RECONFIGURABLE INSTRUCTION-SET APPLICATION TUNING FOR DSP *

B. MESMAN

Philips Research Laboratories Eindhoven, Prof. Holstlaan 4
5656 AA Eindhoven, The Netherlands

Department of Electrical Engineering, Eindhoven University of Technology
Den Dolech 2, Postbus 615, Eindhoven, The Netherlands

Q. ZHAO

Department of Electrical Engineering, Eindhoven University of Technology

N.BUSA

Philips Research Laboratories Eindhoven

K. LEIJTEN-NOWAK

Philips Research Laboratories Eindhoven

Department of Electrical Engineering, Eindhoven University of Technology

Received (received date)
Revised (revised date)
Accepted (accepted date)

In current System-on-Chip (SoC) design, the main engineering trade-off concerns hardware efficiency and design effort. Hardware efficiency traditionally regards cost vs. performance (in high-volume electronics), but recently energy consumption emerged as a dominant criterion, even in products without batteries. "The" most effective way to increase HW efficiency is to exploit application characteristics in the HW. The traditional way of looking at HW design tends to consider it a time-consuming and tedious task, however. Given the current lack of HW designers, and the pressure of time-to-market, clearly a desire exists to fine-balance the merits and effort of tuning your HW to your application. This paper discusses methods and tool support for HW application tuning at different levels of granularity. Furthermore we treat several ways of applying reconfigurable HW to allow both silicon reuse and the ability to tune the HW to the application after fabrication. Our main focus is on a methodology for application-tuning the architecture of DSP data-paths. Our primary contribution is on reusing and generalizing this methodology to application-tuning DSP instruction sets, and providing tool support for efficient compilation for these instruction sets. Furthermore, we propose an architecture for a reconfigurable instruction decoder, enabling application tuning of the instruction set after fabrication.

1. Introduction

*This work is partly supported by the IST-2000-30026 project, Ozone.
The decreasing feature sizes realizable by modern silicon fabrication technology allow an increasing number of processors, memories, and communication busses to be integrated on a single chip. Because these components have to be designed and/or programmed, the technology developments also yield an increase in the design effort. This is just one of the reasons why programmable components are so popular in System-on-Chip design. Portability and a growing need for support of different (eg. multi-media) standards also inspire designers to consider components that are programmable to some degree. On the other hand, programmable, and especially general-purpose components, are often not able to meet requirements regarding performance, cost, and power consumption. As a result, System-on-Chip designers look for opportunities to implement computationally intensive parts of the application in dedicated hardware blocks (at a wide range of granularity), or using application (-domain) specific instruction-set processors (ASIPs). As a result of the combined trend of reuse (flexibility) and hardware efficiency, the heterogeneity in System-on-Chip components and architectures will increase. An example is the introduction of the notion of reconfigurability at different levels of granularity. This paper discusses some commercial and research developments in applying (reconfigurable) application-specific hardware, and is organized as follows. Section discusses the application of HW blocks as co-processors. A way of integrating coarse-grain blocks in the data-path of a VLIW processor is treated in section . Fine-grain HW acceleration is discussed in section 3. Section 9 introduces a code size efficient reconfigurable instruction set, and discusses some problems related to compilation. This sets the stage for the Static Resource Model of an instruction set, treated in section 10. Besides compilation, the SRM model is the basis for instruction-set design in section 14. In section 16 an architecture is proposed for a reconfigurable instruction decoder that matches the SRM model and methodology. The novel technical contribution of our work concerns the SRM model, and correspondingly, compilation, instruction-set design, and a reconfigurable instruction decoder. Conclusions are discussed in Section 17.

2. HW acceleration with dedicated co-processors

The most conventional way of using application specific hardware is to connect an application specific block to a system bus shared with other processors, memory, etc. A typical grain size is that of a motion estimator in an MPEG decoder. The advantage of this approach is that it is relatively easy to integrate dedicated hardware in a plug-n-play like fashion. The disadvantage is that the system bus is burdened with heavy data traffic from and to the application specific block. This reduces the predictability and the performance of the system.

In a reconfigurable context, this block usually consists of FPGA. The number of bits required to configure the block is in the order of 500k bit, and typically takes a few milliseconds to configure at a typical clock frequency of 50-100MHz.
3. HW acceleration with coarse grain functional units

A more convenient way of synchronizing and data communication is achieved by embedding a (coarse grain) hardware block in the data path of a processor. The instructions of the embedding processor controls via its registers the data traffic to the hardware block. This also reduces memory space required to buffer data traffic waiting for access to the bus in the co-processor architecture from section . We discuss in this section an approach based on a VLIW architecture with an embedded FPGA block. In more detail an approach is treated to (hierarchically) embed a VLIW processor.

