

## Chapter 2

# The ASIP Design Space

### 2.1 Introduction

ASIPs represent a growing trend of application oriented processor specialization for computationally intensive embedded applications. The first micro-processors – Intel 4004, TI TMS 1000 and Central Air Data Computer (CADC) – designed way back in the early 1970s were mostly intended for general purpose usage. This trend continues even today as general purpose programmable microprocessors and micro-controllers remain in wide use not only in the server and desktop PC market [8, 82, 130], but also in the embedded computing domain [13, 116]. However, the need for new, application specific programmable devices started to grow with the growth of the Internet, digital multimedia, mobile and wireless technologies, because general purpose processors were often found inadequate to meet the performance requirements of mobile or networking applications under tight cost, area and power consumption budgets. Not surprisingly, a wide array of domain specific programmable devices – such as DSPs [9, 63, 167] and NPU's [147] – are now employed to address the specific computational requirements of these application domains. Domain specific processors contain special micro-architectural features (e.g. multiple memory banks and address generation units in DSPs, multiple parallel processing engines in NPU's) or customized instructions (e.g. multiply-accumulate instruction in DSPs) to accelerate applications from the corresponding domain in hardware. ASIPs take this application oriented design philosophy to the extreme where a processor is designed to optimally execute only a single, or at most, a handful of applications.<sup>1</sup> However, the architecture is expected to be flexible enough to accommodate bug fixes and enhancements without incurring significant performance degradation to ensure longer time-in-market (and shorter time-to-market for subsequent products).

---

<sup>1</sup>The corresponding application will be called the *target application* for the rest of this book.

Although most ASIP designs adhere to the general design principles introduced in the classical computer architecture and organization texts by Hennessy and Patterson [125, 126], the vast gulf of differences in design objectives between ASIPs and general purpose processors usually compels designers to employ design alternatives not commonly found in the general purpose computing domain. This chapter intends to familiarize readers with all such standard and application specific architectural tricks available to the ASIP developers. It discusses standard architectural issues – instruction-set architecture, pipeline, register file (RF) and memory hierarchy design – along with special topics like partially reconfigurable architectures and hardware accelerator development. For each topic, a brief historical perspective and description of standard design practices are followed by an in depth discussion of customization options pertinent to ASIP architectures. The material presented in this chapter provides the necessary background for understanding the various design issues discussed in later parts of this book.

## 2.2 Architectural Alternatives for ASIPs

### 2.2.1 *Instruction-set Architecture*

The instruction-set architecture of a processor represents a simplified programmer's view of the *micro-architecture* (i.e. underlying implementation and organization details of the processor hardware). The ISA specifies the instruction opcodes, instruction encoding, memory/register addressing modes and supported data types. Depending on their ISA structures, processors are usually classified into two categories: *reduced instruction-set computers (RISC)* and *complex instruction-set computers (CISC)* [125]. CISC ISAs are characterized by the presence of several complex instructions (usually created by combining multiple arithmetic/logic computations and memory access operations), special purpose registers, variable length instruction encoding and complicated addressing modes. The absence of general purpose registers as well as the abundance of complicated instructions and addressing modes make CISCs extremely unsuitable for compiler code generation phases like register allocation and code selection.

In contrast to CISCs, RISCs (also called *load-store architectures*) generally employ a uniform *general purpose register (GPR)* file, simple addressing modes, fixed length instruction encoding and simple instructions. One RISC instruction usually contains one arithmetic/logic/memory access operation. Input/output operands of all arithmetic/logic instructions come from the GPR file, while memory accesses are performed by a separate set of load/store instructions. Because it is far easier to design compilers for RISCs than CISCs, almost all general purpose embedded processors [13, 116] today are RISC machines.

Various instruction-set design issues for ASIP architectures are briefly touched upon in the rest of this section.

### 2.2.1.1 Instruction-set Characteristics

As a general rule, ASIP ISAs are closer to many modern DSP instruction-sets which tend to combine properties of both CISC and RISC machines. Most ASIPs generally have a load-store architecture with a GPR file as a *base processor*. The base processor usually implements simple unary or binary arithmetic, logical, relational, shift and memory access operations in a basic instruction-set (a. k. a. base processor instruction-set). However, this basic instruction-set can be augmented using application specific instructions<sup>2</sup> which may use complex addressing modes and hidden special registers. For example, two most prominent configurable processor technologies – MIPS CorXtend [115] and ARC Tangent [11] – allow designers to add special instructions and registers to their RISC ISAs. Similarly, the ASIP design presented in [141] has a set of extremely complex special instructions like a CISC and a GPR file like a RISC machine.

### 2.2.1.2 Base Processor Instruction-set

An important issue for ASIPs is which instructions to include in the basic instruction-set. Normally, all integer arithmetic, logic, shift and relational operations except multiplication and division are included in the basic instruction-set. Integer multiplication and division, as well as floating point operations, are included only when the target application contains them in significant numbers. Leaving out infrequent operations from the basic instruction-set saves area and improves energy efficiency. Left out instructions can be easily emulated in software without hurting performance in any significant way. The operations included in the basic instruction-set are called *base processor instructions (BPIs)*.

A simple example of this application oriented design philosophy can be constructed by considering two different application domains – *multimedia* and *private key cryptography*. Most multimedia applications (e.g. H.264 [177], MPEG-2 [117] and MP3) contain considerable number of multiplications, whereas private key cryptographic algorithms (e.g. AES, DES and Gost [145]) only need addition, subtraction, logical and shift operations. Consequently, multipliers have to be included in the basic ISA of any multimedia ASIP, but can be safely left out from private key cryptographic processors.

### 2.2.1.3 Instruction Encoding

Another vital consideration for ASIP ISAs is the instruction encoding scheme. A compact instruction encoding increases code density and lowers instruction

---

<sup>2</sup>Such application specific instructions are also called *instruction-set extensions (ISEs)* or *custom instructions (CIs)*. For the rest of this book, these three terms will be used interchangeably.

memory requirement. However, such encoding schemes can often adversely affect performance by limiting instruction-set functionalities. Examples of possible restrictions due to a compact instruction encoding include constraints on the number of instruction opcodes, number of bits per register/immediate operand and total number of operands per instruction. As will be seen later in this book, the number of register operands available to application specific ISEs is a key contributing factor for achieving high hardware acceleration. Similarly, the number of bits made available for immediate operands or branch targets must be selected very judiciously. Selection of wider bit-widths for immediate operands can lengthen instruction words and decrease code density. Choosing very narrow fields for immediate values can affect performance, since wider immediate operands have to be loaded to registers before use. For ASIP development, these concerns must be balanced very carefully by considering all pertinent design constraints.

Other possible ways of increasing code density include variable length encoding and dual instruction encoding. Variable length instruction encoding involves different encoding schemes for different classes of instructions – instructions with several operands are encoded using more bits than instructions which have no or few operands. The most common example of dual encoding can be found in ARM processors which support the 16-bit thumb/thumb-2 [81] instruction-sets along with normal 32-bit encoding. The thumb ISA encodes only a subset of the normal ARM instruction-set and limits many instructions to only access half of the register file. During execution, the thumb instructions are translated to normal ARM instructions by a de-compression unit embedded in the ARM processor pipeline. Any one of the above architectural alternatives can be used for ASIPs, if the size of the instruction memory is an important design parameter.

A properly selected instruction encoding can also reduce the overall dynamic energy consumed by the instruction memory and the instruction bus [23, 41, 187]. This optimization is mostly orthogonal to the code density issue – but is equally important for improving the energy efficiency of application specific architectures.

