Application Specific Instruction Set DSP Processors

Dake Liu and Jian Wang

Abstract In this chapter, application specific instruction set processors (ASIP) for DSP applications will be introduced and discussed for readers who want general information about ASIP technology. The introduction includes ASIP design flow, source code profiling, architecture exploration, assembly instruction set design, design of assembly language programming toolchain, firmware design, benchmarking, and microarchitecture design. Special challenges from designing multicore ASIP are discussed. Two examples, design for instruction set level acceleration of radio baseband, and design for instruction set level acceleration of image and video signal processing, are introduced.

1 Introduction

1.1 ASIP Definition

An ASIP is an Application Specific Instruction-Set Processor designed for an application domain. ASIP instruction set is specifically designed to accelerate computationally heavy and most used functions. ASIP architecture is designed to implement the assembly instruction set with minimum hardware cost. An ASIP DSP is an application specific digital signal processor for computing extensive applications such as iterative data manipulation, transformation, and matrix computing.

It is important to understand when the use of an ASIP is appropriate. If the design parameters of an ASIP are well chosen, the ASIP will contribute benefits to a group of applications. An ASIP is suitable when: (1) the assembly instruction set will not be used as an instruction set for general purpose computing and control;

D. Liu (✉) • J. Wang
Linköping University, Linköping, Sweden
e-mail: dake@isy.liu.se; jianw@isy.liu.se

S.S. Bhattacharyya et al. (eds.), *Handbook of Signal Processing Systems*,
DOI 10.1007/978-1-4614-6859-2_21,

(2) the system performance or power consumption requirements cannot be achieved by general purpose processors, (3) the volume is high or the design cost is not a sensitive parameter, and (4) it is possible to use the ASIP for multiple products.

1.2 The Difference Between ASIP and General CPU

Designers of general-purpose processors think of both the maximum performance and maximum flexibility. The instruction set must be general enough to support general applications. The compiler should offer compilation for all programs and to adapt all programmers' coding behaviors.

ASIP designers have to think about applications and cost first. Usually the primary challenges for ASIP designers are the silicon cost and power consumption. Based on the carefully specified function coverage, the goal of an ASIP design is to reach the highest performance over silicon cost, the highest performance over power consumption, and the highest performance over the design cost. The requirement on flexibility should be sufficient instead of ultimate. The performance is application specific instead of the highest one. To minimize the silicon cost, a design of an ASIP aims usually to a custom performance requirement instead of an ultimate possible high performance.

Programs running in an ASIP might be relatively short, simple, with ultra high coding efficiency, requirements on tool qualities such as the quality of code compiler could be application specific. For example, for radio baseband, the requirement on compiler may not really be mandatory.

The main difference between a general-purpose processor and an ASIP DSP is the application domain. A general-purpose processor is not designed for a specific application class so that it should be optimized based on the performance of the "Application Mix". The application mix has been formalized and specified using general benchmarks such as SPEC (standard performance evaluation corporation).

The application domain of an ASIP is usually limited to a class of specific applications, for example video/audio decoding, or digital radio baseband. The design focus of an ASIP is on specific performance and specific flexibility with low cost for solving problems in a specific domain. A general-purpose microprocessor aims for the maximum average performance instead of specific performance.

Except for memories, there were two major families of integrated digital semiconductor components, microprocessors and ASIC. Microprocessors take the role of flexible computing of all programs running in desktops or laptops. An ASIC supplies application specific functionality, and its functional coverage and performance are fixed during the hardware design time, for higher performance, lower silicon area, and lower power consumption. In 1980s to 1990s, ASIC played dominant roles in communications and consumer electronics products, for example, the radio baseband in mobile phones was based on ASIC.

To get high performance and low power consumption at the same time, embedded system designers usually use ARM processors attached with non-programmable

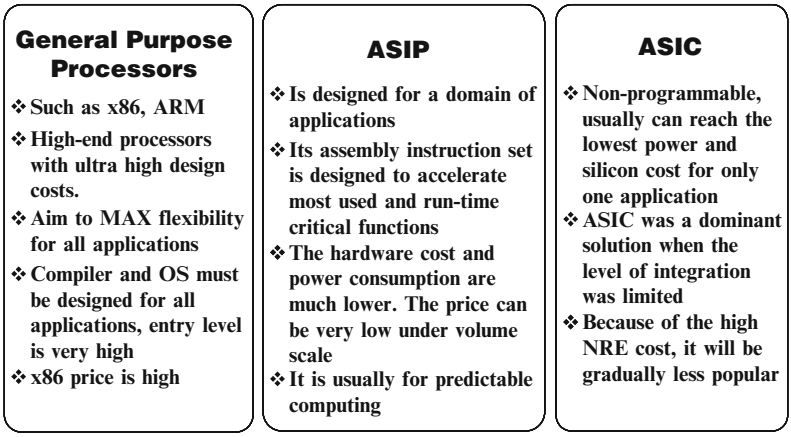

Fig. 1 Comparing ASIP with general processors and ASIC

accelerators, such as a radio baseband modem, a video motion estimator, a DCT module etc. Following trends in industry, there are emerging two problems, not sufficient flexibility and too much NRE (no-return-engineering) cost.

A high-end Smartphone needs multi radio connections such as FDD-LTE, TDD-LTE, Wi-Fi, HSPA, WCDMA, EDGE, and GSM. More links could be needed such as DVB-T and GPS. Multiple baseband modems shall be integrated to support all connections listed. There will be many non-programmable modem accelerators consuming much silicon area, and the verification time will be much longer delaying the TTM (time to market). Moreover, whatever standard modifications and market changes will end up a new silicon design and tapeout, its full mask cost of advanced silicon technologies (less than 45 nm) will be more than 3 million US dollars.

It is obviously, the ASIP, or ASIP controlled by ARM, will be the dominant technology to achieve the high performance, low silicon cost, low power consumption, and low NRE cost. ASIC solutions will therefore be gradually replaced by ASIP. Advantages of ASIP include:

- Sufficient flexibility in an application domain.
- Higher performance from the accelerated instructions for most appearing functions.
- Low power consumption and low silicon cost from domain specific architecture.
- Design time can be very short based on ASIP design tools, such as NoGAP [4].
- Redesign and modification time can be even shorter because most modifications are on software.

To summarize, there are three kinds of essential functional components in embedded systems: the general purpose processor (CPU, MCU, and general purpose DSP), the non-programmable ASIC, and ASIP. Compares were summarized in Fig. 1.

2 ASIP Design Flow for DSP Applications

ASIP DSP design can be divided into three parts, as illustrated in Fig. 2. The first part is the "ASIP architecture design" including architecture exploration, design of the assembly instruction set and the instruction set architecture. The second part is the "design of programming tools for ASIP FW (firmware) coding", including compiler, assembler, linker, and instruction set simulator. The third part is the "ASIP firmware design" including the design of benchmarking codes and application codes. The divided three parts can be identified in the ASIP design flow depicted in Fig. 3. This ASIP design flow gives a guidance of ASIP design from requirement specification down to the microarchitecture specification and design.

Following the ASIP hardware design flow in Fig. 3, the starting point of an ASIP design is to specify the requirements including application coverage, performance, and costs. The inputs of the design are product specification and the collection of underlying standards. By analyzing markets, competitors, technologies, and standards, the output of the step is of course the product requirements including the function coverage, performance, and costs.

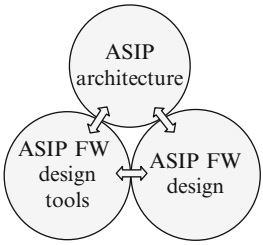

Fig. 2 Knowledge required in ASIP design

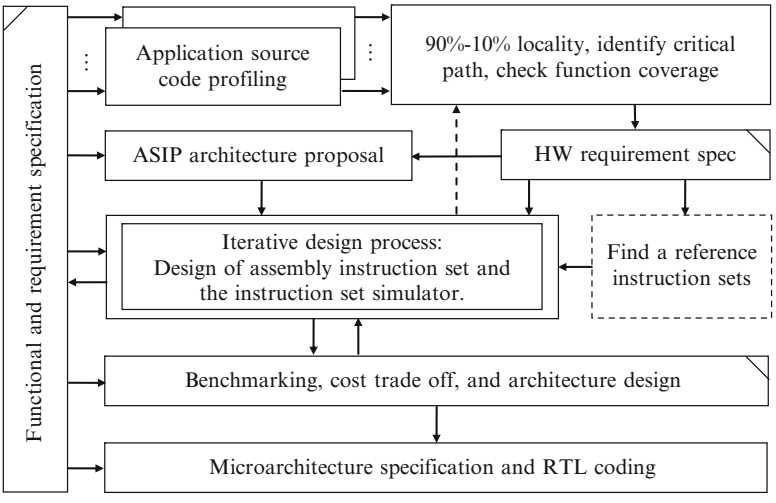

Fig. 3 DSP ASIP hardware design flow

After determining the functional coverage requirement of the ASIP, source code of applications will be collected and source code profiling will be conducted in order to further understand applications. The inputs of the source code profiling are the collected source code and application scenarios. The outputs of the source code profiling are the results of the code analysis including the estimated complexity, computing cost, data access cost, and memory cost. During source code profiling, the early partitioning of hardware and software can be proposed.

Hardware/software partitioning for an ASIP is the trade-off of function allocation to hardware or to software tasks. Implementing a task in software means to write a subroutine of the task in assembly language or in C. Implementing a task in hardware means to implement functions on an accelerated instruction or several accelerated instructions. ASIP instruction set design can be guided by the "10–90% code locality rule", which has been used as the rule of thumb in ASIP design in companies such as Coresonic [3].

The locality rule means that 10% of the instructions run at 90% time and 90% of the instructions appear only 10% of the execution time. In other words, the essential of ASIP design is to find the best instruction set architecture optimized for accelerating the 10% most frequently used instructions and to specify the rest of 90% instructions to fulfil functional coverage.

Based on the exposed 90–10% code locality, hardware requirements can be specified and a specific architecture of the ASIP can be proposed accordingly. ASIP architecture proposal could be based on a selected available architecture with modifications. If flexibility is an essential requirement, selecting an available architecture, such as VLIW architecture might be preferred. A typical case is to select VLIW or master-slave SIMD architecture for a multi-mode image and video signal processing.

An ASIP architecture could also be a dataflow architecture generated from the dataflow analysis during source code profiling. If the requirements on performance and computing latency are exceptionally tough, normal available architectures may not offer sufficient performance. A dataflow architecture might thus be a good choice. For example a dataflow architecture is suitable for a radio baseband processor in an advanced radio base station.

Following a proposed architecture, an assembly instruction set is proposed and its instruction set simulator is implemented for benchmarking. Cost trade-off analysis of the instruction set and the hardware architecture is performed. The inputs of the assembly instruction set design include the architecture proposal, profiling results of the source code, and the 90–10% code locality analysis. If a good reference instruction set can be found from a previous product or from a third party, the design of ASIP will be faster. The output of the assembly instruction set design is the assembly instruction set manual.

