IoT Device Programming and Debugging

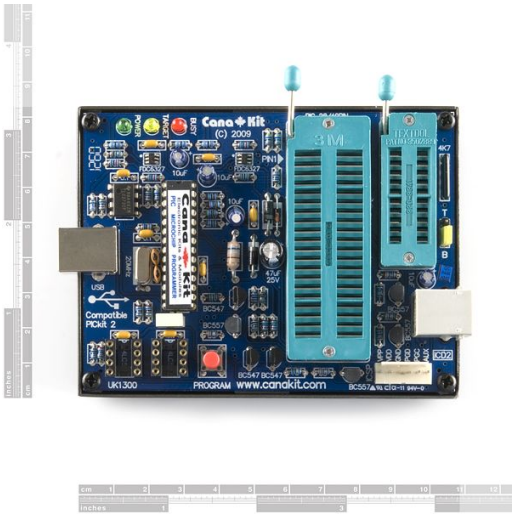
Tom Spink

Programming

Programming is the act of uploading code and data to an embedded device.

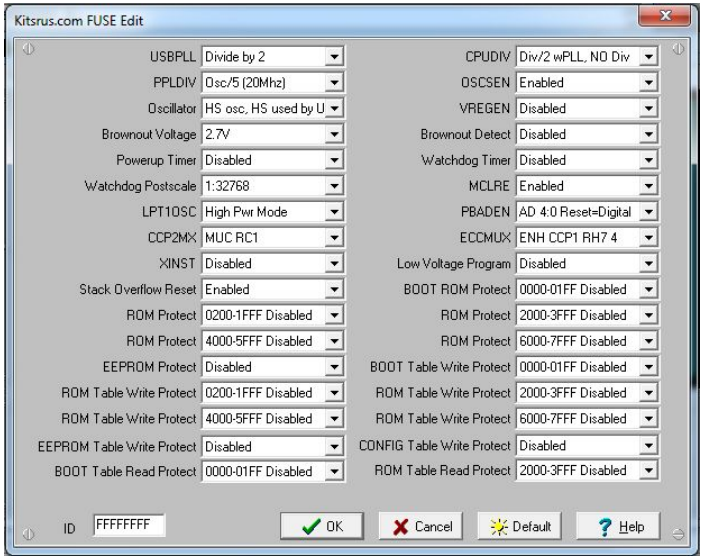
In general, it encompasses writing data into EEPROM and usually the configuration of the target device, via device configuration fuses.

Each microcontroller has its own method of programming, but often a bootloader will be used to enable a more standard or convenient programming method.



Programming

Configuration fuses are a non-volatile memory used to configure the fundamental behaviour of a microcontroller, e.g. the clock source, clock rate, watchdog timer, brown-out detect, etc.



Bootloader

A bootloader is a special (tiny) application that prepares a microcontroller for execution of the main program, by initialising memory and peripherals.

It is the first piece of code executed when the microcontroller comes out of reset.

It can also act as an interface to a more convenient programming method, i.e. expose functionality to enable writing to program and data memory through other means.

Debugging

Debugging an embedded system can be quite a complicated (and expensive!) task, in comparison to debugging a software application.

It’s an invaluable tool during development, as there may be a lot of peripherals to consider.

Ideally, debugging should be rapid so that you don’t have to wait for results.

Debugging

Using an IDE is the ideal approach for debugging an embedded application, as it should contain all the tools and functionality for inspecting the behaviour of the device.

In-circuit debugging allows you to debug the operation of a device, whilst in place on a circuit board - but the microcontroller must support this!

Simulation allows you to run a virtual version of a device, without having to use physical hardware.

Application-specific Debugging

Some applications may implement their own proprietary debugging routines, that are highly application-specific.

For example: implementing a command-line interface that can interrogate device memory.

But! Remember to disable this for production, or you may open up security holes!

Production of a device may leave debug headers available for factory programming, testing and QA, but they should be disabled afterwards.

Simulation

Simulation of a hardware platform is very useful, as you get to work with a virtual version of device that might not yet physically exist.

There is no need to deploy the application to hardware, which takes time, but also may degrade flash memory.

