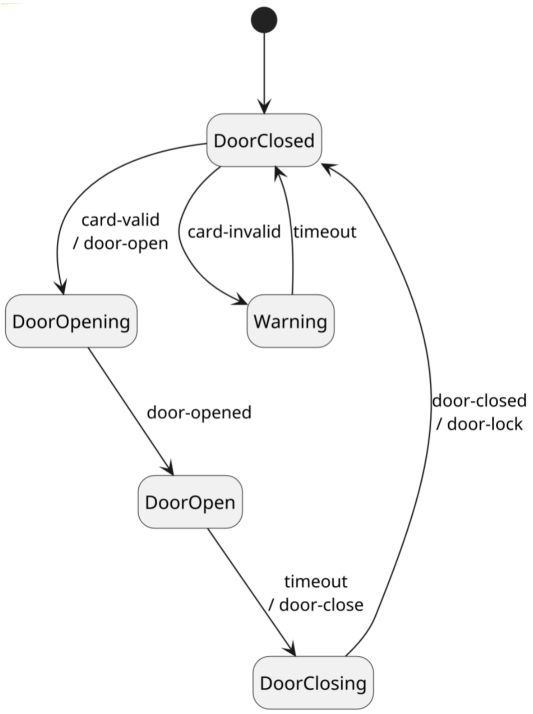
Putting Code Under Test

1. Identify are of code to test (targeted, characterisation, strategic)
2. Find test points (function calls, global variables → encapsulate ASAP, serial)
3. Break dependencies (Solitary tests)



Designing with Models

Event Driven State Machine



With a model there are already some flaws which arise. For example: What happens if the door is closing and a valid card is wiped?

Programm Execution:

- State: PC + Variable Values
- Transitions: Instructions

Time Driven State Machine

PID-Controllers are state machines with "near infinite" states

```
errorInt += error;
errorDeriv = error - lastError;
commandSignal = (Kp*error +
                 Ki*errorInt +
                 Kd*errorDeriv);
```

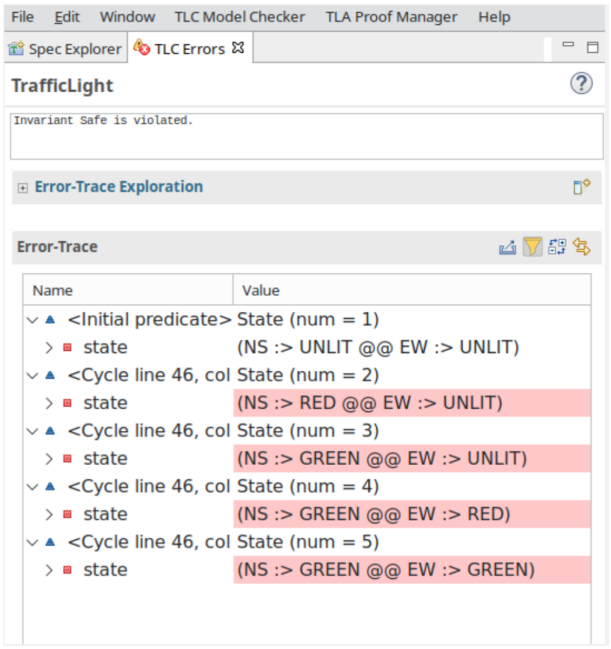
"States"

Use modelling languages

The following example is written in PLUSCAL

```
Light == {UNLIT, RED, GREEN, AMBER}
Direction == {NS, EW}

variables
    state = [dir \in Direction |-> UNLIT];

process Lights \in Direction
begin
    Cycle:
        either await state[self] = RED;
               state[self] := GREEN;
        or await state[self] = GREEN;
               state[self] := AMBER;
        or await state[self] = AMBER;
               state[self] := RED;
        end either;
    goto Cycle;
end process;

// Define a Requirement
define
    Safe =/ ~(state[NS] = GREEN / state[EW] = GREEN)
end define;
```

The model can be checked and we can verify if the requirement is met or not. Add to the model until it meets all requirements



Embedded Software Design

Architecture

Architecture are "important" structures, every structure is important for a specific part of the software. There are several different structures in embedded software systems.

> **Architecture Goals**
>
> - *Understandability* - In Development & Maintenance
> - *Modifiability* - Through "best practices"
> - *Performance* - Reduce Overheads
>
> Other possible requirements: Portability, Testability, Maintainability, Scalability, Robustness, Availability, Safety, Security

Static Structures: Conceptual abstraction a developer works with

Structure	Elements	Relationships
Decomposition	Modules, functions	Submodule of
Dependency	Modules	Depends on
Class	Classes	Inheritance, association

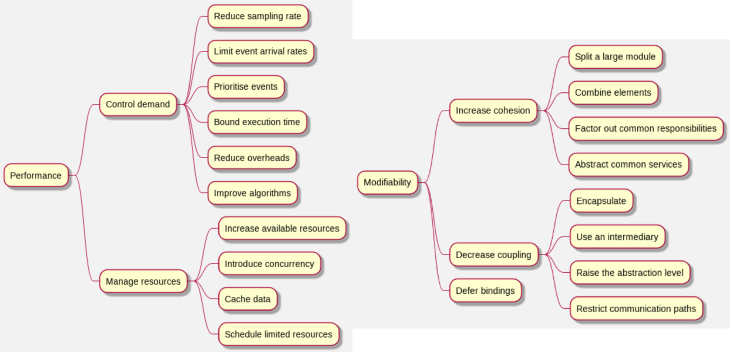
Dynamic Structures: Relationships that exist in executing software

Structure	Elements	Relationships
Collaboration	Components	Connections
Data-flow	Processes, stores	Flows of data
Task	Tasks, objects	Interactions

Allocation Structures: Assignment of software elements to external things

Structure	Elements	Relationships
Memory Map	Data, addresses	Allocated to
Implementation	Modules, files	Allocated to
Deployment	Software, hardware	Allocated to

Patterns are always a combination of tactics, depending on what you're trying to achieve.



Trade-Offs

Quality attributes can conflict with each other. For example:

Quality	Often trades with	Notes
Modifiability	Time performance	Modifiability often requires indirection, which introduces overheads. Removing indirection makes modifications more difficult.
Testability	Time performance	Testability, like modifiability, often requires indirection, which introduces overheads.
Execution speed (performance)	Memory (resource use)	The classic trade-off in software design. Reduce computation time by using more memory, or vice versa.
Responsiveness (performance)	Power (resource use)	Responding quickly may involve avoiding the latency involved in waking from sleep.
Determinism	Responsiveness (performance)	Ensuring deterministic timing of actions may involve polling instead of responding to events as they happen.

Keep Record of Decisions

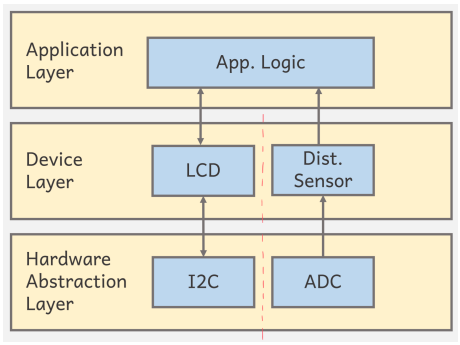
To keep record of decisions and to not loose the overview use tools like:

- *Architecture Haikus*: A onepager overview of your document see here or in the appendix folder.
- *Architecture Decision Records*: A incremental document to record decisions on the go either in a tool or a markdown file.

Layered

Each layer is is providing services to the above layer through well-defined interfaces. Each layer can only interact with the layer directly above or below.

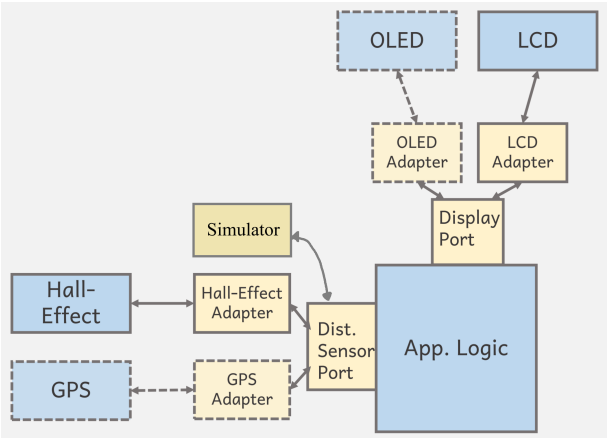
Supports portability and modifiability by allowing internal changes to be made inside a layer without impacting other layers, and isolating changes in layer-to-layer interfaces from more distant layers.



Ports-and-Adapters (or *Hexagonal*)

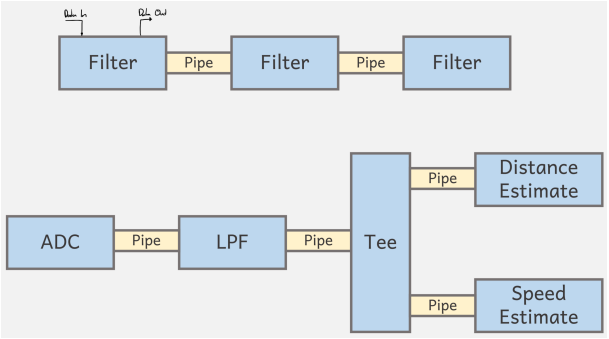
Introduces a single core logic which communicates through abstraction interfaces (**Ports**) to different modules. The **Adapters** map the external interactions to the standard interface of the port.

Supports portability and testability by making the inputs to the ports independent of any specific source, and supports modifiability by creating a loose coupling between components.



Pipes-and-Filters

Supports modifiability through loose coupling between components, and performance by introducing opportunities for parallel execution.

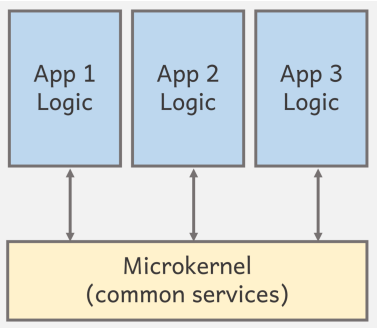


Microkernel

RTOS is a implementation of a Microkernel Architecture. The **Microkernel** includes a set of common core services. Specific

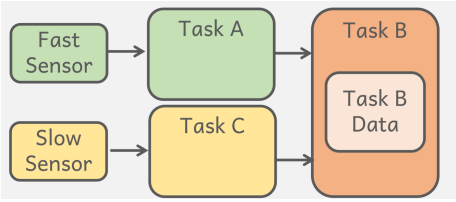
services (**Tasks**) can be plugged into the kernel.

Supports modifiability and portability.



Tasks for Priority and Modularity

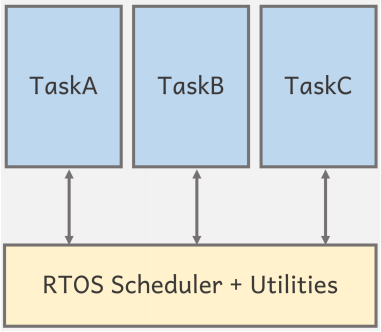
+ Control Priority
+ Control Response Time
+ Modularity
- Concurrency Issues
- Overhead (Scheduler)
- Starvation (Task gets no CPU time)



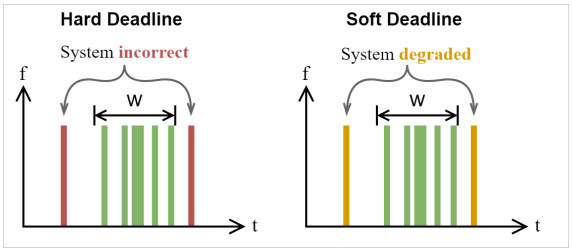
Use **just enough** tasks.

RTOS

To improve **Performance** we introduce **Concurrency** (Run tasks in parallel).



Preemptive approach: *Separation* of concerns, *Scalability*, State is *Managed*.
Do the **right thing** at the **right time** *W*

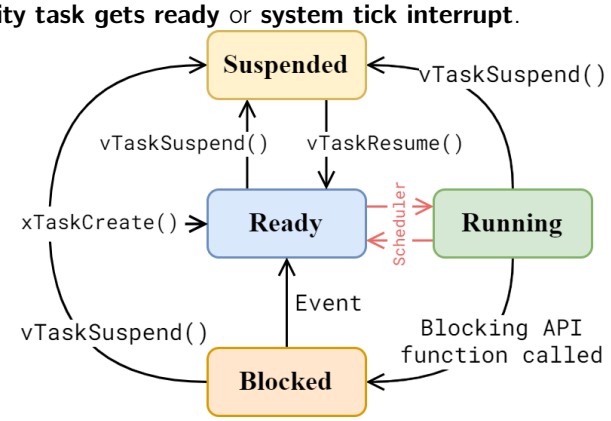


> **RTOS vs. Desktop OS**
>
> - Desktop OSs don't try to achieve *hard* real-time performance
> - In a Desktop OS, programs can be loaded in runtime
> - RTOS is compiled as part of the application, to add a new "program" the application has to be recompiled

Tasks

```
xTaskCreate(
    BlinkTask,     // Function that defines task
    "Blink Task", // Task name (used in debugging)
    STACK_SIZE,    // No. of 4-byte words for stack
    NULL,         // Optional pointer to task argument
    PRIORITY,      // Higher number = higher priority
    NULL);         // Optional pointer to task handle
```

Taskswitches happen at *scheduling points* which occur when a **task is blocked**, **interrupt causes a task**, **priority change**, **higher priority task gets ready** or **system tick interrupt**.



Stack Size

> **!** Stack Size Value
>
> The stack size value passed in `xTaskCreate` is measured in **4-Byte** words.
> Set high margins, something like **300%**

- Minimal: `configMINIMAL_STACK_SIZE`
- Maximal: Device *RAM*
- Actually: Analyse
  - Dynamic: Set something and see if it works / use `uxTaskGetStackHighWaterMark` to measure
  - Static: Use tools (e.g. GCC `-fstack-usage`) to attempt reading on how much stack is needed per function

Priority Task priority has a strong influence on when a task is run and thus on the overall behaviours of the application.

Assign priority based on importance

1. Separate tasks into "critical" (hard deadline) and "non-critical" (soft deadline)
2. Assign low priority to non-critical tasks
3. To be sure about critical tasks meeting their deadlines, apply scheduling theory

Assign non-critical tasks to low priorities

1. Either apply the same priority for all non-critical tasks
2. Or prioritise by *importance*, which depends on
   a. Shortness of Deadline

b. Frequency of Execution
c. Need for Precessor time

Assign critical tasks deadline monotonic priorities
Apply priority based on the size of it's deadline.