Instruction set design consists of the functional specification of instructions, allocation of functions to hardware, and coding of the instruction set. The coding of the instruction set includes the design of the assembly syntax and the design of the binary machine codes. Finally binary codes are executed by the instruction set simulator for benchmarking. The performance of the instruction set and the usage of each instruction is exposed for further optimization.

Before releasing an assembly instruction set, it should be evaluated via instruction set benchmarking. In processor design, benchmark is a type of program designed to measure the performance of a processor in a particular application. Benchmarking is the execution of such a program to allow processor users to evaluate the processor performance and the cost.

Benchmarking methods and benchmark scores of DSP processors can be found at BDTI [6]. More general benchmarking knowledge is available at EEMBC [7]. To support benchmarking, programming tools including compilers, assemblers, and instruction set simulators should be developed. Toolchain design and programming technique for benchmarking will be discussed later in this chapter.

After benchmarking and the instruction set release, the architecture can be designed. The inputs for architecture design are the assembly instruction set manual, the early architecture proposal, and the requirements on performance and costs. The output of the architecture level design is the architecture specification documents. The architecture design of a processor is a specification of the high level hardware modules (the datapath including RF, ALU, and MAC, the control path, bus system and memory subsystem), and interconnections between the top level hardware modules. During architecture design, function allocation to modules and scheduling of functions between modules is performed. The bandwidth and latency of busses is analyzed during the architecture design.

The inputs of microarchitecture design are the architecture specification, the assembly instruction manual, and knowledge of the selected silicon technology. The output of the processor microarchitecture design is the description of the bit accurate and cycle accurate implementation of hardware modules (such as ALU, RF, MAC, and control path) and inter-module connections for further RTL coding of the processor. As a document used in ASIP design flow, the output of microarchitecture design is the input for RTL coding of the processor.

During architecture design, functional modules are specified and the implementation of each functional module was not mentioned. It means that in the architecture level design, functional modules are usually treated as black boxes, in microarchitecture design, missing details in each module will be specified.

Microarchitecture design in general is to specify the implementation of functional modules in detail. Microarchitecture design includes function partition, allocation, connection, scheduling, and integration. Optimization of timing critical paths and minimization of power consumption will be specified during microarchitecture design.

Processor microarchitecture design, in particular, is to implement functions of each assembly instruction into involved hardware modules. It includes:

- The partitioning of each instruction into micro operations.
- The allocation of each micro operation to hardware modules.
- Scheduling of each micro operation into different pipeline stages.
- Performance optimization.
- Cost minimization.

The ASIP hardware design flow starts from the requirement specification and finishes after the microarchitecture design. The design of an ASIP is mostly based on experience. The essential goal of ASIP design flow is to minimize the cost of design iteration. Design of an ASIP is a risky and expensive business with much rewarding when the design is successfully implemented.

After specifying the microarchitecture of an ASIP, the VLSI implementation will be the same as all other silicon backend designs, and can be found from any backend VLSI book. RTL coding and silicon implementation are out of the scope of the chapter.

3 Architecture and Instruction Set Design

In this chapter, the design methodology of an ASIP will be refined. Architecture and assembly instruction set design methods will be introduced.

3.1 Code Profiling

The development of a large application may involve the experiences from hundreds of people over many years. Application engineers take years to learn and accumulate system design experience. Unlike application engineers, ASIP designers are hardware engineers who specialize on circuit specification and implementation as well the design of assembly coding tools. It is impossible that ASIP engineers can really understand all the design details of an application in a short time.

Source code profiling fills the gap between system design and hardware design. The purpose of profiling is not to understand the system design; it is rather to understand the cost of the design including dynamic behavior of code execution and the static behavior of the code structure, the hardware cost, and the runtime cost. Source code profiling is a technique to isolate the ASIP design from the design of applications.

Code profiling can be dynamic or static. Static profiling analyzes the code instead of running it, so it is data (or stimuli) independent. Static code profiling exposes the code structure and the critical (maximum, or worst case) execution time.

A static code profiler is based on a compiler frontend, the code analyzer. Code structure can be exposed in a control flow graph (CFG), the output of the code analyzer. All run time cost including computing and data access can be assigned to each branch of the graph and the total costs can be collected and estimated by accumulating the cost of each path. The critical path with the maximum execution time should be identified via source code analyzers for ASIP design.

However, a critical path may seldom or never be executed and dynamic profiling is thus needed. Dynamic profiling is to run the code by feeding selected data stimuli. Dynamic profiling can expose the statistics of the execution behavior if the stimuli

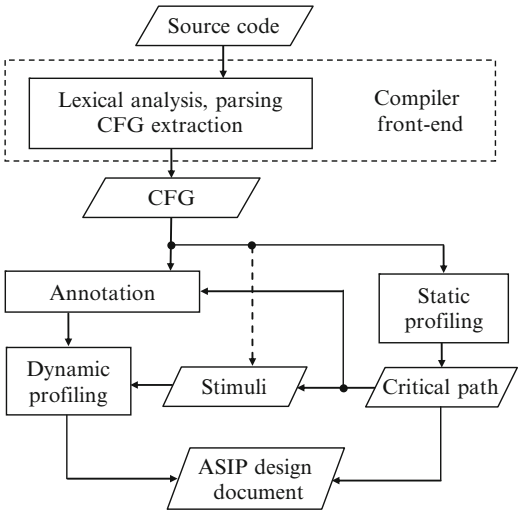

Fig. 4 Static and dynamic profiling

are correctly selected. It is usually used to identify the 90–10% code locality. By placing probes in interesting paths, statistics on critical path appearances can be achieved. Functions that are most timing critical are thus identified as inputs for designing accelerated architectures and instructions. However, dynamic profiling takes much time, so carefully selecting stimuli is an essential ASIP design skill. More detailed information on profiling techniques can be found in Chap. 6 of [1]. An illustrative view of code profiling for ASIP design is given in Fig. 4.

3.2 Using an Available Architecture

Requirements on performance and function coverage were specified during code profiling. If the required performance can be handled by an available architecture, selecting and simplifying of an available architecture will be a good idea to reduce the design complexity and costs.

Principally, there are so many different architectures to select that it is not an easy task to decide a suitable architecture for a class of applications. That is why architecture selection is mostly based on experiences.

Architecture selection can be based on architectural templates given in Fig. 5. Four dimensions: performance, complexity handling, power efficiency and silicon efficiency are taken into account in Fig. 5. Here the performance measurements include arithmetic computing, addressing computing, load store, and control. The control includes data quality control in software running on fixed point datapath hardware, hardware resource control, and program flow control. Control complexity may not be exposed during the source code profiling and it should be predicted according to design experiences.

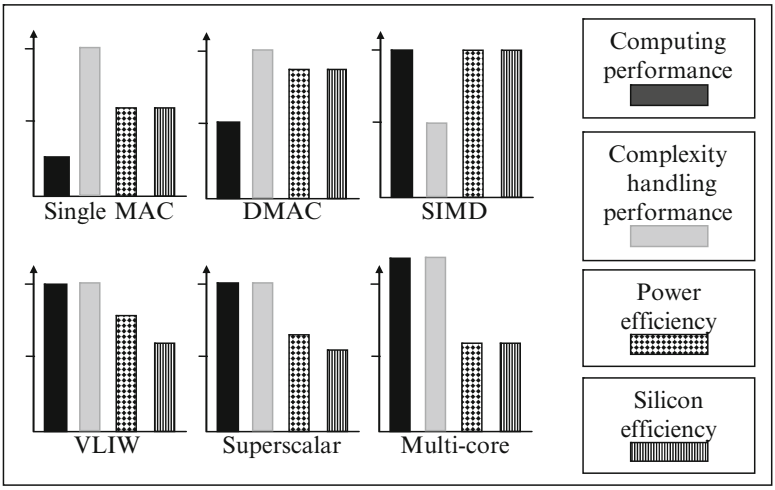

Fig. 5 Available architecture selection

Complexities can be further divided into data complexity, and control complexity. Data complexity stands for the irregularity of data, data storage, and data access. For general computing, the impact of the data irregularity can be hidden by using high-precision data types. For embedded computing a variety of data types can be used to minimize the silicon costs and power consumption.

The data complexity includes also the complexity of memory addressing and data access. Design of data storage and data access is based on the knowledge of memory subsystem (HW) and data structure (SW). Complexities of data storage and data access of DSP applications are different from the complexity of memory subsystem and the data structure of general computing. For general computing, the complexity comes from the requirements of flexible addressing, and large addressing space. The complexity is hidden by hardware cache.

For embedded DSP, the complexity of data storage and data access is induced by parallel data access for advanced algorithms. Successful handling of data complexity can give a competitive edge. Data access complexities cannot be exposed during source code profiling because the complexity of physical addressing is hidden in high level language.

Control complexity is exposed when handling program flow control and hardware resource management. On subroutine level, the control complexity is mixed up with data complexity, such as managing data dependencies. On high level in an application program, the complexity can be algorithm selection, data precision control, working mode (data rate, levels, profiles, etc.) selection, and asynchronous event handling (such as interrupt and exception handling). On top or system level, the complexity can be the hardware resource management through threading control and control for interprocessor communications. The management of control can be efficient if instructions for conditional execution, conditional branches, I/O control, and bit manipulations are carefully designed.

Power efficiency of a system is the average performance over the average power consumption. Low power ASIP design is the process of minimizing redundant operations (to minimize the dynamic power) and minimizing the circuit size (to minimize the static power). When the size of transistors shrinks, power efficiency and silicon efficiency are more related because the static power is no longer negligible and even becomes dominant.

In a modern high-end ASIP design, since the memory consumes most of the power, the design for power efficiency is almost the same as the design of efficient memory architecture. It is about optimizing the memory partition, minimizing the memory size, and minimizing memory transactions.

Silicon efficiency of a system is defined as the average performance over the ASIP silicon area. In general, as discussed previously, low power design is equivalent to designing for silicon efficiency. The on chip memory size of a silicon efficient solution is relatively small. If the on chip memory size is too small, however, the design may induce extra on-chip and off-chip data swapping and introduce extra power consumption.

Figure 5 gives architecture selection among six popular DSP processor architectures. The single MAC architecture is a typical low cost DSP processor architecture with single multiplication-and-accumulation unit in datapath. Only one iterative computing can be executed at a time in the processor. The performance may not be enough for advanced applications. The complexity handling capability can be relatively high if the instruction set is advanced. This architecture consumes the least silicon cost and power consumption is limited.

The dual MAC architecture is a kind of high performance and efficient architecture. There are two multiplication-and-accumulation units in datapath. Two iterative computations or two steps of an iterative computation can be executed at a time by Dual-MAC in the processor. By keeping the same complexity handling capability, the iterative computing performance of a Dual MAC machine can be double. Compared to the single MAC machine, the silicon cost and power consumption may be only few percentage higher, so the silicon efficiency and the power efficiency of a dual MAC machine is much better.