Simulation lets you work on the hardware design in parallel to software, or implement hardware/software co-design. Levels of simulation give different insights into the behaviour of the device:

- Functional Simulation — Fast
- Cycle-accurate Simulation — Slow
- RTL simulation — Very slow

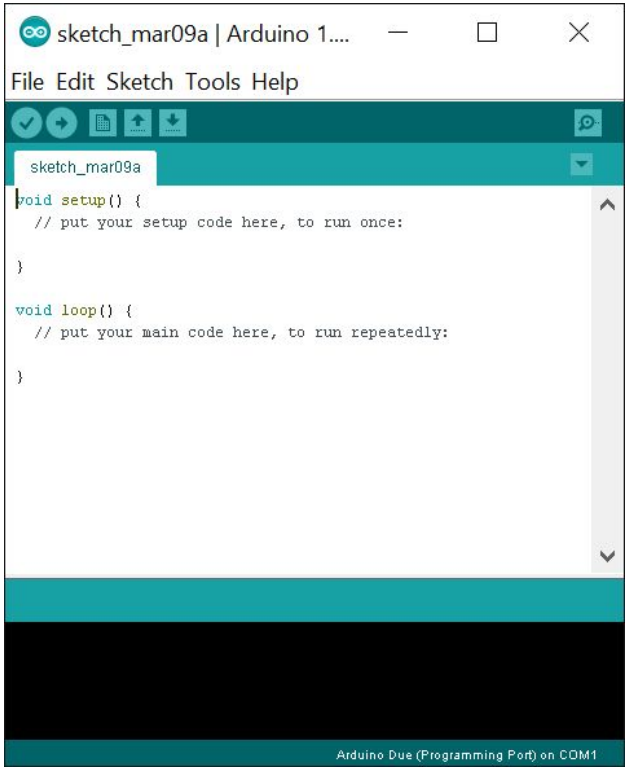
Integrated Development Environments

IDEs are essential for working on large-scale embedded applications.

They can be used for the development of application software and the management of frameworks/libraries.

Most have the facility to program the target device, and some have the facility for in-circuit debugging.

Generic IDEs and manufacturer IDEs exist, with varying levels of support.



Application Frameworks

Frameworks introduce a level of maintainability to an application. By re-using common code, it reduces maintenance burden, improves code quality, and you benefit from (possibly) more scrutinised and tuned implementations.

Drawbacks are security issues, which when discovered in a framework will impact all devices using that framework, and obsolescence when a developer stops supporting it.

Also, performance can be an issue, if core functionality is implemented as function calls, e.g. a HAL which makes calls to toggle GPIO (Arduino's digitalWrite()).

Standard Interfaces

Serial interfaces are typically found on Microcontrollers, e.g. UARTs. These can be used for application-specific communication, debugging or other purposes.

FTDI design and manufacture ICs for bridging a USB serial-port to UART, an extremely straightforward way to get USB connectivity to a device.

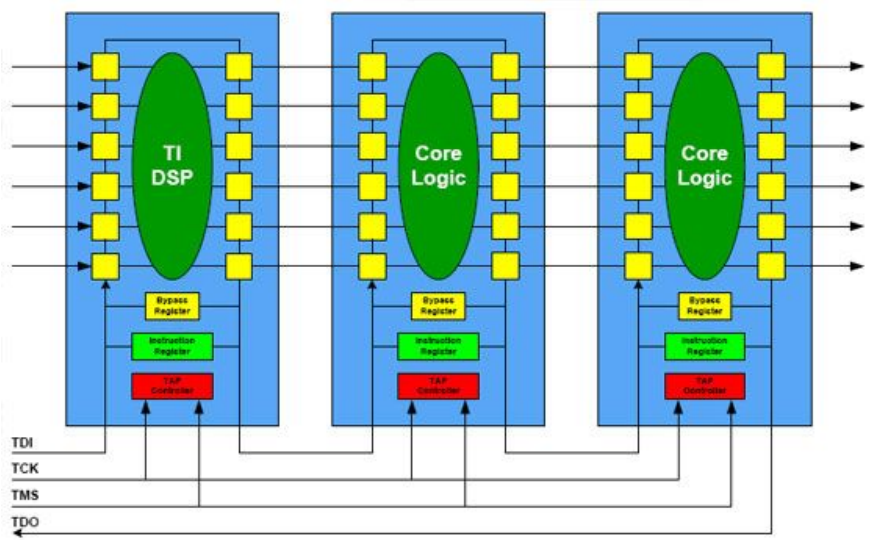
JTAG - Joint Test Action Group - is an industry standard for testing and debugging circuits after manufacture. It specifies a serial debug port that can be used to interrogate various chips, and even upload data to flash memory.

