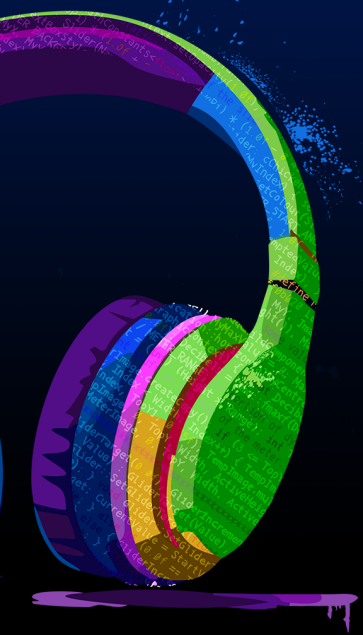


ADC24 Bristol

CRUNCHING THE SAME NUMBERS ON DIFFERENT ARCHITECTURES

MARCO DEL FIASCO

Introduction

Presentation

Embedded systems engineer
Audio, industrial and IoT
DSP, microcontrollers, SoC
Bare metal, RTOS or Linux

Slides

Outline

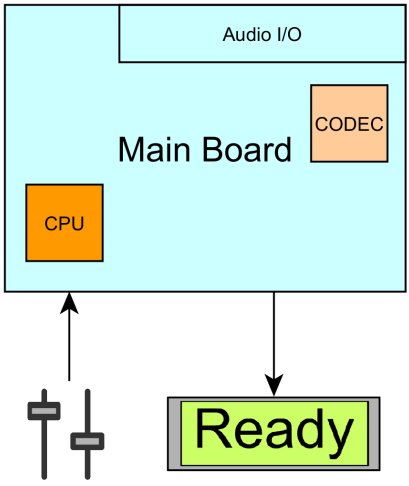
- Audio embedded systems
- CPU and memory concepts
- Architectures for audio
- Coding for DSP
- The OS
- A case study: the FIR filter

Embedded audio products

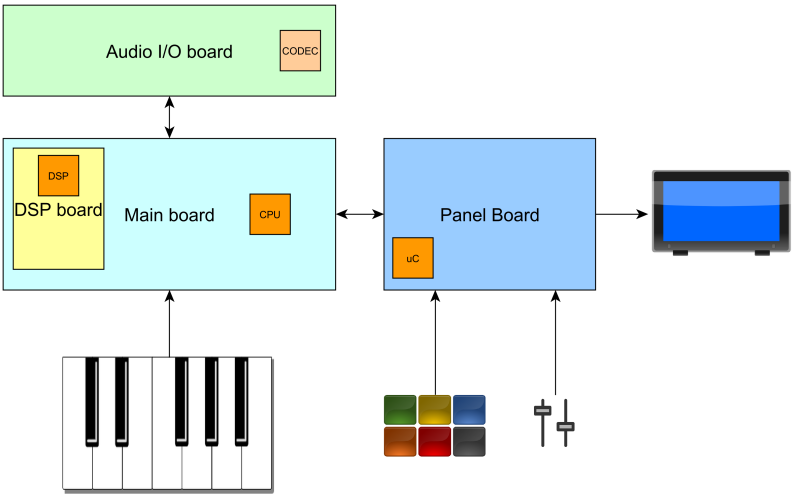
Ingredients

- UI, panel, display, buttons
- Audio I/O connectors
- Codec ADC/DAC
- CPU

Simple product example



Complex product example



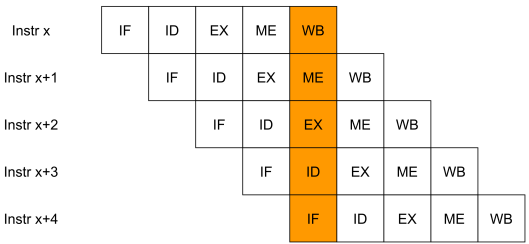
Common design challenges

- Features, BT, WIFI, nice GUI etc.
- Choose the right CPU / OS for each chip
- Inter-chip communication protocols and firmware upgrade
- Debug and testability

Simple is better: **integration**, use less chips!

CPU and memory

The pipeline

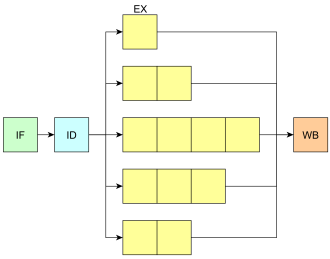


- Allows higher clock
- More cycles to complete
- Latency vs throughput
- Stalls

Pipeline length affects instruction latency: critical for recursive algorithms!

Superscalar CPU

- Fetch and decode of multiple instructions per clock cycle
- Dispatch instructions to several functional units, each with its own pipeline
- In order / out of order execution
- **Complex hardware** logic
- Compiler is **not aware** of underlying hardware

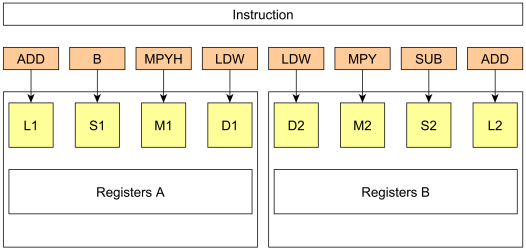


VLIW

Very **L**ong **I**nstruction **W**ord

- Same concept as superscalar but relying on compiler and not hardware
- **Simpler hardware** logic, chip area can be used for other purposes
- Compiler is **aware** of underlying hardware
- Modern DSPs are VLIW
- Hard to write machine assembly code