### 2.2.2 Instruction Pipelining

Since its inception in the 1970s, instruction pipelining has become a universally accepted technique for increasing instruction throughput by lowering *clocks per instruction* (CPI). Pipelining divides the execution of an instruction into several simple, single cycle stages so as to overlap the execution of successive instructions in an instruction stream. The classical pipeline architecture – the template for most modern day RISC processors – described in [126] has five stages: instruction fetch (FE), instruction decode (DE), instruction execution (EX), memory access (MEM) and result write-back (WB) (Fig. 2.1). In this architectural template, the execution of the first instruction in an instruction stream can be overlapped with the decoding of the second and the fetching of the third instruction. Successive stages use *pipeline registers* (① and ④ in Fig. 2.1) for communicating intermediate results.

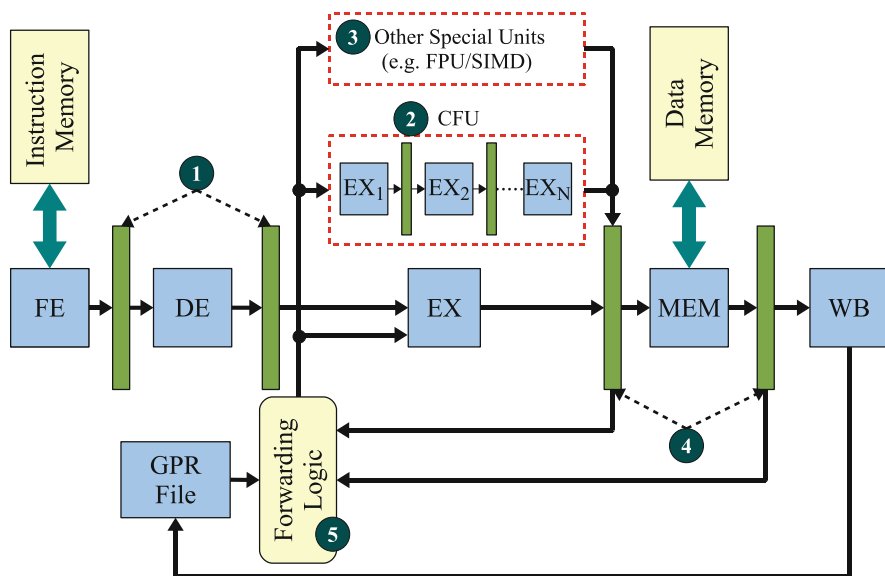


Fig. 2.1 Example pipeline architecture for ASIPs

Depending on the design constraints, ASIP pipelines can be designed completely from scratch or can be based on the classical template described above. An example of a completely customized ASIP pipeline is presented in [141]. The pipeline structure described in this work consists of seven stages. The customary instruction fetch and decode are followed by five stages which have been designed specifically for piecewise approximation of some image processing functions required in retinex like image enhancement algorithms. However, such extreme specialization is not common for most ASIP architectures. Majority of them [11, 89, 115, 182] uses extensions of the classical five stage RISC architectural template shown in Fig. 2.1. The EX stage usually implements simple integer arithmetic/logic instructions and a single cycle integer multiplier. Application specific special instruction data-paths are usually implemented in a *custom functional unit (CFU)* (② in Fig. 2.1) which executes in parallel with the base processor's EX stage. Similarly, other application specific execution units such as FPUs and SIMD data-paths can also be implemented in parallel with the base processor's EX stage (③ in Fig. 2.1).

Extracting the ideal CPI of 1 from a single-issue<sup>3</sup> processor pipeline is usually not possible due to *structural, data and control hazards*. The reasons of these hazards and their possible remedies within the context of ASIP architectures are discussed in the rest of this section.

<sup>3</sup>CPI of less than 1 can be achieved in multiple-issue processors, i.e. processors which can start execution of multiple instructions in parallel. More on this in Sect. 2.2.3.

### 2.2.2.1 Structural Hazards

Structural hazards occur in a processor pipeline when two instructions simultaneously compete for the same processor resource. To construct an example of a structural hazard, a processor architecture with a single memory for storing both instructions and data can be considered. In this processor pipeline, a conflict for the main memory can occur between a memory load instruction in the MEM stage and any other instruction in the FE stage.

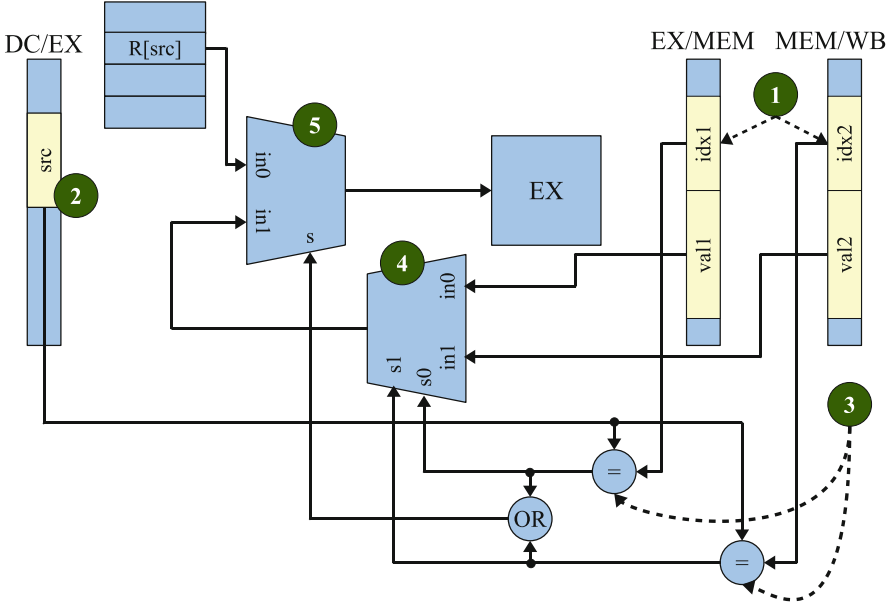
Structural hazards can be handled either by providing more processor resources, or by stopping one of the conflicting instructions from executing. The structural hazard arising out of a single main memory can be avoided if separate memory banks are used for instructions and data (as in Fig. 2.1). Alternatively, the hazard can be handled by preventing any instruction from entering the FE stage when a load enters the MEM stage.

In ASIPs, structural hazards can occur due to multi-cycle, non pipelined special instructions. A conflict for the CFU can develop between a multi-cycle ISE and any other special instruction *immediately following* it. Such conflicts can be resolved either by pipelining the CFU itself (② in Fig. 2.1), or by preventing any other ISE from entering the CFU as long as a multi-cycle ISE is executing inside it. The first solution eliminates any performance penalties due to the structural hazard, but is difficult to implement and increases CFU area. In contrast to this, the second solution is easier to implement, but may cause significant performance degradation because any special instruction following a multi-cycle ISE has to be *stalled* several cycles. The stalling can be implemented through *hardware interlocks*, or can be achieved by having the *compiler* insert an appropriate number of *No OPERATION (NOP)* instructions between a multi-cycle ISE and any other ISE following it.

### 2.2.2.2 Data Hazards

In the instruction pipeline shown in Fig. 2.1, the result of a computation done in the EX stage is percolated through the EX/MEM and MEM/WB pipeline registers (④ in Fig. 2.1) and are written to the GPR file after two cycles in the WB stage. A *data hazard* occurs in this pipeline, if a *read-after-write (RAW)* data dependence exists between two consecutive instructions – i.e. the destination register of the first instruction is one of the source register operands of the second instruction. The effects of the hazard can be understood by considering that the value computed by the first instruction in the EX stage is written back to the corresponding GPR after two cycles, while the second instruction needs the correct value in EX stage in the next cycle. Due to the data dependency, the second instruction ends up reading the old value from the source register instead of the current value computed by the first instruction.

