
Embedded Software in Real-Time Signal Processing Systems: Design Technologies

Presented by Xuanming Dong

For EE249

09/04/2001
Outline

• Core processors in embedded systems
• Why need new suitable design technology?
• Requirements of software compilation tools
• An architecture classification of processor cores
• Issues in software compilation
• A survey of compilation techniques
• Conclusion and Discussion
Different Types of Core Processors

- **General-purpose processors**
  - off-the-shelf programmable processors
  - low/medium production volumes

- **Application-specific instruction-set processors**
  - customize the core’s architecture and instruction-set so that the system’s cost and power dissipation can be reduced significantly
  - high production volumes

- **Parameterisable processors**
  - offering processor cores with a given basic architecture, but that are available in several versions, e.g. with different register file sizes or bus widths, or with optional functional units
A Paradigm Shift from Hardware to Software

• Late specification changes can be included in the design cycle
• It becomes easier to differentiate an existing design by adding new features to it
• The use of software facilitates the reuse of previously designed functions and the platform independence
Software: a Bottleneck in System Design

• Processor cores typically suffer from a lack of supporting tools, such as efficient software compilers or instruction-set simulators
  – in the case of general-purpose cores: a compiler and a simulator are available via the processor vendor
  – in the case of ASIPs: compiler support is normally non-existing
• Standard software compiler techniques are not well suited for the peculiar architecture of DSP processors
• Both for parameterisable processors and ASIPs, the major problem in developing a compiler is that the target architecture is not fixed beforehand
• Programming DSPs and ASIPs by hand-writing machine code leads to a low designer’s productivity. Moreover, it results in massive amounts of legacy code that cannot easily be transferred to new processors
• Finally, the lifetime of a processor is becoming increasingly short and architectural innovation has become key to successful products
Requirements of Software Compilation Tools

• Architectural retargetability
  – compilation tools must be easily adaptable to different processor architectures. This is essential to cope with the large degree of architectural variation, seen in DSPs and ASIPs

• Code quality
  – the instruction and cycle count of the compiled machine code must be comparable to solutions designed manually by experienced assembly programmers
  – a low cycle count (or high execution speed) may be essential to cope with the real-time constraints imposed on embedded systems
  – a low instruction count (or high machine code density) is especially required when the machine code program is stored on the chip, in which case it contributes to a low silicon area and power dissipation
Classify a Processor Architecture: Goal

- Characterise a given compiler (or compiler method), in terms of the classes of architectures that it can handle successfully
- Characterise a given processor, so that one can quickly find out whether suitable compiler support can be found
Classify a Processor Architecture: Parameters

- Arithmetic specialisation
- Data type
- Code type
- Instruction format
- Memory structure
- Register structure
- Control-flow capabilities
Parameters: Arithmetic Specialisation

- **DSPs**
  - use a parallel multiplier/accumulator unit to speed up the execution of correlation-like algorithms (digital filters, auto- and cross-correlation, etc.)

- **ASIPs**
  - the hardware support for a butterfly function in Viterbi decoding, encountered in ASIPs for wireless telecom.
  - the critical sections of the target algorithms (e.g. deeply nested loop bodies) can be executed in a minimal number of machine cycles and without excessive storage of intermediate values
Parameters: Data Type

- Embedded processor cores for consumer and telecom applications normally support fixed-point arithmetic only
  - the reason is that floating-point units (as occurring e.g. in many general-purpose micro-processors) require additional silicon area and dissipate more power

- In a general-purpose DSP, different fixed-point data types are typically encountered

- A comparable variety of data types can typically be found in ASIPs, where the bit-widths of functional units, busses and memories are chosen in function of the application
Parameters: Code Type

• **Data pipeline**
  – to control the operations in the data pipeline, two different mechanisms are commonly used in computer architecture
    • data-stationary coding: controls a complete sequence of operations that have to be executed on a specific data item, as it traverses the data pipeline
    • time-stationary coding: controls a complete set of operations that have to be executed in a single machine cycle

• **Instruction pipeline**
  – microcoded processors: Processors with a time-stationary code type and a single fetch/decode cycle
  – macrocoded processors: processors with multiple instruction pipeline stages, whether of time or data-stationary code type
    • Macrocoded processors may exhibit pipeline hazards
Different Code Types

(a) a pipelined data path

(b) a multiply-accumulate instruction
Parameters: Instruction Format

- **Orthogonal format**
  - consists of fixed control fields that can be set independently from each other

- **Encoded format**
  - the interpretation of the instruction bits as control fields may be different from instruction to instruction
Parameters: Memory Structure