Mihai Sima et. al.\(^\text{19}\) designed a Reconfigurable Functional Unit (RFU) for a Trimedia/CPU64 VLIW datapath. A 16-cycle coarse-grain 8-point IDCT operation has been configured on the above-mentioned RFU. The coarse-grain operation is configured at bit-level and mapped on an ACEX EP1K 100 FPGA from Altera, which is embedded in the VLIW processor’s datapath. In this way, adders and multipliers bit-level architectures are re-configured any time a new coarse-grain operation is configured. Usually, those basic FUs can be re-used and the reconfiguration of their bit-level architectures is not strictly necessary. In this set-up, reconfiguration time is 10-300 clock cycles in a 100MHz processor.

3.1. HW acceleration by embedded instruction-set processors

Busa et al.\(^\text{7}\) propose a synthesizable VLIW architecture (Figure 1) with a coarse-grain functional unit that itself takes the form of a VLIW processor. A substantial speed-up can be achieved as well as a reduction in code size, as will be shown. Secondly, the input and output operands to and from the coarse-grain unit can be individually controlled to obtain advantageous signal lifetimes, thereby reducing the pressure on the datapath registers.

![Fig. 1. A VLIW data path with a coarse grain application-specific unit](image)

The internal schedule of the coarse grain FU will be partially taken into account while scheduling the application. In this way, a FU’s internal schedule could be con-
sidered as embedded in the application’s VLIW schedule. The embedding processor controls only the timing of the I/O operations, thereby limiting the instruction width of the embedding processor. The flexibility available during scheduling I/O operations depends on the holdability of the coarse grain FU; If the coarse-grain FU can be held (frozen) then the I/O operations can be further “stretched” away from each other, as in Figure 2(b), to provide or withdraw data in a ”just in time” fashion. If the coarse-grain FU cannot be held, the timing relations between the I/O operations are fixed, modeled by the additional sequence edges in Figure 2(c).

In Figure 3, we compare the execution length and code size for architectures with and without a (reconfigurable) unit. In the give example, we consider an application containing two critical loops. Each loop requires a different configuration for efficient acceleration. The RISC-like architecture (a) does not provide any hardware acceleration at all. The microcode width is very small, but the execution of both critical loops is long, with a consequent decay in terms of performance and code size. On the contrary, a ”flat” VLIW processor, which contains many fine-grain FUs, will accelerate both loops at the cost of a large instruction width (d). A fixed coarse-grain VLIW processor (b) will provide hardware support for the acceleration of one of the two loops, while keeping the microcode width moderately small. Finally, a reconfigurable coarse-grain VLIW processor will accelerate both loops.

The embedded hierarchical VLIW processors contain a local internal microcode for the execution of the internal schedule of the coarse-grain operation. For a typical coarse-grain operation such as an 8-point DCT, the microcode is about 20 instructions by 25 bits. Note that the “flat” processor in Figure 3 (d) requires several 100 instructions by 25 bits to control the added resources. Most of these bits represent NOPs because the added resources are only used in the critical loops.

In case of reconfigurable coarse-grain units, the configuration latency of the internal microcode varies from 2 to 10 cycles. The controller microcode as well as the RTL VHDL for the RC-FU have been generated using the architectural synthesis
tool A—RT Designer$^1$.

Fig. 3. The compared VLIW datapaths, and their relative controller’s microcodes, as compiled for an application containing two different DSP loops

4. HW acceleration with fine grain functional units

The VLIW architecture shown in Figure 1 may also contain fine grain functional units that perform operations in a few clock cycles. The motivation for these units is that hardwired circuitry provides considerable advantages in speed and power consumption for very specific and small functions. Typical examples are algorithms from cryptology, which contain rather simple bit-level operations, illustrated in 4. These operations would otherwise be implemented with about 8 RISC-instructions
RECONFIGURABLE INSTRUCTION-SET APPLICATION TUNING FOR DSP

for each result bit.

![Bit-level operations diagram](image)

Fig. 4. Bit-level operations

Another typical application for fine-grain FUs is in small conditional constructs. Using the architectural synthesis tool A—RT Designer 1, a GSM viterbi decoder, containing a time critical add-compare-select operation, is implemented on a VLIW processor. The default architecture is depicted in Figure 5. The controller, including the program code, is responsible for 70% of the power consumption. This is largely due to the amount of branching required to perform the add-compare-select.