Like structural hazards, data hazards can be eliminated by stalling the pipeline for appropriate number of cycles – either using NOPs inserted by the compiler, or



**Fig. 2.2** Data forwarding architecture

through hardware interlocks. However, due to the significant performance penalties of this scheme, most processors usually employ data forwarding logic (⑤ in Fig. 2.1) to reduce the effects of data hazards.

An exemplary forwarding architecture is presented in Fig. 2.2 where the indices of the destination registers (①) are carried along with their values through the EX/MEM and MEM/WB pipeline registers. Before starting the execution of an instruction in the EX stage, the index of each of its source operands (② in Fig. 2.2) is compared against the indices of the destination registers of the previous two instructions (③). If one of the destination indices matches with the source index, then the corresponding value is selected as the source operand (④). Otherwise, the value is taken from the GPR file (⑤).

Application specific ISEs in ASIPs often require more GPR input/output operands than available to base processor instructions. Even a very simple ISE like the *multiply and accumulate* (MAC) instruction requires three input operands (compared to two operands required by all arithmetic/logic operations). ISEs designed for ASIPs are usually far more complex than MAC and need several input/output operands. However, the complexity of the data forwarding architecture often becomes a limiting factor in such cases.

Figure 2.2 shows the data forwarding logic for a single source and destination operand. For multiple source operands, the forwarding logic has to be replicated multiple times. In addition to this, the complexity of the forwarding logic for each individual source operand increases with the number of output operands. It can

be easily seen from Fig. 2.2 that four comparison operators instead of two (③) will be required if an instruction is permitted to have two destination operands. Similarly, a 4×1 MUX will be required for selecting the right forwarded value instead of the 2×1 MUX (④). The matter becomes even more complicated if multi-cycles ISEs are taken into account. It is of course possible to implement a partial data forwarding policy, e.g. data forwarding *only* for base processor instructions. However, such policies can increase latencies of special instructions and adversely affect performance.

### 2.2.2.3 Control Hazards

Another important pipelining issue is the handling of *control hazards* arising from conditional branches. In the classical RISC pipeline of Fig. 2.1, the outcome of a conditional branch instruction is not known till the DE stage. However, by that time, another instruction following the conditional branch is already in the FE stage of the pipeline. If the branch is taken, then the effect of this following instruction must be canceled, i.e. it must not be allowed to commit its results to memory or register file. Several schemes for handling control hazards are presented in Fig. 2.3 which shows an example C code and four pseudo assembly code representations of it. The C code contains a for loop (①) and an if-then-else statement (②) – both of which have to be implemented using conditional branches. The four different pseudo assembly codes show four different ways of handling control hazards. The italicized conditional branches in the pseudo assemblies originate from the for loop, while the normal ones originate from the if-then-else statement.

Like structural and data hazards, control hazards can be handled by simply stalling the instruction pipeline and preventing any instruction following the branch from entering the pipeline till the branch outcome is known. The stalling can be done by insertion of a NOP instruction after each conditional branch (③). This results in the so called *branch penalty* – lost execution cycles for each conditional branch instruction.

One commonly used mechanism to eliminate branch penalties is to always execute a fixed number (usually, not more than two) of instructions following a branch irrespective of the branch outcome. These instructions are said to be placed in *delay-slots* of the branch and the corresponding branch instruction is called a *delayed branch*. Obviously, an instruction in a branch delay slot must not be control dependent<sup>4</sup> on the branch, and the branch must not be control or data dependent on it. The load instruction, `val=* (a+i)`, (④ in Fig. 2.3) can be placed in the delay slot of the branch, `if (!t2) goto L2`, because it fulfills both of these conditions. However, the instruction preceding the branch, `t2=i%2`, can not be

---

<sup>4</sup>An instruction is control dependent on a branch if its execution depends on the outcome of the branch. For example, instructions in the if and else portions of the if-then-else statement in Fig. 2.3 are control dependent on the corresponding conditional jump.

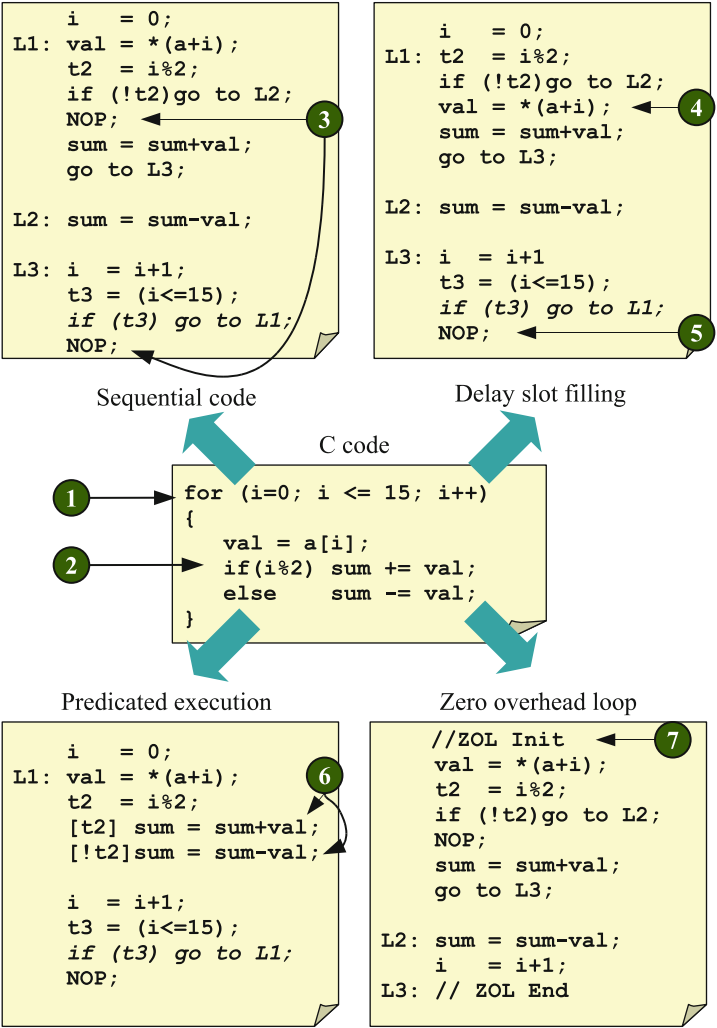


Fig. 2.3 Architectural options for minimizing branch penalty

placed in the delay slot, because the branch is data dependent on it. Usually the task of filling the delay slots is left to the compiler. If suitable instructions for filling the delay slots of a branch are not found, NOPs are inserted instead (⑤ in Fig. 2.3).

Another more complex strategy – known as *branch prediction* – attempts to predict beforehand whether a branch will be taken or not. Depending on the prediction outcome, instructions from either the branch fall-through or the branch target are fetched and speculatively executed. In case of a miss-prediction, the instructions executed after the branch must be annulled.

There are two more strategies which try to eliminate branches altogether. The first of these is called *predicated execution* where a condition – called a predicate – is attached to all instructions control dependent on a branch. The predicate is usually placed in a predicate register. If the predicate is true, then the instruction commits its result to processor storage, otherwise it is annulled. This method works only for *forward conditional branches* (i.e. branches originating from if-then-else or switch-case structures in the source code) and not for loops. An example of this scheme is presented in Fig. 2.3 which shows the predicated version (⑥) of the if-then-else statement of the original C code. Readers can easily observe that this scheme completely eliminates the corresponding conditional jump instruction.

The other scheme reduces the overhead of loop iterations by eliminating the loop condition testing and the jump instruction at the end of a loop. This technique only applies to loops for which the iteration count is known at compile time. In this scheme, a special instruction – usually called a *zero overhead loop (ZOL)* instruction – repeats a certain portion of code a predefined number of times. The portion of code to execute and the iteration count are initialized through some special instructions before the ZOL instruction is triggered (⑦ in Fig. 2.3). Many embedded DSP architectures use this technique for speeding up loops.