Joint Test Action Group (JTAG)

A JTAG interface is a standardised interface for programming and debugging.

It connects between multiple chips on a bus, and allows commands and data to be sent to any chip on the bus.

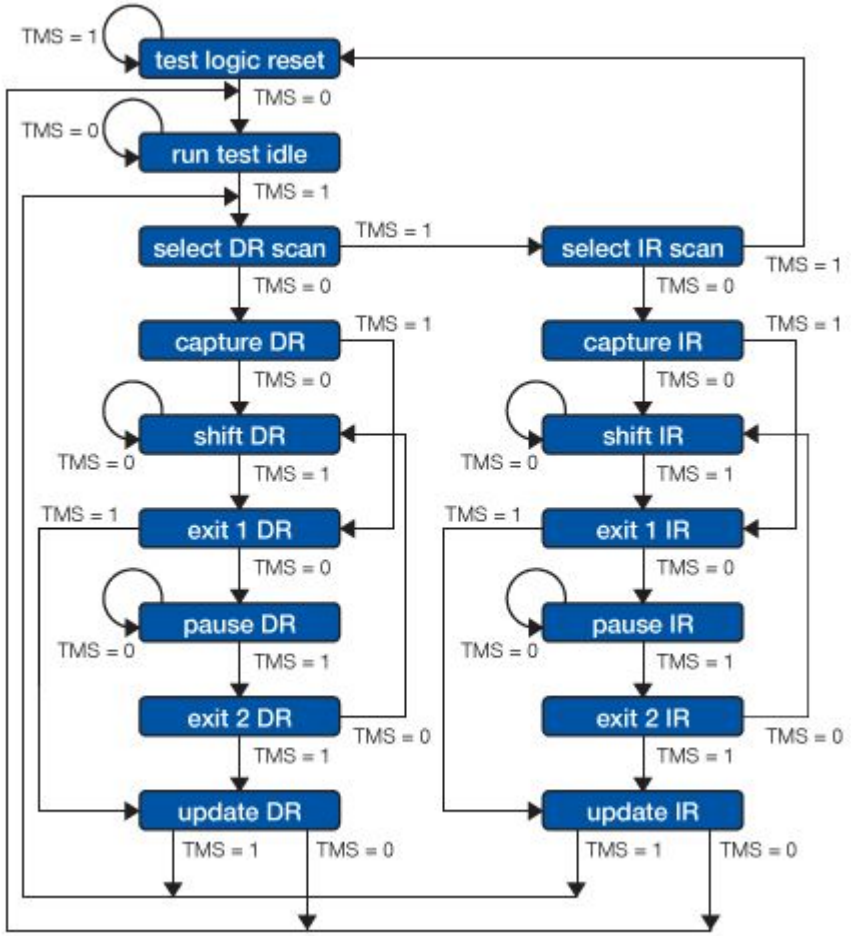
Boundary scanning enables a direct view of the input/output pins on a chip, and thus interconnections between chips (that are difficult to probe) can be tested by interrogating the state of chips' external pins.



JTAG

The JTAG state machine specifies how a device should interpret JTAG communications.

Holding TMS high for at most 5 TCK cycles will guarantee returning to the test logic reset state. (TRST is an optional signal)



Over-the-air Programming & Debugging

Over-the-air (OTA) programming can be convenient for rapid prototyping and testing of devices before or after deployment.

However, there is a danger of bricking the device, and if the device is physically inaccessible, then it is lost.

If programming is interrupted half-way through, or bad data is transmitted or received, it could cause undefined behaviour.

By assuming the worst (defensive programming), the bootloader/programmer should be developed to recover from an interruption.

Over-the-air Programming & Debugging

An on-chip bootloader may implement functionality to handle firmware updates from existing communication devices, enabling existing device infrastructure to be re-used.

The drawback is that if the update fails, and the bootloader becomes corrupt, the device is bricked.

A dedicated off-chip programmer may implement the required protocol for OTA firmware updating, being completely separate to the actual application.

The drawback is that this increases device size, complexity and cost.

Hardware and Software Optimisation

Tom Spink

Optimisation

Modifying some aspect of a system to make it run more efficiently, or utilise less resources.

Optimising hardware: Making it use less energy, or dissipate less power.

Optimising software: Making it run faster, or use less memory.

Choices to make when optimising

Optimise for speed? Do we need to react to events quickly?

Optimise for size? Are we memory/space constrained?