1. Highest ← shortest deadline
2. Lowest ← longest deadline

Deadline / Rate Monotonic Priorities

Deadlines D_i and Period T_i for each task i
Deadline Monotonic: $D_i \leq T_i$
Rate Monotonic: $D_i = T_i$
Futhermore, following assumptions are made:

- Fixed-priority preemptive scheduling
- Hard-Deadline tasks are either:
  - *Periodic* (fixed interval)
  - *Sporadic* (known minimum time between triggering events)

Check Schedulability of Critical Tasks To check if deadlines can be met (schedulable) we calculate the **response time upper bound** R_i^{ub} for each task i. This has to be less than the task deadline D_i

$$R_i^{ub} \leq D_i$$

The tasks are ordered after priority from $i = 1$ (highest priotity) and so on. Then Calculate the upper bound for every task through

$$R_i^{ub} = \frac{C_i + \sum_{j=1}^{i-1} C_j(1 - U_j)}{1 - \sum_{j=1}^{i-1} U_j}$$

C_i worst case execution time (WCET)
U_i utilisation $U_i = \frac{C_i}{T_i}$
T_i task period

Thus $R_1^{ub} = C_1$ ans each lower priority task has a response time that depends on the utilisation of the tasks above it.

Response Time Upper Bound

- If task $R_i^{ub} \leq D_i$ checks, task is practically schedulable
- If task fails test, there is still chance for it to work, as there've been many assumptions
- Response times tests don't account for task interactions and os overhead
- Tests depend on some kind of worst case execution time per task

Task	i	Deadline	Period	WCET	U
A	1	1 ms	10 ms	0.5 ms	0.05
B	2	4 ms	30 ms	3 ms	0.1
C	3	7 ms	25 ms	4 ms	0.16

The response time upper bounds for each task are

$$R_1^{ub} = C_1 = 0.5ms$$
$$R_2^{ub} = \frac{C_2 + C_1(1 - U_1)}{1 - U_1} = 3.66ms$$
$$R_3^{ub} = \frac{C_3 + C_1(1 - U_1) + C_2(1 - U_2)}{1 - (U_1 + U_2)} = 8.44ms$$

Comparing R^{ub} to the deadline:

Task	i	Deadline	R^{ub}
A	1	1 ms	0.5 ms
B	2	4 ms	3.66 ms
C	3	7 ms	8.44 ms

- Task A is schedulable ($R_1^{ub} < D_1$)
- Task B is schedulable ($R_2^{ub} < D_2$)
- Task C is not schedulable according to *this test* ($R_3^{ub} > D_3$)

Estimating WCET To estimate **W**orst **C**ase **E**xecution **T**ime, there are two basic approaches

Static Analysis (Analysis of the source code)

- Relies on good processor model
- Good for simple code & MCU
- Difficult for complex code & MCU

Dynamic analysis (Measurement at runtime)

- Common in industry
- Must be able to exercise worst-case path
- Simple: Toggle GPIO

Concurrency / Gleichzeitigkeit

Tasks "logically happen" at the same time, either physically (multi-core) or through context switches (single-core). This should improve **Responsiveness**.

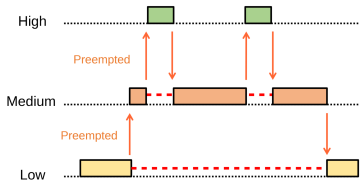


Figure 0.1: Tasks of different priority in a preemptive RTOS

- **Cooperative** multi-tasking: Tasks determine whether they give control back or not
- **Preemptive** multi-tasking: A scheduler takes control of what task gets how much time and also pulls tasks from executing

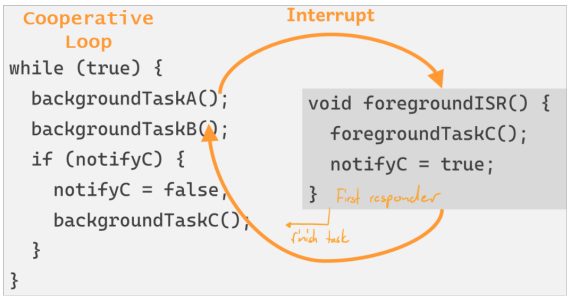
Important Properties

Safety: Nothing bad ever happens
Liveness: Something useful eventually happens
Responsiveness: Eventually is a reasonable amount of time

Cooperative Round-Robin

+ Simple
• No priorities
• Worst case response = sum of all task times
• Scheduling can be deterministic, but task periods must be harmonic
- Must manually manage state of long-running tasks
- Any change may alter response times

Preemptive: Fore-/Background



+ Prioritise tasks
+ Separation of tasks and scheduling eases change
• Worst case response = interrupt time + longest task time
• Time-triggered scheduling deterministic, task harmonic
- Complex task handling / 3rd-party microkernel
- Race conditions for interrupts
- Manual managing of long-running tasks

Preemptive: RTOS Impelementation Each task is written as if it is a *single main loop*.

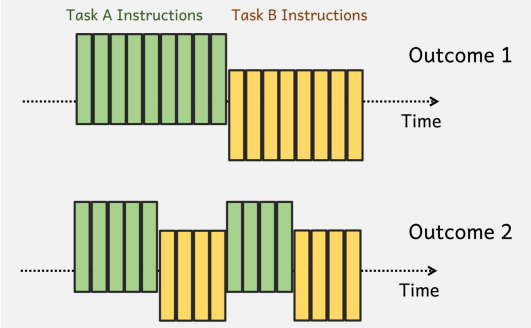
```
// Main Setup
#include <FreeRTOS.h>
#include <task.h>
void main() {
    xTaskCreate(BlinkTask, "BlinkA", STACK_SIZE, NULL,
↪   BLINK_PRIO, NULL);
    xTaskCreate(BlinkTask, "BlinkB", STACK_SIZE, NULL,
↪   BLINK_PRIO, NULL);
    vTaskStartScheduler();
}

// The Task
void BlinkTask(void* pvParameters) {
    while(true) {
        ledInvert();
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}
```

+ Prioritise tasks and responses
+ Separation of tasks and scheduling eases change
+ Long-running tasks are scheduler managed
+ Scheduling is flexible
+ Usefull features (timing-services, protocol stacks, multi-processors,...)
• Worst-case response = interrupt time + scheduler context switch
- Depending on 3rd-party microkernel
- Must manage raceconditions on recourses

- OS overhead costs recources

Concurrency Issues

Race Condition: Outcome depends on timing → Occur when task modify **shared data**



Containment: Keep data within a task
Immutability: Use unchanging data
Atomic Data: Only share data which can be changed atomically
Critical Section: Section of code must execute *atomically* (in one run)
Synchronisation: Concurrency Control

Mutex (Mutual Exclusion) Only one task can take / lock the mutex at a time. Other task trying to acquire the mutex are blocked until the mutex is released. A mutex can only be **released** *by the task that* **acquired** *it*.
If two or more tasks **share a recource**, use a mutex for protection.

```
#include <semphr.h>
SemaphoreHandle_t mutex = xSemaphoreCreateMutex();
...
// in a task
for (int32_t i = 0; i < 1000000; i++) {
    xSemaphoreTake(mutex, portMAX_DELAY);
    counter = counter + 1;
    xSemaphoreGive(mutex);
}
```

Encapsulate Synchronization

- Avoid scattering mutexes around the code
- Prevent client tasks of accessing mutexes directly
- Ensure only a single mutex is hold at a time

```c
// counter_manipultor.c
static SemaphoreHandle_t mutex;
static int32_t counter = 0;

void counterAdd(int32_t value) {
    xSemaphoreTake(mutex, portMAX_DELAY);
    counter = counter + value;
    xSemaphoreGive(mutex);
}
int32_t counterGetValue() {
    xSemaphoreTake(mutex, portMAX_DELAY);
    int32_t local = counter;
    xSemaphoreGive(mutex);
    return local;
}

// counter_task.c
void CounterTask(void* pvParameters) {
    for (int32_t i = 0; i < 100; i++) {
        counterAdd(1);
    }
    int32_t final = counterGetValue();
    printf“(%”d, final);
    vTaskSuspend(NULL);
}
```

Semaphore A semaphore can be **given** *by any task*. To receive and wait for a signal use `xSemaphoreTake(...)`.
If two or more tasks need to **coordinate actions**, use a semaphore to send signals.

```c
SemaphoreHandle_t signal = xSemaphoreCreateBinary();
void TaskA(void* pvParameters) {
    for (;;) {
        printf("Ready!");
        xSemaphoreGive(signal);
        vTaskSuspend(NULL);
    }
}
void TaskB(void* pvParameters) {
    for (;;) {
        xSemaphoreTake(signal, portMAX_DELAY);
        printf("Go!");
        vTaskSuspend(NULL);
    }
}
```

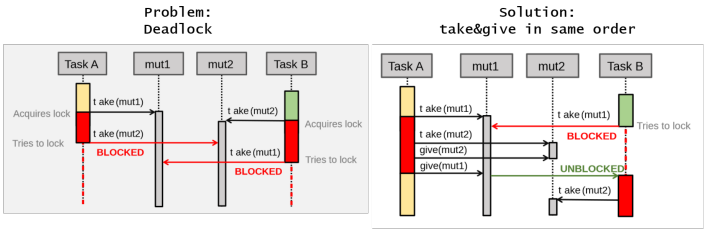
Task Notification Task notifications are FreeRTOS specific and offer a *light weight* alternative to a semaphore.

Queues Used to send data from one task to the other. Data is written to a queue **as copy**.

```c
// create queue
#include <queue.h>
QueueHandle_t msgQueue = xQueueCreate(QUEUE_SIZE,
↪  sizeof(msg_t));

// send a message
xQueueSend(msgQueue, &toSend, 0);

// receive a message
xQueueReceive(msgQueue, &received, portMAX_DELAY);
```

`xQueueReceive(..., portMAX_DELAY)` is blocking until something is put into the queue. The time can be adjusted by the last argument, usually `portMAX_DELAY`, veeeeery long.

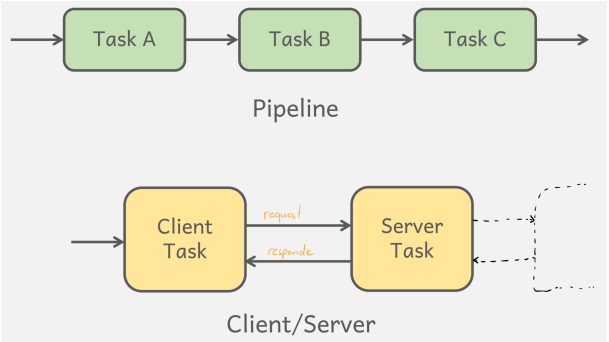
Deadlock .



We can also design tasks to only block in one place and thus deadlocks are less likely

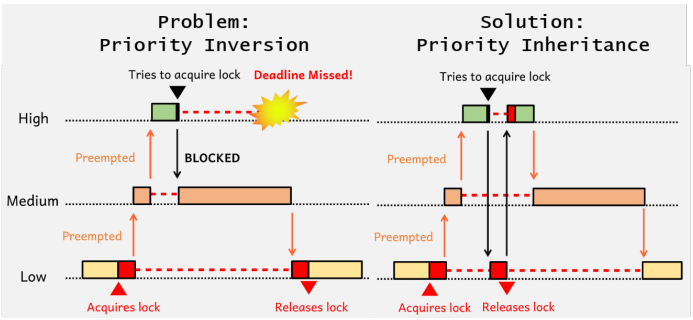
```c
void Task(void* pvParameters) {
    for (;;) {
        xQueueReceive(...); // single block
        switch (received.msgType) {
            case MSG_A: // Handle A events
            case MSG_B: // Handle B events
            case TIMER_1: // Handle timer
            case default: // Assert?
        }
    }
}
```

Or use a structure which prevents deadlocks generally



Use a **Pipeline** to minimise circular dependencies and a **Client-Server** structure to restrict the directionality of connections.

Priority Inversion .



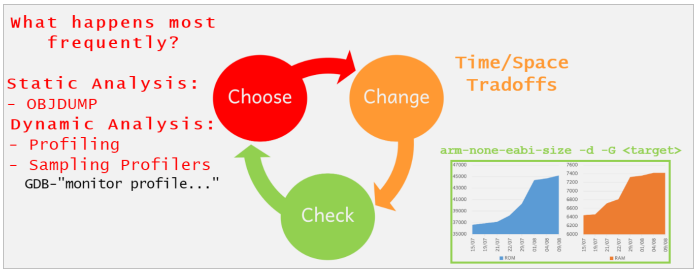
Resources

Ausgangslage TIVA

32KB RAM, 256KB ROM, Low-Power
FreeRTOS needs: 5-10KB ROM, RAM: 236bytes (scheduler), 76bytes + storage size (queue), 60 bytes + stack size (task)

Premature optimization is the root of all evil.

- Don't optimize before you know your constraints.
- Don't waste time optimizing things that won't help.
- Don't add complexity you don't need



Choose

Get the memory map (static)

```
// add this to CMake
set(CMAKE_EXE_LINKER_FLAGS
    "-T\"${SCRIPTS_DIR}/link.ld\" \
    -Xlinker -Map=${CMAKE_CURRENT_BINARY_DIR}/%
    ")

// output
...
.text 0x00000470 0x280 accl_manager.c.obj
      0x00000470       acclInit
      0x00000488       acclProcess
...
.data 0x2000005c 0x4   __atexit_dummy
      0x20000060 0x4   __atexit_recursive_mutex
...
.bss 0x2000081c 0x10 accl_manager.c.obj
...
```

The distance between two functions equals their size (mostly).