A SIMD (single instruction multiple data) machine can perform multiple operations while executing one instruction. The computing performance is high. However, a SIMD machine can handle data level parallel computing only when computation is regular. While handling irregular operations for complex control functions, a SIMD machine exposes its weaknesses. Power and silicon efficiency can be very high if a SIMD machine handles only regular vector data signal processing. In most systems, SIMD is a slave processor to accelerate signal processing with regular data structure. A master is used to handle complex control functions. Because control and parallel data processing are allocated to two processors, extra computing latencies induced by inter-processor communications should be taken into account.

When requirements on both the computing performance and the complexity handling are high at the same time, a VLIW (very long instruction word) machine may be a solution. A VLIW machine can execute multiple operations in one clock cycle. However, it does not have a data dependency checker in hardware. Advanced compiler technology must be available together with VLIW technology.

Because a VLIW machine handles both performance and complexity the same time. A VLIW machine is therefore popular for general purpose DSP with high throughput. However, when designing an ASIP, the application is known and a design based on master-slave SIMD or accelerators might be more efficient.

When applications are unknown and there is no sufficient compiler competence, VLIW machine architecture cannot be used. The superscalar architecture might thus be a choice. A superscalar processor can check data dependencies dynamically in hardware. At the same time, the extra hardware cost is needed for data dependency check. The hardware cost of a superscalar is therefore relatively higher. The power efficiency and silicon efficiency is relatively lower than that of VLIW. However superscalar processors offer higher performance while handling complex control. Superscalar ASIP can be found for searching based applications.

When control complexity cannot be separated from vector computing, a VLIW or a superscalar processor should be used. If control complexity can be separated from vector computing, and if there is a master processor available for handling the top-level DSP application program, a SIMD architecture is preferred to enhance the performance of vector/iterative processing. SIMD, VLIW, and superscalar all have only one control path issuing instructions, so these machines are ILP (instruction level parallel) machines.

Finally, a multicore solution is needed if the computation cannot be handled by one processor. A multicore solution introduces extra control complexities of inter-task dependence, control, and communications between processors. A multicore solution also introduces the partition induced overheads. Multicore ASIP is beyond the scope of the chapter.

A selected architecture can be a template of an ASIP. Architectural level modification and instruction level adaptation based on the selected template will be followed. The final architecture of the ASIP might be none of the architecture listed in Fig. 5.

3.3 Generating a Custom Architecture

A normal processor architecture is a control flow architecture (von Neumann architecture). A control flow architecture machine can handle only limited number of operations in one clock cycle. A data flow architecture might be introduced as an ASIP architecture to handle many operations, such as hundreds of operations, per clock cycle. In this case, optimized algorithm codes can be directly mapped or partly mapped to hardware to form a dataflow machine.

Task processing in a classical dataflow machine is controlled by a new event; it could be the system input "probe" or the data output "tag" of previous execution. The execution control of classical dataflow machine is not by program counter based FSM. However, dataflow architectures have been successfully merged into normal control flow architectures as slave machines of the main processor. In Fig. 6, a slave machine could be a function solver, a FSM driven datapath, or a programmable slave processor.

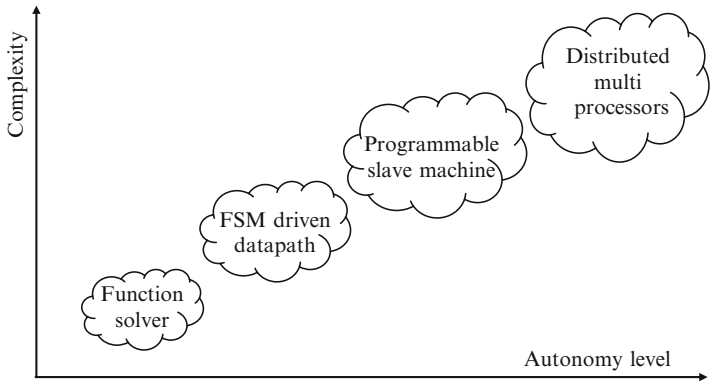

Fig. 6 Possible custom architectures

Except for data flow machines, there are other kinds of custom architectures. A function solver can be a part of datapath in an ASIP. It is a simple hardware module directly controlled by the control path of the processor. A typical function solver carries a single function such as 1/x, square root, functional module for complex data processing etc. It is usually driven by an I/O instruction or an accelerated instruction. The autonomy level of a function solver is low.

A FSM driven datapath usually offers autonomous execution of a single algorithm. The execution might be triggered by arriving data or by the master machine. This architecture is typically used to process functions requiring ultra high performance, or functions relatively independent of the master machine.

A programmable slave machine is controlled by a local programmable control path, and the autonomy level is rather high. However, task execution in a programmable slave machine is controlled by the master processor. A programmable slave machine is usually an ASIP. SIMD is a typical programmable slave machine.

Finally, a distributed system could be based on distributed ASIP processors. In this system, no one is assigned as master and no one is principally a slave. Each machine runs its own task and communications between tasks are based on message passing.

A FSM driven datapath in Fig. 7 could be a task flow machine. The task flow architecture is a typical custom architecture. It is also called function mapping hardware or algorithm mapping architecture. The architecture is the direct implementation of a DSP control flow graph. In Fig. 7, the method of hardware implementation of a control flow graph is depicted.

The left part in Fig. 7 is the behavioral task control flow graph of the DSP application. If the control flow graph is relatively simple and will not be changed through the life time of the product, the control flow graph on the left part of Fig. 7 can be implemented using HW on the right part of Fig. 7b. In Fig. 7b, the datapath is the mapping of the control flow graph in Fig. 7a. The FSM controls the execution of the DSP task in each hardware block step by step. Memories are used as the computing data buffer passing data between the hardware function blocks.

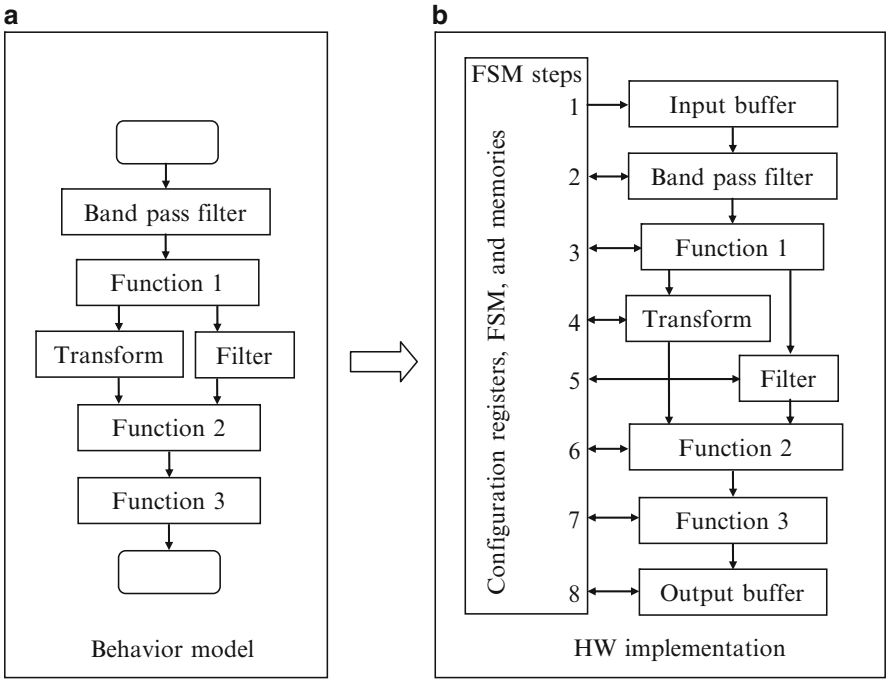

Fig. 7 Implementation of task flow architecture

The control flow graph architecture (a kind of dataflow architecture) is preferred when the complexity is not high and the data rate is very high. In a control flow graph architecture, hardware multiplexing may not be necessary, configuration of the hardware is easy, and programmability is not mandatory. The data flow architecture is mostly used for function accelerations at the algorithmic level.

Here configurability refers to the ability that the system functionality can be changed by an external device. Programmability stands for the ability of the hardware to run programs. The difference between configurability and programmability is the way the hardware is controlled; configuration control is relatively stable and it is not changed in each clock cycle and the running program changes the hardware function in each clock cycle.

4 Instruction Set Design

Different from a general purpose processor, an ASIP instruction set consists of two parts, the first part for function coverage and the second part for function acceleration. To simplify the ASIP design, the first part of the instruction set should

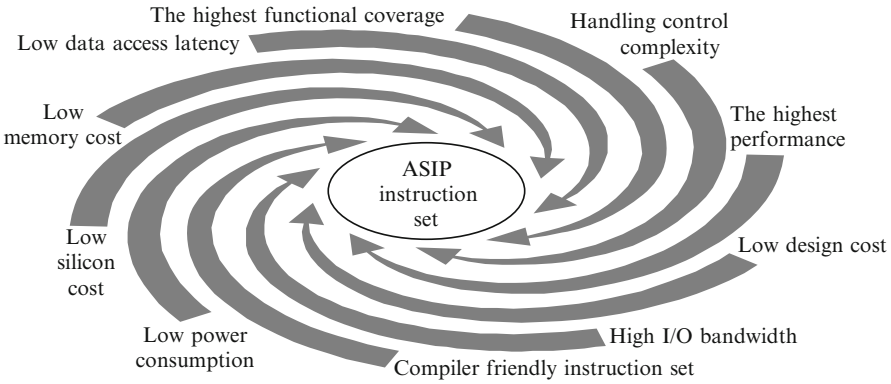

Fig. 8 Design space of an ASIP assembly instruction set

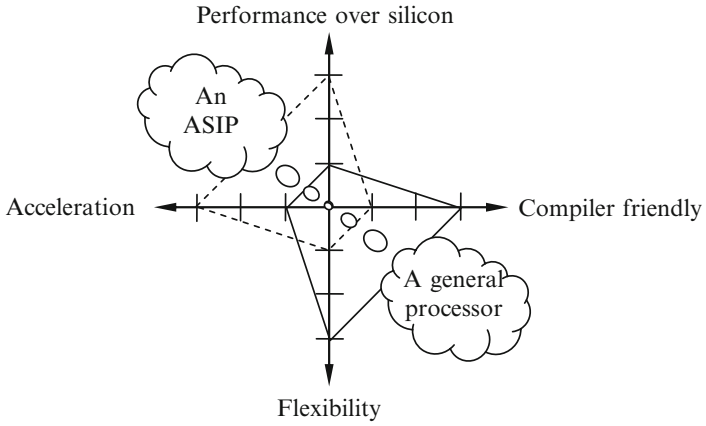

Fig. 9 Trade-off between performance and flexibility

be relatively simple. The second part of the instruction set offers both accelerations for computing extensive functions and for most frequently executed functions.

It is always expected that an assembly instruction set should be optimized for all possible applications. However, no instruction set can perfectly reach that goal. An optimization promotes some features and restrains other features. Figure 8 exposes different requirements to an ASIP instruction set. Obviously, not all can be reachable at the same time. When designing a processor, we specify the main goal and the cost constraints. Under limited power consumption and silicon cost, one can only focus on and optimize some features or parameters. At the same time, other features will be restrained. The trade-off among the four most important dimensions is illustrated in Fig. 9.