Optimise for power? Is there limited scope for power dissipation?

Optimise for energy? Do we need to conserve as much energy as possible?

Some combination (with trade-off) of all of these?

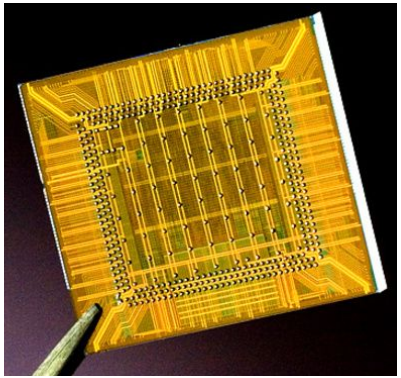
Hardware Optimisation

Additional Processors

DSPs implement specialised routines for a specific application.

FPGAs are a popular accelerator in embedded systems, as they provide ultimate re-configurability. They are also invaluable during hardware prototyping.

ASICs are the logical next step for a highly customised application, if gate-level re-configurability is not a requirement.



Field Programmable Gate Arrays (FPGAs)

FPGAs implement arbitrary combinational or sequential circuits, and are configured by loading a local memory that determines the interconnections among logic blocks.

Reconfiguration can be applied an unlimited number of times - and “in the field”!

Useful for software acceleration, with the potential for further upgrades, or dynamic adaptation.

Very useful for hardware prototyping. Experimental data can feed into ASIC design.

Application-specific Integrated Circuits (ASICs)

- Designed for a fixed application, e.g. bitcoin mining.
- Designed to accelerate heavy and most used functions.
- Designed to implement the instruction set with minimum hardware cost.
- Goals of ASIC design:
  - Highest performance over silicon and over power consumption
  - Lowest overall cost
- Involves:
  - ASIC design flow, source-code profiling, architecture exploration, instruction set design, assembly language design, tool chain production, firmware design, benchmarking, microarchitecture design.

ASIC Specialisations

Instruction Set Specialisation

- Implement bare minimum required instructions, omit those which are unused.
  - Compress instruction encodings to save space.
  - Keep controller and data paths simple.
- Introduce new, possibly complex, application-specific instructions.
  - Combinations of common arithmetic operations (e.g. multiply-accumulate)
  - Small algorithmic operations (e.g. encoding/decoding, filtering)
  - Vector operations.
  - String manipulation/matching.
  - Pixel operations/transformations.
- Reduction of code size leads to reduced memory footprint, and hence better memory utilisation/bandwidth, power consumption, execution time.

ASIC Specialisations

Functional Unit and Data-path Specialisation

After the instruction set has been designed, it can be implemented using a more or less specific data path, and more or less specific functional units.

- Adaptation of word length.
- Adaptation of register count.
- Adaptation of functional units.

Highly specialised functional units can be implemented to deal with highly specialised instructions, such as string manipulation/matching, pixel operations, etc

ASIC Specialisations

Memory Specialisation

- Number and size of memory banks + number and size of access ports
  - Both influence the degree of parallelism in memory accesses.
  - Having several smaller memory blocks (instead of one big one) increases parallelism and speed, and reduces power consumption.
  - Sophisticated memory structures can drastically increase cost and bandwidth requirements.
- Cache configurations
  - Separate or unified instruction and data caches?
  - Associativity, cache size, line size, number of ways.
  - Levels/hierarchy.
- Very much dependent on application characteristics.
  - Profiling very important here!
- Huge impact on performance/power/cost.

ASIC Specialisations

Interconnect Specialisation

- Interconnect of functional modules and registers
- Interconnect to memory and cache
  - How many internal buses?
  - What kind of protocol? Coherency?
  - Additional connections increase opportunities for parallelism.

Control Specialisation

- Centralised or distributed control?
- Pipelining? Out-of-order execution?
- Hardwired/microcoded?

Digital Signal Processors (DSPs)

Designed for a specific application, typically arithmetic heavy. Offers lots of arithmetic instructions/parallelism.

- Radio baseband hardware (4G)
- Image processing/filtering
- Audio processing/filtering
- Video encoding/decoding
- Vision applications

Choose to implement a DSP that performs processing faster, and uses less memory and power, than if the algorithm was implemented on a general purpose processor.

Heterogeneous Multicores

A heterogeneous multicore is a processor that contains multiple cores that implement the same underlying architecture, but have different power and performance profiles.