Among the techniques described above, delayed branches and ZOLs are usually preferred candidates for ASIP architectures. Predicated execution is not commonly found in single-issue RISC architectures, but is a prominent feature in many VLIW architectures [176]. The predicated execution scheme converts control flow to data flow which lets VLIW compilers easily discover parallelism.

Hardware branch prediction is a scheme which is not commonly found in embedded architectures, although high performance general purpose processors often use it. Branch prediction significantly complicates the pipeline implementation and is not suitable for data dominated embedded applications.

### 2.2.3 Instruction and Data Parallelism

In order to meet the stringent performance and power efficiency requirements of most portable consumer electronic gadgets, SoC designers usually try to exploit any form of available parallelism in the target applications. Readers may recall from Sect. 1.1 that these applications are usually composed of several coarse-grained tasks. Parallelism in such an application may exist between two independent tasks (inter-task parallelism) as well as within a single task (intra-task parallelism). While inter-task parallelism can only be exploited at the SoC architecture level, intra-task parallelism must be handled in the corresponding PEs. In this section, some standard techniques to deal with fine-grained, intra-task parallelism in programmable processors are discussed briefly.

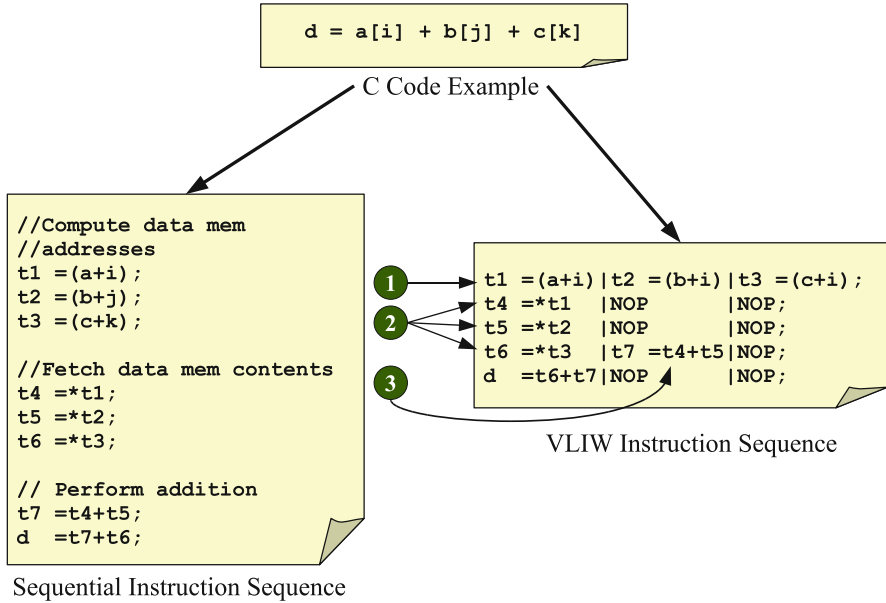
### 2.2.3.1 Exploiting Instruction Level Parallelism

An application exhibits *instruction level parallelism (ILP)*, if, at any point in its execution, there exists a set of yet-to-be-executed instructions which are mutually independent (i.e. there exists no control or data-dependence between any two instructions in the set) and therefore, can be executed in parallel. The parallelism available in an application can be increased by re-ordering its sequential instruction stream through *instruction scheduling*.

Instruction level parallelism can be exploited by a processor if it can simultaneously *issue* multiple instructions in each clock cycle. Multiple issue architectures can be categorized into two classes: *very long instruction word (VLIW)* [58] and *superscalar* [148] processors. In a VLIW processor, as the name suggests, multiple instructions packed into each single *very long instruction word* are executed in parallel. The pipeline structure for such a processor usually consists of the classical RISC pipeline (Sect. 2.2.2) replicated as many times as the maximum number of instructions in an instruction word. Each such parallel execution pipeline is called an *issue slot*. For a VLIW machine, the compiler tool-chain *statically* schedules a sequential stream of instructions into parallel issue slots. In contrast to this, superscalar architectures use *dynamic scheduling*, i.e. scheduling and parallel issue decisions are taken by the processor hardware during program execution. The runtime scheduling and issue mechanisms used in superscalars are extremely complex and usually not suitable for embedded processors due to area and energy efficiency reasons. On the other hand, the VLIW philosophy has found its way into several embedded digital signal processor architectures [10, 168, 176] due to its apparent simplicity.

Figure 2.4 presents a comparison between sequential and VLIW instruction schedules generated for a given piece of C code. The VLIW architecture in question allows only one memory access per cycle (assuming a single data memory with a single read/write port), but can simultaneously execute three arithmetic/logical/comparison/shift operations in three parallel issue slots. The sequential pseudo assembly code takes eight cycles in total to implement the C code fragment, whereas the VLIW schedule achieves the same in only five cycles. The first three add instructions for generating data memory addresses in the sequential code are packed in the same instruction word in the VLIW operation schedule (① in Fig. 2.4). The memory load instructions (② in Fig. 2.4), which use these computed addresses, can not be scheduled in parallel due to memory access restrictions. However, the first addition operation following the memory loads can be scheduled in parallel with the last load instruction (③ in Fig. 2.4). Thus, a total saving of three execution cycles w.r.t. the sequential schedule is achieved.

VLIWs are attractive design alternatives for target applications which contain high degrees of fine-grained parallelism. Still, the VLIW option must be weighed carefully against other possible performance enhancement techniques such as special instructions or hardware co-processors due to its high *design complexity*. Although VLIWs are not as complex as superscalar processors, designing them



**Fig. 2.4** Comparison between sequential and VLIW instruction schedules

from scratch still requires considerable design effort. The key challenge is to design a suitable optimizing compiler that can identify parallelism in an in-order instruction stream and effectively schedule it.

For embedded consumer electronic applications, the low *code density* of the VLIWs is another major concern. During instruction scheduling, a VLIW compiler may not find enough parallel instructions to fill each issue slot in an instruction word. Such unused slots are filled using NOP instructions (e.g. seven unused issue slots have been filled with NOPs in the VLIW instruction schedule shown in Fig. 2.4). This policy adversely affects code density.

To alleviate this problem, some VLIW processors like Philips Trimedia [176] apply code compression on the scheduled instruction stream. The compressed instruction words are decompressed using a special hardware unit after instruction fetch from the instruction memory. However, any such code compression technique further raises the design complexity of the corresponding architecture.

### 2.2.3.2 Exploiting Data Level Parallelism

*Single instruction multiple data (SIMD)* – originally described in Flynn’s famous taxonomy [60] of computer architectures – is a type of parallel computer organization where multiple processing elements simultaneously apply the same operation on several data elements. Unlike scalar architectures which process one single data item at a time, a SIMD computer contain instructions which can operate

on entire data vectors (i.e. one dimensional arrays of data items). The SIMD philosophy was the basis of most vector processing supercomputer architectures designed in 1970s and 1980s. Today, SIMD instructions are commonly used to exploit the data level parallelism in a variety of multimedia – especially video processing – algorithms. Two very common applications of SIMD instructions are computation of *sum of absolute differences (SAD)* and *sum of absolute transformed differences (SATD)* functions. Most video compression/de-compression algorithms use some variant of these functions for block matching in their motion estimation kernels. Similar examples can be found for audio and static image processing algorithms, too. Not surprisingly, a large number of desktop general purpose processors provide SIMD extensions to their normal instruction-sets (e.g. MMX, SSE and SSE-2 from Intel, AltiVec for PowerPC and 3DNow! for AMD) or use SIMD based *graphics processing units (GPUs)* (e.g. NVidia GeForce and ATI Radeon) for accelerating media processing and graphics rendering. Due to the recent convergence of various multimedia applications onto handled consumer electronic gadgets, SIMD instructions have also become very common in embedded DSP (e.g. C6000 series of processors from Texas Instruments [168] and Tigershark from Analog Devices [10]), and configurable processor cores (e.g. TenSilica Xtensa [182] and ARC Tangent [11]).