• Memory access
  – Von Neumann architecture
    • a single memory space is used to store both data and program
  – Harvard architecture
    • data and program are accessible through separate hardware

• Addressing modes
  – Immediate, Direct, and Indirect addressing

• Operand location
  – Load-store architecture (also called register-register architecture).
    • all arithmetic operations get their operands from, and produce results in addressable registers
  – Memory-memory and memory-register architecture
    • arithmetic instructions can be specified with data memory locations as operands
Parameters: Register Structure

• Homogeneous register set
  – all registers are interchangeable

• Heterogeneous register set
  – consists of special-purpose registers
  – only serve as an operand or result register of specific instructions

• The register set of a processor can be partitioned into different register classes
  – A register class is a sub-set of the processor’s register set, that can be viewed as homogeneous from the point of view of a certain instruction’s operand or result
Parameters: Control Flow

- Many DSPs and ASIPs support standard control-flow instruction
- Branch penalties are usually small
- Zero loop overhead
- Conditionally executable arithmetic instructions
- Residually controlled
- Interrupt controller
## Classification of Existing DSP-ASIPs
(Based on Six Parameters)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic specialisation</td>
<td>Plug-in applic.-spec. unit</td>
<td>Viterbi ALU</td>
<td>Dual multipl.-accumulator</td>
<td></td>
</tr>
<tr>
<td>Data type</td>
<td>Fixed point</td>
<td>Fixed point</td>
<td>Fixed point</td>
<td>Fixed point</td>
</tr>
<tr>
<td>Code type</td>
<td>Time-stationary</td>
<td>Data-stationary</td>
<td>Data-stationary</td>
<td>Time-stationary</td>
</tr>
<tr>
<td>Instruction format</td>
<td>Encoded</td>
<td>Encoded</td>
<td>Encoded</td>
<td>Orthogonal</td>
</tr>
<tr>
<td>Memory structure</td>
<td>Harvard with 2 data memories</td>
<td>Harvard with 2 data memories</td>
<td>Harvard with 2 data memories</td>
<td>Harvard with 4 data memories</td>
</tr>
<tr>
<td></td>
<td>Load-store</td>
<td>Memory-reg.</td>
<td>Memory-reg.</td>
<td>Load-store</td>
</tr>
<tr>
<td></td>
<td>Address. modes w/ post-modification</td>
<td>Address. modes w/ post-modification</td>
<td>Address. modes w/ post-modification</td>
<td>Address. modes w/ post-modification</td>
</tr>
<tr>
<td>Register structure</td>
<td>Heterogeneous</td>
<td>Heterogeneous</td>
<td>Heterogeneous</td>
<td>Heterogeneous</td>
</tr>
</tbody>
</table>
# Scope of Retargetability of the Chess Compiler

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Supported values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data type</strong></td>
<td>Fixed &amp; floating point</td>
</tr>
<tr>
<td></td>
<td>Standard &amp; User-defined data types</td>
</tr>
<tr>
<td><strong>Code type</strong></td>
<td>Time-stationary</td>
</tr>
<tr>
<td><strong>Instruction format</strong></td>
<td>Encoded &amp; Orthogonal</td>
</tr>
<tr>
<td><strong>Memory structure</strong></td>
<td>Harvard, multiple data memories</td>
</tr>
<tr>
<td></td>
<td>Load-store</td>
</tr>
<tr>
<td></td>
<td>Addressing modes with post-modification</td>
</tr>
<tr>
<td><strong>Register structure</strong></td>
<td>Heterogeneous &amp; Homogeneous</td>
</tr>
<tr>
<td><strong>Control flow</strong></td>
<td>Zero-overhead loops</td>
</tr>
<tr>
<td></td>
<td>Residual control...</td>
</tr>
</tbody>
</table>
Evolution of Retargetable Compiler Research in the Past Decades

1970:
- Software compilers (CISC)
  - Code generation (dynamic programming)
  - Code selection (LR parsing)
  - Code selection (combiner)
  - Register allocation (colouring)
  - Code selection (tree automata)
  - Scheduling (trace)
  - Scheduling (software pipelining)
  - Models for retargetability
  - Phase coupling
  - Register allocation (heterog. reg.)