An example is ARM big.LITTLE, of which a configuration is available comprising four Cortex-A7 and four Cortex-A15 (for a total of eight) cores.

This enables threads to run on the low energy cores, until they need a speed boost, or where multithreaded applications may have threads with different performance requirements, running on different cores.

Run-state Migration

Clustered Switching: Either all fast cores or all slow cores are being used to run threads.

CPU Migration: Pairs of fast and slow cores are used for scheduling, and each pair can be configured to execute the thread on either the fast core or the slow core.

Global Task Scheduling: Each core is seen separately, and threads are scheduled on cores with the appropriate performance profile for the workload.

Software Optimisation

Optimisation Targets

Optimise for speed Generate code that executes quickly; at the expense of size

Optimise for size Generate least amount of code; at the expense of speed

Optimise for power/energy Combination of optimisation strategies

Use analysis, debugging, simulation, prototyping, monitoring, etc to feedback into the optimisation strategy.

Compiler Optimisation Choices - Speed/Size

Optimise for Speed (-O3)

- May aggressively inline
- Can generate MASSIVE amounts of code for seemingly simple operations
  - e.g. separate code to deal with aligned vs. unaligned array references
- Re-orders instructions
- Selects complex instruction encodings that can be quite large

Optimise for Size (-Os)

- Will penalise inlining decisions
- Generates shorter instruction encodings
- Affects instruction scheduling
- Fewer branches eliminated (e.g. by loop unrolling)

Custom Optimisation (-On)

- You might be interested in very specific optimisation passes
  - LLVM is best for this
- You may want to insert assembly language templates.

Code Size

Does it matter?

128-Mbit Flash = 27.3mm^2 @ 0.13µm

ARM Cortex M3 = 0.43mm^2 @ 0.13µm

RISC architectures sacrifice code density, in order to simplify implementation circuitry, and decrease die area.

Code Size

Possible solution: Dual Instruction Sets

Provide a 32-bit and a 16-bit instruction set:

- Thumb/Thumb-2
- ARCompact
- microMIPS

...but 16-bit instructions come with constraints!

- Only a subset of the registers available
- Must explicitly change modes
- Range of immediate operands reduced

Code Size

Possible solution: CISC Instruction Set

Provide complex instruction encodings that can do more work:

- x86
- System/360
- PDP-11

...but CISC instruction sets are by definition complex, and require more complex hardware.

Often the support for generating more exotic instructions doesn't exist in the compiler, negating some benefit.

Instruction Level Parallelism

- Normally considered a way to increase performance.
- Dynamic Hardware-based Parallelism
  - Hardware decides at runtime which instructions to execute in parallel.
  - Pipelining, out-of-order execution, register renaming, speculative execution, branch predication.
- Static Software-defined Parallelism
  - Compiler decides at compile time which instructions should be executed in parallel.

Very Long Instruction Word (VLIW) Computing

- Instead of executing individual instructions, hardware executes bundles of instructions.
- Unused parallel units must be filled with a nop.