Here "Compiler friendly" means that the distance between IR (intermediate representation) and assembly language is relatively small. It also means that the

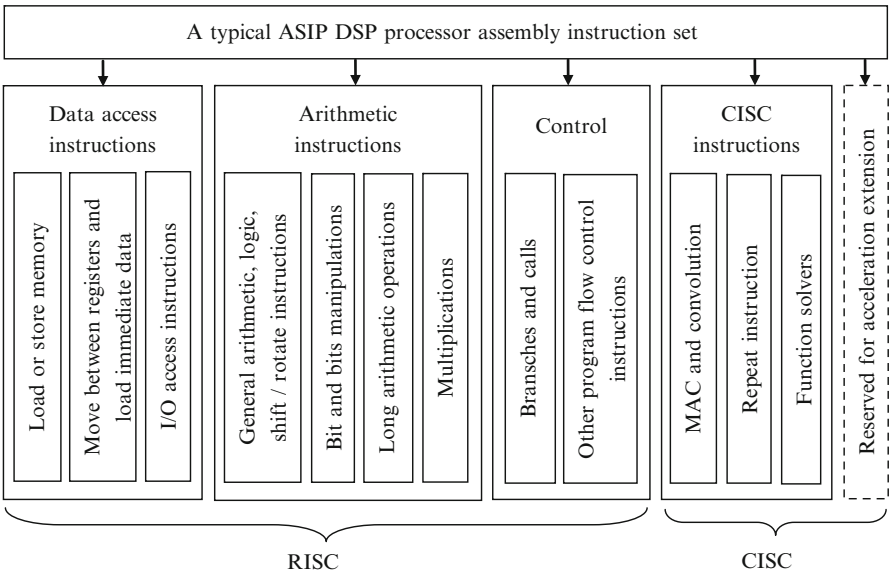

Fig. 10 Classification of an instruction set

compiler (usually a C compiler) can translate high level source code to assembly code and use the assembly instruction set in an efficient way. A "compiler friendly" instruction set is usually characterized by generality and orthogonality (for example, any instruction can use any register in the same way). Unfortunately in ASIP instruction sets may not be general enough if high performance and low power are required at the same time.

4.1 Instruction Set Specification

A DSP instruction set is usually divided into two groups, the RISC subset and the CISC subset, as illustrated in Fig. 10. All DSP processors, both ASIP DSP processors and general-purpose DSP processors, need RISC subset instructions for handling general arithmetic and control functions. The function coverage is offered by the RISC instruction subset. CISC subset instructions are used for function accelerations.

If a processor is required to supply a rather wide set of complex control functions, the RISC instruction subset could be similar to an instruction set of a general RISC processor. The RISC instruction subset is usually divided into three groups: instructions for basic load and store, instructions for basic arithmetic and logic operation and instructions for program flow control. Design of RISC instruction subset can be found from most computer architecture books.

4.2 Instructions for Functional Acceleration

The decision to introduce CISC instructions is based on 10–90% code locality analysis during source code profiling. CISC subset instructions can be divided into two categories: normal CISC instructions and instructions for accelerated extensions. The normal CISC instructions are specified in the assembly instruction set manual during the design of the ASIP core. The normal CISC instructions are completely decoded by the control path of the ASIP core.

ASIP might be designed as an SIP (Silicon Intellectual Property) for multi-products. Different products may have different requirements on function accelerations. Further instruction level function extension should be possible even after the design release of the ASIP RTL code and the programming toolchain. The method to add CISC instructions without touching the released ASIP design was discussed in detail in Chap. 17 of [1].

One group of CISC instructions are specified by merging a chain of available RISC arithmetic and load store instructions into one CISC instruction. In this way, acceleration is achieved without adding extra arithmetic hardware. The execution time is reduced to the time to run one instruction. The pipeline depth of the accelerated instruction is increased.

If the available hardware used by RISC instructions cannot support the acceleration, extra hardware should be added to support accelerated instructions. Another group of CISC instructions thus need extra hardware. By adding the first group of CISC instructions, we only increase the control complexity of the ASIP control path. By adding the second group of CISC instructions, we add complexity both to the control path and to the datapath.

A typical CISC instruction example is the MAC (multiplication and accumulation) instruction. It merges the multiplication instruction and accumulation instruction to one instruction with double pipeline depth. The execution of the instruction usually needs two clock cycles. The behavior description of the instruction is:

```
Buffer <= (Operand A) x (Operand B) //in the first clock cycle
ACR <= ACR + Buffer // in the second clock cycle
```

This is the most used arithmetic computation in DSP applications. There are opportunities to dramatically improve DSP performance by fusing several instructions into a “convolution instruction”, an iterative loop, including memory address computing, data and coefficient loading, multiplication, accumulation, and loop control. By adding the convolution instruction, both data access overheads and loop control overhead are eliminated. A convolution instruction CONV N M1(AM) M2(A) is therefore introduced. The behavior description of the convolution instruction is:

```
01   // CONV instruction: iteration from N to 1
02   OPB <= TM (A); // load filter coefficient
03   OPA <= DM (AM);     // load sample
04   if AM == BUFFER_TOP then AM <= BOTTOM // Check FIFO bound
05      else INC (AM);  // update sample pointer
06   INC (A);        // update coefficient pointer
07   BFR <= OPA * OPB;  // Save compute product in buffer
```

```
08   ACR <= ACR + BFR;  // accumulate product
09   DEC (LOOP_COUNTER);        // DEC loop counter
10   if LOOP_COUNTER != 0 then jump to 01 // one more iteration
11     else Y <= Truncate (round (ACR));    // final result
12   end.
```

The loop counter LOOP_COUNTER, the data memory address pointer AM, and the coefficient memory address pointer A must be configured by running prolog instructions. In each operation step, two memory accesses, the multiplication, and the accumulation consumes one clock cycle, N taps of convolution will consume only N+2 clock cycles. Without the special CISC instruction CONV an iterative computing of a N-tap FIR could consume up to 12N clock cycles, 6N cycles for arithmetic computing, addressing, and data access, 3N clock cycles for checking the FIFO bound, and 3N clock cycles for checking the loop counter (consuming extra two clock cycles when a jump is taken).

4.3 Instruction Coding

Coding includes assembly coding and binary machine coding. Assembly coding is to name assembly instructions and make sure that the assembly language is human-readable. Hardware machines can only read binary machine code and the assembly code must be translated to binary machine code for execution.

The purpose of assembly coding is to let the assembly code be easy to use, easy to remember, and without ambiguity. Assembly coding is usually based on suggestions from the IEEE std. 694-1985. As the lowest level hardware dependent machine language, assembly code gives function descriptions based on micro-operations. Assembly code also specifies the usage of hardware such as the arithmetic computing units, physical registers, the physical data memory addresses, and the physical target address for jump instructions. The assembly language exposes micro-operations in the datapath, addressing path and memory subsystem, some micro-operations in the control path, such as test of jump conditions and target addressing.

It is not necessary to expose all micro-operations. Some micro-operations are not exposed in the assembly manual because they are not directly used by assembly programmers, such as bus transactions and the details of instruction decoding. In Fig. 11, explicit and implicit micro operations are classified.

All micro-operations that the assembly users (programmers) need to know will be exposed in the assembly language manual. However, to efficiently use the binary machine code, some micro-operations that are specified by the assembly language manual, will not be coded into assembly and binary machine code. Examples of such operations are flag operations for ALU computing and PC=PC+1 to fetch the next instruction. When writing instruction set simulator and the RTL code, all hidden micro-operations must be exposed and implemented.

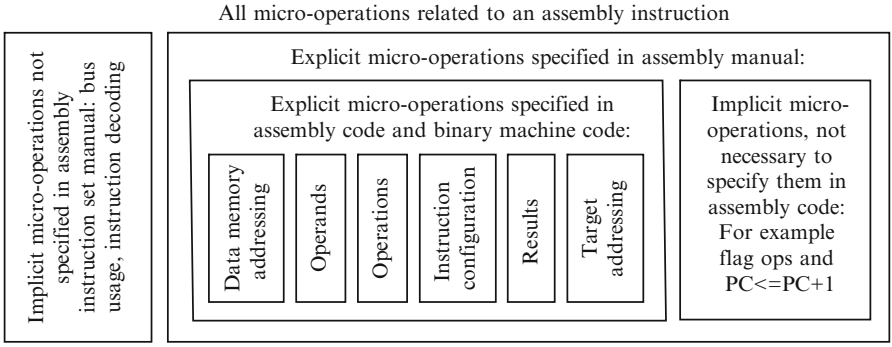

Fig. 11 Assembly and binary coding convention

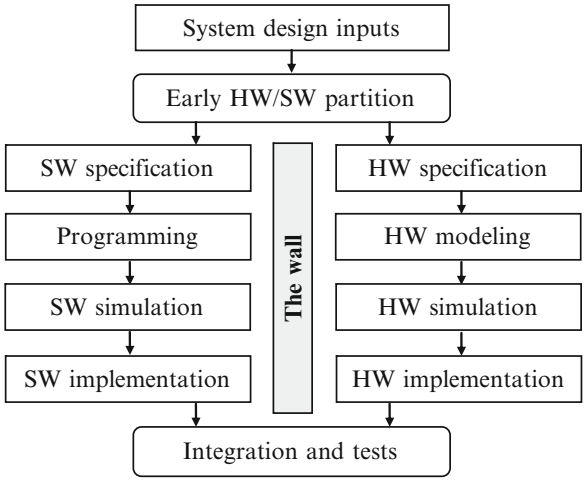

Fig. 12 Traditional embedded system design flow

4.4 Instruction and Architecture Optimization

An ASIP instruction set shall be optimized for a group of applications with cost in mind. The optimization process is so called software–hardware co-design of the instruction set architecture.

Traditional embedded system design flow is given in Fig. 12. To simplify a design, system design inputs are partitioned and assigned to the HW design team and the SW design team during the functional design. The SW design team and HW design team design their SW and HW independently without much interwork. In the SW design team, functions mapped to SW will be implemented in program codes to be executed in hardware. The implemented functions will be verified during simulations and finally the simulated programs are translated to machine language during the SW implementation.

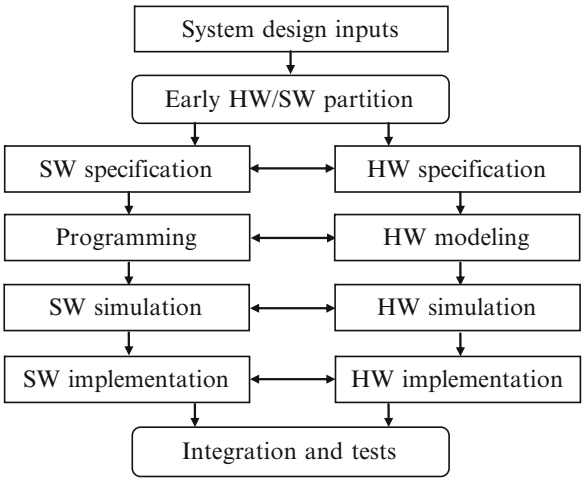

Fig. 13 Hardware/software co-design flow

From HW design perspective, functions assigned to HW team are allocated to HW modules. Modules could be either processors or functional circuits. The behaviors of programmable HW modules can be described by assembly language simulator. The behaviors of non-programmable HW modules can be described by HW description languages. A function implemented in programmable hardware will be an assembly language instruction. Functions implemented in non-programmable hardware will be hardware modules or parts of a hardware module.

