Digital Signal Processors: fundamentals & system design

Lecture 3

Maria Elena Angeloletta
CERN

Topical CAS/Digital Signal Processing
Sigtuna, June 1-9, 2007
Lectures plan

Lecture 1
introduction, evolution, DSP core + peripherals

Lecture 2
DSP peripherals (cont’d), s/w dev’pment & debug.

Lecture 3
System optimisation, design & integration.

Lecture 3 - outline

Chapter 8: RT design flow – analysis & optimisation

Chapter 9: RT design flow – system design

Chapter 10: RT design flow – system integration

Chapter 11: Putting it all together ...
Chapter 8 topics

RT design flow: analysis & optimisation

8.1 Introduction
8.2 Optimiser ON
8.3 Analysis tools
8.4 Optimisation guidelines

Summary
8.1 Introduction

- Optimisation: speed, memory usage, I/O BW, power consumption.
- Steps: Debug → set optimiser ON → analyse & optimise (if needed).
- Debug & optimise: different & conflicting phases!

- Tuneable configurations.
  - Debug: debug features enabled.
  - Release: optimised (size/speed) version.
  - Custom
8.2 Optimiser ON

- Compilers very good @optimising: efficient code!
- Many optimisation phases (levels): size vs. speed.
- Power consumption often critical factor, too!
- Careful: optimiser rearranges code!
  - Assembly looks different
  - Desired action may be modified:
    - `unsigned int *ctrl;
      while (*ctrl != 0xFF);`
    - `volatile unsigned int *ctrl;
      while (*ctrl != 0xFF);`
    - **BAD!**
    - **OK**
    - Disables locally memory optimisation!
8.2 Optimiser ON [2]

- Recommended code development flow:
  - **Phase 1**: write C/C++ code.
  - **Phase 2**: optimize C/C++ code
  - **Phase 3** (if needed): code time-critical areas in **linear assembly**.
  - **Phase 4** (if needed): code time-critical areas by **hand in assembly**.

### Comparison of programming techniques

<table>
<thead>
<tr>
<th>Source</th>
<th>Efficiency</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>C / C++</td>
<td>80-100%</td>
<td>Low</td>
</tr>
<tr>
<td>Optimising Compiler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linear ASM</td>
<td>95-100%</td>
<td>Med</td>
</tr>
<tr>
<td>Assembly Optimiser</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASM</td>
<td>100%</td>
<td>High</td>
</tr>
<tr>
<td>Hand Optimised</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
8.3 Analysis tools

- Know what to optimise! 20% of the code does 80% of the work.
- Know when to stop! → diminishing returns.
- TI tools:
  - **Compiler consultant**: recommendations to optimize performance.
  - **Cache tune**: optimizes code size vs. cycle count.
  - **Code size tune**: graphical visualisation of memory reference patterns, to identify conflict areas.

**NB**: tools limitations with h/w. Use simulator!
8.4 Optimisation guidelines

... i.e. how to write more efficient code from the start

- Make the common case fast.
- Allocate memory wisely (→ linker!) & use DMA.
- Keep pipeline full.
  - Small code may fit in internal memory
  - Software pipelining: memory has edges!

- Native vs. emulated data types: faster execution on native data types (h/w vs. emulated arithmetic). → **KNOW YOUR DSP !**

- Function calls: pass few parameters (if no more registers available, parameters passed via stack → slow!)
8.4 Optimisation guidelines [2]

- Data aliasing: multiple pointers may point to same data → compiler doesn’t optimise → compilation switches to state aliasing YES/NO.

- Loops:
  - Avoid function calls & control statement inside loops.
  
  ```c
  for {
    if {...} else {...}
  }

  No pipeline breaks, but bigger code.
  
  ```c
  if {
    for {...}
  } else {
    for {...}
  }

  Min/max/abs often single-cycle instructions

  ```c
  k = k - 1;
  if( k < -1) 
  k = -1;

  k = max (k-1, -1);
  
  ```

- Move operations inner → outer loops (compilers focus on inner loops)
- Keep loop code small (local repeat optimisation).
- Loop counter: int/unsigned int instead of long.
8.4 Optimisation guidelines [3]

- Time-consuming operations:
  - **division.** Often no h/w support for single-cycle division. Use *shift* when possible.
  - **cos, sin, atan:** (+ high resolution) often needed by accelerator systems!