Vectorisation

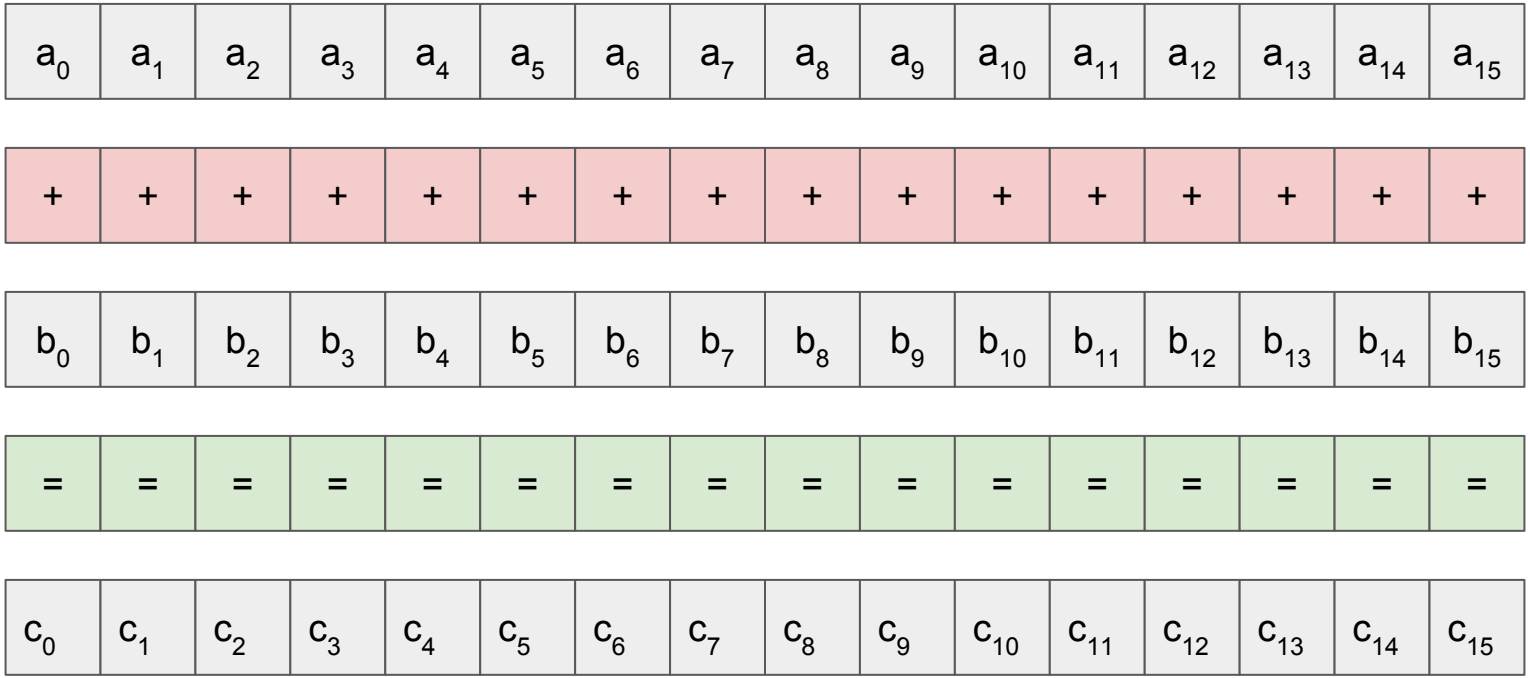
```
for i = 0 while i < 16 step 1
    c[i] = a[i] + b[i]
```

Choose a vector width, and divide total count to vectorise the loop (or increase step count)

```
for i = 0 while i < 16 step 4
    c[i:i+3] = a[i:i+3] + b[i:i+3]
```

Be careful if vector width doesn't divide iteration count exactly! Need extra code complete the operation.

Vectorisation: Scalar Operations



Vectorisation: Vector Operations

Vector width: 4

Vectorisation

Exploit parallel data operations in instructions: **c[i] = a[i] + b[i]; // 0 ≤ i < 16**

```
no_vectorisation:
  xor    %eax, %eax
1:
  mov    (%rdx, %rax, 1), %ecx
  add    (%rsi, %rax, 1), %ecx
  mov    %ecx, (%rdi, %rax, 1)
  add    $0x4, %rax
  cmp    $0x40, %rax
  jne    1b
  retq

avx512:
  vmovdqu32  (%rdx), %zmm0
  vpaddd     (%rsi), %zmm0, %zmm0
  vmovdqu32  %zmm0, (%rdi)
  vzeroupper
  retq
```

```
with_vectorisation:
  movdqa (%rsi), %xmm0
  paddd  (%rdx), %xmm0
  movaps %xmm0, (%rdi)
  movdqa 0x10(%rsi), %xmm0
  paddd  0x10(%rdx), %xmm0
  movaps %xmm0, 0x10(%rdi)
  movdqa 0x20(%rsi), %xmm0
  paddd  0x20(%rdx), %xmm0
  movaps %xmm0, 0x20(%rdi)
  movdqa 0x30(%rsi), %xmm0
  paddd  0x30(%rdx), %xmm0
  movaps %xmm0, 0x30(%rdi)
  retq
```

```
with_vectorisation_no_unroll:
  xor    %eax, %eax
1:
  movdqu (%rsi, %rax, 4), %xmm0
  movdqu (%rdx, %rax, 4), %xmm1
  paddd  %xmm0, %xmm1
  movdqu %xmm1, (%rdi, %rax, 4)
  add    $0x4, %rax
  cmp    $0x10, %rax
  jne    1b
  retq
```