Finally, the implemented SW and HW is integrated. The implemented binary code of the assembly programs will be executed on the implemented HW. The design flow in Fig. 12 is principally correct because it follows the golden rule of design: divide-and-conquer. However, when designing a high quality embedded system, we actually do not know how to make a correct or optimized early partition. In other words, the early partition may not be good enough without iterative optimizations through the design.

To optimize a design, we need to interactively move functions between SW and HW design teams during the embedded system design. Under such a challenge, "HW/SW co-design" appeared in the early 1990s.

The HW/SW co-design flow is depicted in Fig. 13. Following the figure, the idea of the new design flow is to optimize the partition of HW and SW functions cooperatively at each design step during the embedded system design. HW/SW co-design is to trade-off function partition and implementation between SW and HW through embedded system design. Eventually, the results will be optimized following certain goals. Re-partition is not difficult. The difficulty is the fast and quantitative modeling for functional verification as well as performance and cost estimation of the new design after each re-partition. Fast processor prototyping is therefore a challenging research topic [4].

The implementation of an algorithm level function in ASIP HW is to design accelerated instructions for the specific function. Implementing a function in SW

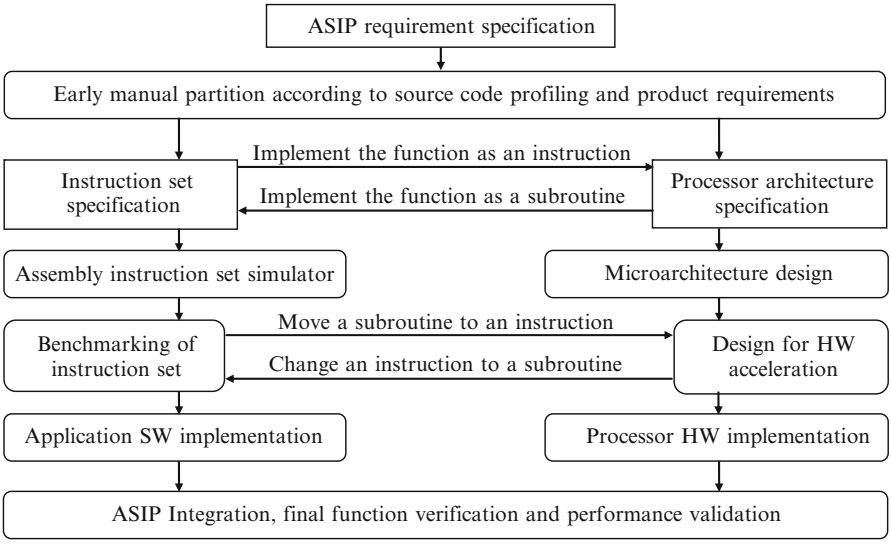

Fig. 14 Hardware/software co-design for an ASIP

means to design software subroutine of the function. After source code profiling, an instruction set architecture can be proposed. Further HW/SW co-design of ASIP in Fig. 14 is to modify the proposal by trading off HW/SW partition through all design steps of an ASIP.

An ASIP assembly instruction set should be efficient enough to offer performance for the most frequently appearing functions and be general enough to cover all functions in the specified application domains. When the performance is not enough, the most used (kernel) subroutines or algorithms should be hardware-accelerated. Those subroutines or algorithms that are seldom used should be implemented as SW subroutines. Performance and coverage can be fulfilled by carefully trading off the HW/SW partitioning.

However, functional coverage evaluation is not easy. So far, there is no better way than running real applications. The cost is high and it takes a long time. HW/SW co-design flow in Fig. 14 is part of the design flow in Fig. 3.

4.5 Design Automation of an Instruction Set Architecture

Researchers are working towards multi-objective-oriented evaluation tools to compare and select among different solutions. Many researchers also work on identification of functions to be accelerated and propose instructions for functional acceleration. Also many researchers are working on ASIP synthesis.

There are two main approaches in research of ASIP synthesis. One is to synthesis an ASIP based on fixed template, and the other one is to synthesis an ASIP without template, only based on architecture description language (ADL) [4]. ASIP will have a bright future when it can be synthesized with or without a template architecture. In such a case, quantitative requirements including performance, flexibility, silicon costs, power consumption, and reliability can be reached. However, there are marketing, and project dependent problems that can not yet be modeled by EDA tools, such as:

- How to model market requirements
- How to model project requirements
- How to model human competence and availability
- How to balance SW design costs and HW design costs
- How to differentiate from competitors
- How to convince customers
- How to distribute weighting factors among requirements
- How to define the product life time
- How to trade-off between costs and flexibility.

At present, design of an ASIP DSP is based on experience. There are other non-technical decision factors. For example, the life time of ASIP may not be "the longer the better", as a longer product life time may block sales of new products.

5 An ASIP Instruction Set for Radio Baseband Processors

Baseband signal in a radio system is the signal which carries the information to be transmitted and the signal is not modulated onto the carrier wave of radio frequency. A baseband signal can be analog or digital. The sub system processing on a digital baseband signal is called digital baseband or DBB. The sub system between the DBB and the RF (radio frequency) sub system is called mixed analog baseband or ABB. In this section we only discuss DBB. A DBB in radio system is presented in Fig. 15.

A DBB module includes both the transmitter and the receiver. In a transmitter, the DBB codes binary streams from application payload to symbols on the radio channel. The coding includes efficient coding (the digital modulation, to code as many bits as possible in unit bandwidth) and redundant coding (to carry information for error correction on receiver side). The digital modulation is a way to carry multiple bits to a symbol.

The receiver recovers a radio signal from the analog to digital converter (ADC) in ABB to payload bits. The received waveforms carry both transmitted signals with distortions and interferences. Interferences are from external interference sources, such as glitches and white noise, and from the received signal itself. The received signal is a combination of the wave directly transmitted and waves reflected by different objects coming to the receiver at different times, giving interference

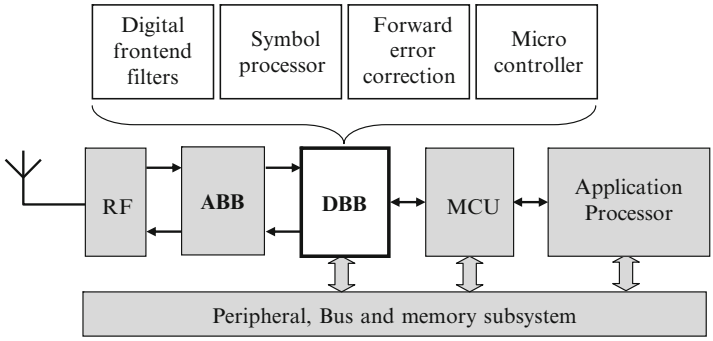

Fig. 15 A digital baseband DBB in a radio system

with each other. Signals arriving to the antenna at different times induce time domain distortion. Signals carried by different frequencies induce frequency domain distortion. Distortions and interferences must be cancelled before signal detection, which requires a pair of channel estimator and equalizer to remove the noise and distortions. Relative movement of a radio receiver and its transmitter changes the relative positions between the transmitter and the receiver. An estimated channel will not be valid as soon as the relative position is changed. If the mobility is high (e.g. a moving vehicle), the life time of a channel model will be on millisecond level, requiring recalculation of the channel in a millisecond.

To guard against glitches and white noise, forward error correction (FEC) algorithms must be executed. Computing features of advanced forward error correction algorithms include high computing load, computing based on short data precision, advanced addressing algorithms for parallel data access, and feasibilities for data and task level parallelization.

All these computing tasks, filtering signals, signal synchronizer, channel estimator, channel equalizer, signal detector, forward error correction, signal coding, and symbol shaping, need heavy computing. Moreover, radio baseband signal processing must conducted in real-time. A mobile channel should be estimated and updated within milliseconds, and all other signal processing for a receiver and transmitter must be done in each symbol period. For example, the symbol period of IEEE 802.11a/g/n is 4 μs, so several billions of arithmetic operations per second are required [2].

Some tasks have heavy requirements on computing performance and low requirements on flexibility, such as simple filtering, packet detection, and sampling rate adaptation. These tasks can be allocated to the DFE (digital frontend) module. Functions of these filters do not change in each clock cycle, thus configurability is sufficient for complexity handling of DFE. In the same way, error correction computing under acceptable low data precision can be allocated to FEC (forward error correction) modules without requiring programmability.

Table 1 Profiling and function allocation for IEEE 802.11a/g receiver

Tasks requiring high computing performance	Algorithms	Number of arithmetic operations per task	Appearance (μs)	Million arithmetic operations per second
Packet synchronization	Cross correlation	~ 3,000	~ 100	30
Sampling rate offset estimation	FFT/phase decision	2,500	100	25
F domain channel estimation	FFT, interpolation, 1/x	6,640	24	277
Payload Rotation	Vector product	~ 384	4	96
IFFT or FFT	64 points FFT/IFFT	1,984	4	496
F-domain channel equalization	Vector product	~ 384	4	96
Total cost				1,010

Table 2 Five most used task level (CISC) instructions accelerated on vector processing level

Instructions	Functional Specification
Conjugate complex convolution (auto correlation)	For I = 1 to N do {Complex REG = Complex REG + V1[i] * Conjugate of V1(or V2)[i]}
Complex convolution	For I = 1 to N do { Complex REG = Complex REG + V[i] * V2[i]}
Product of conjugate complex vectors	For I = 1 to N do {V3 [i] = V[i] * Conjugate of V2[i]}
Product of complex vectors	For I = 1 to N do {V3 [i] = V[i] * V2[i]}
Product of complex data vector and complex data scalar	For I = 1 to N do {V2 [i] = Scalar * V2[i]}

Except for DFE, FEC, and hardware control functions (to be allocated to the micro controller), the rest of the tasks are allocated to the symbol processor. Sufficient flexibility is required for signal processing on complex data. The main tasks allocated in the symbol processor include synchronization, channel estimation, time and frequency domain channel equalization, and transformation. The cost of 802.11a/g symbol signal processing in symbol processor is listed in Table 1. The required computing performances are calculated based on single precision integer data. Associated performance requirements for data access are not included in the table.

By offloading the majority of computing to DFE and FEC, the symbol processor only handles the small part of computing requiring programmability. However, the small part of the programmable computing requires more than 1 Giga arithmetic operations in a second, not yet including the data access cost. By analyzing tasks listed in Table 1, the most used ASIP task level instructions for signal processing of radio symbols are listed in Table 2.