VLIW assembly example




```
L2: ; PIPED LOOP KERNEL

   [ B1]   SUB   .S2     B1,1,B1        ; <0,8>
||         ADD   .L2     B9,B5,B9       ; |21| <0,8> ^ sum0 += a[0] * b[0]
||         ADD   .L1     A6,A0,A0       ; |22| <0,8> ^ sum1 += a[1] * b[1]
||         MPY   .M2X    B8,A4,B9       ; |19| <1,6> a[0] * b[0]
||         MPYH  .M1X    B8,A4,A6       ; |20| <1,6> a[1] * b[1]
|| [ B0]   B     .S1     L2             ; |32| <2,4> if (!I) goto loop
|| [ B1]   LDW   .D1T1   *A3++(8),A4    ; |24| <3,2> load a[2-3] bankx+2
|| [ A1]   LDW   .D2T2   *B6++(8),B8    ; |17| <4,0> load a[0-1] bankx
```

Vector processing

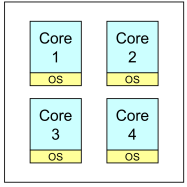
Single **I**nstruction **S**ingle **D**ata - SISD

Single **I**nstruction **M**ultiple **D**ata - SIMD

SIMD examples

- SHARC+ **2**x32 bit
- ARM NEON **4**x32 bit
- Intel AVX2 **8**x32 bit

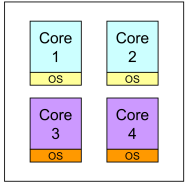
Multicore SMP



Symmetric **M**ulti **P**rocessing

- Homogeneous multicore
- Shared address space
- Same OS taking care of all the cores

Multicore AMP



Asymmetric **M**ulti **P**rocessing

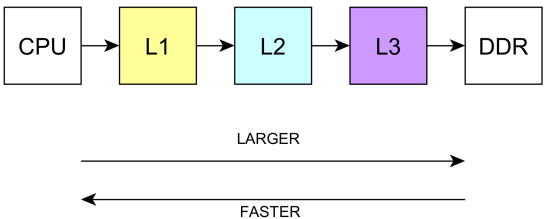
- Heterogeneous multicore (can be homogeneous)
- Different address space
- Usually running different OS, like Linux + FreeRTOS
- Examples: CPU + microcontroller, CPU + DSP, microcontroller + DSP

Why AMP

- Offloading the main OS for specific purposes
- Easier to write firmware than a Linux driver + userspace stack
- Some amount of vendor lock-in

Remember the importance of integration and using less chips?

Memory levels



Access time

- L1: 1 cycle, 16kB+
- L2: 2+ cycles, 256kB+
- L3: 20+ cycles, some MB
- DDR: 100+ cycles, some GB

L1, L2, L3

- SoC: always **caches**
- DSP & MCU: **caches** or **static RAM** for critical functions and data placement

Cache thrashing

- Happening when most of the data is not in the cache
- Generates **jitter** and **instability**

Access time of DDR about **100** times slower than L1

Cache thrashing

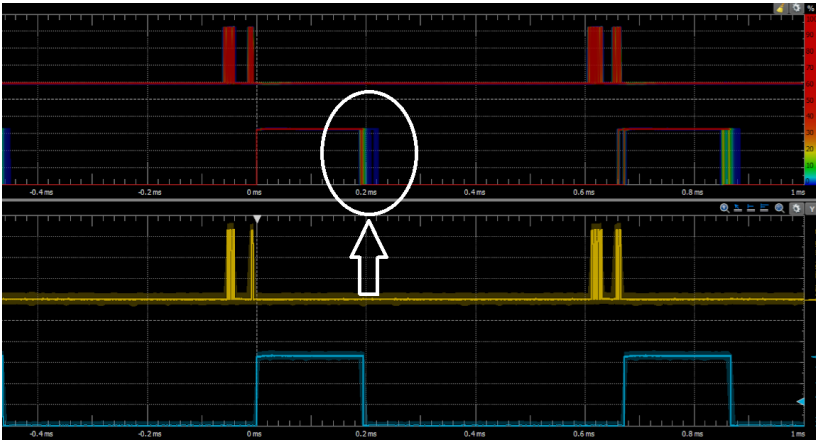
Causes

- Other processes running on same or different CPU **even of lower priority!**
- Low program or data locality

Facts

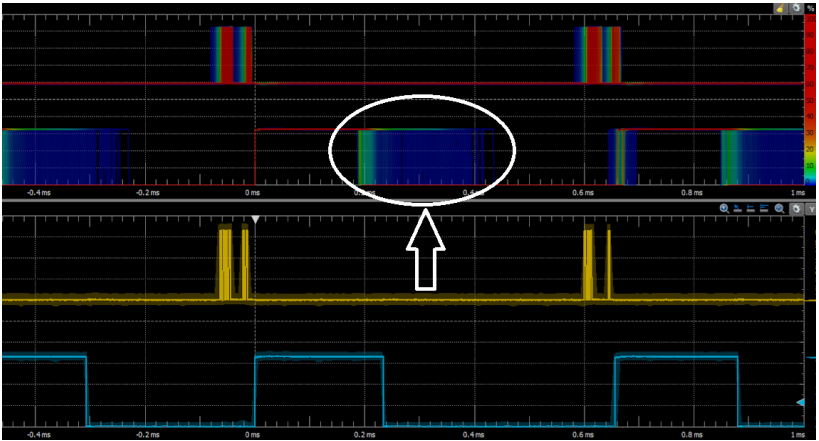
- Shared memory will make the problem worse, i.e. shared L2 cache
- Systems using cache **always** have some amount of processing jitter and instability
- Cache thrashing can be mitigated on some systems, i.e. Intel CAT