  → **CERN LEIR LLRF:** *polynomial* instead of Taylor-expansion. Resolution comparable to VisualDSP++ *emulated double floating point but faster!*

<table>
<thead>
<tr>
<th>Function</th>
<th>CERN single precision implementation</th>
<th>VisualDSP++ single precision implementation</th>
<th>VisualDSP ++ double precision implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>cosine</td>
<td>0.25</td>
<td>0.59</td>
<td>5.5</td>
</tr>
<tr>
<td>sine</td>
<td>(for a sine/cosine couple)</td>
<td>0.59</td>
<td>5.3</td>
</tr>
<tr>
<td>atan</td>
<td>0.4125</td>
<td>1.4</td>
<td>5.6</td>
</tr>
</tbody>
</table>

**CERN LEIR LLRF:** optimised & high-resolution functions implementations.

---

8.4 Optimisation guidelines [4]

- Use libraries: optimisation done @ algorithmic level (FFT, FIR, IIR...).
  - Sometimes data format not fully IEEE compatible for speed opt.
    - ADI Blackfin BF533: IEEE-compliant vs. non IEEE-compliant library functions.

<table>
<thead>
<tr>
<th>operation</th>
<th>fast-ft [cycles]</th>
<th>IEEE-ft [cycles]</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiply</td>
<td>93</td>
<td>241</td>
<td>0.4</td>
</tr>
<tr>
<td>add</td>
<td>127</td>
<td>264</td>
<td>0.5</td>
</tr>
<tr>
<td>subtract</td>
<td>161</td>
<td>329</td>
<td>0.5</td>
</tr>
<tr>
<td>divide</td>
<td>256</td>
<td>945</td>
<td>0.3</td>
</tr>
<tr>
<td>pow</td>
<td>8158</td>
<td>17037</td>
<td>0.5</td>
</tr>
</tbody>
</table>

- Power optimisation: s/w plays big role!
  - Minimise access to off-chip memory.
  - Use power-management API (not task!).
  - RTOS can help.

Power management (PWRM) added to DSP/BIOS for 'C5x DSPs.
Chapter 8 summary

- Code optimisation: size, speed, power.

- Compiler optimisation rearranges code → turn optimisation ON after debugging!

- If compiler optimisation not enough → linear / hand-coded assembly.

- Development environment provides analysis tools: compiler consultant, cache/code size tune.

- Write efficient code from the start → optimisation guidelines.
9.1 Introduction: DSP & architecture choice.
9.2 DSP: fixed vs. floating point
9.3 DSP: benchmarking
9.4 Architecture: multi-processing option
9.5 Architecture: radiation effects
9.6 Architecture: interfaces
9.7 Code design: interrupt-driven vs. RTOS
9.8 Code design: good practices
9.9 General recommendations

Summary
9.1 Intro: DSP & architecture

- DSP choice in accelerator sector: “Power consumption” factor negligible.
  - Standardisation in laboratory.
  - Possible synergies.
  - System evolution / migration to other machines.
  - Existing know-how / tools / hardware.

- **Global** decision needed for new systems

  3 main actors: DSP, FPGA, f/end computer.

Know them to:
  - Decide which to use.
  - How to split tasks among them.

Motorola PowerPC + Altivec extension. Altivec: SIMD expansion to PowerPC.

9.1 Intro: DSP & architecture [2]

Some points to be considered:

**Which DSP:**
- Fixed-point vs. floating point
- Benchmarking

**System architecture**
- Multi-DSP/multi-core
- Radiation effects
- Interfaces

**DSP: how**
- Interrupt-driven vs. RTOS
- Good practices

(code design)
9.2 DSP: fixed vs. floating point

- Number format: influences DSP architecture

  **Fixed point:** integer arithmetic
  - 🎨 Scaling operations needed (*but* DSP features help, *ex* saturation)
  - 😊 Fast (*but* scaling operations needed...)

  **Floating point:** integer/real arithmetic
  - 😞 High power consumption & slow speed (*but* scaling NOT needed)
  - 😞 Expensive. DSP format often not fully IEEE-compliant (*speed*).
  - ☀️ High dynamic range helps many algorithms (*ex*.: FFT).

  **NB:** Variable gap between numbers.
  - Large numbers → large gaps; small numbers → small gaps.
9.2 DSP: fixed vs. floating point [2]

Floating point: often choice for accelerator but ... CAREFUL!