In Table 2, V1[i], V2[i], and V3[i] are complex data vectors. If a datapath of complex data is used, one step of signal processing on complex data equals about 4–6 times of signal processing on integer data, and the performance will be

enhanced 4–6 times. By further accelerating ASIP instructions on vector level, loop overheads can be eliminated by hardware loop controller. All functions listed in Table 1 can be processed on 200 MHz in one datapath with complex data type [2].

The digital baseband of IEEE802.11a/g is relatively easy. The computing cost for LTE and WiMAX will be at least 5–10 times higher. At least 5 more datapath modules for complex data computing are needed in an ASIP for LTE and WiMAX baseband signal processing.

6 An ASIP Instruction Set for Video and Image Compression

An image or video CODEC carries-out signal processing, eliminates redundancies of image and video data, and is a kind of applications of data compression [5]. Compressed images and video can be stored and transferred with low cost. The most frequently used image and video compression methods are lossy compression to get a high compression ratio. Generally, lossy compression techniques for image and video are based on 2D (two dimensional) signal processing, as illustrated in Fig. 16.

Figure 16 shows a simplified video compression flow, and the shaded part in Fig. 16 is a simplified image compression flow. The first step in image and video compression is color transform. The three original color planes R, G, B (R=red, G=green, B=blue) are translated to the Y, U, V (Y=luminance, U and V=chrominance) color planes. The human eye has high sensitivity to luminance ("brightness") and low sensitivity to chrominance ("color"). Normal people will not notice the difference of the resolution if the color planes U and V are down-sampled to $\frac{1}{4}$ of their original size. Therefore, a frame with a size factor of 3 (3 RGB

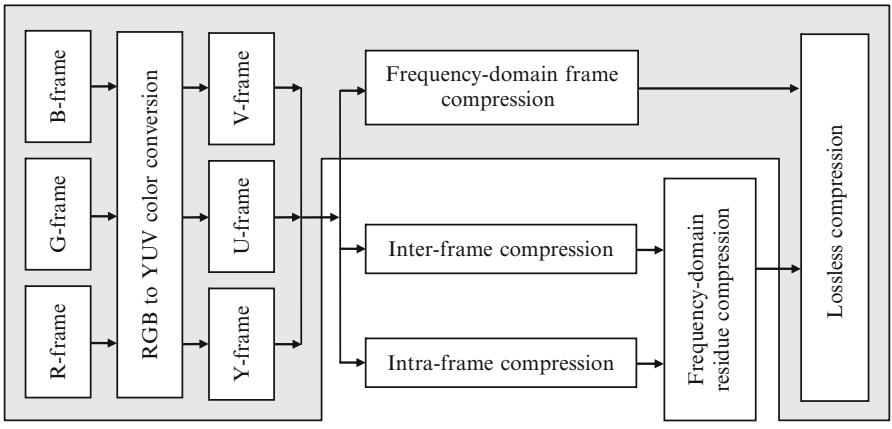

Fig. 16 Image and video compression

color planes) is down-sampled to a frame of YUV planes with the size factor of $1 + \frac{1}{4} + \frac{1}{4} = 1.5$, yielding a compression ratio of 2.

Frequency-domain compression is executed after the RGB to YUV transform. The discrete cosine transform (DCT) is used to translate the image from time-domain to frequency-domain. Because the human sensitivity to spatially high-frequency details is low, the information located in the higher frequency parts can be quantized with lower resolution. People will not notice that the resolution of the higher frequency part is low. After quantization, the resolution of the higher frequency part will only be 1 or 2 bits instead of 8 bits. The image or video signal in frequency domain is further compressed.

After the frequency-domain compression, continuous strings of 1's and 0's appear because of quantization. A lossless compression such as Huffman coding is finally used to compress the most frequent patterns, such as a chain of zeroes. Huffman coding assigns shorter bit strings for more frequent symbols and longer bit string for less probable symbols, and ensures that each code can be uniquely decoded.

Video compression is an extension of image compression, and it further utilizes two types of redundancies: spatial and temporal. Compression of the first video frame, the so called reference frame, is similar to the compression of a still image frame. The reference frame is used for the further compressing of later frames, the inter-frames. The difference between a reference frame and a neighbor frame is usually small. If only the identified inter-frame difference is compressed and transferred, the neighboring frame can be recovered by applying the differences to the reference frame. The size of inter-frame differences is very small compared to the size of the original frame. This coding method is temporal coding.

If one part of a video frame is the same or similar to another part in the same frame, only one part of the compressed code will be stored or transferred. It can be used to replace another part of the frame. Transferring or storing another part of the video frame will therefore not be necessary. Only the difference of the two parts will be transferred. This coding method is spatial coding.

In a video decoder, to comfort visual feeling of human, interpolation and de-blocking algorithms are introduced. To further improve video resolution, the corresponding sample is obtained using interpolation to generate non-integer positions. Non-integer position interpolation gives the best matching region comparing to integer motion estimation. Interpolation algorithms are needed on both the encoder and decoder sides. A de-blocking filter is applied to improve the decoded visual quality by smoothing the sharp edges between coding units.

The classical JPEG (joint picture expert group) compression for still images can reach 20:1 compression. Including the memory access cost, JPEG encoding consumes roughly 200 operations and decoding roughly 150 operations, for one RGB pixel (including three color components). If multiple pictures with high resolution should be compressed in a high resolution (e.g. $3,648 \times 2,736$) camera, accelerated instructions will be essential. Because the processing of Huffman coding and decoding can vary a lot between different images, the JPEG computing cost of a complete image cannot be accurately estimated.

Table 3 Accelerated algorithms for image and video signal processing

Algorithms	Kernel operations
SAD	Result=$\sum \lvert Ai - Bi \rvert$
Interpolation	Result=round $((a_1x_1 \pm a_2x_2 \pm a_3x_3 \pm a_4x_4 \pm a_5x_5 \pm a_6x_6)/32)$
De-blocking filter	Result=round$((a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4 + a_5x_5 + a_6x_6)/8)$
8×8 Discrete cosine transform	Result=Integer butterfly computing and data access
Color transform	Result=$a_1x_1 \pm a_2x_2 \pm a_3x_3$

The classical MPEG2 (Moving Picture Expert Group) video compression standard can reach a 50:1 compression ratio on average. The advanced video codec H.264/AVC standard can increase this ratio to more than 100:1. The computing cost of a video encoder is very dependent on the complexity of the video stream and the motion estimation algorithm. Including the cost of memory accesses, a H.264 encoder may consume as much as 4,000 operations for a pixel and the decoder consumes about 500 operations for a pixel on average. As an example, for encoding a video stream with QCIF size (176×144) and 30 frames per second, the encoder requires about 3 Giga operations per second. The corresponding decoder requires about 400 Mega operations per second. A general DSP processor may not offer both the performance and low power consumption for such applications. An ASIP will be needed especially for handheld video encoding.

Obviously, function acceleration is needed both for encoding and decoding of images and video frames. The heaviest computing to accelerate is motion estimation. Many motion estimation algorithms have been proposed, most of them based on SAD (Sum of absolute difference). Some accelerated algorithms are listed in Table 3.

If data access and computing of the listed algorithms in Table 3 can be implemented with accelerated instructions, more than 80% of image/video signal processing can be accelerated.

7 Programming Toolchain and Benchmarking

As soon as an assembly instruction set is proposed, its toolchain should be provided for benchmarking the designed instruction set. An assembly programming toolchain includes the C-compiler, the assembler, the linker, the ISS (instruction set simulator), and the debugger. A simplified firmware design flow is given in the following Fig. 17a. The toolchain to support programming is given in Fig. 17b. Seven main design steps for translating the behavior C code to the qualified executable binary code are shown in Fig. 17a, and the six tools in the programmer's toolchain in Fig. 17b are used for the code translation and simulation. Each tool in the toolchain in Fig. 17b is marked with numbers. The ways that tools are used in each design step in the flow are annotated by numbers on the design steps in Fig. 17a.

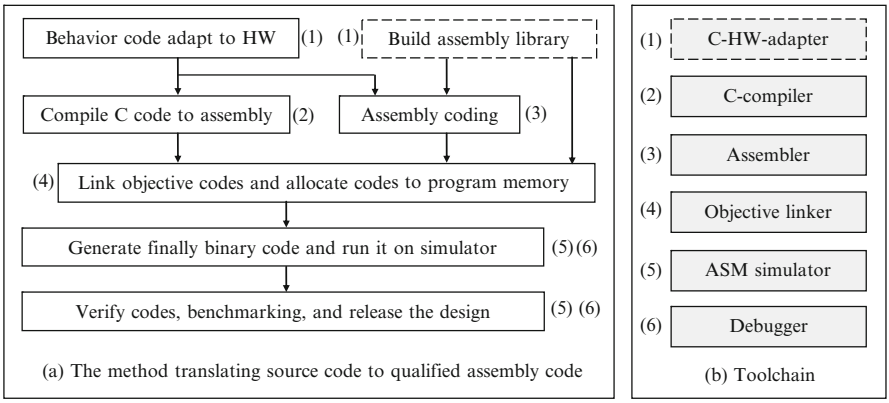

Fig. 17 Firmware design flow using programmer's toolchain

A C-code adapter might be a tool or a step of programming methodology. To get quality assembly code from C-compiler, the source C-code should adapt to hardware including:

- Change or modify algorithms to adapt to hardware.
- Adapt data types to hardware.
- Add annotations to guide instruction selections during compiling.

After adapting the source C-code to hardware, the C-code is compiled to assembly code by the compiler. A compiler consists of its frontend with source code analyzer and optimizer, and its target code synthesizer, or the backend. Most ASIP compilers are based on gcc (GNU Compiler Collection). Compiler design for ASIP DSP can be found in other chapters in this handbook.

The assembler further translates assembly code into binary code. The translation process has three major steps. The first step is to parse the code and set up symbol tables. The second step is to set up the relationship between symbolic names and addresses. The third step is to translate each assembly statement; opcode, registers, specifiers, and labels into binary code. The input of an assembler is the source assembly code. The output of an assembler is the relocatable binary code called object file. The output file from an assembler is the input file to a linker, which contains machine (binary) instructions, data, and book-keeping information (symbol table for the linker).

The assembly instruction set simulator executes assembly instructions (in binary machine code format) and behaves the same way as a processor. It is therefore called the behavioral model of the processor. A simulator can be a processor core simulator or a simulator of the whole ASIP DSP chip. The simulator of the core covers the functions of the basic assembly instructions, which should be "bit and cycle accurate" to the hardware of the core. The simulator of the ASIP chip covers the functions of both the core and the peripherals, which should be "bit, cycle, and pin accurate" meaning the result from the simulator should match the result from the whole processor hardware.

Once the firmware is developed, the programmer must debug it and make sure it works as expected. Firmware debugging can be based on a software simulator, or a hardware development board. In any cases, the programmer will need a way to load the data, run the software, and observe the outputs.