Processing jitter



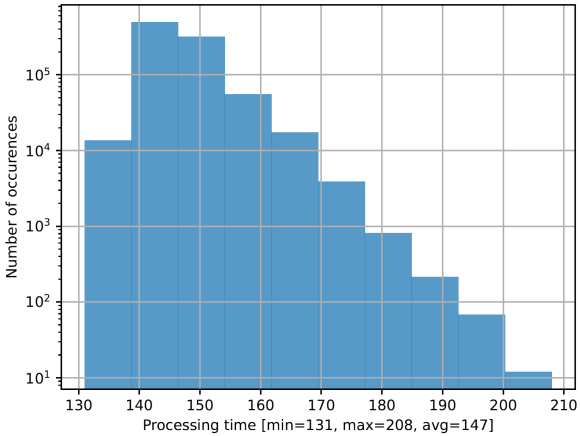
Run period with low jitter

Processing jitter



Run period with high jitter

Processing jitter



Run period histogram

Architectures for audio DSP

Some common choices

- **D**igital **S**ignal **P**rocessor - DSP
- **M**icro **C**ontroller **U**nit - MCU
- **S**ystem **o**n **C**hip - SoC

Specialized ones

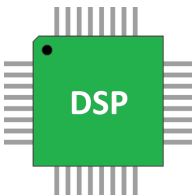
- DSP coprocessors (like Teensilica HiFi DSP)
- Low power DSPs
- XMOS
- FPGA, ASIC

The selected architectures for our test

- DSP: **ADSP21569** 1GHz **SHARC**+
- uC: **iMXRT1176** 1GHz ARM **Cortex M7**
- SoC: **BCM2711** 1.5GHz ARM **Cortex A72** (RPi4)

DSP

- Ultra low latency
- Predictable timing
- Relatively large internal memory
- Hardware accelerators
- Usually require some low level programming for DMA / accelerators
- External control CPU is usually required



The first DSP

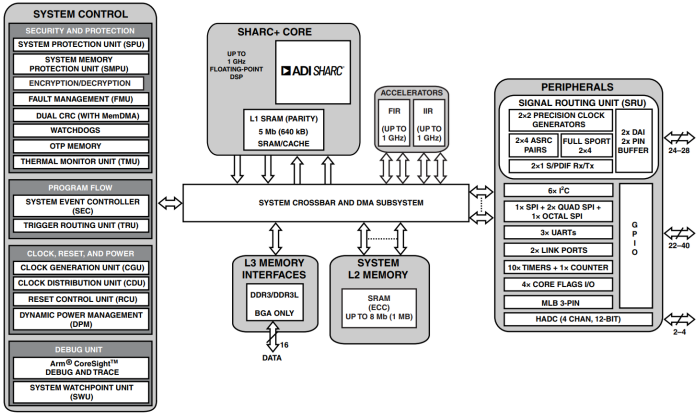
1978: The TMS5100 was the first commercial audio DSP

Speak & spell child computer

DSP examples

- Motorola 56k: 250MHz, 24 bit fixed point
- Analog Devices SHARC: 1GHz, 40 bit floating point
- Texas Instruments C6000: 1GHz, 32 bit fixed/floating point

DSP - ADSP21569 system



ADSP21569 processor (taken from ADSP21569 device data sheet)

DSP - ADSP21569 system

Key features

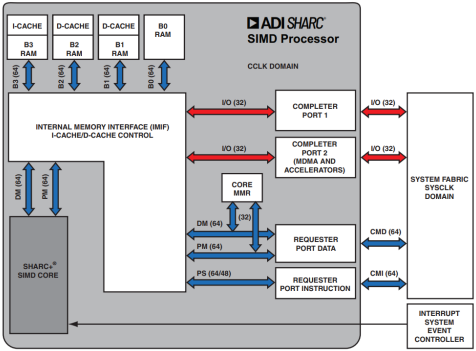
- Audio oriented peripherals: Clock generators, ASRC, SPDIF, SPORT
- General purpose peripherals: UART, I2C, SPI, timers, ADC
- Large internal memories: 640kB L1 + 1MB L2
- FIR and IIR hardware accelerators

Missing

- USB, ethernet, display

DSP - ADSP21569 processor

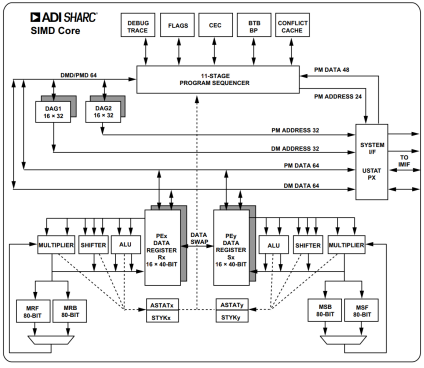
Super **H**arvard **ARC**hitecture



- 64 bit DM and PM buses
- 4 memory banks
- Optional cache

SHARC processor (taken from ADSP21569 device data sheet)

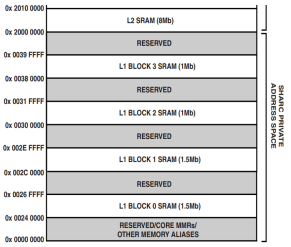
DSP - ADSP21569 processor



SHARC SIMD core (taken from ADSP21569 device data sheet)

- VLIW
- 11 stages pipeline
- 2 way SIMD

DSP - ADSP21569 internal memory

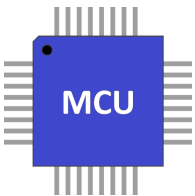


SHARC memory map (taken from ADSP21569 device data sheet)

- Memory banks allow faster and parallel load/store operations
- Banks not contiguous in memory
- Memory bank aware programming is needed!