LHC BC example

- Cavities @ ~400.78 MHz.
- F out:
  - format: unsigned int (16 bits)
  - resolution 0.15 Hz
  - range: 10 kHz from 400.7819 MHz
- Single floating: number spacing > 1 @400 MHz!
- TigerSHARC: h/w single-float, emulated double → loops calculations as offset from 400.7819 MHz.

LHC beam control: zoom onto beam loops part.

9.3 DSP: benchmarking

- Performance commonly judged via metric set

<table>
<thead>
<tr>
<th>METRIC</th>
<th>UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max clock frequency</td>
<td>[MHz]</td>
</tr>
<tr>
<td>Power consumption</td>
<td>[W or W/MIPS]</td>
</tr>
<tr>
<td>Execution speed</td>
<td>[MIPS, MOPS]</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>[Mbytes/s]</td>
</tr>
<tr>
<td>Memory latency</td>
<td>[clock cycles]</td>
</tr>
</tbody>
</table>

Typical metric set & corresponding unit.

- Metrics often give peak/projected values. Difficult comparison!
  - **Clock frequency** can differ from instruction frequency.
  - **MIPS**: VLIW DSPs have simple instruction set → one instruction does less work.
  - **MOPS**: often based on MAC. Not included: control instructions & memory bottlenecks.

- Option: benchmark with kernel functions (ex: FFT, FIR, IIR...)

---

9.4 Architecture: multi-processor option

a) Multi-DSP

- Many DSPs collaborate to carry out processing.
- Essential: good application partition across DSPs.
- Inter-DSP communication channels: essential!
  - **Cluster bus**: resource sharing (ex: memory) & info broadcasting.
  - **Point-to-point bus**: direct communication among processing elements.

ADI SHARC DSP EXAMPLE

Clusters bus configuration

Point-to-point (ex: linkports) configuration

9.4 Architecture: multi-processor option [2]

b) Multi-core

- Multiple cores in same physical device: performance increase without architectural change.
  - Boosts effective performance: more gain for small core freq. increase.
  - Already-used philosophy: coprocessors (ex: Viterbi decoders).

DSP evolution: multi-core & coprocessor.

Two main flavours:
- Symmetric Multi-Processing (SMP): similar/identical DSPs.
- Asymmetric Multi-Processing (AMP): DSP + MCU.

Architecture options:
- Cores operate independently (DSP farm).
- Core interaction for task completion.
9.4 Architecture: multi-processor option [3]

b) Multi-core

- Resource partitioning:
  - @board level (like single-core case)
  - @ device level (added complexity).

Example of multi-core bus & memory hierarchy.

- Inter-core communication must be available.

- Programming complex: re-entrancy rules.
  - Needed to keep one’s core processing from corrupting data of another core’s processing.
  - Single-core follows re-entrancy rules for multitasking, too.

b) Multi-core

SMP example:
TMS320C5421 multi-core DSP
9.5 Architecture: radiation effects


- General mitigation techniques:
  - **Device level**: extra doping layers to limit substrate charge collection.
  - **Circuit level**: decoupling resistors/diodes/transistors... for SRAM.
  - **System level**:
    - Error Detection & Correction (EDAC) code...
    - Algorithm-based fault tolerance (ex: Weighted Checksum Code). Rounding errors + weights overflow → difficult with floating point!

- ADI/TI: no rad-hard DSP offered (third-party companies offer ADI/TI rad-hard versions).

- **Application example**: CERN LHC power supply controllers.
  - TI DSP 'C32 + MCU (non rad-hard).
  - EDAC for SRAM protection.
  - Watchdog to restart system if it crashes.
9.5 Architecture: interfaces

- DSP interface to define:
  - DSP-DSP
  - DSP-FPGA
  - Timing
  - DSP-Master VME (control + diagnostics)
  - DSP-daughtercards

- Don’t hard-code addresses in DSP code – use linker!
- Create data access libraries → modular approach (upgradeable!).

Remember:
good fences make good neighbours!

Digital system: typical building blocks

Digital I/O, fast data links
9.6 Code design: interrupt-driven vs. RTOS

- Fundamental choice. Depends on:
  - System complexity
  - Available resources

- **Interrupt-driven**: threads defined / triggered by interrupts.
  - Optimum resource use
  - OK for limited interrupt number.