![GSM viterbi decoder: default solution](image)

Fig. 5. GSM viterbi decoder: default solution

The add-compare-select operation is performed on a dedicated unit in the architecture in Figure 6, resulting in a speed-up with a factor 5 and a power reduction
with a factor 20.

![Program code](image)

Fig. 6. GSM viterbi decoder with an add-compare-select unit

In this approach, the designer has the responsibility to organize his source code with function calls as API to use the hardwired unit. In order to identify candidates for hardwiring, the designer is supported with detailed schedule feedback concerning the resource and register occupation in the time-critical loops.

5. Parameterizing the number of functional resources

![Flowchart](image)

Fig. 7. Parameterizing the number of Functional Units in the High-Level Synthesis of ASICs

Besides offering hardwired functional units, also the number of functional units in the data-path can be adapted to the application. In the context of High-Level Synthesis, a rather trivial 'best practice' approach is illustrated in Figure 7: The designer analyzes the time critical loops for bottlenecks in the availability of processor resources required to obtain the target schedule throughput. These bottlenecks can be identified by scheduling the loop and examining the resource utilization. In
Figure 6 for example, the resource add-compare-select (acs\_asu\_1) is 83% active, suggesting that this unit may represent a bottleneck for the required performance. This bottleneck is then relieved by allocating additional resources in Figure 8; Adding 3 more add-compare-select units results in an additional speed-up with a factor 5 and a power reduction with a factor 3.

In this way an ASIC is designed from a minimalistic viewpoint in order to optimize the design criteria (power consumption, area) while just satisfying the performance requirements. Often a high-level power estimator is used to identify bottlenecks for power consumption and to choose among viable alternatives when allocating additional resources. During High-Level Synthesis also the register and communication infra-structure of the data-path is parameterized. These are for a large part responsible for the area and power consumption. Since the functional resources are responsible for the performance, their parameterization is considered the primary objective during High-Level Synthesis. It is also the main focus of this section, because this 'best practice' approach will be abstracted in section 14 to perform instruction-set design.

![Diagram of EXU with activ, area, power values]

**Fig. 8.** GSM viterbi decoder with four add-compare-select units

A quick & dirty alternative to scheduling is available for performance analysis on critical loops in Figure 7. It uses a well-known performance bound based on available processor resources, which is explained next.

When applying software pipelining techniques to loop kernels, the *initiation interval* \((II)\) is an important criterion for measuring the performance. An \(\text{II}\) is the period between the start times of the execution of two successive loop-body iterations. The
minimum initiation interval (MII) is the lower bound of the II. The MII can be
determined either by a critical resource that is fully utilized or a critical chain of
dependencies running through the loop iterations. One lower bound, the resource
constrained MII (ResMII), is derived by calculating, in total, the usage requirements
for each resource imposed by one iteration of the loop.

Suppose a loop containing 14 add operations is mapped on a data path contain-
ing three adders and each adder takes one clock cycle to execute. Then we need at
least \( \left\lfloor \frac{14}{3} \right\rfloor \) = 5 clock cycles to execute the loop iteratively. By doing this calculation
for every available resource, we obtain the lower bound ResMII on the initiation
interval II of a pipelined schedule of the loop. The general experience is that this
bound is very tight. The lower bound indicates critical functional resources in the
data path. Suppose in the example above that the performance requirements dictate
a new loop iteration to start every 4 clock cycles. Then the lower bound of 5 clock
cycles indicates that at least one additional adder should be allocated. This lower
bound estimate can therefore be used for bottleneck identification. The method of
estimating the II and allocating more critical resources is extended in section 10 for
the design of (reconfigurable) instruction sets.

5.1. IS-Application tuning in the Concise architecture

In a method is proposed to integrate a reconfigurable HW acceleration unit
in the data path of a Risc microprocessor, see Figure 9. The instruction set of
this microprocessor has some room for additional (application-specific) instructions
performed on the RFU, which is placed next to the ALU. This provides a practicable
path for integration, but restricts the flexibility in using the RFU: Only operations
with two input operands and one output operand can be accelerated on the RFU.
This requirement matches particularly well with bit-level operations in cryptology
(see Figure 4). Several algorithms are proposed to automatically identify suitable
targets for HW acceleration, resulting in a speed-up of 10-60% on the complete
application for several DSP benchmarks. The RFU has a configuration memory in
the order of 5kbit, so reconfiguration time can be neglected.
6. IS-Application tuning in the Real architecture