- **.text** - my code, vector table plus constants
- **.data** - initialized variables and counts for RAM and FLASH *(linker allocates data in FLASH which is then copied from ROM to RAM on startup*
- **.bss** - uninitialized data in RAM *(initialized with zero on startup)*

Dynamic Allocation Analysis

Dynamically allocated memory is difficult to analyse statically and dynamic analysis is really hard on embedded systems. Usually measure the dynamic memory allocation with FreeRTOS macros `traceMALLOC`and `traceFREE`. They keep track on the allocated memory.
Most important: Detect Memory Leaks! (MALLOC but no FREE)

Use `objdump -d` to disassemble executable

```
00000798 <readADC>:
     798:       b580            push    {r7, lr}
     79a:       b082            sub     sp, #8
     79c:       af00            add     r7, sp, #0
     79e:       2300            movs    r3, #0
     7a0:       607b            str     r3, [r7, #4]
     7a2:       2300            movs    r3, #0
     7a4:       807b            strh    r3, [r7, #2]
     7a6:       2300            movs    r3, #0
     7a8:       807b            strh    r3, [r7, #2]
     7aa:       e00c            b.n     7c6
     7ac:       4b0c            ldr     r3, [pc, #48]
     7ae:       681b            ldr     r3, [r3, #0]
     7b0:       4618            mov     r0, r3
     7b2:       f000 fa67       bl      c84 <readCircBuf>
     7b6:       4603            mov     r3, r0
```

Profiling

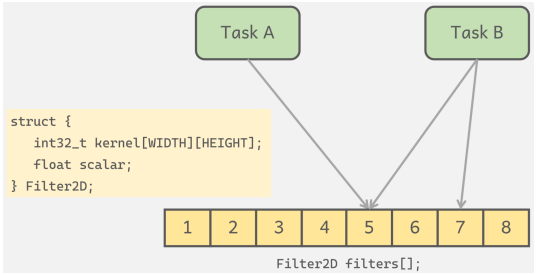
- Use GPIO and oscilloscope to profile how long certain parts of a function run
- Use a DIY sampling profiler

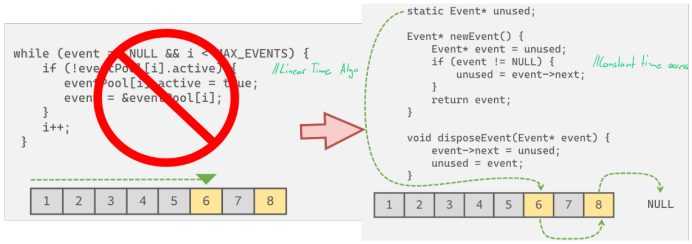
```
(gdb) continue Continuing.
^C Program received signal SIGINT, Interrupt.
↪  prvIdleTask (pvParameters=0x0
↪  <vPortValidateInterruptPriority>)
    at FreeRTOS/Source/tasks.c:3487
3487                vApplicationIdleHook();
```

Change

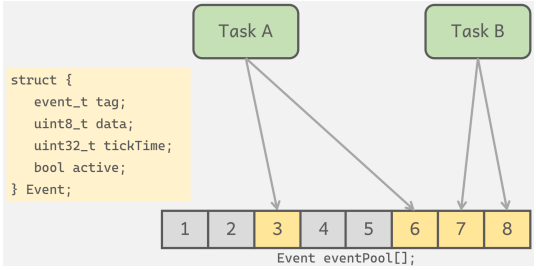
- Reduce RAM requirements by removing *concurrency*
- Trade *time* for *space*
- Increase *coupling* by sharing data
- Add structural or computational *complexity* to allow reusing limited memory
- Turn off unused peripherals (energy)
- Use lower clock rates or put MCU to sleep (energy)

Flyweight pattern Task refers to same immutable data over and over again





Memory Pool pattern Prebuilt datapool for mutable data → Control over size and avoids memory fragmentation



Cache Data .

```
...
int degrees = 2*PI*freq*time;
output = amplitude*sin((float)degrees);
...
```

```
static float sinLookup[360] = {
    0.0, 0.017, 0.034, ...
};

...
int degrees = (2*PI*freq*time) % 360;
output = amplitude*sinLookup[degrees];
...
```

Compiler and Linker help

```
// Optimize for size
gcc -Os <your file>.c
// Use link-time optimizations
gcc -flto
// Use linker to remove unused code and data
gcc -ffunction-sections -fdata-sections
Link with --gc-sections
// Optimize for speed
gcc -O2 <your file>.c
gcc -O3 <your file>.c
```

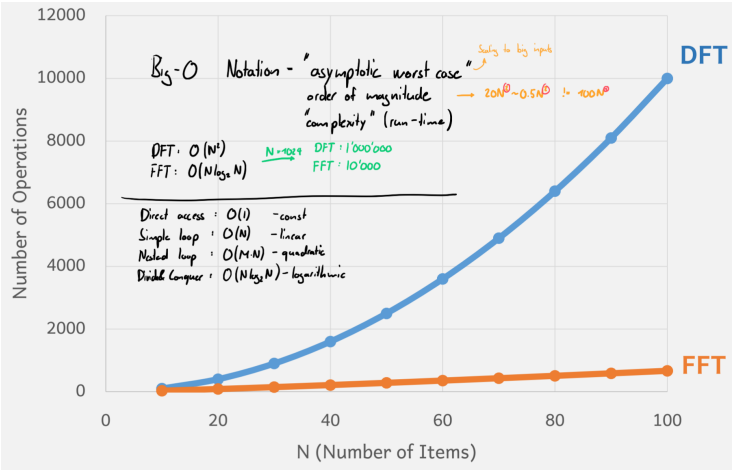
More

- Minimize data passing by **passing by reference**
- **Datacompression** with Differencing (store difference of two values in sequence), or Run length encoding
- Task notifications instead of queues
- Limit scope of local variables
- **Reduce Overhead** through removing layers
- Reduce sampling rate

Engineering in the face of uncertainty:

1. Use large margins (e.g. RAM 200%-300%)
2. Establish resource budgets for major elements
3. Estimate resource usage for each element early
4. Track resource use against budget (e.g. dashboard)
5. Finish with 20%-50% margin for future growth

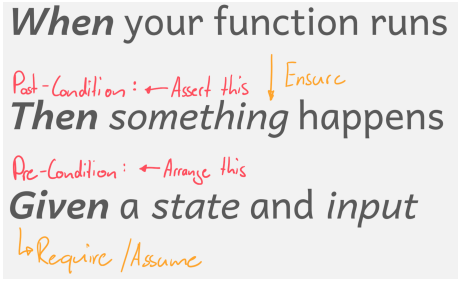
Use better algorithms .



Write better algorithms .

Performance

Preemptive Debugging




```
float squareRoot(float x) {
    // Require:
    // Failure equals happened before function gets
    ↪  called
    assert(x >= 0);
    y = ... ;
    // Ensure:
```

```
    // Failure equals error in this function
    assert((y*y) == x);
    return y;
}
```

With FreeRTOS `configAssert( x )` can be called. The behaviour is user dependent (standard `while(true);` → Fail-Safe.

```
/* Custom implementation */
void vAssertCalled( const char * pcFile, unsigned long
↪  ulLine ) {
 (void)pcFile; // unused
 (void)ulLine; // unused
 while (true); // LED, EEPROM,...
}
```

Don't **R**epeat **Y**ourself - **DRY**

```
ASSUME("event in set of handled events")

switch (event) {
  case ev1: handleEv1(); break;
  case ev2: handleEv2(); break;
  case ev3: handleEv3(); break;
}
```

```
switch (event) {
  case ev1: handleEv1(); break;
  case ev2: handleEv2(); break;
  case ev3: handleEv3(); break;
  default:
    // Should never get here
    ASSERT(false);
}
```

Don't do this

```
ASSERT(kbPress != 'j');
```

Avoid asserting **expected errors** → Handle directly in code

```
ASSERT(i++ < 5);
```

Avoid **Sideeffects**

Static Analysis

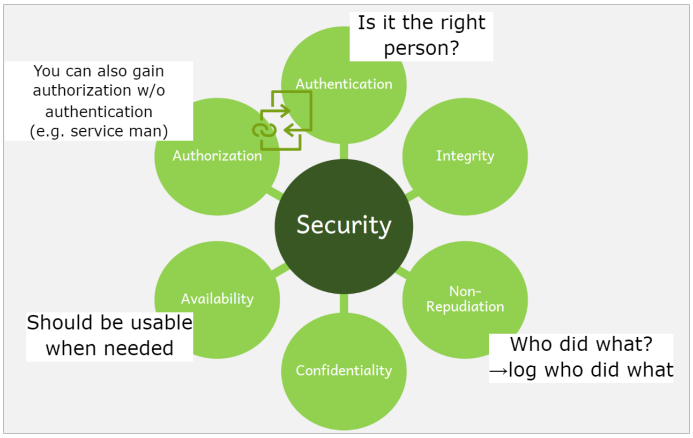
Happens **at compile-time**:

```
_Static_assert(BLINK_STACK_SIZE >
↪  configMINIMAL_STACK_SIZE, "Stack size too small");
```

Use a framework like Frama-C:

```
/*@
    requires y > 0
    ensures \result > y
*/
int makeLarger(int y) {
    int x = y*2;
    return x;
}
```

Security



C is not a "safe" language

Array overflow is not prevented, you could access the `state` variable in the following example:

Code

```
uint32_t inputBuffer[128];
uint8_t state;
```

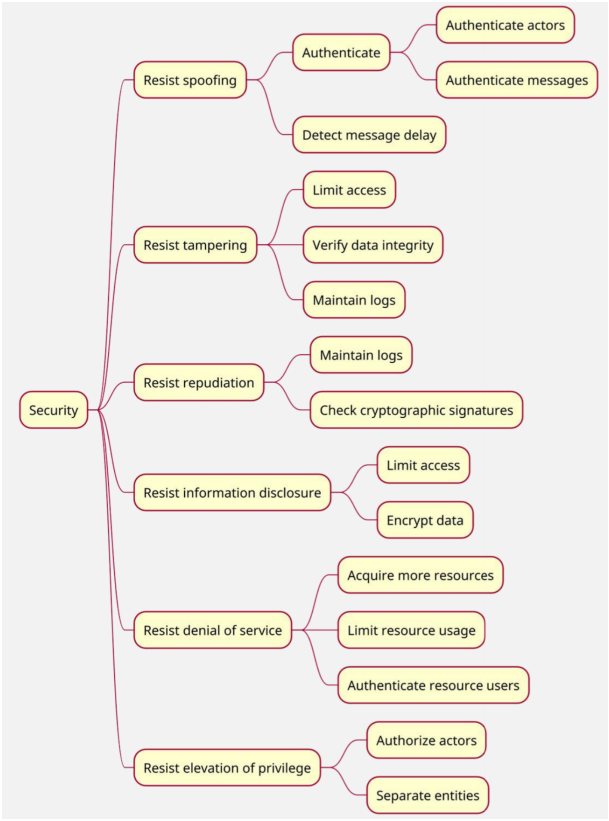
Memory map

```
.bss    0x00009040      ...
        0x00009040      inputBuffer
        0x00009240      state
```

Security failures are often design failures

Use the checklist **STRIDE** and design a architecture to avoid running into problems

- **S**poofing
- **T**ampering
- **R**epudiation
- **I**nformation Disclosure
- **D**enial of Service
- **E**levation of Privilige



Evaluate Threads

This is time consuming → **in Projektmanagement beachten**
https://emb3d.mitre.org/

Element	Interaction	Threat	Mitigation
Fitness Monitor	Transmit data to app	Tamper with data in transmit	Cryptographically sign data
	Transmit data to app and website	Information disclosure	End-to-end encryption of data
...	...	...	...
Mobile App	Poll FM for data	Spoofing of FM packets	Explicitly paired Bluetooth connection
...	...	...	...

Follow "secure" coding standards

https://securecoding.cert.org/

EXP12-C. Do not ignore values returned by functions
EXP19-C. Use braces for the body of an if, for, or while statement

CON35-C. Avoid deadlock by locking in a predefined order
CON43-C. Do not allow data races in multithreaded code
CON01-C. Acquire and release synchronization primitives in the same module, at the same level of abstraction

Use static analysis to find problems

```
// Turn on all warnings
gcc -Wall -Wextra <your file>.c
// Use static analyzer
gcc -fanalyzer <your file>.c
```

Consider exceptional CHILDREN

- **C**omputation
- **H**ardware
  - Transient Faults
  - Memory Corruption
- **I**/O
  - Running out of file space?
- **L**ibrary
  - Handle error returns
- **D**ata input
  - Buffer input overflows
  - More / Less data than expected
- **R**aces and deadlocks
- **E**xternal user
  - Wrong, Late, Other input
- **N**ull pointer and memory

Testing

Testing is for *Finding Bugs*, *Reduce risk to user* **and** *business*, *reduce development costs*, *keep code clean*, *improve performance* and to *verify that* **requirements are met**. There are different test which can be performed:

- *Unit Testing*: Verify behaviour of individual units (modules)
- *Integration Testing*: Ensure that units work together as intended
- *System Testing*: Test **end-to-end** functionality of application
- *Acceptance Testing*: Verify that the requirements are met (whole system)
- *Performance Testing*: Evalutate performance metrics (e.g. execution time)
- *Smoke Testing*: Quick test to ensure major features are working

To make testing efficient, we implement automatic testing routines. They act as a **live** documentation. Allows for **refactoring with confidence**.

Unit Test

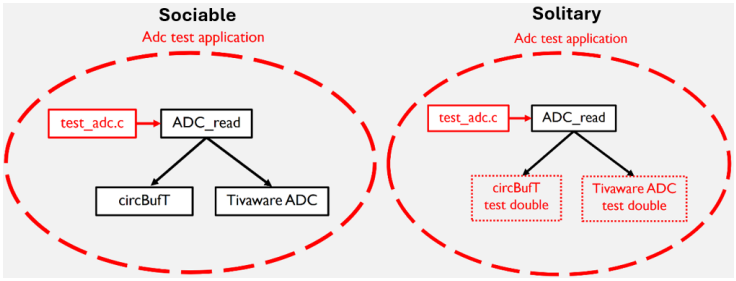
A good test case checks **one behaviour** under **one condition**, this makes it easier to localise errors.

```
void test_single_element_in_single_element_out(void)
↪ {
    // Arrange: given buffer has a single element
    writeCircBuf(&buff, 11);
    // Act: when buffer is read
    uint32_t value = readCircBuf(&buff);
    // Assert: then the same value is returned
    TEST_ASSERT_EQUAL(11, value);
}
```

i Testing Frameworks

- Unit Test Framework Unity
- Test Double Framework fff

Unit Test with Collaborators



Test Doubles

Implement test doubles through the fake function framework (fff). There are different variations of test doubles:
Stub: Specify a return value - *Arrange*

```
// Set single return value
i2c_hal_register_fake.return_val = true;
// Set return sequence
uint32_t myReturnVals[3] = { 3, 7, 9 };
SET_RETURN_SEQ(readCircBuf, myReturnVals, 3);
```

Spy: Capture Parameters - *Arrange* / *Assert*

```
// Arrange, e.g. get passed function
adc_hal_register(ADC_ID_1, dummy_callback);
void (*isr) (void) = ADCIntRegister_fake.arg2_val;
// Assert Parameter
TEST_ASSERT_EQUAL(3,
↪  ADCSequenceDataGet_fake.arg1_val);
```

Mock: Can act as a *Stub*, *Spy*, and much more (from fff). Implemented as follows:

```
// in some_mock.h
VALUE_FUNC(uint32_t *, initCircBuf, circBuf_t *,
↪  uint32_t);
VOID_FUNC(writeCircBuf, circBuf_t *, uint32_t);
```

Fake: Provide a custom fake function - *Arrange*

```
// Define Fake Function
int32_t ADCSequenceDataGet_fake_adc_value(uint32_t
↪  arg0, uint32_t arg1, uint32_t *arg2) {
    *arg2 = FAKE_ADC_VALUE;
    return 0;
}
// Apply Fake Function - Arrange
ADCSequenceDataGet_fake.custom_fake =
↪  ADCSequenceDataGet_fake_adc_value;
```

Continuous Integration

CI is used to automate the integration of code changes. These are automated scripts running all the tests. This is usually implemented in the code hoster (e.g. *GitLab*) and is executed after every push. It also runs before every merge and **blocks a merge** if one of the tests fails.

