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

```python
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)

```python
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

\[ \begin{align*} a_0 &+ a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + a_8 + a_9 + a_{10} + a_{11} + a_{12} + a_{13} + a_{14} + a_{15} = c_0 + c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + c_{10} + c_{11} + c_{12} + c_{13} + c_{14} + c_{15} \end{align*} \]
Vectorisation: Vector Operations

<table>
<thead>
<tr>
<th>a₀</th>
<th>a₁</th>
<th>a₂</th>
<th>a₃</th>
<th>a₄</th>
<th>a₅</th>
<th>a₆</th>
<th>a₇</th>
<th>a₈</th>
<th>a₉</th>
<th>a₁₀</th>
<th>a₁₁</th>
<th>a₁₂</th>
<th>a₁₃</th>
<th>a₁₄</th>
<th>a₁₅</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

+ + + +

| b₀ | b₁ | b₂ | b₃ | b₄ | b₅ | b₆ | b₇ | b₈ | b₉ | b₁₀ | b₁₁ | b₁₂ | b₁₃ | b₁₄ | b₁₅ |
|----|----|----|----|----|----|----|----|----|----|------|------|------|------|------|
|    |    |    |    |    |    |    |    |    |    |      |      |      |      |      |

= = = =

<table>
<thead>
<tr>
<th>c₀</th>
<th>c₁</th>
<th>c₂</th>
<th>c₃</th>
<th>c₄</th>
<th>c₅</th>
<th>c₆</th>
<th>c₇</th>
<th>c₈</th>
<th>c₉</th>
<th>c₁₀</th>
<th>c₁₁</th>
<th>c₁₂</th>
<th>c₁₃</th>
<th>c₁₄</th>
<th>c₁₅</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Vector width: 4
### Vectorisation

Exploit parallel data operations in instructions: \( c[i] = a[i] + b[i]; \quad 0 \leq i < 16 \)

#### no_vectorisation:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xor %eax, %eax</td>
<td>1: xor %eax, %eax</td>
</tr>
<tr>
<td>mov (%rdx, %rax, 1), %ecx</td>
<td>add (%rsi, %rax, 1), %ecx</td>
</tr>
<tr>
<td>mov %ecx, (%rdi, %rax, 1)</td>
<td>add $0x4, %rax</td>
</tr>
<tr>
<td>cmp $0x40, %rax</td>
<td>jne 1b</td>
</tr>
<tr>
<td>retq</td>
<td></td>
</tr>
</tbody>
</table>

#### with_vectorisation:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movdqa (%rsi), %xmm0</td>
<td>paddd (%rdx), %xmm0</td>
</tr>
<tr>
<td>movaps %xmm0, (%rdi)</td>
<td>movdqa 0x10(%rsi), %xmm0</td>
</tr>
<tr>
<td>paddd 0x10(%rdx), %xmm0</td>
<td>movdqa 0x20(%rsi), %xmm0</td>
</tr>
<tr>
<td>movaps %xmm0, 0x10(%rdi)</td>
<td>paddd 0x20(%rdx), %xmm0</td>
</tr>
<tr>
<td>movdqa 0x30(%rsi), %xmm0</td>
<td>movaps %xmm0, 0x30(%rdi)</td>
</tr>
<tr>
<td>paddd 0x30(%rdx), %xmm0</td>
<td>movdqa %xmm0, 0x30(%rdi)</td>
</tr>
<tr>
<td>movaps %xmm0, 0x30(%rdi)</td>
<td>retq</td>
</tr>
</tbody>
</table>

#### with_vectorisation_no_unroll:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xor %eax, %eax</td>
<td>1: movdqu (%rsi, %rax, 4), %xmm0</td>
</tr>
<tr>
<td>movdqu (%rdx, %rax, 4), %xmm1</td>
<td>paddd %xmm0, %xmm1</td>
</tr>
<tr>
<td>movdqu %xmm1, (%rdi, %rax, 4)</td>
<td>movdqu %xmm1, 0x10(%rdi)</td>
</tr>
<tr>
<td>add $0x4, %rax</td>
<td>cmp $0x10, %rax</td>
</tr>
<tr>
<td>jne 1b</td>
<td>retq</td>
</tr>
</tbody>
</table>

#### avx512:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vmovdqu32 (%rdx), %zmm0</td>
<td>vpadd %zmm0, %zmm0</td>
</tr>
<tr>
<td>vpadd %zmm0, %zmm0</td>
<td>vmovdqu32 %zmm0, (%rdi)</td>
</tr>
<tr>
<td>vzeroupper</td>
<td>retq</td>
</tr>
</tbody>
</table>
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!

Limitations

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

```assembly
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
```
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

\[
\begin{array}{cccccccc}
  b_8 & b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\
\end{array}
\]

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

\[
\begin{array}{cccccccc}
  -b_4 2^4 & b_3 2^3 & b_2 2^2 & b_1 2^1 & b_0 2^0 & . & b_{-1} 2^{-1} & b_{-2} 2^{-2} & b_{-3} 2^{-3} & b_{-4} 2^{-4} \\
\end{array}
\]
### Example - $Q_{5.4}$

<table>
<thead>
<tr>
<th>$-2^4$</th>
<th>$2^3$</th>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
<th>.</th>
<th>$2^{-1}$</th>
<th>$2^{-2}$</th>
<th>$2^{-3}$</th>
<th>$2^{-4}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>.</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>.</td>
<td>.5</td>
<td>0</td>
<td>.125</td>
<td>.0625</td>
</tr>
</tbody>
</table>

\[2^3 + 2^1 + 2^{-1} + 2^{-3} + 2^{-4} = 10.6875\]

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