Although SIMD instructions for GPPs and GPUs can be as complex as supporting operations on entire data vectors, most embedded processor exploit simple sub-word parallelism through SIMD operations. This technique takes advantage of the fact that many multimedia applications use 8 bit or 16 bit wide data elements (e.g. image processing algorithms may use 16 bits per pixel) which can be easily stored as sub-words of 32 bit wide general purpose registers. Operations like simultaneous addition/subtraction of all the sub-words of two registers can be accomplished by simply stopping the carry propagation path between consecutive sub-words. An example of this is presented in Fig. 2.5 which shows a piece of C code that adds two arrays containing four 8 bit wide `char` elements (① in Fig. 2.5a). The sequential implementation of this C code (② in Fig. 2.5a) requires a loop construct containing six instructions per iteration. This amounts to the execution of a total of 24 instructions. The same example can be implemented using only four instructions which exploit the sub-word parallelism of the data elements. In this implementation, all the elements of the two input arrays – `a` and `b` – are loaded into the sub-words of two GPRs using normal memory load instructions (③ in Fig. 2.5b), added in parallel using the SIMD add operation (④), and stored back to memory using a normal memory write operation (⑤). The SIMD add has very little hardware overhead, because the four 8 bit adders can be implemented simply by deactivating the carry path of the normal 32 bit adder at the 8 bit sub-word boundaries. This example clearly demonstrates why SIMD instructions are so attractive for embedded ASIP architectures.

SIMD and VLIW are complementary paradigms for exploiting fine-grained, intra-task parallelism in programmable architectures. These two approaches are often combined together [10, 168] to take maximum advantage of available parallelism in target applications.

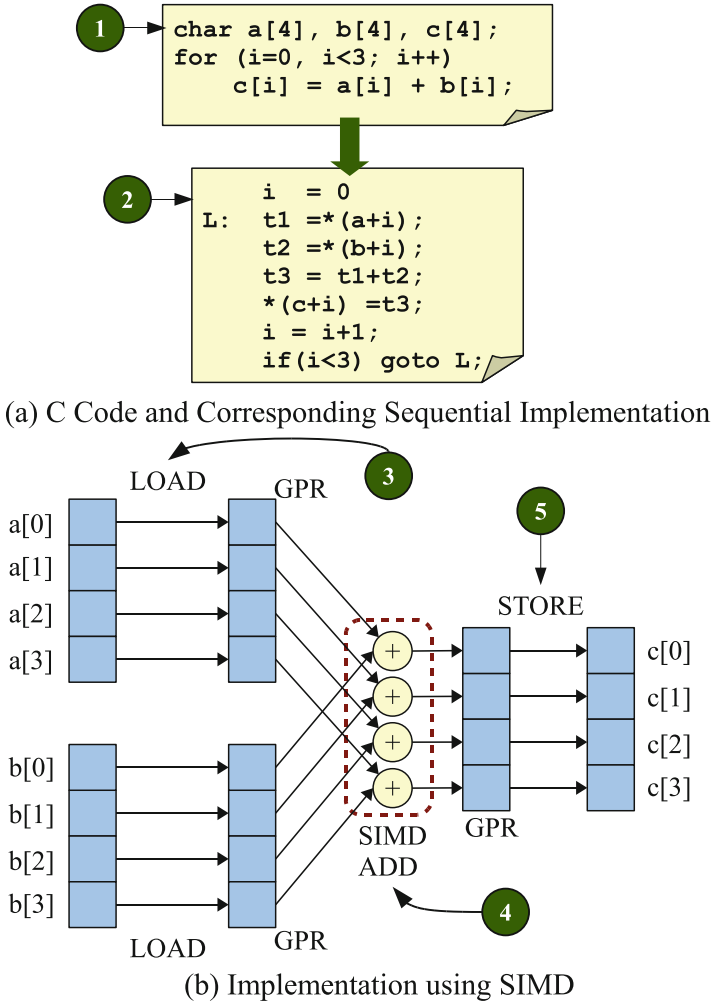


Fig. 2.5 Example of SIMD using sub-word parallelism

### 2.2.4 Hardware Acceleration Technologies

General purpose processors are often found inadequate to meet the performance and energy efficiency requirements of applications with high-computational complexity. As a consequence, designers have traditionally off-loaded computationally intensive tasks to customized hardware accelerators. This trend continues even today for ASIPs where specialized hardware accelerators continue to remain the primary means of performance optimization. The prevalent approaches for accelerator design are discussed in this section.

As a general rule of thumb, a large portion of the execution time of a program is spent inside only a few critical segments of code. Hardware accelerators are normally designed to speed-up execution of such critical segments. One can categorize various accelerator design techniques into two different approaches based on the *granularity* of these critical program segments. The first approach advocates migration of performance critical coarse-grained subtasks of the original target application to dedicated ASIC based *co-processors*. In a co-processor based system, the main-processor usually retains the control intensive software code of the original application and co-ordinates the execution of various computation heavy tasks on different co-processors. The controller commonly initiates the execution of a co-processor by initializing a number of interface registers, and then waits for the co-processor to return results after successful completion of the designated task. Translating a coarse-grained task to a hardware description is usually done using *high level synthesis (HLS)* [113]. A large number of commercial offerings of HLS tools are already available in the market [19, 36, 42, 61, 111, 165].

The second approach of accelerator design, which has been already mentioned several times in preceding sections, is to integrate application specific ISEs into the original processor core. An ISE combines program elements of the finest possible granularity, i.e. individual program statements, into hardware blocks. Readers may recall from Sect. 2.2.2 that ISEs are usually implemented inside a tightly coupled CFU within the base processor pipeline and are permitted to access the GPR file and base processor resources.

Examples of both of these approaches are presented in Fig. 2.6. Figure 2.6a shows a small code snippet from an implementation of the edge and corner detection algorithm *SUSAN* (an acronym for *smallest univalue segment assimilating nucleus*) [162]. The code snippet, encapsulated inside the `corner_draw` function, marks the detected edges/corners of a given image by black points. In the co-processor based implementation, the entire functionality of the code snippet is executed on a dedicated hardware block. The main processor initializes interface registers of the co-processor with the required execution parameters and then waits for the hardware accelerator to finish execution. The co-processor block directly modifies the shared memory regions holding the `corner_list` structure during its execution. The processor/memory/accelerator communications are done over a shared bus.

In the ISE based implementation, source line ① of Fig. 2.6a is implemented using a special instruction, `get_pixel_address`, inside the main processor's pipeline. `get_pixel_address` is allowed to access GPRs and main memory of the base processor core to finish execution, and does not need to go over a shared system bus to access the pixel data.