Higher Level Testing

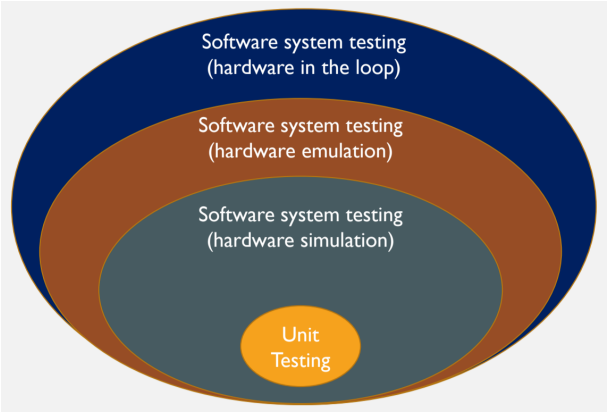
Unit tests only verify small elements of a system in isolation.

Automated Acceptance Testing

- Verifys system requirements
- Live documentation of high-level requirements
- Understandify behaviour
- Acceptance test pass → requirement met
- Written by PM or QA (≈ customer)
- Written in natural scripting language
- Non-Technical stakeholder in the loop
- Called: **B**ehaviour-**D**riven **D**evelopment **BDD**

Automated System Tests

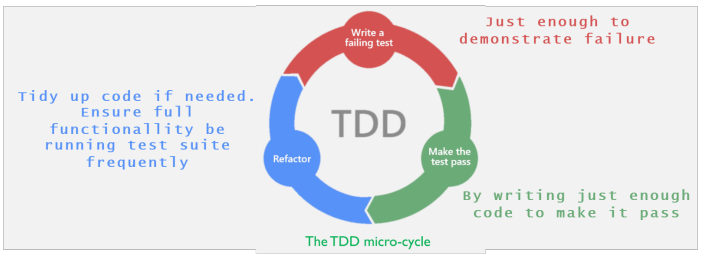
Hardware Simulation: Developed on PC, no need to know specific hardware implementation yet, limitations with hardware peripherals.
Hardware Emulation: Emulate processor on PC, needs resources for emulator. Tools: QUEMU
Hardware in the Loop: Runs on target, test scripts on a test enclosure to manipulate hardware, expensive setup. Tools: NI DAQ, Labview



Manual Testing

Sometimes automated test setups are more expensive.
Manual testing can involve **user interaction**, **Debugger**, **direct Signal Probing** (Oscilloscope, Multimeter, Logic Analyzer).

Test Driven Design TDD



Applying TDD through writing *unit tests* during development, benefits:

- **Reduce Debug Time**: small feedback loop

- **Courage to make changes**: tested code is changeable code
- **Tests are Reliable**: high level of coverage
- **Good Architecture**: Writing test implies decoupling

For each new unit:

1. Come up with a **set of requirements**
2. Generate a **rough test list**
3. Implement unit by going through the list with **TDD**
4. Tick off, remove, or add items to/from the list in the process

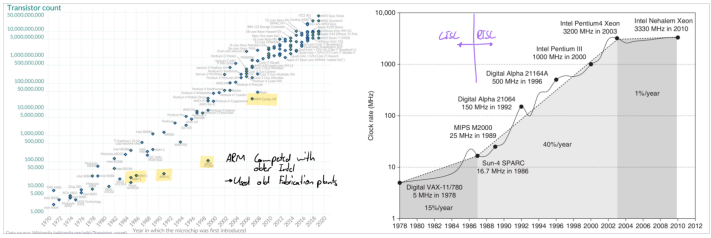
Use **ZOM** to come up with tests:

- **Z**ero case(s): simplest scenario → build interface
- **O**ne case: simplest scenario to transition from **Z**ero to **O**ne
- **M**any cases: generalise design, each test case adds a scenario

Computer History

Moor's Speculation

The *prediction* of Gordon Moore states "doubling the number of components on a IC each year". This is also largely true.



The exponential growth turned out to be true, for how much longer?

Early History of Digital Computers

The early history of digital computers and microprocessors highlights the progression from large, power-hungry machines to more efficient, smaller designs:

- **1950s**: The largest computer, IBM AN/FSQ-7, used 55,000 vacuum tubes and consumed 3 MW of power.
- **1964**: DEC introduced the first minicomputer, the PDP-8, followed by the PDP-11, which played a key role in developing Unix and C.
- **Intel 4004 (1971)**: The first microprocessor with a 4-bit CPU and 2300 transistors, originally designed for calculators.
- **Intel 8008 (1972)**: A more advanced 8-bit CPU with 5000 transistors.
- **Intel 8080 (1974)**: Improved performance (10x over 8008) and used in the first PC (Altair).
- **Texas Instruments TMS1000 (1974)**: First microcontroller, used in calculators and appliances.
- **MOS Technology 6502 (1975)**: Popular in early PCs (Commodore, Apple, Atari) for $25.
- **Intel 8086/8088 (1978)**: Intel's first 16-bit processor, chosen for the IBM PC despite flaws.
- **Motorola 68000 (1979)**: A 32-bit processor used in the Apple Macintosh.
- **Intel 386 (1985)**: Introduced linear addressing and virtual memory support, popular in PCs by 1990.

> **Why x86?**
>
> The **IBM PC** is the first personal computer, the engineers wanted to use the *Motorola 68000* chip, but management choose the *Intel 8088*, that's the dawn of Intels dominance and thus the dominance of the *x86-Architechture*

Fundamentals of Microprocessors and Architectures

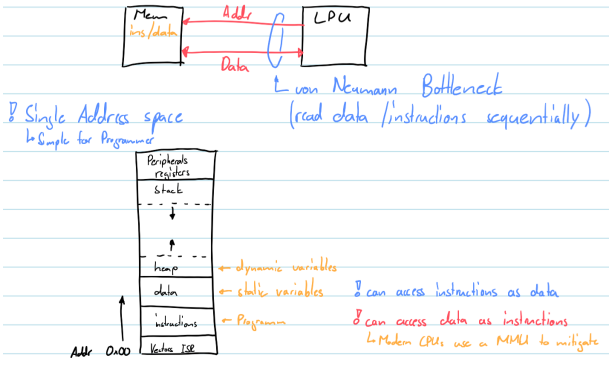
Microprocessor architectures

Flynn's Classification processor architectures into subgroups

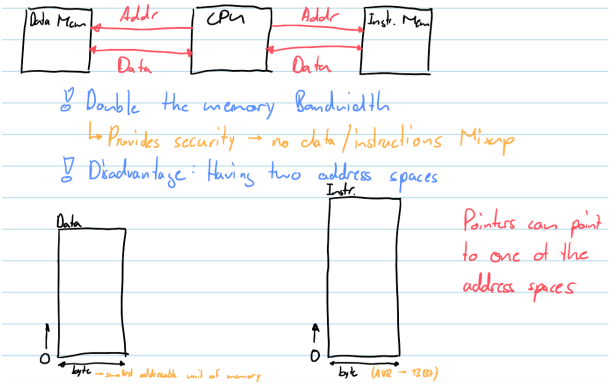
SISD - Single Instruction / Single Data

Processors with a SISD-Architecture are **Uniprocessors (von Neumann)**, they consist of a single processor. They are partitioned into *input devices, output devices, memory, ALU and control unit

Princeton Memory stores *data and instructions*, thus the **von Neumann Bottleneck** appears. There is also the risk of **accessing data as instructions** and vice verca. This can stall the CPU and is mitigated through memory management units (MMU) in modern processors.



Harvard The memory is seperated as data / instruction. This adds security, but harder for the programmer (What memory space does a pointer point to?).



Modified Harvard There are separate instruction and databuses, but share the same memory space. Application: **DSP**

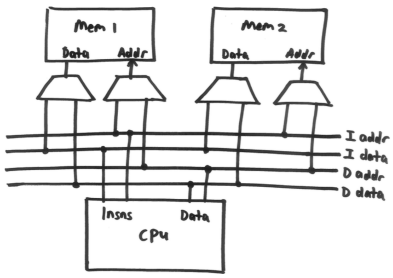
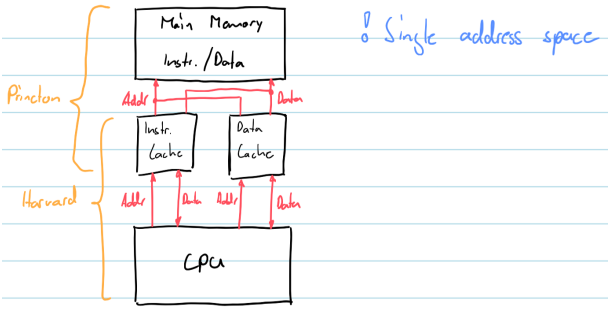


Figure 2.3: The modified Harvard architecture uses bridges to link the buses. These are implemented using multiplexers and bidirectional tri-state buffers (transceivers).

Hybrid There is a single memory space and data/insn-bus, but to gain speed, caches are used to mimic a harvard architecture.



SIMD - Single Instruction / Multiple Data

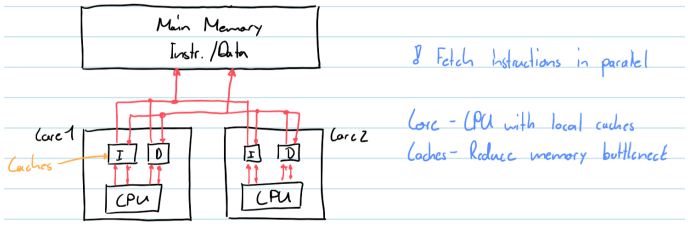
Typical SIMD-Processors are **Array-/Vector-Processors** and are often used in *GPUs* or *special CPUs*. Each processor simultaneously performs the same instruction on different data.

> **i** SIMD Instructions
>
> TO operate on **vector registers**...
> - ... *ARM* Processors have **NEON** instructions
> - ... *Intel* Processors have **SEE** instructions

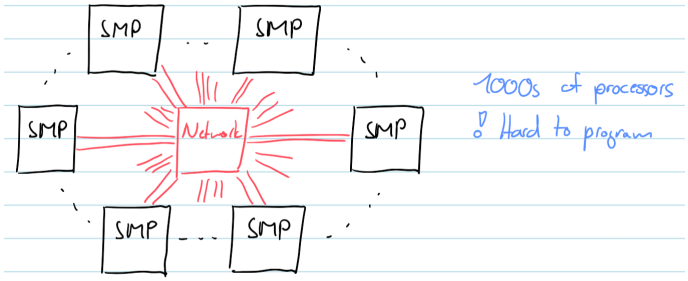
MIMD - Multiple Instruction / Multiple Data

MIMD-Processors are multi-core processors, with several cores, often accessing the same memory. This is done to improve computer performance through **parallelism**.

SMP - Symmetrical Multiprocessor Few identical processors share a common memory. Caches are used to mitigate the *von Neumann Bottleneck*.



MPP - Massively Parallel Processor Many processors using distributed memory and communication through a network *(many topologies, e.g., star, ring, ...)*.



Accessing Program Memory on AVR

AVR is a manufacturer who uses the *Harvard*-Architecture. To access the program memory (flash) the `PROGMEM` macro from `avr/pgmspace` is used. *(support of that is introduced in GCC 4.8)*

```
#include <avr/pgmspace.h>

// for flash access
PROGMEM const char flash_str[] = "Hello world";
char flash_read_byte (const char *p) {
    return pgm_read_byte(p);
}

/* translates to following assembly */
flash_read_byte:
    movw r30 , r24
    lpm r24 , Z    // load program memory
    ret

// for SRAM access
const char sram_str[] = "Hello world";
char sram_read_byte (const char *p) {
    return *p;
}

/* translates to following assembly */
sram_read_byte:
    movw r30 , r24
    ld r24 , Z
    ret
```

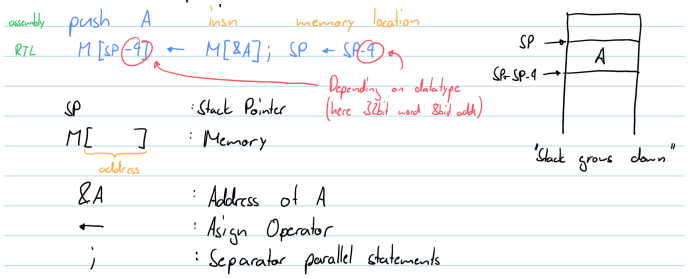
Instruction Set Architectures (ISA)

The *instruction set architecture* (ISA) is the assembly language of a specific processor. The ISA includes things such as:
- Instruction Set (NOP, BRA, LOAD, ADD, ...)
- Data Types (u/s-int, float, addresses)

- Registers
  - PC: programm counter
  - SP: stack pointer
  - R0, R1, R2, . . . : general purpose registers
  - W, Z, V, C: status registers
- Addreassing nodes, for accessing memory

Register Transfer Language (RTL)

Describes how a cpu instruction behaves



Stack ISA

Stack ISAs operands are pushed onto a stack before operators are applied. Source operands are popped off the stack, and destination operands are pushed back onto it, resulting in very short instructions. However, the stack can become a bottleneck, as it is not randomly addressable.

```
Assembler   RTL
PUSH  A     ; SP <- SP - 4; M[SP] <- M[&A];
PUSH  B     ; SP <- SP - 4; M[SP] <- M[&B];
ADD         ; M[SP + 4] <- M[SP + 4] + M[SP]; SP <- SP + 4
POP   C     ; M[&C] <- M[SP]; SP <- SP + 4
```

Accumulator ISA

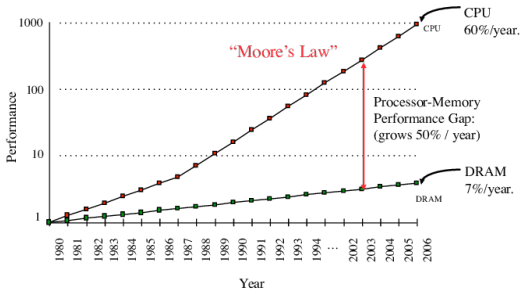
Accumulator ISAs use a single accumulator register for storing the results of all ALU operations, leading to short instruction lengths where only one operand is specified. Memory traffic is high since the accumulator is the primary storage, but this design was popular in early microprocessors when memory and CPU speeds were comparable.

```
Assembler   RTL
LOAD  A     ; ACC <- M[&A]
ADD   B     ; ACC <- ACC + M[&B]
STORE C     ; M[&C] <- ACC
```

General Purpose Register

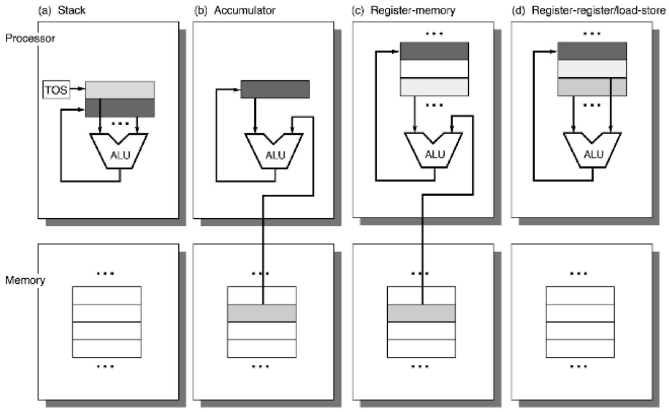
Issue with frequent memory access

Because the speed of **DRAM** didn't keep up with the speed of **CPUs**, memory is a bottleneck of performance.