MCU

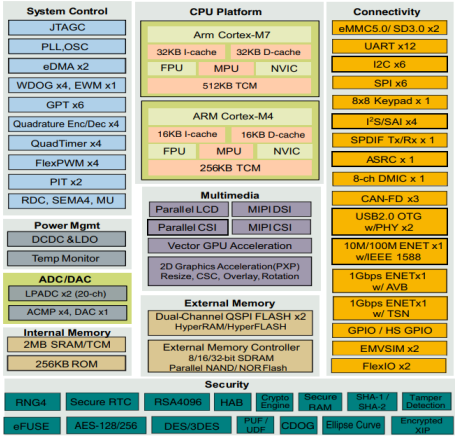
- Comprehensive set of peripherals and drivers
- Low cost, BoM savings
- Still decent amount of fast internal memory
- Low power consuption
- Suitable for lower end, single chip solutions



MCU examples

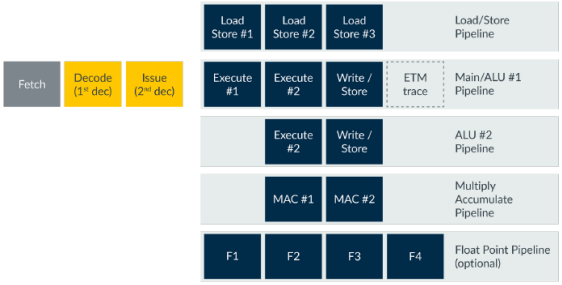
- NXP iMXRT
- ST Microelectronics STM32
- Espressif ESP32
- ...

MCU - iMXRT1170



iMXRT1170 system block diagram (taken from NXP iMXRT1170 data sheet)

MCU - Cortex M7 pipeline

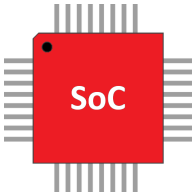


Cortex M7 pipeline (taken from Arm Cortex-M7 Processor Datasheet)

- ARMv7-M architecture
- Superscalar in-order with 6 stages pipeline
- SIMD instructions but only for 8bit and 16bit, not attractive for modern audio

SoC

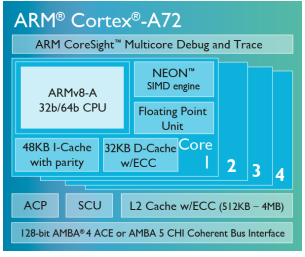
- High performance, higher clock, multicore
- Large peripheral set, including high speed buses, i.e. PCIe
- High integration, HDMI, GPU, NPU
- Suitable for high end, single chip solutions



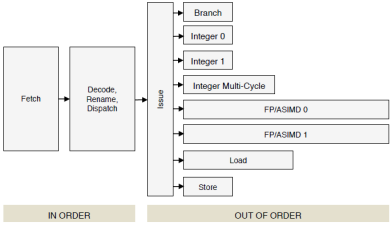
SoC examples

- NXP iMX8/iMX9
- TI AM62
- Rockchip, MediaTek and **many** others

SoC - Cortex A72 core



Cortex A72 block diagram (taken from ARM website)



Cortex A72 pipeline (taken from Cortex-A72 Software Optimization Guide)

- ARMv8-A architecture
- 128 bit NEON SIMD (4 x 32 bit)
- Per core L1, shared L2

- Superscalar 15 stages pipeline
- Out of order execution

DSP coding and debugging

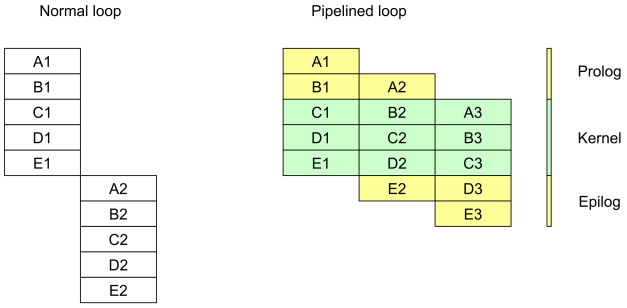
Universal rules

- Avoid branches
- If you know the loop count, tell the compiler or use templates
- Instruct the compiler of no pointer aliasing (e.g. `restrict`)
- Do not forget about locality if you have a cache

Software pipelining

- Loop optimization
- Overlap instructions from successive loop iterations
- Works better if loops are not dependent each other
- It is the key for optimization, especially for VLIW

Software pipelining



- No dependencies between successive loop iterations
- Prolog and epilog overhead, better with high loop count
- Will optimize parallel use of hardware

Loop vectorization / SIMD

- Uses machine vector instructions
- Works under specific conditions of alignment
- Usually requires `#pragma`

The compiler

- Compiler role is crucial for VLIW and SIMD
- It needs to assert certain conditions to apply optimizations like alignment and loop count
- Turning on compiler feedback will help (`-save-temps`)