The ASIC based coarse grained co-processor design does not really fit well with the philosophy of the ASIP's. One of the major advantages of ASIPs over ASICs – added flexibility and programmability – is lost with the usage of coarse grained accelerators, although some amount of flexibility can be incorporated into co-processor based accelerators by using FPGA or eFPGA devices instead of ASICs. This is illustrated in Fig. 2.6b which shows a modified version of the `corner_draw`

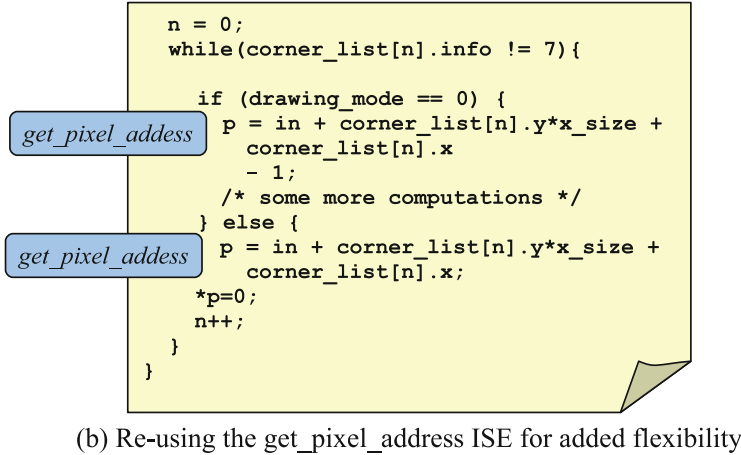
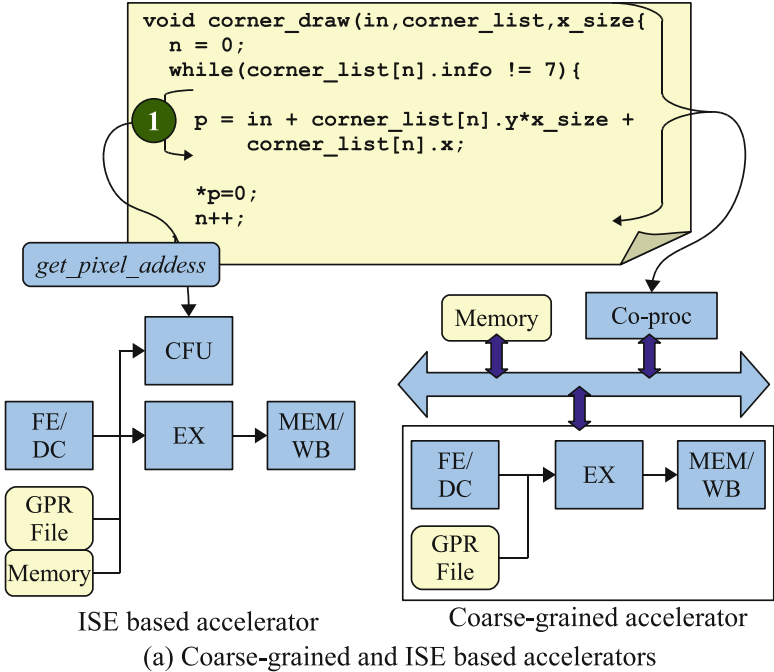


Fig. 2.6 Comparison of coarse-grained and ISE based hardware accelerators

function which can employ an enhanced drawing mode to mark edges/corners in black with white borders (instead of simple black points as in the original). This drawing mode can not be supported by the coarse grained co-processor and has to be executed in software. However, the same `get_pixel_address` instruction can be used to provide some hardware acceleration for even this modified drawing mode.

The usage of co-processors and ISEs for hardware acceleration are not mutually exclusive. For example, Sun et al. [161] proposes a technique to find the right balance between ISEs and co-processors during ASIP customization. Similarly, the GNSS receiver ASIP presented in [89] uses ISEs, and ASIC and eFPGA based co-processors for hardware acceleration.

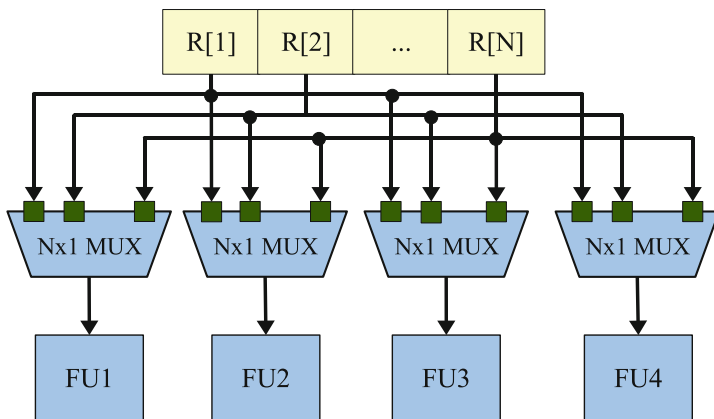
One interesting commercially available co-processor design technology is offered by the *Cascade* technology from Critical Blue [50]. Cascade claims to generate *programmable co-processors* with high degree of instruction level parallelism for accelerating computation intensive, coarse grained code segments. Designers can even specify some efficient application specific *functional units (FUs)* that can be embedded in the co-processor hardware. Still, the usage of co-processors for hardware acceleration is more in the domain of system design than processor design, and is beyond the scope of this work. For the rest of this book, we will concentrate on the issues concerning ISE based accelerator generation.

### 2.2.5 Register File Architecture

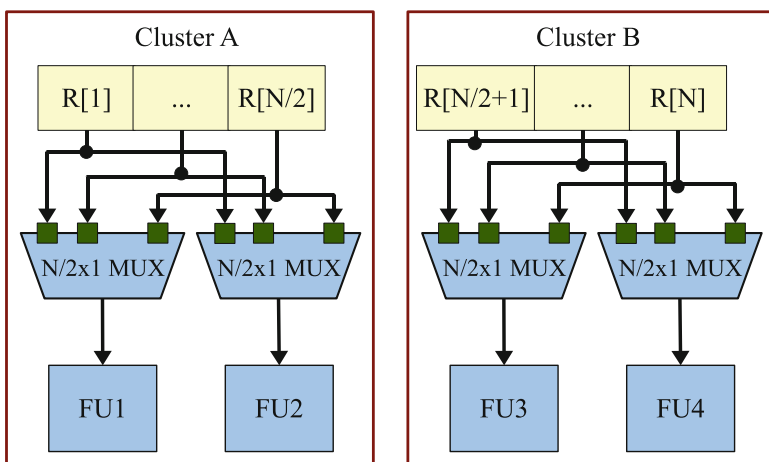
The design of the register file needs a special mention within the ASIP design context, because it has special ramifications for ISEs and instruction level parallelism, among other things. We have already mentioned that most ASIP architectures today are built around pipelined RISC base processors. As a natural consequence, an ASIP usually contains a GPR file which is used by both base processor instructions and ISEs. However, unlike the unary or binary BPIs, most ISEs require more than two input operands and one output operand from the GPR file. These extra operands are often provided via special registers which do not appear in the instruction encoding. Since these special registers are invisible to the BPIs, they are usually placed inside the CFU instead of the base processor pipeline. The usage of these special registers for ISE design will be revisited again in Chaps. 6 and 8.

The GPR file design of an ASIP might itself be different from those used in general purpose processors. One common GPR file design strategy found in many embedded VLIW architectures [58] is called clustering. VLIW architectures need several RF input/output ports due to multiple parallel FUs. However, the area and latency of a RF increases with increasing number of ports. In a clustered architecture, this problem is solved by dividing the monolithic GPR file into several smaller register files. Each smaller RF is grouped together with a set of FUs to form a *cluster*. Each FU from a particular cluster can access the entire RF of the same cluster, but is only granted limited access to registers from other clusters through restricted inter-cluster interconnection networks.

The advantages of a clustered register file over a non-clustered one can be easily understood from Fig. 2.7. Figure 2.7a shows a VLIW with four parallel FUs and a monolithic register file with  $N$  registers. Figure 2.7b shows the same architecture divided into two clusters – each containing  $N/2$  registers. For the sake of simplicity,



(a) Register Ports in a Non-clustered VLIW



(b) Register Ports in a Clustered VLIW

**Fig. 2.7** Register file ports and data forwarding architectures in clustered and non-clustered VLIWs

no inter-cluster communication network has been shown here. Each read port of the non-clustered register file requires a  $N \times 1$  vector MUX, whereas the same in the clustered version requires only a  $N/2 \times 1$  vector MUX. For typical values of  $N$ ,<sup>5</sup> the savings in register file area and latency due to clustering can be considerable.