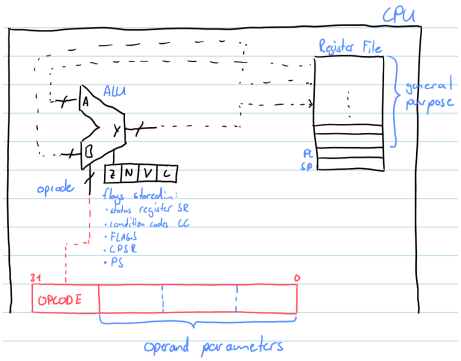


Most high-performance computers use general-purpose register ISAs because registers are faster than memory. These ISAs have many registers, explicit operands, and longer instructions. Load/store ISAs, where ALU instructions only operate on registers, are popular despite requiring more instructions because they are simpler and more suited to pipelining. ISAs can be classified into memory-memory, register-memory, and register-register, based on how many memory operands an ALU instruction can have.

CISC		RISC
memory-memory	**register-memory**	**register-register**
`ADD A,B,C`	`LOAD A,R0`	`LOAD A,R0`
	`ADD B,R0,R1`	`LOAD B,R1`
	`STORE R1,C`	`ADD R0,R1,R2`
		`STORE R2,C`
IC = 1	IC = 3	IC = 4



Instruction encoding

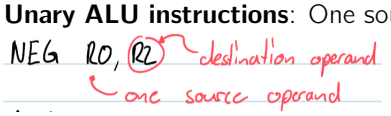


ALU flags

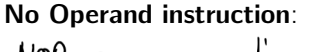
Z : result is *zero*
N : result is *negative*
V : result has *signed overflow*
C : result has *unsigned overflow* **carry**

Opcodes encode the operations to perform. There are different kind of opcodes, depending on how many operands are involved:
Binary ALU instructions: Two source operands

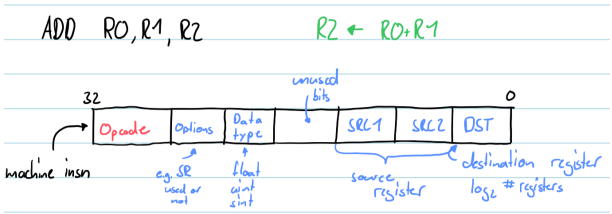
Unary ALU instructions: One source operand



No Operand instruction: No operand, doens't use ALU



Binary ALU Instruction Encoding

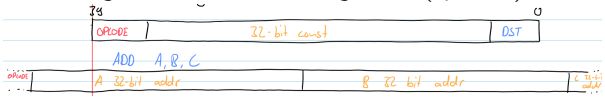


Operands Operands can be a **Register**, a **Constant**, or at a **Memory Location**.

Register, operand in register
```
MOV  R1, R0        ; R0 <- R1
```
Immediate, constant in insn
```
MOV  1, R0         ; R0 <- 1
```
Direct, memory addr in insn
```
MOV  A, R0         ; R0 <- M[&A]
```
Register inderect, memory addr in register
```
MOV  *R7, R0       ; R0 <-M [R7]
```
Displacement, memory addr in reg + constant
```
MOV  *(R7+4), R0   ; R0 <- M[R7+4]
```
Indexed, memory addr is sum of two regs
```
MOV  *(R7+R4), R0  ; R0 <- M[R7+R4]
```

⚠ Length of Instructions

If you want to load large constants (32-bit words), you won't have enough space in your instruction encoding. There are two solutions:
CISC: Variable length instructions
This introduces the issue of different instruction having different lengths, and thus needing more (specific) hardware.

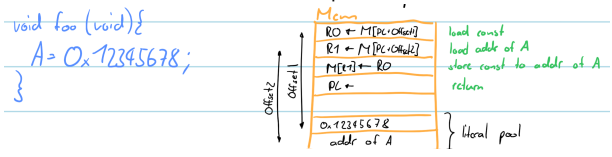


RISC: Fixed length instructions
This is great for Pipelining, and thus makes a processor faster. To fix the size issue we can use:

1. Synthesize large constants using multiple insn

```
R0<-0x12345678
        ⬇
R0<-0x1234 ; R1<-0x5678 ; R2<-R0<<16 ; R0<-R1|R2
```

2. Load a constant from literal pool



CISC to RISC

! CISC issues

- Initially ISAs were designed for easy assembly language programming
- This introduced many...
  - ... datatypes: byte/short/int/float/double/BCD/bit fields
  - ... instructions: crc (cyclic redundancy check)/polynomial extension/linked list insertion/...
  - ... addressing modes: R0<-M[3]/R0<-M[R1]/R0<-M[R1+R2]/R0<-M[M[R1+R2]]

Computer Performance Measures

To measure the compute performance, execute a benchmark and take the time:

$$\begin{aligned}\text{Program Execution Time} &= \frac{\text{CPU Clock Cycles}}{\text{Program}} \times \frac{1}{\text{Clock Rate}} \\ &= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{CPU Clock Cycles}}{\text{Instruction}} \times \frac{1}{\text{Clock Rate}} \\ &= \text{Instruction Count} \times \text{CPI} \times \frac{1}{\text{Clock Rate}} \qquad (5.1)\end{aligned}$$

depends on ISA +compiler; depends on processor organisation; depends on HW technology

To minimise program execution time, reduce the instruction count, reduce the CPI (Clocks per Instruction), or increase the Clock rate.
Clock Rate: Hardware technology and processor organisation
CPI: Processor organisation and instruction set architecture
Instruction Count: Instruction set architecture and compiler technology

⚠ Misleading MIPS

Million **I**nstructions **P**er **S**econd is a missleading measure, as a processor without a FPU can have higher MIPS (simpler architecture), but takes longer to calculate. Furthermore, it doesn't take *chip cost* and *power consumption* into account.

$$\text{MIPS} = \frac{\text{Clock Rate}}{\text{CPI} * 10^6} = \frac{\text{Instruction Count}}{\text{Program Execution Time}}$$

Amdahl's Rule

Make the common case fast

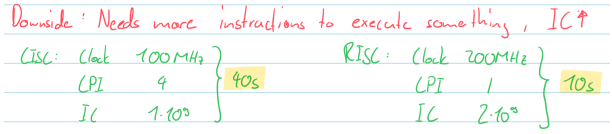
To compare performance improvements, we need the *speedup* and *utilisation*

$$\text{Exec Time New} = \text{Exec Time Old}\left[(1 - \text{Utilisation}) + \frac{\text{Utilisation}}{\text{Speedup}}\right]$$

$$\text{Speedup Overall} = \frac{\text{Exec Time Old}}{\text{Exec Time New}} = \frac{1}{(1 - \text{Utilisation}) + \frac{\text{Utilisation}}{\text{Speedup}}}$$

Enhancement	Speed up	Utilisation	Overall Speedup	
A	10	50%	1.82	
B	4	60%	1.82	Management decision
C	2	90%	1.82	

With this we have a absolute number of the overall speedup. RISC was developed with this in mind, which lead to a overall improvement, even though the instruction count went up



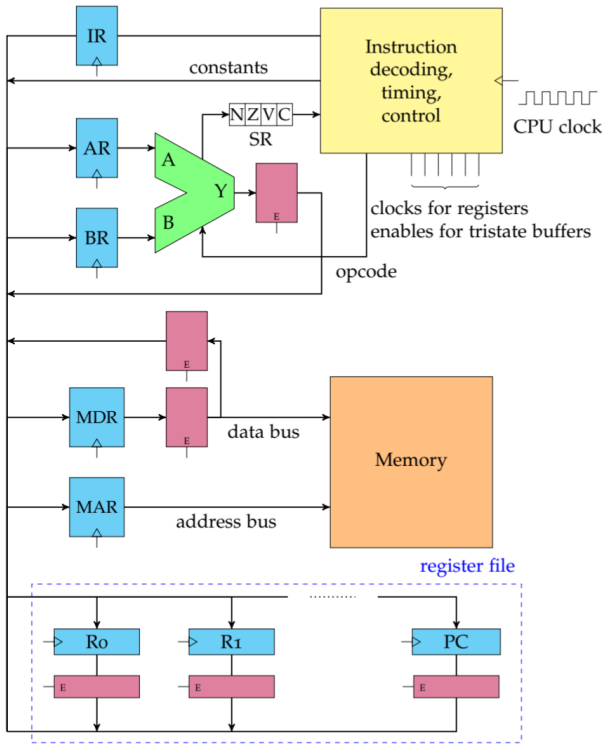


Figure 0.2: General Purpose Register CPU

Pipeline and Parallelism

Computer pipelining

For bigger throughput, separate stages should take the same amount of time.

A CPU needs the basic 5 stages (modern CPUs have way more):

1. *Fetch*: fetches insn from memory
2. *Decode*: Determine opcode & operands
3. *Read*: Reads source operands
4. *Execute*: ALU performs operation
5. *Write*: ALU result is written back to register

A 5 stage pipeline provides a CPI=1, but a latency of 5 clock cycles.

```
MUL     R7, R8, R0      ; R0 <- R7 * R8
ADD     R7, R8, R1      ; R1 <- R7 + R8
SUB     R7, R8, R2      ; R2 <- R7 - R8
OR      R7, R8, R3      ; R3 <- R7 | R8
AND     R7, R8, R4      ; R4 <- R7 & R8
```

Cycle	Fetch	Decode	Read	Execute	Write
1	MUL	-	-	-	-
2	ADD	MUL	-	-	-
3	SUB	ADD	MUL	-	-
4	OR	SUB	ADD	MUL	-
5	AND	OR	SUB	ADD	MUL

Pipeline Hazards

There are **Resource**-, **Data**-, and **Control**-Hazards.

Resource Hazards

Occur when two stages need the same resource (e.g. memory, ALU). To mitigate this issue, the pipeline is **stalled**, which introduces a *pipeline bubble* and raises the CPI.

```
LOAD    A, R3           ; R3 <- M[&A]
MUL     R7, R8, R0      ; R0 <- R7 * R8
ADD     R7, R8, R1      ; R1 <- R7 + R8
SUB     R7, R8, R2      ; R2 <- R7 - R8
```

Cycle	Fetch	Decode	Read	Execute	Write
1	LOAD	-	-	-	-
2	MUL	LOAD	-	-	-
3	-	MUL	LOAD	-	-
4	ADD	-	MUL	LOAD	-
5	SUB	ADD	-	MUL	LOAD

Pipeline Bubble

A example for this hazard is the *division operation*, as it requires the N ALU-operations for N-bit operands.

Data Hazards

RAW, Read After Write $I1$ & $I2$ have a **true dependency** on `R0`.

```
I1      MUL     R1, R2, R0      ; R0 <- R1 * R2
I2      ADD     R0, R3, R4      ; R4 <- R0 + R3
```

Problem: `R0` is written in stage 5 for $I2$, but read in stage 3 for $I2$
Solution: **Stalling** pipeline, until `R0` is written by $I1$

WAR, Write After Read $I1$ & $I2$ have a **anti-dependency** on `R0`.

```
I1      MUL     R0, R1, R2      ; R2 <- R0 * R1
I2      ADD     R3, R4, R0      ; R0 <- R3 + R4
```

Potential Problem: If $I2$ writes `R0` before $I1$ reads
-> this doesn't happen in the usual case
-> can occur with out of order execution (**superscalar**)

WAW, Write After Write $I1$ & $I2$ have a **output dependency** on `R0`.

```
I1      MUL     R1, R2, R0      ; R0 <- R1 * R2
I2      ADD     R3, R4, R0      ; R0 <- R3 + R4
```

This is very **uncommon**, only with badly optimised out of order execution.

Resource and Data hazard avoidance

Static Hazard Detection (compiler)

1. Place useful instructions between $I1$ and $I2$, that don't alter the program
2. Otherwise stall the pipeline with `NOP`

Dynamic Hazard Detection (CPU)

1. Place useful instructions between $I1$ and $I2$, that don't alter the program (out-of-order processors)
2. Otherwise stall the pipeline using a *pipeline interlock*

Control Hazards

Occur due to *changes in the program flow* (branches/jumps/interrupts/function calls), which change the *PC* often unpredictably.

```
I1  LOOP:   MUL     R1, R2, R2      ; R2 <- R2 * R1
I2          SUB     1, R0, R0       ; R0 <- R0 - 1
I3          BNE     LOOP
I4          ADD     10, R2, R2      ; R2 <- R2 + 10
I5          STORE   R2, C           ; M[&C] <- R2
```

The hazard occurs, because the CPU doesn't know what to fetch after the branching `BNE`.

Control Hazard Avoidance To avoid a control hazard, the cpu can:

1. **Stall** the pipeline until it's known what path is taken (CPI goes up!)