The Real architecture, depicted in Figure 10, contains a data-path with two address computation units (ACUs), two data memories, two multipliers, and two ALUs. The use of special-purpose registers (e.g., the accumulator registers) and dedicated communication restricts the control of the data-path to 96 bits. The 16 and 32 bit instructions encode a limited but sufficient subset of these (96 bit) control words to efficiently support the largest part of the application and obtain a small program code. Some of the critical DSP loops may however only be scheduled efficiently when the full range of control over the data-path is available. For these loops, 96-bit application-specific instructions (ASIs) are available in a reconfigurable look-up table (128 words). The Real instruction-set is very efficient with regard to code size, but the irregular instruction set and data-path is not a convenient compiler target, as will be explained in the next section.

![REAL Processor Architecture](image)

Fig. 10. The REAL processor architecture

7. The SRM model

Processes like the Real (section 9) encode instructions in such an efficient way that the resulting instruction sets show a very irregular structure. This adds the necessity to ‘recognize’ in the application, instructions supported in the instruction set. Often this task of recognition (code selection) is performed by the programmer himself, either by writing assembly or with the use of APIs in the source code to call
the dedicated instructions or hardware. Both require a lot of source code rewriting and low-level programming, and both are detrimental for the maintainability and portability of the code.

![Diagram](image)

Fig. 11. Instruction selection prior to scheduling may yield inferior results

Alternatively, a compiler can be used. This compiler contains a code selection phase that recognizes valid instructions in a Data Flow Graph (DFG), often using pattern matching and graph covering techniques. An example is given in Figure 11. This task makes the compiler more complex, and introduces the issue of phase coupling: On the one hand, if scheduling is performed first, the available instructions may not be able to implement the schedule. On the other hand, if code selection is performed prior to scheduling, the optimal schedule can easily be eliminated as a result of the choices made during instruction selection. The code selection made in Figure 11 for example, eliminates the optimal schedule depicted in Figure 12. The problem of phase coupling is held partly responsible for the considerable overhead in both schedule length and code size (reported in the order of 800% of compiler generated code compared to manually written assembly for ASIPs. Optimal schedules can be obtained using more exact methods. In for example, an Integer Linear Programming (ILP) approach is proposed for the combined problems of code selection and scheduling. The resulting ILP instance necessarily contains many decision variables, causing run-times in the order of minutes or hours.

It should be noted however that the mentioned graph covering methods are sufficiently general to deal with a wide range of tricks to keep instruction widths small. These tricks include:

- The use of macro instructions. Similar to CISC instructions, they efficiently encode frequently occurring sequences of operations, like the well-known multiply-accumulate instruction.
- Special-purpose registers or restricting the operand register addressing range per operation.
• Restricting opcode bits by allowing only certain combinations of operations to be executed in parallel.

We will now propose a more satisfying solution for the last mentioned trick (the trick illustrated in the example in Figure 11). We hope to extend this idea to the other main encoding tricks in future research.

The solution is offered by the Static Resource Model (SRM). The SRM approach targets the instruction sets that are minimized by restricting certain combinations of operations that the data path can execute in parallel. Instruction sets in this class, which contains e.g. the so-called issue slot machines, can be modeled in terms of virtual resources, easily interpreted by classic resource constrained schedulers such as the popular list-scheduling algorithm. This is illustrated in Figure 12 for the example in Figure 11. The instructions in the instruction set IS have been augmented by the use of the virtual resources \(\{\text{ld}, \text{mul}\}\), \(\{\text{mul}, \text{shl}\}\) and \(\{\text{shl}, \text{add}\}\), which are abbreviated as \(LM, MS\) and \(SA\). We have one instance available of each virtual resource. Each operation uses all the virtual resources that it is contained in, e.g., operation \(\text{mul}\) uses virtual resources \(LM\) and \(MS\) simultaneously. In this way, the \(LM\) resource for example models the instruction set constraints that the \(ld\) and \(mul\) operations may not occur simultaneously. For this so-called static resource model, a list scheduler generates the schedule on the right hand side of Figure 12. The reader can verify that the operations at each clock cycle can be implemented with instructions from the original instruction set IS. The resulting schedule has length 5, whereas the schedule in Figure 11 has length 6. This example demonstrates that better schedules can be obtained using a static resource model instead of covering the DFG with valid instructions. Because most instruction selectors target instruction sets with a number of tricks (mentioned above), little comparison can be made to 'similar' methods. We can mention that the example used in the integer linear programming approach in \(^{11}\) has been scheduled optimally using the SRM model approach in only a fraction of a second. The Trimedia scheduler \(^{11}\) uses a form of consistency checking when scheduling an operation during at a certain clock cycle. A bipartite matching algorithm is used for that purpose. It works for
cycle-based schedulers and requires adaptation of the scheduling algorithm with a bipartite matching algorithm.