<sup>5</sup>Typical values of  $N$ , i.e. the size of the GPR file, are usually 8, 16 or 32 for most embedded processors.

Unfortunately, the improvements due to clustering come at the cost of register access restrictions. As an example, let us suppose that all the FUs in cluster B in Fig. 2.7b in a particular cycle need to read all their sources from cluster A. Due to the limited number of inter-cluster interconnection paths, all these read operations can not be supported in a single cycle. This issue can be resolved either by inserting special move operations which copy source registers from cluster A to cluster B, or by scheduling some of the competing instructions in later cycles. Both of these solutions lead to a loss of execution cycles. Minimizing such performance penalties due to clustering is an active area of compiler research.

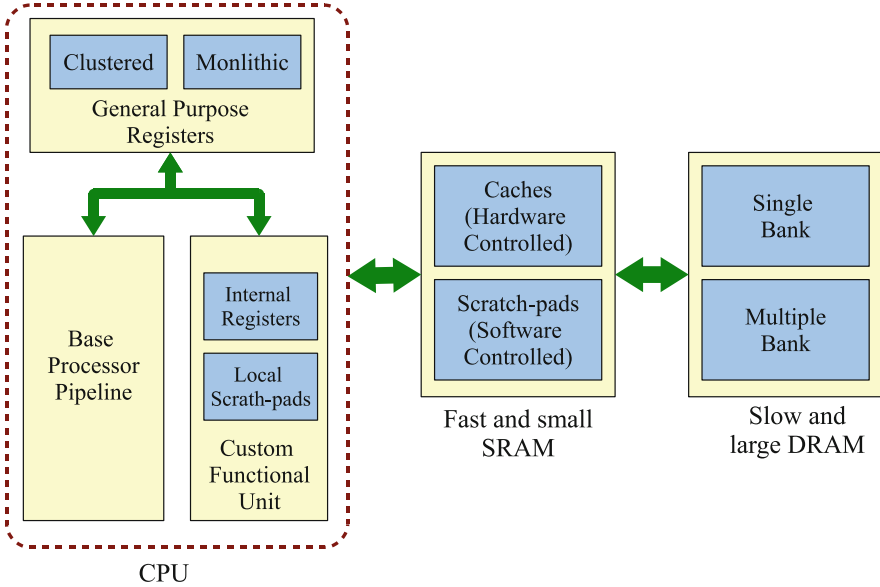
The concept of clustering can also be used to provide multiple input/output GPR operands to ISEs in single issue RISC processors. This technique will be described in detail in Chap. 9.

### 2.2.6 Memory Subsystem Design

Nowadays, the memory subsystem in any computer system constitutes a primary bottleneck w.r.t. the two most important design criteria – power consumption and performance. It is the main bottleneck for system performance due to the continually increasing speed-gap between memories and processors. For battery driven embedded applications, a more urgent concern is the energy efficiency of memory devices. Memory accesses account for a vast bulk of power consumption in embedded systems. For example, in real time signal processing applications – such as speech processing in a mobile phone – 50–80% of the power is consumed in the traffic between the CPU and the off-chip memory [151]. Naturally, the memory subsystem is one of the most important architectural concerns for ASIP design.

The performance of the memory subsystem can be optimized using a variety of *hierarchical memory* configurations. The most common configuration preferred in almost all existing desktop GPP architectures consists of one or more levels of fast, hardware controlled cache memories between the processor and the main memory [125]. Caches are generally implemented using SRAMs which are usually far faster (by a factor of 10 or more) and more expensive (by a factor of 20 or more) than the DRAMs used for main memories. Due to the cost of SRAMs, they are normally far smaller in capacity than main memories and are used to store only those memory objects (instructions or data) which are most likely to be accessed during the execution of a program. In desktop systems, caches are used for lowering the *average memory access latency* whereas the main memory is used for building capacity.

Besides the standard memory hierarchy design for GPPs, several other memory subsystem alternatives common in embedded systems can also be employed for ASIP architectures. This section briefly discusses these alternatives. A pictorial overview of these alternatives is presented in Fig. 2.8.



**Fig. 2.8** Options for ASIP register file and memory hierarchy design

### 2.2.6.1 Software Controlled Caches

In desktop systems, the policies that govern the placement and allocation of instruction and data objects in caches are implemented in hardware. For embedded systems, the area and energy consumption overhead of such hardware cache controllers makes caching quite unattractive. The other alternative is to use fast SRAMs as *scratch-pads* where the placement of memory objects is controlled by software – usually by the compiler. Due to the absence of complicated cache controllers, scratch-pad based memory systems can be far smaller in area and far more energy efficient than caches. For example, the study by Banakar et al. [22] demonstrates that a scratch-pad memory is around 34% smaller and 40% more energy efficient than a cache of the same capacity.

Scratch-pads are promising alternatives of caches for ASIPs. Unlike a general purpose processor designed to run a large variety of applications with varying memory footprints, memory access patterns of target applications are known in advance for an ASIP. A suitable scratch-pad allocation policy – selected by using such a priori knowledge – can even outperform hardware caching. For example, Banakar et al. [22] observed a 18% reduction in cycle count (compared to a cache) using a simple knapsack based static allocation algorithm. It is, therefore, not surprising that the compiler assisted software scratch-pad allocation has become an area of active research [22, 124, 170].

Many embedded applications contain data arrays which are initialized once and are not written thereafter during program execution. Some common examples of these are the coefficients used in digital filters [135] and the so called S-Boxes used in private key block cipher algorithms [145]. Such arrays are also good candidates for placement inside scratch-pads. Often enough, application kernels, which make heavy use of such data arrays, are accelerated using ISEs. In such cases, the corresponding scratch-pads can be placed directly inside the CFU. This issue will be revisited again in Chaps. 6, 7 and 8.

### 2.2.6.2 Multiple Memory Banks

Another possible architectural feature for ASIPs is to add multiple memory banks. In many signal processing applications, elements from multiple arrays have to be processed simultaneously. The classical example of this is found in digital FIR or IIR filters where the elements of input sample arrays (in case of IIR filters also previously computed output sample arrays) are multiplied with elements of coefficient arrays. Efficient implementations of such filters require instructions which can simultaneously access both the sample array and the coefficient array. For supporting such filter applications, many DSP architectures contain dual X-Y memory banks, and provide MAC or multiply instructions which can use two memory operands – one from each memory bank.

### 2.2.7 Arithmetic Data Types

A very important consideration for ASIPs is which data types to support for a given target application. This design decision determines the bit-widths of GPRs, FUs and data bus in an architecture. General purpose desktop and embedded processors use integer data types for representing positive and negative whole numbers, and floating point data types for real numbers. Normal practice is to use 32 bit wide integers, and IEEE 754 [69] standard compliant single precision (32 bit), double precision (64 bit) or extended precision (more than 64 bit) floating point numbers. These values can be entirely different for an ASIP.

The ideal size of the integer data type for an ASIP depends on the target application. As an example, recall from Sect. 2.2.3 that many multimedia programs use only 8 or 16 bit data elements. The integer data type, as well as the FUs and GPRs, can be appropriately resized for such algorithms. Operations on wider integers can be emulated in software in such architectures.