Cycle	Fetch	Decode	Read	Execute	Write
1	MUL	-	-	-	-
2	SUB	MUL	-	-	-
3	BNE	SUB	MUL	-	-
4	-	BNE	SUB	MUL	-
5	-	-	BNE	SUB	MUL
6	-	-	BNE	-	SUB
7	-	-	BNE	-	-
8	-	-	-	BNE	-
9	-	-	-	-	BNE
10a	MUL	-	-	-	-
10b	ADD	-	-	-	-

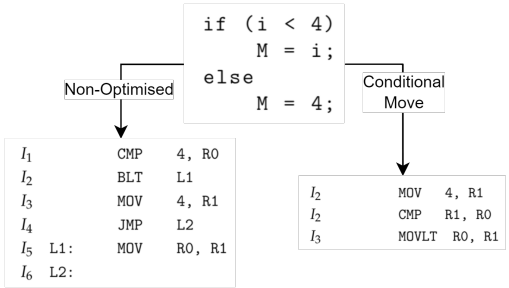
2. **Assume** the pipeline increments normally -> process after branch -> **flush** pipeline if wrong

Cycle	Fetch	Decode	Read	Execute	Write
1	MUL	-	-	-	-
2	SUB	MUL	-	-	-
3	BNE	SUB	MUL	-	-
4	ADD	BNE	SUB	MUL	-
5	STORE	ADD	BNE	SUB	MUL
6	I6	STORE	ADD	BNE	SUB
7	I7	I6	STORE	ADD	BNE
8a	I8	I7	I6	STORE	ADD
8b	MUL	flush	flush	flush	flush

3. **Predict** which way the branch goes (**flush** if wrong) -> **Speculative Execution**
   i. *prediction on heuristics*: backwards branch is more likely to be taken (loops)
   ii. *record previous branch prediction*: CPUs have branch prediction HW

Branch Prediction CPUs have a bit that indicate which way the branch previously went. They're assuming it goes the same way next time. This is a good assumption for e.g. `for(i=0;i<100;i++){ }`. Modern CPUs use multiple branch prediction bits and the PC is used as a hash key.

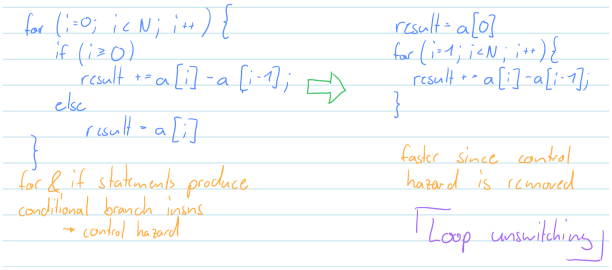
Conditional Move Instructions To avoid frequent flushing of pipelines, a conditional move instruction was introduced, which moves depending on the result of the previous insn.



Interrupts

- Tricky since PC changes unpredictably
- When interrupt occurs:
  1. pipeline needs to be flushed
  2. pipeline restarts with ISR code
- When interrupt finishes
  3. pipeline restarts at the point the program was interrupted

Program to reduce control hazards With modern CPUs and compilers (if no aliasing [nicht abtasttheorem] is detected), the difference in performance will be minimal. A technique is the **loop unswitching**. Loop optimisation, should start from the inner to the outer loops.



Instruction level parallelism

The goal is to reduce

$$\text{CPI} < 1$$

This requires multiple instructions executed at same time. For this multiple **Functional Units** (execution units) are used

VLIW Processor

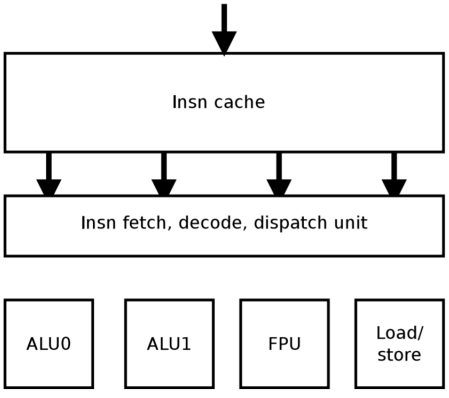
Very **L**ong **I**nstruction **W**ord processor explicitly state what each execution unit does. Thus the instructions are much longer

This relies on the compiler and it can also happen that not every functional unit is utilised. For this NOPs have to be inserted, the CPI goes up and memory is wasted.

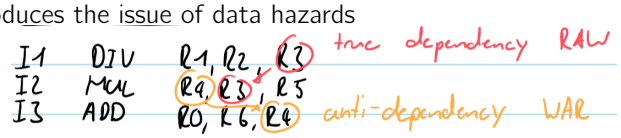
Superscalar Processor

This CPU fetches and decodes a standard stream of instructions and dispatches them to the appropriate functional unit. To make efficient use of this, a *instruction cache* is used. From the n different instructions can be fetched, decoded and dispatched per cycle.

***n*-way superscalar** $\text{CPI} = \frac{1}{n}$

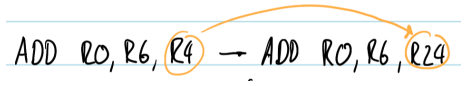


Data Dependencies When running several instructions in parallel introduces the issue of data hazards

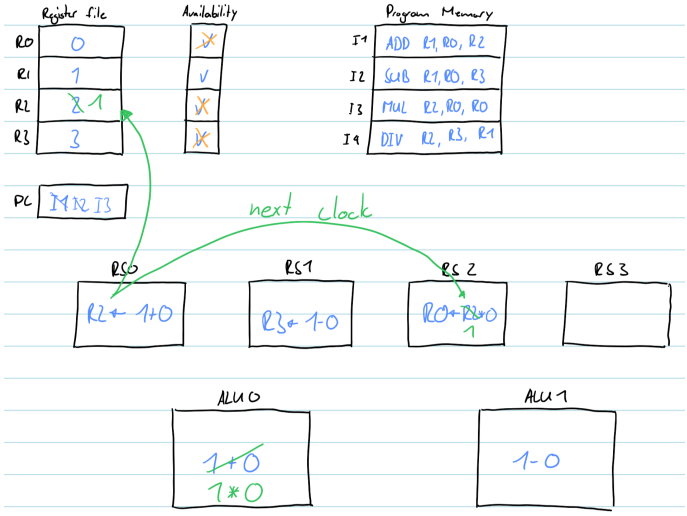


- R3: cannot execute I2 until I1 is finished
- R4: cannot execute I3 before I2 is finished

Register Renaming CPUs have additional registers to use in place, which the programmer has no access to. Any followup instructions using the R4 result become dependent on R24.



Tomasulo's Algorithm Tomasulo's algorithm schedules instructions on superscalar CPUs.
It handles RAW, WAR, WAW hazards.
It introduces and handles *out-of-order-execution*
It uses **reservation stations** (RS) to hold instructions before they are executed



1. While RS is not available: *Stall Pipeline*
2. Copy instructions and available operands into RS
3. Mark destination register as unavailable
4. If RS has all operands and a suitable FU available: issue instruction to FU and free RS
5. Copy the result to destination register and any waiting RS
6. Mark destination register as available

The **availability**-register states if the operand is available for further processing, or if a operation is performed for it right now. (avoid RAW hazard)
Instructions sit in a RS until a FU is available and all register source operands are known.
WAW & WAR hazards are avoided by copying register values into RS
Reservation stations have a high cost in comparators, needed to compare all results returned from processing units with all stored addresses.

Multithreading CPU

To maximise FU usage to get the lowest CPI we allow multiple threads to run in parallel, sharing the CPU. For this we need:

- Two program counters, for each thread one
- Two sets of registers, for each thread one
- Threads share functional units and reservation stations

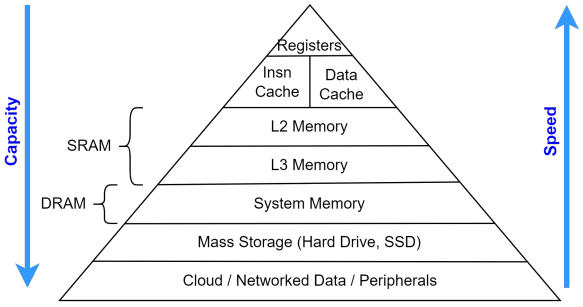
This results in better utilisation of the functional units (e.g.: one thread does integer (gcc), the other floating point (simulations) / one thread executes, one thread is waiting for memory)

Memory Systems and Optimization

Because CPUs increased in speed at a faster rate than DRAM, **Memory Caches** are introduced to mitigate a memory bottleneck.

Cache memory systems

DRAM	SRAM	Registers
+ cheap (1T+1C)	+ faster	+ flip-flops
- C leaking	+ no refresh	+ fastest
- dynamic refreshing	+ static	- only small memory
- refresh after read	- expensive (6T)	
- slow (200 cycles)		



- **L1**-cache: split in insn and data, to mitigate von neumann-bottleneck
- **L2**&**L3**-cache: unified data caches for one or several cores

Average Memory Access Time

The goal of a good memory hierarchy is to keep the total memory cost down whilst still trying to keep the average memory access time fast. This is measured by the *Average Memory Access Time*:

$$\begin{aligned}\text{Average Memory Access Time} &= \text{Hit Time} * \text{Hit Ratio} + \text{Miss Time} * (1 - \text{Hit Ratio}) \\ &= \text{Hit Time} + \text{Miss Penalty} * (1 - \text{Hit Ratio})\end{aligned}$$

- Hit Time: Memory access time for data in cache
- Hit Ratio (Typically 85 – 95%) : Proportion of memory access that is cache

$$\text{Hit Ratio (aka Hit Rate)} = \frac{\text{Cache Hits}}{\text{Total Memory Requests}}$$

Cache Parameters

From the cache parameters it's obvious, that frequent reading from the system memory should be avoided if possible

Memory	Size	Read cycles (clocks)	Line size (B)	Associativity
L1 insn cache	32 KB/core	4 cycles	64	8
L1 data cache	32 KB/core	4 cycles	64	8
L2 cache	256 KB/core	11 cycles	64	4
L3 cache	8192 KB	35 cycles	64	16
DRAM	16 GB	135 cycles	-	-

Cache Locality

- **Temporal** locality: Recently read locations are often re-read *(loops)*
- **Spatial** locality: Likely to read memory *close* to previously read location *(iterate over array)*

Iterating over multi dimensional arrays

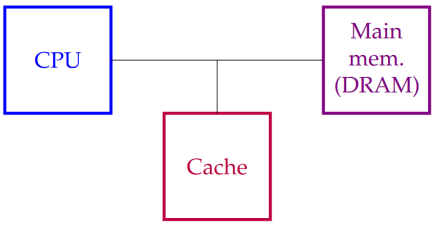
When looping over multidimensional arrays, the innermost loop should always loop over data next to each other.

```c
/* Good data cache usage */
double matsum1(const double *mat, unsigned int
↪  rows, unsigned int cols) {
    unsigned int i, j;
    double total = 0.0;
    for (j = 0; j < cols; j++)
        for (i = 0; i < rows; i++)
            total += mat[j * rows + i];
    return total;
}

/* Poor data cache usage */
double matsum2(const double *mat, unsigned int
↪  rows, unsigned int cols) {
    unsigned int i, j;
    double total = 0.0;
    for (i = 0; i < rows; i++)
        for (j = 0; j < cols; j++)
            total += mat[j * rows + i];
    return total;
}
```

Cache architectures

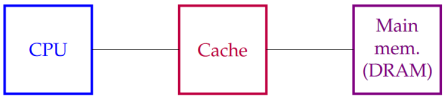
Look Aside

The **Look Aside** cache connects the CPU directly to all memory, and thus requests it from all memory. If the data is in cache, the *acces cycle is terminated*.



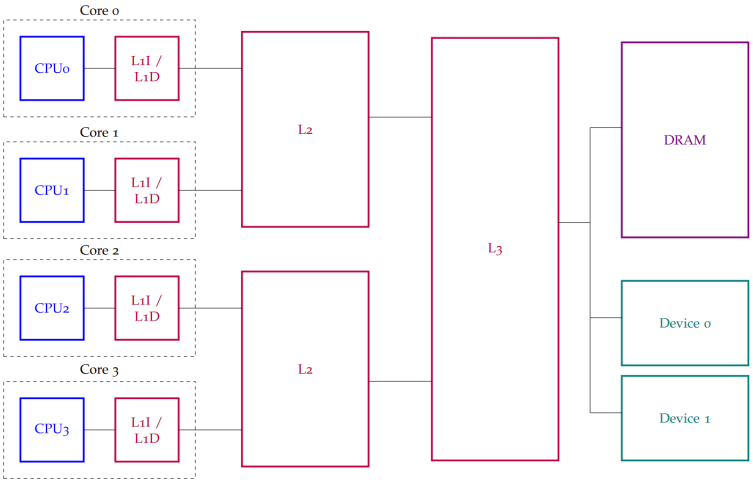
Look Through

In a **Look Through** configuration, the CPU is only connected to the L1 cache and the L1 cache acts as the master for the next higher memory access. There is *less traffic* on the main system bus, but the hardware is more complicated and it's slower.



Multicore Cache Architecture

The stages of **L1/L2/L3**-caches are all *look through*.
Cache Coherence: All caches must be kept consistent (The same data spread over several caches/memory locations must stay the same).

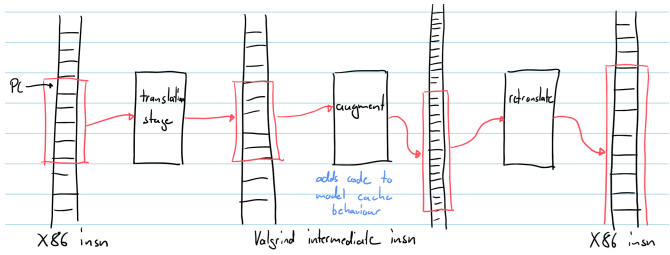


Monitoring Tools

To monitor cache acces behaviour tools like
- **Valgrind** - Finding memory leaks or read from uninitialised memory
- **Cachegrind** - Analysis of cache & branch prediction

Cachegrind: Uses just in time translation and runs the modified program (slower). It records information to a file which can be analysed afterwards.

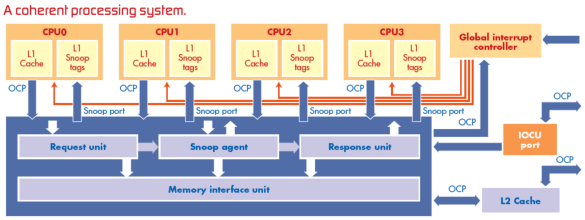
```
valgrind --tool=cachegrind -v ls
```



Cache Coherence

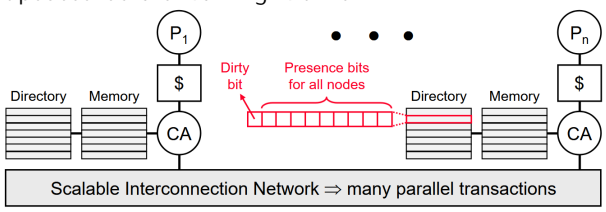
- Chaches store a local copy of main memory, but must be consistent (over all caches)
- If both caches have a copy of the same memory location and
  1. cache A is modified
  2. cache B can be updated or invalidated
     - if invalidated, the cache is updated when the CPU wants to read the previously cached location

Bus Snooping Each cache monitors all the other caches bus traffic to see if another cache changed.



Problem: This doesn't scale well for many processors and caches.

Directory Based Coherence Directory holds an entry for each memory line to say which caches have a copy. The dirty bit then indicates if the memory block is in a modified state and has to be updated before working with it.