The main advantages of the SRM model are the following:

- **Run time.** The SRM has to be computed only once. On the other hand, instruction selection has to be performed for each basic block of operations. Consistency checking\(^\text{11}\) has to be performed for each operation.

- **Schedule freedom.** The scheduler is not restricted by a specific code selection and therefore has more opportunity to minimize register requirements.

- **Pipelined schedules.** The scheduler also has the opportunity to produce loop-pipelined schedules (also called loop folding or software pipelining). In the DSP world this is a must to exploit instruction-level parallelism.

- **No additional compiler step is required.**

- **Any resource-constrained scheduler can be used without adaptation.**

Our approach is motivated by the observation that both resource constraints and instruction set constraints can be expressed by inequalities. For example, if an architecture contains two ALUs and each ALU can be used as an adder or a subtractor, then the resource constraints can be expressed by the following inequality: \( N(ALU) \leq 2 \), which is equivalent to \( N(A) + N(S) \leq 2 \), where \( N(A) \) and \( N(S) \) denote the number of add and subtract operations allowed to execute in parallel. Any schedule satisfying at any time the above inequality indicates a valid resource usage. Similarly, if an instruction set contains instructions \([\text{add, add}], [\text{add, sub}]\) and \([\text{sub, sub}]\), the operation type usage can also be expressed as an inequality: \( N(\text{add}) + N(\text{sub}) \leq 2 \), assuming any subinstruction is also a valid instruction.

![Instruction set IS1 expressed as points in the operation type space](image_url)

Fig. 13. Instruction set IS1 expressed as points in the operation type space
In general, operation types are associated with axes in a multi-dimensional numerical space $\mathcal{R}^d$, where $d$ is the dimension corresponding to the number of operation types, and instructions can be geometrically represented as points in this operation type space. An example is depicted in Figure 13. This figure gives an instruction set using three operation types. The four instruction words at the bottom of Figure 13(a) represent the instruction set. Instruction $I_1 = \{\text{add, add, mul, shift}\}$ represents the parallel execution of two add operations, one multiply, and a shift operation. It is drawn as point $p^7$ with coordinates $I_1(\text{add})$, $I_1(\text{mul})$ and $I_1(\text{shift})$. Only the “maximum” instructions are listed; instructions that are fully contained in other instructions are not explicitly represented. It is our aim to capture this subspace spanned by the instructions via inequalities. Inequality $N(\text{mul}) + N(\text{shift}) \leq 3$, for example, is one of the inequalities needed to capture this subspace. The complete set of inequalities is listed in Figure 13(b). They represent a model of the instruction set. The SRM model can be derived directly from this set of inequalities, and is given in Figure 13(c). For example, inequality (5): $N(\text{mul}) + N(\text{shift}) \leq 3$ corresponds to virtual resource MS, of which three instances are available. Both the multiply and the shift operation use one instance. This ensures that no more than three multiply or shift operations will ever execute in parallel.

In general, deriving the inequalities and the resulting virtual resources for an instruction set can be perceived as a convex hull problem $^3$, for which well-understood and efficient algorithms are available. We use the cdd $^9$ package for deriving the convex hull. The details of the computation are outside the scope of this paper.

7.1. Instruction-set design using the SRM model

Besides its advantages for code generation, the SRM approach also enables in-
struction set design (space exploration) with the almost trivial method treated in section 6. The observation that this approach works on instruction sets as well as for High-Level Synthesis stems from the potential equal treatment of functional resources (in the data-path) and virtual resources (modeling the instruction-set).

Fig. 15. DFG and the MII estimation

In case of instruction set constraints and the corresponding SRM, virtual resources should be taken into account for the performance estimation. The resource constrained minimum initiation interval ResMII is no longer simply totaling of resource usages for each resource because some operations are mapped into more than one instance of virtual resources. Thus the ResMII has to be updated with respect to the multiple usage of the virtual resources. For example, for the instruction set in Figure 14, inequality (4) in Figure 14 (b) indicates that the shift operation uses two instances of the resource MS, of which four are available. For a DFG in Figure 15, the number of operations using each operation type add, mul and shift can be expressed as $n_{add} = 2, n_{mul} = 2, n_{shift} = 4$ respectively. As illustrated in 16 (left-hand side), the ResMII is upper bounded by $\frac{n_{max} + 2 \times n_{shift}}{4} = 3$ clock cycles,
based on the critical resource $MS$. Figure 15 shows an optimal scheduling result.