- **RTOS-based**: RTOS manages threads + priority + trigger.
  - Some resources used by RTOS
  - Clean design + built-in checks

Interrupt-driven control & background tasks.
9.7 Code design: good practices

- Digital system *NOT* black box. Add diagnostics buffer: user-selectable / post-mortem / circular.
- Add version number to identify significant code release.
- Add check on execution duration. Essential for interrupt-driven systems.

64-bit counter incremented unconditionally @every instruction cycle.

```
ustat3 = emuclk;  // read time @execution start

CRITICAL ACTION

r0 = emuclk;      // read time @execution end
r1 = ustat3;

r1 = r0 - r1;     // calculate execution time
```

ADI SHARC: *emuclk* registers and their use.

9.8 General recommendations

- Careful with new DSPs - could be beta-versions!
- Look @ DSP anomalies list.
- Gain s/w experience with development environment & simulator.
  - ADI & TI give 90-days fully-functional free evaluation of their tools.
- Gain s/w + h/w experience with eval. boards: easy prototyping.
  - Helps solving technical uncertainties!

TI C6713 DSK: picture & block diagram.
Chapter 9 summary

- DSP & architecture choice: influenced by many factors.
- Which DSP:
  - Fixed-point vs. floating point: LHC BC example
  - Benchmarking: careful, often misleading!
- System architecture:
  - Multi-DSP / multi-core.
  - Radiation effects.
- DSP: code design
  - Interrupt-driven vs. RTOS
  - Good practices
- General recommendations:
  - β-beware, anomalies
  - Fully functional s/w evaluations
  - Use evaluation boards!
Chapter 10 topics

RT design flow: system integration

10.1 Introduction
10.2 Good practices
10.1 Introduction

- System integration = system commissioning, limited to data exchange with control infrastructure & applic. prog.

- Typically digital developer works here

- Different developers (groups) involved: Instrumentation + Controls + Operation → coordination & specification work needed.

- Possibly slow: developers (groups) have different priorities.
10.2 Good practices

- **Work in parallel**
  - Start planning all layers *asap* → do not wait for low-level completion!

- **Interfaces**
  - Clear, documented & agreed upon.
  - Useful: recipes on how to setup/interact etc.
  - Keep documents updated & on server!

- **Input checks**
  - Check input validity → warnings/alarms

- **Spare parameters (in/out)**
  - Mapped DSP ⇄ application prg.
  - Small upgrades / debug added without new iterations.

- **Code releases**
  - Save current release + description. Going back easier if troubles.

- **Code validation**
  - Define data set + procedure for sub-system validation.
Chapter 11

Putting it all together...

A digital system example:

CERN LEIR LLRF

i.e. how now you know more on DSP fundamentals &
system design than two days ago!
11. Example: CERN LEIR LLRF

- **Components:**
  - DSP: beam loops implementation
  - FPGA: fast processing + glue-logic.
  - PowerPC: LLRF management & controls interface.
  - [*chapter 9*]

- **Architecture:** interrupt-driven + multi-DSPs
  - [*chapter 9*]

- **DSP-DSP comms.:** linkports + chained DMA.
  - [*chapters 4, 3, 9*]

- **SRAM:** shared DSP-Master VME (FPGA access arbitration).
  - [*chapter 4*]

---

M. E. Angoletta, “DSP fundamentals & system design – LECTURE 3”, CAS 2007, Sigtuna  34/36

- **DSP boots** from on-board FLASH memory. [→ chapter 4]
- **Languages**: C + assembly (ISR + shadow registers). [→ chapter 6]
- **Diagnostics buffers**: four, 1024-word buffers / DSP board. User-selectable decimation & signal (~50 / DSP board) [→ chapter 9]

*CERN LEIR LLRF system*: radial position (red line) & B field (blue line) during a cycle.

*M. E. Angoletta, “DSP fundamentals & system design – LECTURE 3”, CAS 2007, Sigtuna 35/36*
DSP fundamentals & system design: summary

- DSPs born early ‘80s: evolution in h/w + s/w tools.
- DSP core architecture shaped by DSPing.
- DSP peripherals integrated & varied.
- RT design flow: s/w development.
  - Languages: assembly, C, C++, graphical. RTOS
  - Code building process: compiler, assembler, linker
- RT design flow: debugging
  - Simulation/Emulation
- RT design flow: analysis & optimisation
- RT design flow: system design & integrations