Cache Organisation

- Cache needs to be **fast**
- Need a fast way to determine if a memory location is in cache

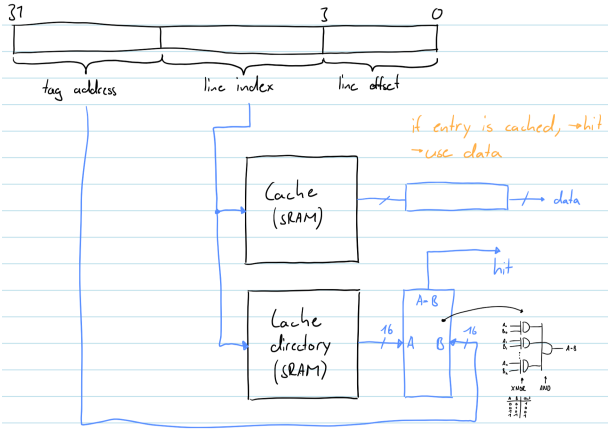
If use **LUT**, assume 32-bit addr, 1 bit per LUT-entry: *not feasable*

$$2^{32} \cdot 1bit = 4Gbits$$

Use **Hashing Techniques** with a cache directory (SRAM).

Direct Mapped (DM) Cache

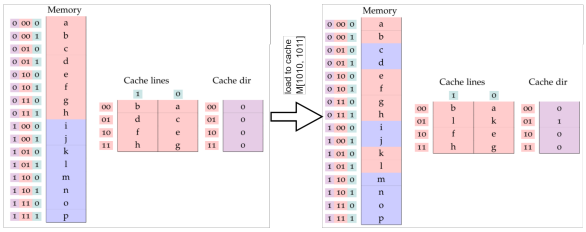
Cache controllers must quickly determine if memory is cached. For this the (actual) memory address is split into three fields:
Line Offset Bits = $\log_2$(Line Size)
Line Index Bits = $\log_2$(Cache Size) − $\log_2$(Line Size)
Tag Address Bits = Address Size Bits − $\log_2$(Cache Size)



- The **Line Index** is used to index the cache and the cache directory.
- The **Cache Directory** stores the last cached **Tag Address** and outputs this on addressing with the line index.
- The **Line Offset** describes in what part of the line the actual byte is.
- A **Hit** occurs when the *requested* tag address matches the *stored* tag address.

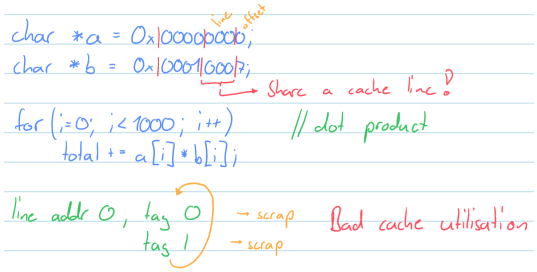
Example

1. First 8 bytes are cached
2. CPU tries to read M[1011]
3. line: 01, tag: 1
4. tag in cache dir: 0 → **Miss**
5. cache controller loads line 01, with tag 1: M[1010, 1011]
6. cache directory line 01 is filled with tag: 1



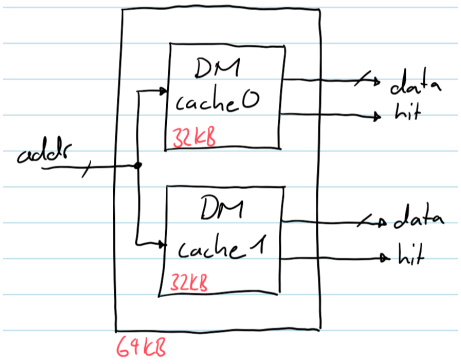
Cache Thrashing

- There is a many to one mapping (address → line address)
- There is only one spot to cache a given memory address



Set-associative Caches

- There are several choices in cache for a given address
- Equivalent to multiple DM caches in parallel
- *Reduces* cache thrashing



2-Way Set-associative

- Two DM cahces in parallel
- Two choices for a given address

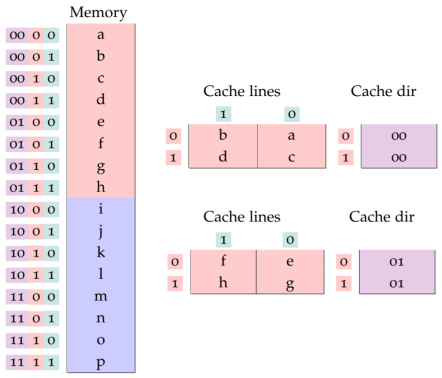


Figure 0.3: 2-way accociative cache example

N-Way Set-associative

- Provides *N* choices in cache
- When cache line is loaded, cache needs to choose which of *N*-Lines to use:
  1. LRU - Least Recently Used
  2. Random

Fully associative cache

- *N*-Way Set-associative cache for *N*-lines in cache
- no restrictions on cache line placement → **avoids thrashing**
- *N* DM caches in parallel with one line each

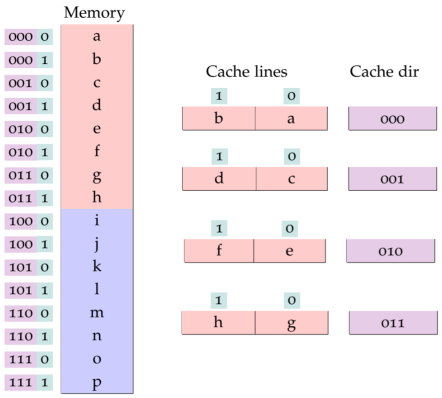


Figure 0.4: Fully associative cache example

Virtual Memory Systems

If we work with data that can't fit into RAM anymore, we use a *virtual memory system*. The system uses the Harddisk as a higher level cache. The CPU is working with a virtual address, which has to be translated to a physical address.

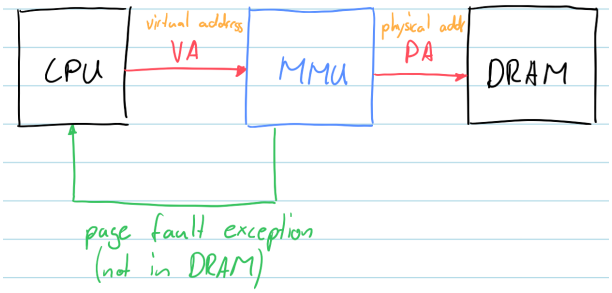
Virtual Memory acts like a cache for DRAM in the Hard Disk

- cache: 8-64 byte lines
- VM: 4kB/4MB pages
- VM: Has horrendous miss penalty (better with SSD)
- VM creates illusion that CPU has more memory

Advantages of Virtual Memory Systems:

- Coexistence of multiple programs with separate address spaces with protection from each other
- Better system security: Accessing invalid pages (e.g. device registers) generates page fault (Segmentation Violation)
- Simpler program loading (All processes can start at address 0)
- Faster program startup, since pages are loaded on demand
- Extension of the address space of a running process
- Retirement of DRAM pages that have memory faults

MMU - Memory Management Unit



- MMU translates VA (virtual addr) → PA (physical addr)
- MMU generates a page fault exception if VA is not in DRAM
- Page fault exception is handled by OS → *paging*

Virtual Addresses / Page Tables

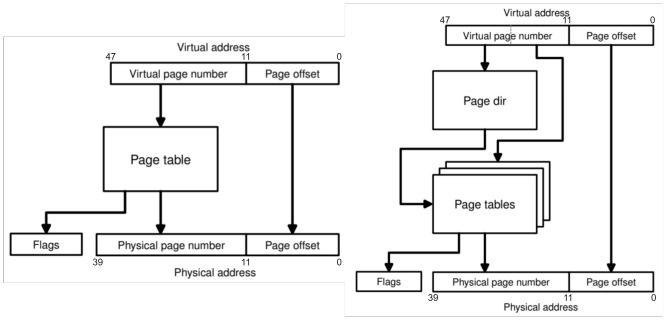


Figure 0.5: Page Table (left) and Hierarchical Page Table (right)

Virtual Addresses (VA) are split into a Virtual Page Number (VPN) and a Page Offset. VPN is used to page the Physical Page Number (PPN).
VPN consist of 48 bits and if we'd address 39 bit addresses, we'd need a lookup table of size

$$\frac{2^{48} \cdot 39 \text{ bits}}{8} = 1248 \text{ TB}$$

Through the use of pages (4 KB) the size can be reduced to

$$\frac{2^{48} \cdot 39 \text{ bits}}{4 \cdot 2^{10} \cdot 8} = 312 \text{ GB}$$

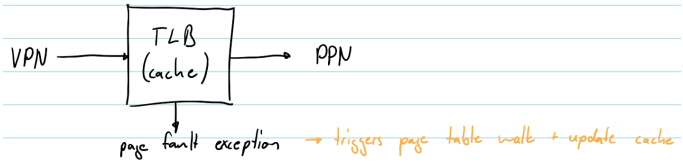
or 312 MB with 4 MB pages.
These pages are still really big, that's why we use **Hierarchical Page Tables**:

- Page tables and page directories stored in DRAM (can be paged to disk)
- Reduces page table memory
- Perform page table walk to get final addr

TLB - Translation Lookaside Buffer

- Special cache to avoid cache table walks
- Some CPUs doe page table walks in HW → most in SW

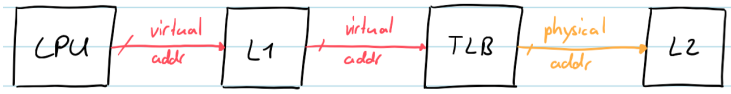
  - It's not done very often (Amdahl's Rule)
- Page table walk follows pointers to find PPN from VPN



- TLB is fully associative (no restrictions where a addr can be stored (penalty is to big if it's not in cache)

TLB and Caches

Normally TLB look ups are only performed if there is a cache miss on L1 cache.



VIVT **V**irtual **I**ndex **V**irtual **T**ag

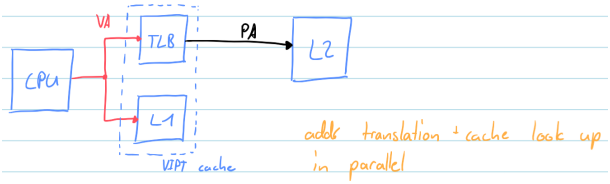
- L1 Cache
- Translation only occur for cache miss
- Has aliasing problem: multiple VA → PA

PIPT **P**hysical **I**ndex **P**hysical **T**ag

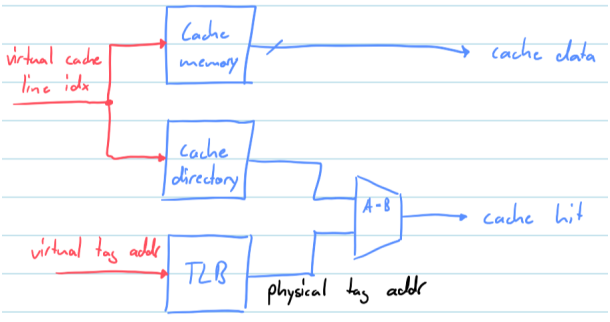
- L2 Cache
- Uses translated PPN addresses

VIPT **V**irtual **I**ndex **P**hysical **T**ag

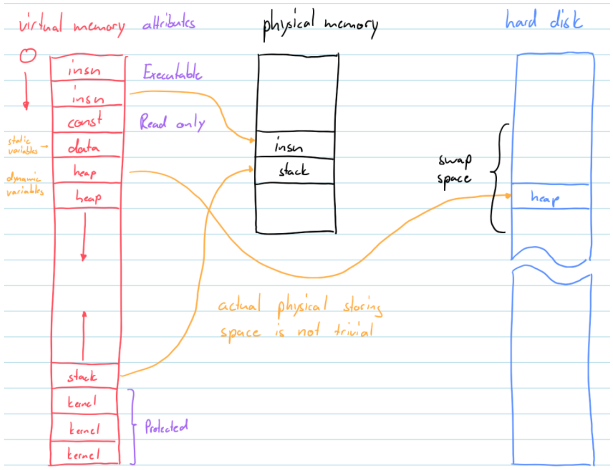
Do the addr translation and cache look up in parallel, that way the PA is ready if there is a cache miss.



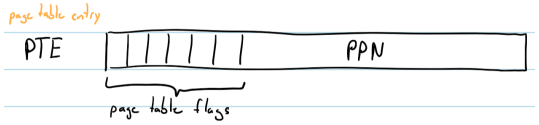
This is implemented with 3 SRAMs in parallel



Virtual Memory Paging (swapping)



VM Page Table Attributes



- **Presence** - Set if page resident in DRAM
  - page fault exception if try to read a page if bit not set
- **Modified** - Indicates page written (dirty)
  - needs to be written to disk
- **Reference** - Set if page accessed
  - OS uses LRU (least recently used) algorithm to determine if page should be in DRAM

Security improving Attributes:

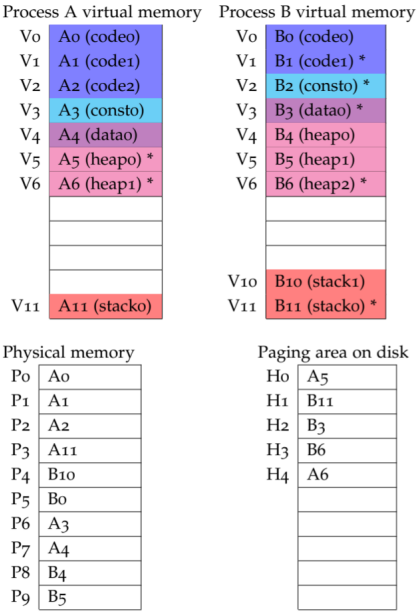
- **Read Only** - Page fault exception if you try to write to page
- **Execute** - Indicates page contains instructions
- **Owner** - Indicates a page that can only be accessed by OS kernel

IOMMU

MMU for IO devices (e.g. mapping GPU memory to physical addr space)

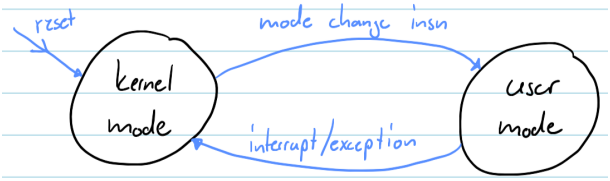
Multiple virtual addr spaces

In a multitasking system, each process has its own page table and the CPU selects the current page table using a page table register.



Processor Modes

Different CPU modes with different privileges maintain system security.



Kernel Mode

- Unrestricted access to hardware, page tables, CPU registers, and privileged instructions
- Entered when interrupt occurs, the previous processor mode is restored when the ISR executes a return-from-interrupt instruction

User Mode

- Used by applications so they cannot directly access hardware devices or memory belonging to other tasks
- The application must make a system call to access the hardware

Profiling

To determine the *hotspots* of a program, the code parts which can improve the execution speed the most, we use profiling techniques.