Avoiding Branch Delay

Use predicated Instructions

```
test:
  cmp     r0, #0
  addne   r2, r1, r2
  subeq   r2, r1, r2
  addne   r1, r1, r3
  subeq   r1, r1, r3
  bx      lr
```

```
test:
  cmp     r0, #0
  bne     2f
  sub     r2, r1, r2
  sub     r1, r1, r3
1:
  bx      lr
2:
  add     r2, r1, r2
  add     r1, r1, r3
  b       1b
```

Function Inlining

Advantages

- Low calling overhead
  - Avoids branch delay!

```
test_and_set:
  mov     $0x1, %eax
  xchg    %eax, (%rdi)
  retq

acquire:
  mov     %rdi, %rdx
1:
  mov     %rdx, %rdi
  callq   test_and_set
  test    %eax, %eax
  jne     1b
  retq
```

```
acquire:
  mov     $0x1, %edx
1:
  mov     %edx, %eax
  xchg    %eax, (%rdi)
  test    %eax, %eax
  jne     1b
  retq
```

Limitations

- Not all functions can be inlined
- Code size explosion
- May require manual intervention with, e.g. inline qualifier/function attributes.

Opportunistic Sleeping

To conserve energy, applications should aim to transition the processor and peripherals into the lowest usable power mode as soon as possible.

Similar to stop/start in cars!

Interrupts for events should wake the processor, to perform more processing.

Need to balance energy savings vs. reaction time, depending on application.

Floating Point to Fixed Point Conversion

Algorithms are developed in floating point format, using tools such as Matlab

Floating point processors and hardware are expensive!

Fixed point processors and hardware are often used in embedded systems

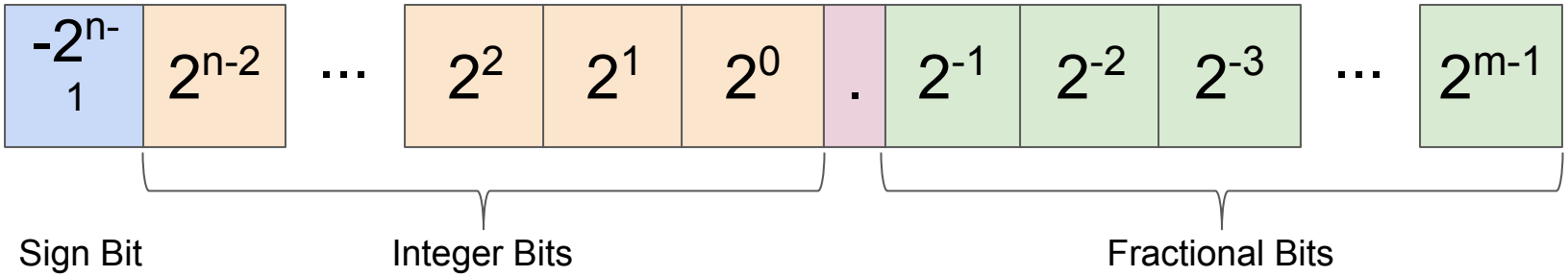
After algorithms are designed and tested, they are converted into a fixed point implementation.

Algorithms are ported onto a fixed point processor, or ASIC

Qn.m Format

Qn.m is a fixed positional number system for representing fixed point numbers

A Qn.m format binary number assumes n-bits to the left (including the sign bit), and m-bits to the right of the decimal point.



Qn.m Format

$$Q_{2.10}$$

2 bits are for the 2's complement integer part.

10 bits are for the fractional part.

Conversion to Qn.m

1. Define the total number of bits to represent a Qn.m number
   e.g. 9 bits

b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

2. Fix location of decimal point, based on the value of the number.
   e.g. assume 5 bits for the integer portion

$-b_4 2^4$	$b_3 2^3$	$b_2 2^2$	$b_1 2^1$	$b_0 2^0$	.	$b_{-1} 2^{-1}$	$b_{-2} 2^{-2}$	$b_{-3} 2^{-3}$	$b_{-4} 2^{-4}$

Example - $Q_{5.4}$

-2^4	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}
0	1	0	1	0	.	1	0	1	1
0	8	0	2	0	.	.5	0	.125	.0625

$2^3 + 2^1 + 2^{-1} + 2^{-3} + 2^{-4} = 10.6875$

Hexadecimal representation: 0xab

A 9-bit $Q_{5.4}$ fixed point number covers **-16** to **+15.9375**
Increasing the fractional bits, increases the precision

Range Determination for Qn.m Format

Run simulations for all input sets

Observe ranges of values for all variables

Note minimum + maximum value each variable sees, for Qn.m range determination