The job: **compile**, read **feedback**, **adjust**, iterate

The compiler - feedback example

```
//-----------------------------------------------------------------------
//   Loop at "/home/marco/repo/dsp-benchmark/src/adsp/fir_circular.h" line 88 col 9
//-----------------------------------------------------------------------
//   This loop executes 2 iterations of the original loop in estimated 2
//   cycles.
//-----------------------------------------------------------------------
//   Unknown Trip Count
//   Successfully found modulo schedule with:
//     Initiation Interval (II)                                  = 2
//     Stage Count (SC)                                          = 3
//     MVE Unroll Factor                                         = 1
//     Minimum initiation interval due to recurrences (rec MII)  = 2
//     Minimum initiation interval due to resources (res MII)    = 2.00
//-----------------------------------------------------------------------
//   This loop's resource usage is:
//     dm dag                          used   2 out of 2 (100.0%)
//     memory access                   used   4 out of 4 (100.0%)
//     multifunction alu               used   2 out of 2 (100.0%)
//     multifunction float multiply    used   2 out of 2 (100.0%)
//     multifunction integer multiply  used   2 out of 2 (100.0%)
//     multifunction mult              used   2 out of 2 (100.0%)
//     pm dag                          used   2 out of 2 (100.0%)
//-----------------------------------------------------------------------
//   Loop was vectorized by a factor of 2.
//-----------------------------------------------------------------------
```

Coding for SIMD

Do this

```
struct iir_t {
  float b[3][SIMD];
  float a[2][SIMD];
};
```

Not this

```
struct iir_t {
  float b[3];
  float a[2];
};
struct iir_t iir[SIMD];
```

- Will make your code less human readable but will run much faster
- Optimal `SIMD` value might depend on each algorithm even on the same machine

Watch A More Intuitive Approach to Optimising Audio DSP Code

System optimizations - DSP and microcontrollers

- Proper memory placement: L1, L2, shared RAM, external memory
- If you're running out of fast memory things will get hard soon
- Advanced techniques: overlays, cache freeze

System optimizations - SoC with Linux

- Use PREEMPT_RT, supervisor or dual kernel like **XENOMAI**
- Core isolation, tasks and IRQ pinning
- Avoid using CPUs with shared L2 cache, or use cache isolation

Debugging tools

DSP and microcontrollers

- IDE
- GPIO probes, histograms, oscilloscope
- Console

SoC / Linux

- System timer
- File I/O, logging
- GDB

The OS

Baremetal OS

- Programmer relies on CPU modes for scheduling (timer, audio ISR)
- Usually a control loop + interrupt handlers

```
void audio_callback(float *in, float *out)
{
  // Process audio data
}

void main(void)
{
  setup_audio(audio_callback);
  while (1)
  {
    main_task();
  }
}
```

RTOS

Services

- Scheduling, task management
- Communication and sync services
- Memory management
- Timing and delay API
- Debugging and monitor

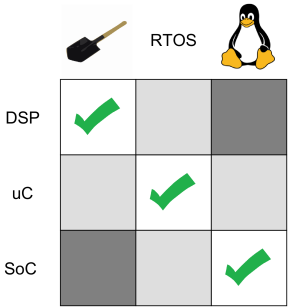
Examples

- FreeRTOS, Zephyr

Linux

- A full featured software stack
- Stable and maintained API
- System level protection
- Virtually unlimited memory
- Allows high levels of **abstraction from underlying hardware**
- Tons of ready to use drivers and libraries

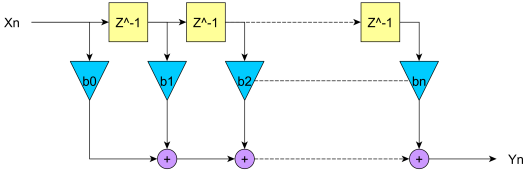
Typical choices

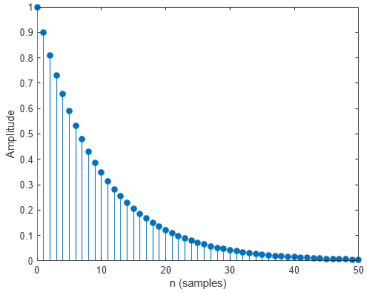


DSP $\leftrightarrow$ baremetal
Microcontrollers $\leftrightarrow$ RTOS
SoC $\leftrightarrow$ Linux

Testing with a FIR filter

The FIR filter





Why the FIR filter

- Inherently parallel
- High performance expected from SIMD
- An upper bound to the MAC / cycle
- Will stress memory and ALU

Goals

- Easy and straightforward implementation
- No assembly code
- Benchmark the architecture, single core

Naive implementation

```
// Circular buffer load (circular buffer size is fir_size + input buffer size)
memcpy(&state[fir_size - 1], input, buffer_size * sizeof(float));

for (int i = 0; i < buffer_size; i++)
{
    float sum = 0;
    float *taps = &state[fir_size - 1 + i];
    for (int j = 0; j < fir_size; j++)
    {
        sum += coeff[j] * (*taps--);
    }
    *output++ = sum;
}

// Rotate circular buffer
memmove(&state[0], &state[buffer_size], (fir_size - 1) * sizeof(float));
```

Circular buffer implementation

```
for (int i = 0; i < buffer_size; i++)
{
    state[pos++] = *input++;
    pos = pos % fir_size;

    float sum = 0;
    int taps_pos = pos;
    for (int j = 0; j < fir_size; j++)
    {
        sum += coeff[j] * state[taps_pos++];
        taps_pos %= fir_size;
    }
    *output++ = sum;
}
```

- Load and rotate the circular buffer for each input sample
- Coefficients and taps arrays dot product, but taps rotate
- Expected to work good on DSP having circular buffer addressing

SHARC optimization

Tested using some compiler guidance directives

1. Using `restrict` on all pointers
2. Dual bank load using `pm` and `dm` buses
3. `all_aligned` pragmas (inform about double-word alignment)
4. `loop_count` pragmas on loops
5. Used the FIR in SHARC library