8 ASIP Firmware Design and Benchmarking

Firmware by definition is the fixed software in embedded products, for example an audio decoder (MP3 player) in a mobile phone, where it is visible to end users. On the other hand, firmware can also implicitly exist in a system not exposed to the user, e.g. the firmware for radio baseband signal processing. Firmware can be application firmware or system firmware. The latter is used to manage tasks and system hardware. A typical system firmware is the real-time operating system, RTOS. Qualified DSP Firmware design is based on deep understanding of applications, a DSP processor, and its assembly instruction set.

8.1 Firmware Design

The firmware design flow was briefly introduced on the left part in Fig. 17. It is further refined in Fig. 18. A complete firmware design process consists of behavior

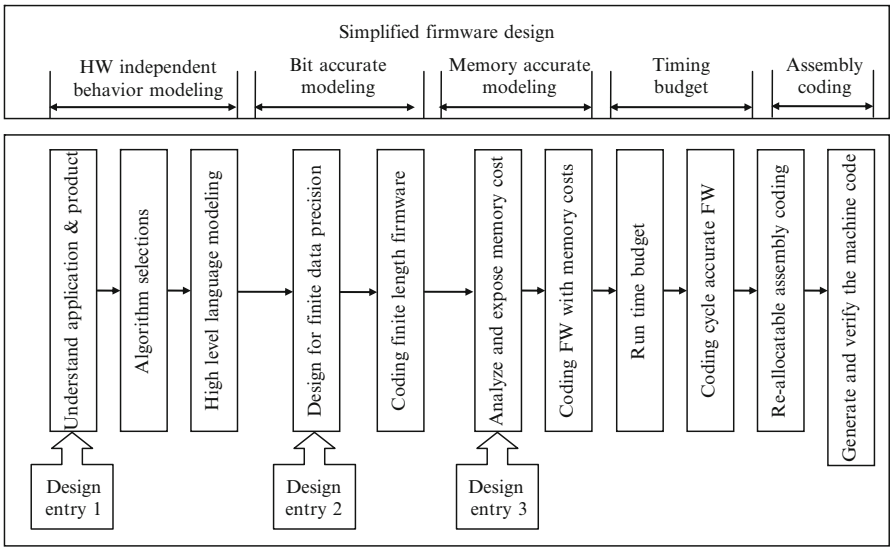

Fig. 18 Detailed firmware design flow

modeling, bit accurate modeling, memory accurate modeling, timing budget and real-time modeling, and finally assembly coding. Firmware design has three starting points according to the freedom of the design.

If designers have all freedoms including selection of algorithms, the design will start from design entry 1, the step of algorithm design and selection. For example, standards of radio baseband modems specify only the transmitter and do not regulate the implementation of receivers. During the implementation of radio baseband receivers, we have the freedom to select either a frequency-domain channel estimator or a time-domain channel estimator according to the cost-performance trade off, the computing latency, and the hardware architecture. Here the cost means mostly the computing cost, data access cost, and the memory cost. For example, some architecture will be suitable for transform and frequency domain algorithms. High level behavior of a DSP system is modeled based on high precision data types and a behavior modeling language, such as C.

However, if there is no freedom to select algorithms, a design starts from design entry 2. A typical example is video decoding for H.264 or audio decoding for MP3. Algorithms and C code are available from the standard. The C code is based on floating-point with excessive high data precision. For embedded applications, hardware with fixed point and limited precision will be used. The design following entry 2 starts from bit-accurate modeling. In this case, the freedom to decide the data precision is available. Data or intermediate results can be scaled and truncated during processing. During the bit accurate modeling phase, data quality control algorithms, such as data masking to emulate the finite hardware precision, signal level measurements, and gain control algorithms, will be added to the original source code.

Freedom of data precision control may not be available in some designs. When the freedoms of algorithm selection and data precision are not available, the firmware design entry is 3 in Fig. 18. In this case, the bit accurate model is available when the firmware design starts, for example the bit accurate source code could be from a standard committee. A typical case is the implementation of some voice decoder, which starts from available bit-accurate C code. Voice is usually compressed before transfer or storage to minimize the transfer bandwidth or storage cost. A voice decoder will decompresses the compressed voice and to recover the voice waveform at the voice user side.

In most ASIP, scratchpad memories instead of cache are used to minimize the data access latency. The time cost of data access and the memory cost is roughly exposed during source code profiling. The early estimate of memory cost might not be correct. Memory cost can be further exposed when the data types are decided after bit accurate modeling. A memory accurate model can therefore be achieved. Cache is not much used for embedded DSP computing because cache is expensive and data access in embedded computing is relatively explicit and predictable. A memory accurate model is an essential step to expose the data access cycle cost of scratchpad memories and finally to minimize it. The first step of memory accurate modeling is to re-model the data access in real hardware based on the constraint of the on chip memory sizes. After the first step, the extra memory transaction cost

will be exposed. In the next step, moving data between memories and other storage devices will be modeled using program code or DMA transactions.

The firmware has runtime constraints when processing real-time data, especially streaming data. From profiling of the source code, runtime cost can be estimated. Extra run time cost is used to execute added subroutines for handling finite data precision, extra memory transactions, and task management. If the total runtime is less than the new data arrival time, the system implementation is feasible. Otherwise, enhancing hardware performance and selecting low cost algorithms are required.

Finally, the assembly code is generated and the functionality is verified. The assembly code as the firmware can be released when the final run time of the assembly code follows the specification and the precisions of the results are acceptable.

8.2 Benchmarking and Instruction Usage Profiling

Benchmarking by definition is some kind of measure on performance. In this context, benchmarking is used to measure the performance of the instruction set of an ASIP DSP. In general, benchmarking is to measure the run time (the number of clock cycles) for a defined task. Other measurement can also be important, for example memory usage and power consumption. Memory usage can be further divided into the cost of program memory and data memory required by a benchmark. DSP benchmarking is usually conducted by running the assembly benchmark code on the instruction set simulator of the ASIP.

DSP Benchmarking can be further divided into the benchmarking of DSP algorithm kernels and the benchmarking of DSP applications. By benchmarking DSP kernels, the 10% codes taking 90% runtime, essential performance and cost can be estimated. When benchmarking the kernel algorithms, kernel assembly code such as filters FIR, IIR, LMS (least mean square) adaptive filter, transforms (FFT and DCT), matrix computing, and function (1/x, for example) solvers are executed on the instruction set simulator of the processor. Benchmarking of an application runs the application codes on assembly instruction set simulator of the processor. The best way to evaluate a processor is to run applications because all overheads can be taken into account. However, the coding cost for an entire application can be very high. The benchmarking of an application should only be conducted on part of the application, the cost extensive part.

To benchmark a DSP instruction set, two kinds of cycle cost measurement are frequently used. One is the cycle cost per algorithm per sample data. For example "30 clock cycles are used to process a data sample by a 16-tap FIR filter". Another kind of measurement is the data throughput of an application firmware per mega Hertz. For example, "In one million clock cycles (1 MHz), up to 500 voice samples can be processed in a 2048-tap acoustic echo canceller". If the voice sampling rate is 8 kHz (it usually is), the computing cost of 16 MHz will be required in this case.

Table 4 Examples of kernel DSP algorithms running on a single MAC DSP

Algorithm kernel	Descriptions or specifications	Typical cycle cost
Block transfer	Move N data words from one memory to another	$\sim 3N+3$
256p FFT	256 point FFT including computing and data access	$\sim 11,000$
Single FIR	A N-tap FIR filter running one sample	$\sim N+12$
Frame FIR	A N-tap FIR filter running K samples	$\sim K(N+6)+8$
Complex data FIR	A N-tap complex data FIR filter running one sample	$\sim 8N+15$
LMS Adaptive FIR	A N-tap least significant square adaptive filter	$\sim 3N+10$
16/16 bits division	A positive 16bits divided by a 16bits positive data	~ 50
Vector add	C[i] $\leq$ A[i] + B[i] Here i is from 0 to N-1.	$\sim 3+3N$
Vector window	C[i] $\leq$ A[i] * B[i] Here i is from 0 to N-1.	$\sim 3+3N$
Vector Max	R $\leq$ MAX A[i] Here i is from 0 to N-1.	$\sim 2+2N$

The benchmarks of basic DSP algorithms are usually written in assembly language. However, if the firmware design time (time to market) is very short, benchmarks written in high-level language will be necessary. In this case, the benchmarking checks the mixed qualities of the instruction set and compiler.

BDTI (Berkeley Design Technology Incorporation [6]) supplies benchmarks based on handwritten assembly code. EEMBC (EDN embedded microprocessor benchmark consortium [7]) allows two scoring methods: Out-of-the-box benchmarking and Full-Fury benchmarking. Out-of-the-box (do not requiring any extra effort) benchmarking is based on the assembly code directly generated by the compiler. Full-Fury (also called optimized) benchmarking is based on assembly code generated and fine tuned by experienced programmers.

It is not easy to make an ideally fair comparison by benchmarking low level algorithm kernels on target processors. Each processor has dedicated features and is optimized for some algorithms, while not optimized for some other algorithms. A processor holding a poor benchmarking record of an application might have very good benchmarking record of another application. A typical case is that a radio baseband processor will never be used as a video decoder processor. For fair comparison, processors from different categories should not be compared.

DSP kernel algorithms consist of 10% of an application code that takes 90% of the runtime. Benchmarking on kernels is relevant because DSP kernel algorithms will take the majority of the execution time in most DSP applications. Well accepted DSP kernels are listed in Table 4. In the table, the typical cycle cost is measured on a DSP processor with single MAC unit and two separated memory blocks, a simple and typical DSP processor. It exposes the average performance among single MAC commercial DSP processors. If the benchmarking result of an ASIP designed by you is much behind scores in the table, you may need to understand why and try to improve your design.

Cycle cost and code cost consist of three parts while coding and running a kernel benchmarking subroutine, the prolog, the kernel, and the epilog. The prolog is the code to prepare and start running a kernel algorithm, it includes loading and configuring parameters of the algorithm. The kernel is the code of the algorithm,

including loading data, algorithm computing, and storing intermediate results. The epilog is the code to terminate running of an algorithm, including storing the final result and restoring the changed parameters. Cycle cost for running the prolog, the kernel, and the epilog should be taken into account during benchmarking.

Instruction usage profiling is another measure on using an instruction set. Instead of measuring the performance, it measures the appearance of each instruction in application codes and how often it is executed in each million cycles. The usage profiling tells which instruction is used the most in an application. It also tells which instruction is the least used by an application. If several instructions are not used by a class of applications and the processor is only designed for this class of application, these instructions can be removed to simplify the hardware design and reduce the silicon costs. However, sometimes the saving of silicon cost and design cost is almost negligible while removing an instruction. The cost reduction could be significant only when many instructions are removed. On the other hand, since the most used instructions can be identified, optimizing them can significantly improve the performance or reduce the power consumption.

9 Microarchitecture Design for ASIP

The microarchitecture design of an ASIP specifies the design of hardware modules in an ASIP. In particular, it is the hardware implementation of the ASIP assembly instruction set to each function module of the core and peripheral modules of the ASIP. The input of the microarchitecture design is the ASIP architecture specification, the assembly instruction set manual, and the requirements on performance, power, and silicon cost. The output of the microarchitecture design is the microarchitecture specification for RTL coding, including module specifications of the core and interconnections between modules. The microarchitecture design of an ASIP can be divided into three steps.