1980:
- Software compilers (RISC, VLIW, superscalar)
  - Compaction (list)
  - Code selection (heuristic)
  - Allocation (clique covering)
  - Scheduling (list)
  - Scheduling (force-directed)
  - Allocation (regularity-based)

1990:
- Micro-programming
  - High-level synthesis (ASIC)
  - Retargetable compilation (embedded processors)
Anatomy of a Software Compiler

Data base

Algorithmic specification

Processor specification

FRONT END

DATA FLOW ANALYSIS

REFINEMENT & PROCESSOR-INDEPENDENT OPTIMISATIONS

CODE SELECTION

REGISTER ALLOCATION

COMPACTION

CODE EMISSION

HIGH-LEVEL OPTIMISER

BACK END CODE GENERATOR

Machine code
Compilation Techniques: Processor Specification Languages

• Netlist-based languages
  – describe the processor as a netlist of hardware building blocks, including data path, memories, instruction decoder, and controller

• High-level languages
  – contains a structural skeleton of the processor (essentially a declaration of storage elements and data types), and a description of the actual instruction set
Compilation Techniques: Processor Models

- **Template pattern bases**
  - by means of a template pattern base
  - Each partial instruction is represented as a pattern, expressed by means of the algorithm intermediate representation

- **Graph models**
  - represent the processor by means of a graph model
Part of a Tree Pattern Base ( Derived for the ADSP-21xx Instruction Format)

Above each tree the corresponding grammar representation is shown
Compilation Techniques: Code Selection

• Dynamic programming
  – based on a stepwise partitioning of the code selection problem, using dynamic programming.
  – Two phases: Tree pattern matching and Tree covering

• Left-right (LR) parsing
  – parse the subject tree using the specified regular tree grammar

• Graph matching
  – pattern matching algorithms that directly support DAG structures.

• Bundling
  – the required patterns are constructed on the fly during a traversal of the intermediate representation.

• Rule-driven code selection
  – a set of rules is provided in a well-structured programming environment which guides each transformation.
Code Selection for a Symmetrical Filter
(Using the Tree Pattern Base)

(a) CDFG of the application
(b) Tree-structured intermediate representation, with a possible cover
Code Selection Using a Bundling Approach

(a) CDFG
(b) Initial mappings of CDFG on ISG vertices
(c) Construction of bundles
Compilation Techniques: Register Allocation

• **Graph colouring**
  – the execution order determines a live range
  – based on these live ranges, an interference graph is constructed
  – Register allocation then is equivalent to finding an acceptable vertex colouring of the interference graph, using at most $N$ colors

• **Data routing**
  – most practical processors have a heterogeneous register structure
  – To transfer data between functional units via intermediate registers, specific routes may have to be followed
  – An efficient mechanism for phase coupling between register allocation and scheduling becomes essential
Register Allocation Based on Graph Colouring

<table>
<thead>
<tr>
<th>Value</th>
<th>Live range</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>__________</td>
</tr>
<tr>
<td>$v_2$</td>
<td>__________</td>
</tr>
<tr>
<td>$v_3$</td>
<td>__________</td>
</tr>
<tr>
<td>$v_4$</td>
<td>__________</td>
</tr>
<tr>
<td>$v_5$</td>
<td>__________</td>
</tr>
<tr>
<td>$v_6$</td>
<td>__________</td>
</tr>
</tbody>
</table>

(a)

Live ranges, displayed on a time axis

Interference graph
Three Alternative Register Allocations for the Multiplication Operand in the Symmetrical FIR filter.

(a) Storage in AR

(b) Storage in AR followed by MX

(c) Spilling to data memory DM
Compilation Techniques: Memory Allocation and Address Generation

- Typically a pointer is maintained to the stack frame
- DSP processors and ASIPs typically have specialised address generation units which support address modifications in parallel with normal arithmetic operations
- By pointer post-modification, the address pointer can be updated without an instruction cycle penalty
Compilation Techniques: Scheduling

• DSPs and ASIPs require a high degree of instruction-level parallelism and high code quality

• Local versus global scheduling
  – local scheduler: a scheduler that operates at the level of basic blocks (i.e. linear sequences of code without branching) in the intermediate representation
  – global scheduling: partial instructions can be moved across basic block boundaries. These moves are also termed code motions
Several Important Types of Code Motions (I)

• In the presence of conditional branches
  – A useful code motion: moves instructions across a complete conditional branch
  – Speculative execution: a conditional instruction will be executed unconditionally
  – Copy up and copy down motions result in code duplication into conditional blocks
  – Code hoisting means that identical instructions in mutually exclusive conditional branches are merged and executed unconditionally

• In the presence of iterators
  – Loop unrolling: a standard transformation whereby consecutive iterations of a loop are scheduled as a large basic block
  – Software pipelining: a transformation that restructures the loop, by moving operations from one loop iteration to another
Several Important Types of Code Motions (II)
Conclusion and Discussion

- Retargetable software compilation for embedded processors is a very important design technology issue
- There are still some other important design technology issues
  - System level algorithmic optimisations
  - System partitioning and interface synthesis
  - Synthesis of real-time kernels