SHARC optimization - naive FIR code

```
static void fir_basic_run_dual_bank_aligned_loop_count(
    const struct fir_basic_t *fir, const float *restrict input, float *restrict output, int buffer_size)
{
  pm float *restrict coeff = (pm float *)fir->coeff;
  dm float *restrict state = (dm float *)fir->state;
  int fir_size = fir->size;

  memcpy(&state[fir_size - 1], input, buffer_size * sizeof(float));
  #pragma all_aligned
  #pragma loop_count(BUFFER_LEN_MIN, BUFFER_LEN_MAX, BUFFER_LEN_MUL)
  for (int i = 0; i < buffer_size; i++)
  {
      float sum = 0;
      float *taps = &state[fir_size - 1 + i];
      #pragma all_aligned
      #pragma loop_count(FIR_LEN_MIN, FIR_LEN_MAX, FIR_LEN_MUL)
      for (int j = 0; j < fir_size; j++)
      {
          sum += coeff[j] * (*taps--);
      }
      *output++ = sum;
  }
  memmove(&state[0], &state[buffer_size], (fir_size - 1) * sizeof(float));
}
```

SHARC optimization - circular FIR code

```
static void fir_circular_run_optimized(struct fir_circular_t *fir, const float *restrict input,
                                       float *restrict output, int buffer_size)
{
  pm float *restrict coeff = (pm float *)fir->coeff;
  dm float *restrict state = (dm float *)fir->state;
  int fir_size = fir->size;
  int pos = fir->pos;

  #pragma all_aligned
  #pragma loop_count(BUFFER_LEN_MIN, BUFFER_LEN_MAX, BUFFER_LEN_MUL)
  for (int i = 0; i < buffer_size; i++)
  {
    state[pos++] = *input++;
    pos = pos % fir_size;

    float sum = 0;
    int taps_pos = pos;
    #pragma all_aligned
    #pragma loop_count(FIR_LEN_MIN, FIR_LEN_MAX, FIR_LEN_MUL)
    for (int j = 0; j < fir_size; j++)
    {
      sum += coeff[j] * state[taps_pos++];
      taps_pos %= fir_size;
    }
    *output++ = sum;
  }
  fir->pos = pos;
}
```

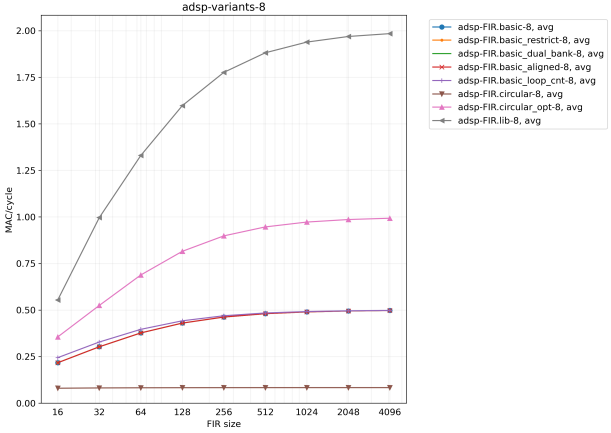
SHARC optimization - wrapping SHARC library

```
#include <filter.h>

static void fir_lib_run(struct fir_lib_t *fir, const float *input,
                        float *output, int buffer_size)
{
  firf(input, output, (const pm float *)fir->coeff, (dm float *)fir->state,
       buffer_size, fir->size);
}
```

SHARC optimization - results

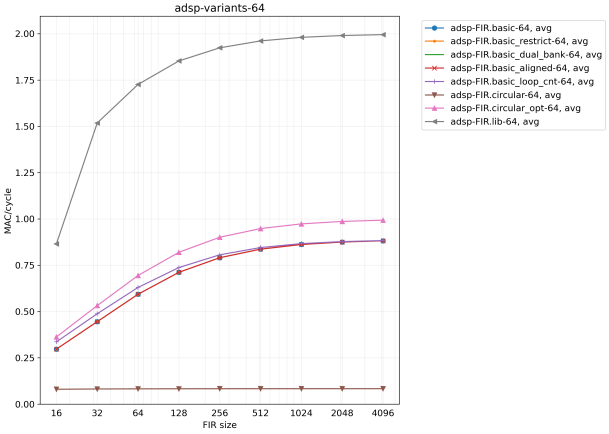
BETTER



ADSP21569 FIR variants: MAC / cycle with 8 samples buffer size

SHARC optimization - results

BETTER

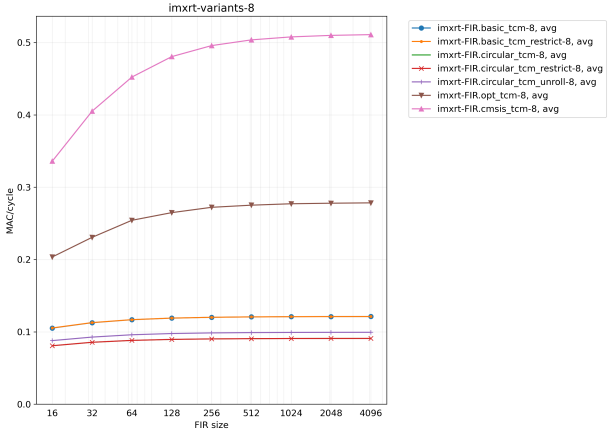


ADSP21569 FIR variants: MAC / cycle with 64 samples buffer size

iMXRT optimization

- Used `restrict`
- Manual unrolling `#pragma GCC unroll`
- Vectorization pragmas `#pragma GCC ivdep`
- Used an optimized library **CMSIS-DSP**