(1) To partition each assembly instruction into micro-operations and to allocate each micro-operation to corresponding hardware modules and corresponding pipeline steps. A micro-operation is the lowest level hardware operation, for example, operand sign bit extension, operand inversion, addition, etc. If there is no such a module for some micro-operations, adding a module or adding functions to a module is needed.
(2) For each hardware module, collect all micro operations allocated in the module, allocate them to hardware components, and specify control for hardware multiplexing for RTL coding.
(3) Fine tune and implement inter-module specifications of the ASIP architecture and finalize the top-level connections and pipelines of the core.

Similar to architecture design, there are also two ways to design microarchitecture, one way is to use reference microarchitecture, which is an available design of a hardware module from publications, other design teams, and available IP.

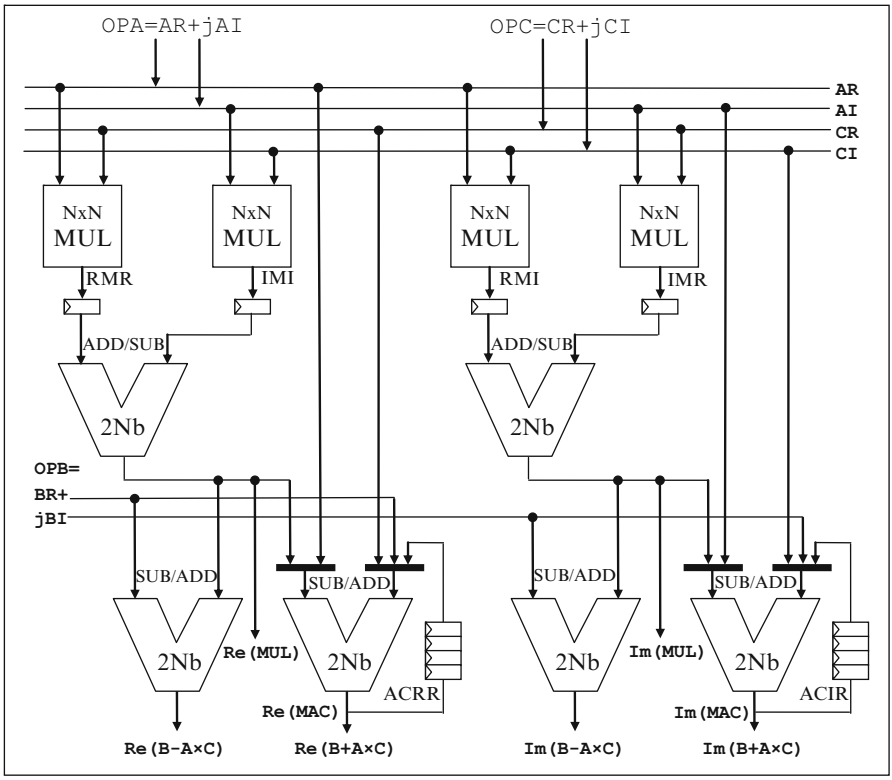

Fig. 19 A radix-2 data path for complex data computing

The reference microarchitecture is close to the requirements of the module to be designed and can execute most micro-operations allocated to the module.

Another method is to design a custom microarchitecture, which is to generate a custom architecture dedicated for a task flow of an application. This method is used when there is no reference microarchitecture or the reference microarchitecture is not good enough. The method is to map one or several CFG (control flow graph, the behavior flow chart of an application) to hardware. A typical case is to design accelerator modules for special algorithms, such as lossless compression algorithms, forward error correction algorithms, or packet processing protocols.

A typical example in Fig. 19 shows an example of a simplified datapath cluster for complex-data computing in a radio baseband signal processor. N in this figure is the word length of real or imaginary data. The cluster can execute the following functions one step in a clock cycle. These functions are computing a FFT butterfly, iterative computing for complex data multiplication and accumulation, complex data multiplication, and complex data addition/subtraction.

A Radix-2 decimation in time (DIT) butterfly can be executed in each clock cycle for FFT. The input data of the butterfly is data OPA = AR + jAI and OPB = BR

+ jBI, the input coefficient is OPC = CR + jCI. The outputs of a butterfly are Re(B+A*C) + jIm(B+A*C) and Re(B − A ∗ C) + jIm(B − A ∗ C). When executing the complex-data MAC (Multiplication and accumulation) or convolution, the input will be OPA and OPC. The output is thus Re(MAC) + jIm(MAC). When executing the complex multiplication, the input is also the OPA and OPC. The output is Re(MUL) + jIm(MUL). Finally, complex data addition and subtraction can be executed when the inputs are OPA, and OPC. Results are OPA + OPC = REA(MAC) + j Im(MAC) or OPA − OPC = REA(MAC) − j Im(MAC).

10 Multicore Parallel Processing

With the rapid development of radio standards, multimedia application, and graphics computing, the signal processing workload of modern embedded system has increased dramatically. For example, a high-end mobile phone that supports wideband radio communication and HD video codec demands a computation throughput of 100 GOPS [9]. The high computing load leads to multicore architecture with multiple ASIP processors to do parallel signal processing. The design of such a multiprocessor includes the design/selection of ASIP processors, memory hierarchy design, and interconnection architecture design.

10.1 Heterogeneous Chip Multiprocessing

Modern DSP system integrates multiple DSP cores to do signal processing in parallel. The wide range of applications and algorithms has led to the use of heterogeneous multiprocessor with specialized cores. Various ASIPs could be selected for different applications. For example, traditional single scalar DSP can be used for scalar algorithms like audio codec. SIMD style data parallel processors are used for computing tasks with rich vector operations such as radio baseband and multimedia. Since those ASIPs are optimized for data processing, a group of ASIPs are usually controlled by a microcontroller to form a DSP subsystem. The microcontroller handles the computing flow and data movements. Figure 20 shows a multiprocessor DSP system with one microcontroller and multiple application specific DSPs.

10.2 Memory Hierarchy

Multiprocessor DSPs usually utilize multi-level parallel memory hierarchy to improve performance by hiding memory access latency and avoiding access conflict. The memory system uses both distributed memory and shared memory, as shown in

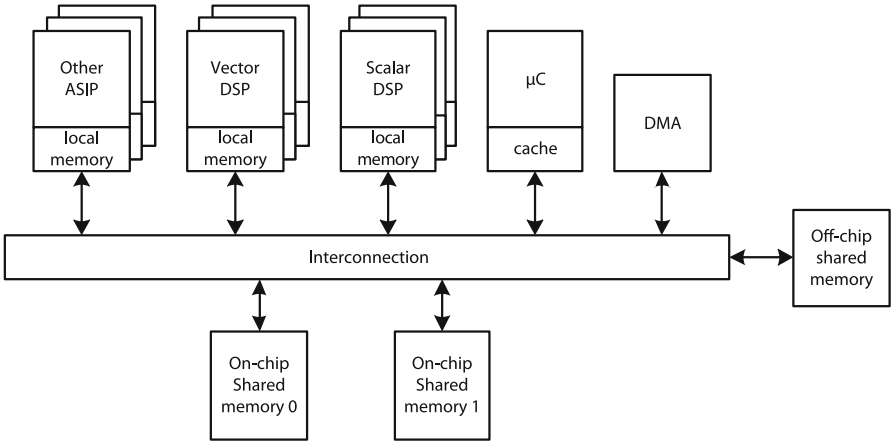

Fig. 20 Heterogeneous multiprocessor DSP subsystem

Fig. 20. The distributed memories are small and private on-chip memories of each ASIP core. The larger shared memory is mapped to the global address space of the system. The shared memory can have multiple levels; include both on-chip memory and off-chip memory. Unlike the general purpose processor uses cache to hide the complexity of data accesses, embedded DSP processors use scratch pad memory (SPM) as the local storage and rely on the software programmer to explicitly control the data movements.

The complex memory hierarchy and the use of SPM make a big challenge for software developments of multiprocessors. In a multiprocessor DSP system, it is common to have tens of memory modules in a single chip. Most of these memories are under explicit program control by software. A programmer needs to not only read and write memory data for computing but also use DMA functions to move data between these memories. The software programmer needs to handle multicore synchronization and data dependency. Multiprocessor platform should provide software development tools to trace memory access activities to help software programmers. Such tool can be found in the IBM Cell SDK [10] as an example.

10.3 Interconnection Architecture

Data communication in the multicore memory hierarchy is across the interconnection network. The design of interconnection architecture depends on requirements on the communication throughput and latency. If the communication throughput is not very high relative to the computing load on the processors, simple and low cost shared bus with narrow bus width can be used. If inter-processor communication

is critical in a multiprocessor system, the interconnection design can use multi-connection and multi-layer busses (for example crossbar). The shared bus allows only one bus master to transfer data at one time. While a multi-connection and multi-layer bus allows concurrent data transfer between different master-slave pairs.

11 Further Reading

The ASIP DSP design flow was briefly introduced in this chapter. Architecture exploration and design, assembly instruction design, design for toolchain, firmware design, benchmarking, and microarchitecture design were introduced in the chapter. Two examples, radio baseband ASIP DSP and video ASIP DSP were briefly introduced. Readers are encouraged to get more ASIP design knowledge from [1]. ASIP design is so far experience based. If all requirements on performance, power consumption, and silicon cost cannot be reached by COTS, ASIP design is essential. ASIP enables performance computing and dramatically minimizes the cost of volume products.

This chapter is provided for readers who want to get fundamental knowledge in ASIP. For professional ASIP designers, more detailed information is provided in [1, 8], and [4].

References

1. D. Liu, *Embedded DSP Processor Design, Application Specific Instruction Set Processors*, Elsevier 2008 ISBN 9780123741233
2. D. Liu, A Nilsson, D Wu, J Eilert, and E Tell, *Bridging dream and reality: Programmable baseband processor for software-defined radio*, IEEE Communication Magazine, pp 134–140, September 2009
3. Coresonic Inc., http://www.coresonic.com/
4. P. Karlstrom, D. Liu, NoGAP a Micro Architecture Construction Framework, SAMOS IX: International Symposium on Systems, Architectures, MOdeling and Simulation, July 2009.
5. David Salomon, *Data Compression, The Complete Reference, 3rd Ed.*, Springer, ISBN: 9781846286025, 2006.
6. Berkeley Design Technology, Inc., http://www.bdti.com
7. Embedded Microprocessor Benchmarking Consortium, http://www.eembc.org
8. M. Hohenauer, H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, *Compiler-in-loop Architecture Exploration for Efficient Application Specific Embedded Processor Design, magazine*, Design and Elektronik. WEKA, Verlag 2004.
9. D. Liu, et al, ePUMA, *Embedded Parallel DSP Processor with Unique Memory Access*, ICICS 2011, Singapore.
10. IBM, Cell Broadband Engine Programming Handbook, Version 1.11, May.2008.