External Timing

1. Drive GPIO-Pin **High** at start of function
2. Drive GPIO-Pin **Low** at end of function
3. Use *Oscilloscope* to measure the time

Internal Timing

Use integrated clock functions that approximate the time used by the CPU (`#include <time.h>`).

Attention: Poor resolution and no notice of interrupts
→ average over many iterations

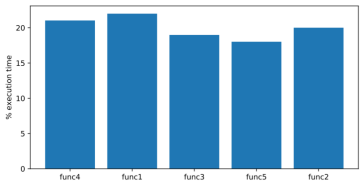
Subroutine Profiler

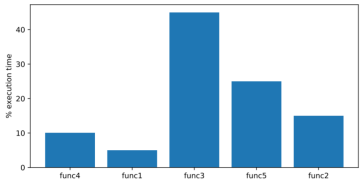
Used to find which fucntion takes the most time (Tool: **gprof**).

1. Compile with profiling enabled
   a. `gcc -pg` (profiling flag)
   b. Compiler adds extra code to start of every function to count how often it's called
2. Run program as usual → on exit info & additional timing info gets dumped to file
3. Run profiling analysis (**gprof**) to analyse saved info and annotate the C-code

Flat timing profile

As a result of the analysis, we get a **flat timing profile**, which shows how much time is spent in each function.
If no predominant function (first example) it's harder to determine where to start optimising, where a dominant function (`func3` second example) gives a better clue.





A flat profile comes as a text summary:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 84.44      0.31     0.31     1000     0.31     0.31  dvdot
 16.34      0.37     0.06   100000     0.00     0.00  dgauss
  0.00      0.37     0.00        1     0.00    60.47  dvrand_gauss


                     Call graph


granularity: each sample hit covers 2 byte(s) for 2.68% of 0.37 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.37                 main [1]
                0.31    0.00    1000/1000        dvdot [2]
                0.00    0.06       1/1           dvrand_gauss [4]
-----------------------------------------------
                0.31    0.00    1000/1000        main [1]
[2]     83.8    0.31    0.00    1000         dvdot [2]
-----------------------------------------------
                0.06    0.00  100000/100000      dvrand_gauss [4]
[3]     16.2    0.06    0.00  100000         dgauss [3]
-----------------------------------------------
                0.00    0.06       1/1           main [1]
[4]     16.2    0.00    0.06       1         dvrand_gauss [4]
                0.06    0.00  100000/100000      dgauss [3]
-----------------------------------------------


Index by function name

   [3] dgauss                  [2] dvdot                   [4]
       dvrand_gauss
```

Deterministic Profiler

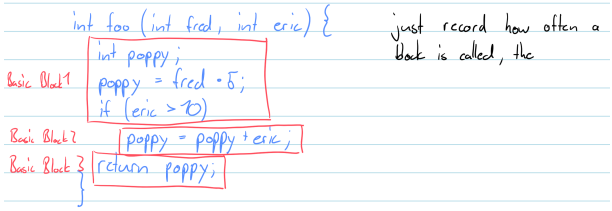
The compuler adds code at the start and end of functions to read the elapsed time.
Attention: This adds alot of additional code.

Statistical Profiler

Periodic interrupt handle that **samples Program Counter** of the interrupted function and stores this in profiler file. This is simple to implement, but subject to sampling artefacts (missing frequent but very short functions).

Block Profiler

Used to find which line is executed the most →$ assume you have to optimise this line.
Tools: **gcov**, **oprofile** (both Linux)



> **Basic Block**
>
> A group of statements that are always executed together.
> No *flow-changing* statements (e.g. `if`, `for`, ...)

Code Coverage With *basic block profiling* we can work out the **code coverage**. We can work out how many times code is executed.
This is useful to detect **dead code**.

```
        -:    0:Source:coverage.c
        -:    0:Graph:coverage.gcno
        -:    0:Data:coverage.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include "stdlib.h"
        -:    2:
     1000:    3:static double dvdot (double *src1, double *src2,
        -:    4:                      int size)
        -:    5:{
     1000:    6:    double total = 0.0;
        -:    7:    int i;
        -:    8:
100001000:    9:    for (i = 0; i < size; i++)
100000000:   10:        total += src1[i] * src2[i];
     1000:   11:    return total;
        -:   12:}
        -:   13:
   100000:   14:static double dgauss (void)
        -:   15:{
        -:   16:    int j;
   100000:   17:    double total = 0.0;
        -:   18:
 10200000:   19:    for (j = 0; j <= 100; j++)
 10100000:   20:        total += rand () / 100.0;
   100000:   21:    return total;
        -:   22:}
```

Just-in-time Profilers

Just-in-Time profilers can perform the analysis dynamically, generating detailed information, but making the program running slower.
Tools: **callgrind** (from *valgrind*)

Simulators

A accurate simulator of a CPU can give deep insights about **pipeline stalls**, **cache statistics**, **Clock Cycles per Instruction**, and much more.
They are extremely complex and slow and hard to get for modern (e.g. X86) CPUs.

Hardware Profilers

Modern CPUs contain registers, that store information like **Clock Cycles per Instruction**, **Wrong Branch Predictions**, **Cache Misses**, ...
Tools: **perf** (Linux, X86)

Optimisation

Optimisation is mainly done by compilers and optimally produce smaller and faster code. Some speed optimisations do tradeoff with size (e.g. loop unrolling).

- **Avoid** premature optimisation
- Use **profiling** tools to find what to optimise first
- Use a **better algorithm** (e.g. FFT vs. DFT)

GCC optimisation levels

GCC has built in optimisation (100's of flags), but there are predefined optimisation levels:

- `-O1` - enable simple optimisations
- `-O2` - enable `-O1` with additional optimisation, *no speed/memory tradeoff*
- `-O3` - enable `-O2` with additional optimisation, may make program larger (e.g. loop unrolling)
- `-Og` - enables optimisation that do not interfere with debugging
- `-Os` - optimise for size, uses `-O2` optimisations that do not increase code size
- `-Ofast` - enable `-O3` with additional non standard compliance optimisation

Aliasing

```
int foo (int *pa, int *pb)
{
    *pa = 1;
    *pb = 2;
    return *pa == 1;
}
```

gcc optimised

```
int foo (int *pa, int *pb)
{
    *pa = 1;
    *pb = 2;
    return 1;
}
```

Problem: If `*pa` and `*pb` point to the same address, the return value should be 0, that's why the compiler has to **play safe**.
Solution: Restricted Pointer

```
/* pa and pb do not alias since they are specified
   with the restrict qualifier.  When using gcc
   you need to compile with -std=c99 or later.  */
int foo (int * restrict pa, int * restrict pb)
{
    *pa = 1;
    *pb = 2;
    return *pa == 1;
}
```

C Aliasing Rules

1. A pointer to `char` can alias with any other pointer
2. Pointers to data types that differ only by qualifier can alias. (e.g. `unsigned int` can alias `int`)
3. Pointers to different built-in types do not alias. (`int` and `double`)
4. Pointers to aggregate or union types with differing tags do not alias (two separately defined unions)
5. Pointers to aggregate or union types which differ only by name may alias (two unions based of the same definition)

Fast and Loose math

FLoating point is a poor aproximation of real numbers, they are special rational numbers of the form

$$\frac{M}{2^N}$$

with M and N as integers. Thus numbers such as 0.1 can't be represented correctly.

```
for (i = 0; i < N; i++)
    total += a[i] / 10.0;
```
"optimised"
```
for (i = 0; i < N; i++)
    total += a[i] * 0.1;
```

This operation doesn't give the same result, thus the compiler won't optimise it.
If numeric accuracy is *not important*, this optimisation can be enabled with the `-fast-math` option.
Furthermore, float point arithmetic is non associative, the order matters, because:

$$1e16 + 1 - 1e16 \to 0$$
$$1e16 - 1e16 + 1 \to 1$$

Assiting the compiler to optimise

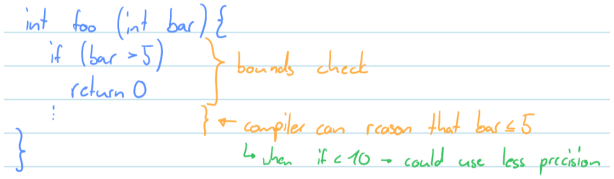
Data Qualifier:

- `unsigned` - value never negative
- `const` - value never change
- `volatile` - value can unpredictably change (shared ISR variable HW registers) → useful to tell the compiler not to optimise

```
for (i=0; i<1000; i++){
    continue;
}
// i=1000, would be otpmised away
// but might be used at other point!
```

- `restrict` - pointer not aliased

Bounds checking



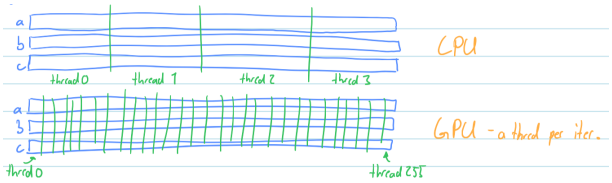
GPU - Graphics Procesing Unit

Originally designed for graphics, including VRAM and many pipelined FUs *(shaders, texture mapping, render output, special functions [sine/cosine/...], ray tracers)*.
NVIDIA introduced the **GPGPU** - **G**eneral **P**urpose **GPU**, which got popular for AI. It consists of many more cores than a CPU (1000's).

```
void vmul (double * restrict a, double * restrict b,
           double * restrict c)
{
    for (int i = 0; i < 256; i++)
        c[i] = a[i] * b[i];
}
```

CUDA Code

```
__global__ void mul (double * restrict a, double * restrict b,
                     double * restrict c)
{
    int i = threadIdx.x;
    c[i] = a[i] * b[i];
}
```



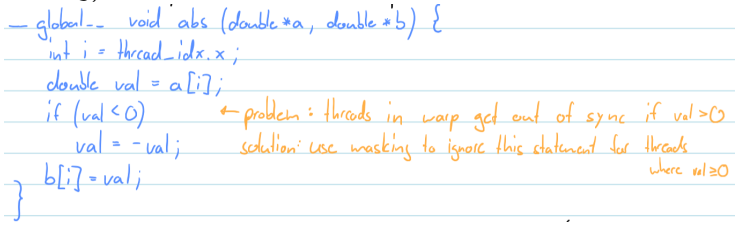
Architecture

Each Core consists of many functional units (typically 32), but have a lot less cache memory than a CPU.

> **Latency Hiding**
>
> Due to the smaller cache size, memory has to be read from main memory more often. To mitigate the introduced latency, **latency hiding** is introduced.
> Whenever a thread is waiting for memory, another thread can be run (i.e. multithreading).
> This requires less fanc caching, but **fast context switches**.
> This is achieved by having a set of registers for each thread.

SIMT - Single Instruction Multiple Threads

- Group block of threads together, typically 32. *(NVIDIA: warp, AMD: wavefront)
- Special warp core (NVIDIA: streaming multiprocessor, AMD: SIMD)
- Each warp has a shared single instruction pointer
- Threads in a warp execute in lock-step
- ! Divergent paths are bad for utilisation of the FUs (branching)



- Latency hiding: If a warp is blocked, another queued warp is executed.

GPU Programming

Threads are organised as a grid. It is often **faster to recalculate results, than read them from memory.**

- CUDA - Compute Unified Device Architecture: NVIDIA framework

- OpenCL - Open Computing Language: Apple framework, for heterogeneous crossplatform execution (CPUs, GPUs, DSPs, FPGAs)
- OpenACC - Open Accelerators: Programming standard for parallel computing on heterogeneous CPU/GPU systems

Advanced Topics and Future Technologies

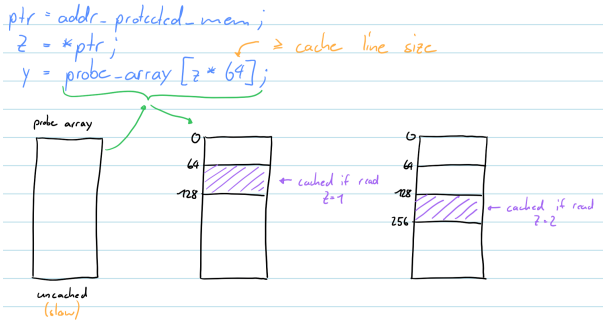
Computer exploits

Typical computer exploits are based on reading protected memory through a side channel.

Side Channel

The Side Channel is the central part of computer exploits and utilises the fact that a cached memory location is faster to read.

1. Prepare side channel: allocate a probe array and ensure it's not cached *(by reading lots of other memory)*
2. Fool the CPU to read a protected memory location *(e.g. kernel)* and use the result as an index into the probe array



Meltdown

Relies on a flaw with out of order execution.

> **Page Fault Exception**
>
> When we try to access protected memory through `Z=*ptr;` a page fault exception occurs, because access is not allowed. **Meltdown** works around that with utilising a signal handler.

1. Set up a signal handler to capture page fault exception otherwise the OS aborts the program
2. Prepare probe array
3. Read from protected memory location

```
Z = *ptr; // Some CPUs take a long time to check page
↪  permissions (pipelining)
y = probe_array(Z*64); // in the meantime this insn is
↪  already executing and loading the `Z` part into
↪  cache
```

4. Through timing analysis on accessing the probe array, the value of `Z` can be figured out

Meltdown can be avoided by:

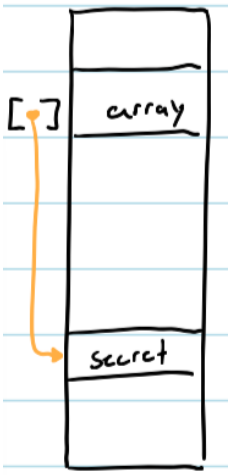
- Clearing cache on page fault (slow)
- Disable cache (very slow)
- Disable out of order execution (slow)
- Randomize timing [apple]
- Only map a few of the kernel pages in the virtual memory space (e.g. at least the interrupt vectors)

Spectre

Relies on a flaw with speculative execution. The attacker needs to find a special function that runs with privileges in the kernel or web server. This function must have the form of

```
char attack (int x, char *probe_array){
    char y=0;
    if (x<array_size)
        y = probe_array[array[x]*64]; // 64 must be
↪   int >= 64
    return y;
}
```

1. Prepare branch predictor by calling `x<array_size` repeatedly
2. Ensure probe array is not cached
3. Call attack function with value of x outside of array bounds
4. Use memory timing to infer secret value



Spectre can be avoided by:

- Disable cache (really slow)
- Disable speculative execution (slow)
- Randomize timing
- Use compiler to avoid generating code that can be exploited

Instruction set architecture problems

The ARM Cortex A-15

Quantum computing

Quantum computers (superposition)

Quantum computers (entanglement)