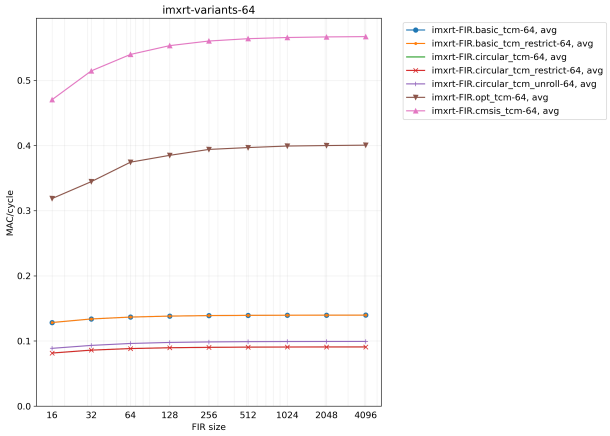
iMXRT optimization - results TCM



iMXRT1176 FIR variants: MAC / cycle with 8 samples buffer size

iMXRT optimization - results TCM

BETTER



iMXRT1176 FIR variants: MAC / cycle with 64 samples buffer size

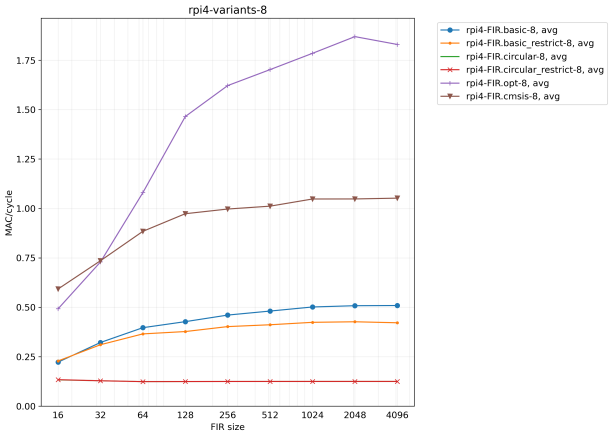
RPi4 optimization

Same techniques as for the microcontroller (it's still GCC)

- Used `restrict`
- Manual unrolling `#pragma GCC unroll`
- Vectorization pragmas `#pragma GCC ivdep`
- Used an optimized library **CMSIS-DSP**

RPi4 optimization - results

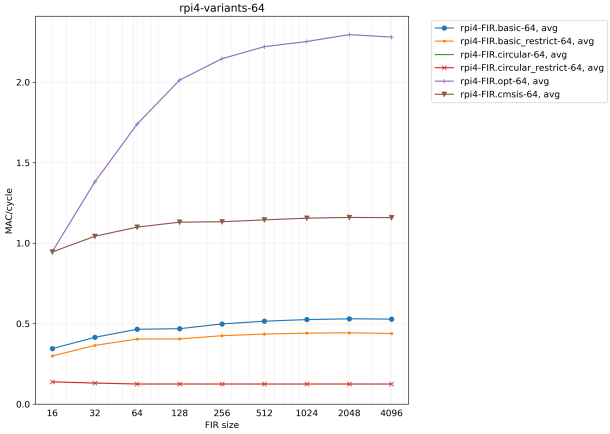
BETTER



BCM2711 FIR variants: MAC / cycle with 8 samples buffer size

RPi4 optimization - results

BETTER



BCM2711 FIR variants: MAC / cycle with 64 samples buffer size

Comparison 1 - best of

Time to **normalize the clock speed** now and see real execution time

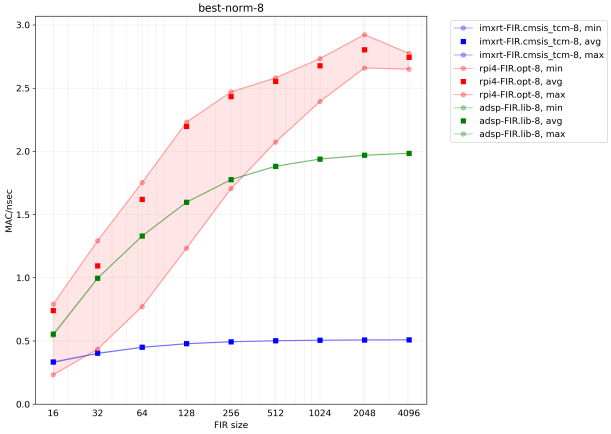
- SHARC: 1GHz
- iMXRT: 996MHz
- RPi4: 1.5GHz

Normalizing to 1GHz using **MAC / nsec** seems the most natural choice

We will compare the best of (fastest) FIR version between different architectures