A floating point unit can be optionally incorporated in an ASIP for floating point multimedia and signal processing applications. Still, due to the design complexity and area/power consumption overheads of a fully IEEE 754 compliant FPU, only

limited and absolutely necessary functionalities are included in such cases. A more widely accepted practice in embedded system domain is to represent fractional numbers using *fixed point data types*. A fixed point data type, as the name suggests, uses a fixed number of bits for representing the fractional part of a real number. This is in contrast to a floating point type which uses a mantissa and an exponent to represent a real number. A floating point type can represent a far wider range of values than a fixed point type of the same width, but also has far higher implementation complexity.

A fixed point type is essentially an integer type scaled down by a specific factor which is determined by the number of bits in the fractional part. As an example, let us consider a 8 bit wide signed fixed point type with 3 bits reserved for the fractional part. The scaling factor in this fixed point type is  $2^{-3}$ , (or, in decimal, 0.125) and all the numbers represented in this type are multiples of this scaling factor. The range of values for a 8 bit wide signed integer lies between  $-128 (= -2^7)$  to  $+127 (= 2^7 - 1)$ . The range of values of the corresponding fixed point type, which lies between  $-16 (= \frac{-2^7}{2^{-3}})$  and  $-15.875 (= \frac{2^7-1}{2^{-3}})$ , can be derived by simply scaling the ranges of the original integer down by the scaling factor. As a consequence of this, fixed point operations can be simply implemented in an ASIP using slightly modified integer arithmetic FUs. The optimal bitwidth of the fractional part can be determined by profiling the real numbers used in the target application.

## 2.2.8 Partially Reconfigurable ASIPs

In the past few years, one of the most interesting developments in the ASIP design landscape has been the emergence of partially *reconfigurable ASIPs* (*rASIPs*). An *rASIP* combines a processor's data-path with a tightly coupled reconfigurable fabric. While the base instruction-set of the processor provides flexibility in software, the reconfigurable fabric offers flexibility in hardware.

The Stretch [158] architecture is a good representative example of *rASIPs*. It embeds an *instruction-set extension fabric (ISEF)* inside an Xtensa LX processor (which by itself is configurable!). The ISEF is a programmable fabric that offers designers the opportunity to implement computation intensive data-paths. Re-tuning the architecture for newer applications, or bug-fixing older applications can be simply achieved by re-programming the ISEF. Other prominent *rASIPs* from industry and academia include Xilinx Microblaze [179], Altera NIOS [5], ADRES [110], MOLEN [172] etc. Most of these architectures are very close to the configurable processors in their design philosophy because they also try to lower the verification effort through their pre-designed base-architectures and reconfigurable data-paths.

Although *rASIPs* represent a promising development for future SoC designs, their acceptance is still very limited due to the high area and energy consumption generally associated with reconfigurable fabrics. A complete discussion of design

issues related to rASIPs is beyond the scope of this book. Interested readers may refer to [40] for a more in-depth overview of reconfigurable architectures. However, for the rest of this work, we will only confine ourselves to fixed ASIPs without reconfigurable fabrics.

### 2.3 Cross Cutting Issue: Designing Optimizing Compilers

In the preceding sections of the current chapter, several standard and application specific customization options for ASIPs have been discussed. We will like to conclude this chapter by discussing a very important cross-cutting issue – that of designing optimizing compilers which can take advantage of the architectural alternatives discussed so far. Today, the embedded system design community overwhelmingly uses high level programming languages like C/C++ for system specification, and depends on software compilers to convert such specifications to the final executable representations. Naturally, the performance and energy efficiency of the executable is largely determined by the optimizations performed during compilation.

A normal *high level language (HLL)* compiler consists of a generic front-end and a target processor dependent back-end (Fig. 2.9). The front-end converts a

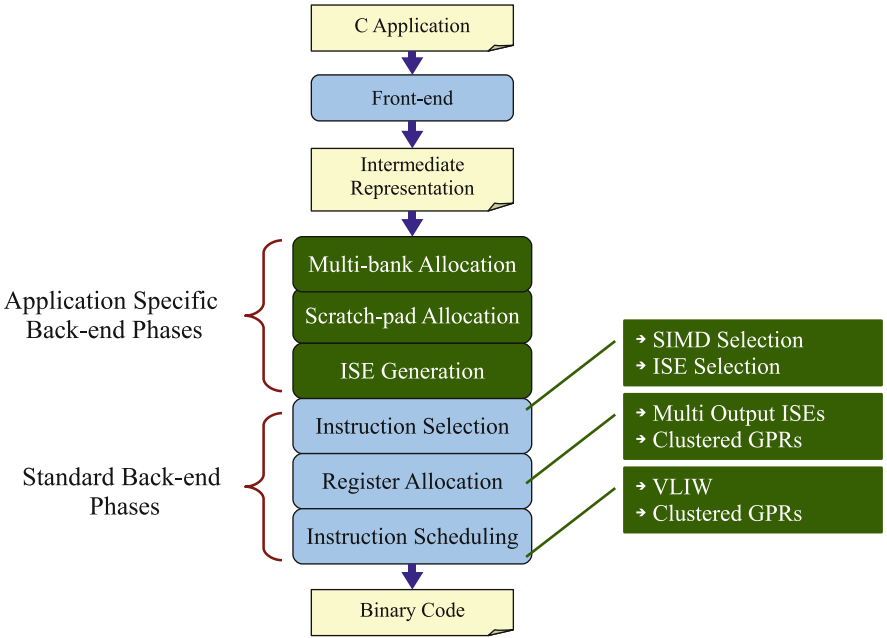


Fig. 2.9 Optimizing compilers for ASIPs

HLL application into an *intermediate representation (IR)* and applies several target independent high level optimizations on it. Interested readers may consult the classic compiler construction text by Aho, Sethi and Ulman [4] for a detailed treatment of the front-end design issues. However, for ASIPs, the back-end optimizations have a higher relevance.

The standard compiler back-end for any processor – embedded or otherwise – consists of three phases – *instruction selection*, *register allocation* and *instruction scheduling*. Instruction selection converts the IR representation of an application to a sequence of processor instructions, register allocation maps program variables to processor registers, and instruction scheduling reorders the sequential instruction stream so as to avoid pipeline hazards and maximize parallelism. Depending on the selected architectural alternatives, an ASIP's compiler may have several additional back-end phases, or may need to incorporate special optimization capabilities in the traditional back-end steps. This is due to the fact that most of the architectural customizations introduced in the previous sections can not be utilized properly without appropriate compiler support. For example, the performance of a processor with scratch-pads or multiple memory banks largely depends on how the compiler allocates program variables to them, and clustered VLIWs require special instruction scheduling and register allocation techniques to minimize the performance penalties incurred due to clustering. Similarly, addition of SIMD operations to a processor's ISA necessitates changes in the instruction selector.

Another important task of an ASIP's compiler is ISA customization, i.e. the process of automatically adding application specific ISEs to an ASIP core. It involves identification of promising special instructions from a given application's source code (called *ISE generation*), integration of the identified instructions into the target processor's architecture (called *ISE implementation*), and insertion of the special instructions into the final executable (called *ISE utilization*). The ISE generation step is a computationally complex software-hardware partitioning problem and is extremely difficult to solve manually. Automated ISE generation can be integrated as a separate back-end phase to the ASIP compiler. These techniques for ISA customization will be discussed in detail in Chaps. 6–9.

ISE utilization is another complex problem which requires modifications in register allocator and instruction selector. However, this problem is less urgent than ISE generation, because it can be partially alleviated by manually inserting the special instructions into the C code using assembler functions. In practice, this policy works fairly well because the special instructions are usually needed for small fragments of computation heavy code. The rest of the application is compiled to the instruction-set of the base processor using standard back-end techniques.

Application Analysis Tools for ASIP Design  
Application Profiling and Instruction-set Customization  
Karuri, K.; Leupers, R.  
2011, XXII, 232 p., Hardcover  
ISBN: 978-1-4419-8254-4