Test 1 - best of comparison

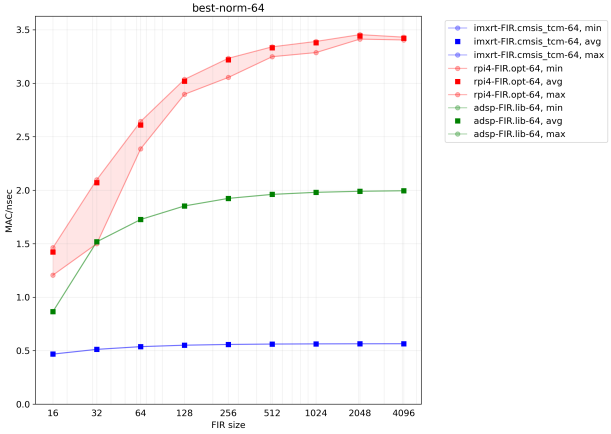
BETTER



Best of FIR comparison: MAC / nsec with 8 samples buffer size

Test 1 - best of comparison

BETTER



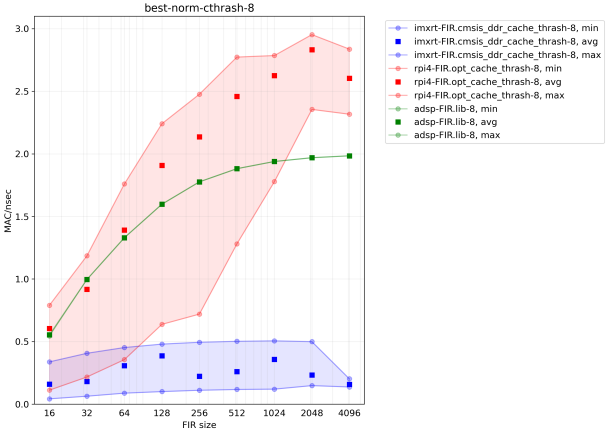
Best of FIR comparison: MAC / nsec with 64 samples buffer size

Test 2 - cache thrashing

Considerations

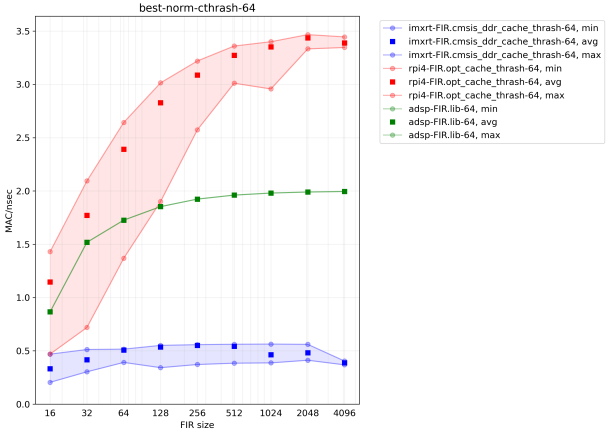
- Using SHARC as a benchmark (not running on cache)
- Added cache thrashing call after each processing loop - 256 cache lines
- Time to show external memory performance on the microcontroller

Test 2 - cache thrashing



Best of FIR with cache thrash comparison: MAC / nsec with 8 samples buffer size

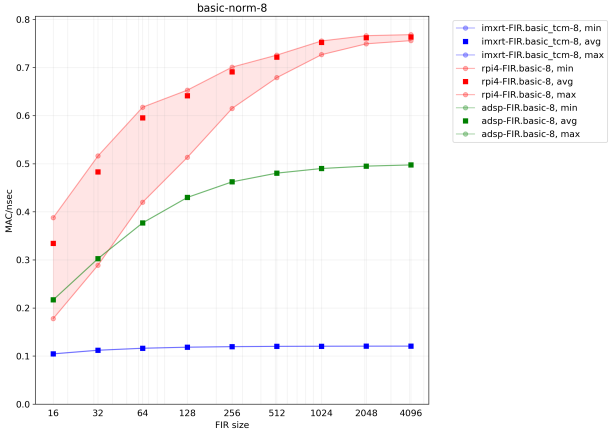
Test 2 - cache thrashing



Best of FIR with cache thrash comparison: MAC / nsec with 64 samples buffer size

Test 3 - naive comparison

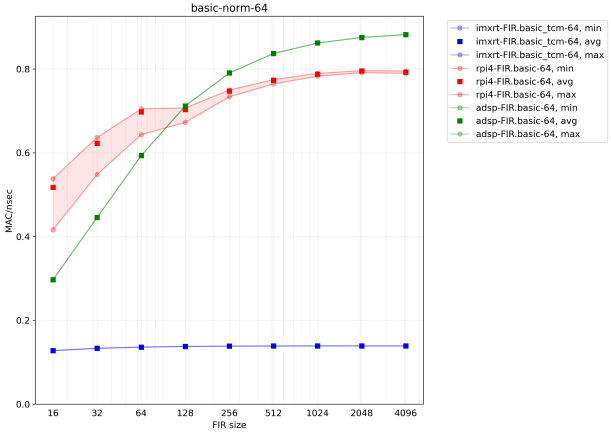
BETTER



Naive FIR comparison: MAC / nsec with 8 samples buffer size

Test 3 - naive comparison

BETTER



Naive FIR comparison: MAC / nsec with 64 samples buffer size

Is this a final statement?

No!

Tested one simple algorithm only

Recursive algorithms may behave differently

Architectures still evolving

- ARMv8-M Helium 4x32bit SIMD - Cortex M55, M85
- New SHARCs 8x32 SIMD - SHARC-FX
- SoC with higher number of cores / greater clock speed

Takeaways

Be a better DSP programmer

- Keep in mind your architecture, parallelism, cache and memory size constraints
- Try out different implementations for the algorithm first
- Put some expectations on your CPU then benchmark, monitor, optimize
- 80/20 rule

CPU selection criteria

- If you need a few samples latency and no jitter use a **DSP**
- If need high performance / multicore then use a **SoC**
- For simple and low cost designs use an **MCU**

- Consider a SoC with AMP for high levels of integration and flexibility
- Beware of inherent non predictability of cache behaviour
- Double check that internal memory is large enough for your code and data with DSP or MCU

CPU selection influencing factors

- Company background
- The internal team
- Designing for the future
- Younger minds, abstraction

Q&A

Q&A

Questions?

marcodelfiasco@gmail.com

Realtime Linux audio OS
https://www.elk.audio

Code github.com/marcodelfiasco/dsp-benchmark