<table>
<thead>
<tr>
<th></th>
<th>initial design</th>
<th>new design</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SRM</strong></td>
<td>$#A = 2$, $#M = 2$</td>
<td>$#A = 2$, $#M = 2$, $#S = 2$</td>
</tr>
<tr>
<td></td>
<td>$#AM = 3$, $#AMS = 4$</td>
<td>$#AM = 3$, $#MS = 4$</td>
</tr>
<tr>
<td>mapping</td>
<td>$\text{add} \rightarrow A, \text{AM}, \text{AMS}$</td>
<td>$\text{add} \rightarrow A, \text{AM}, \text{AMS}$</td>
</tr>
<tr>
<td></td>
<td>$\text{mul} \rightarrow M, \text{AM}, \text{MS}, \text{AMS}$</td>
<td>$\text{mul} \rightarrow M, \text{AM}, \text{MS}, \text{AMS}$</td>
</tr>
<tr>
<td></td>
<td>$\text{shift} \rightarrow \text{MS}, \text{AMS}$</td>
<td>$\text{shift} \rightarrow \text{MS}, \text{AMS}$</td>
</tr>
<tr>
<td><strong>MII</strong></td>
<td>$\tau(2 + 4*2)/4\tau = 3$</td>
<td>$\tau(2 + 4*1)/\tau = 2$</td>
</tr>
<tr>
<td>inequality</td>
<td>$N(\text{mul}) + 2 N(\text{shift}) \leq 4$</td>
<td>$N(\text{mul}) + N(\text{shift}) \leq 3$</td>
</tr>
<tr>
<td>IS</td>
<td>[add, add, mul, shift]</td>
<td>[add, add, mul, shift]</td>
</tr>
<tr>
<td></td>
<td>[add, add, shift, shift]</td>
<td>[add, add, shift, shift]</td>
</tr>
<tr>
<td></td>
<td>[add, mul, mul, shift]</td>
<td>[add, mul, shift, shift]</td>
</tr>
</tbody>
</table>

Fig. 16. Modification of the SRM

The bottleneck can be relieved by allocating additional (virtual) resources. In addition to the allocation of additional resources, in our instruction set design flow we also have the possibility to decrease the resource usage of a critical resource in order to relieve the bottleneck. This is illustrated in Figure 16. We like the $\text{shift}$ operation to use only one $MS$ to alleviate the burden on this critical resource. So the relevant inequality $n_{\text{mul}} + 2 \times n_{\text{shift}} \leq 4$ is modified to $n_{\text{mul}} + n_{\text{shift}} \leq 3$ at the right-hand side of Figure 16. The new set of inequalities now define a different instruction set, depicted in Figure 13. In this case, the change in the resource usage of $MS$ results in the addition of instruction $[\text{add, mul, shift, shift}]$ (point p11). This process of bottleneck identification and resolving is iterated until the timing is met for each resource, and for each loop. We have shown only one such iteration due to paper area considerations.

### 7.2. A reconfigurable instruction-decoder

In this section we will introduce a reconfigurable instruction decoder to implement instruction sets that can be modeled with the SRM model treated in section 10. A method for run-time configuring the instruction set is based on the instruction-set design approach treated in section 14.

The relevant parts of the data-path and the instruction decoder are depicted in Figure 17. The communication and register infrastructure is not depicted for it is outside the scope of this treatment. The purpose of the instruction decoder is to efficiently control the (possibly large) number of functional units in the data path. An instruction word consists of a header followed by a number of issue slots. One issue slot controls the execution of an operation on a functional unit. The header bits control the demultiplexors at each of the issue slots, so they determine which exact functional unit is controlled by each issue slot. Issue slot $i$ is connected (via the demultiplexer) to a limited cluster $C_L_i$ of functional units in order to restrain the
area and delay complexity and to maintain scalability of the architecture. These clusters may overlap to ensure sufficient flexibility in configuring the instruction set. In the example architecture in Figure 17, each issue slot is connected to 8 functional units. The multiplexors account for ~200 gates in the example. In order to restrict the number of header bits, the control of the demultiplexor associated with slot i is limited to a subset $S_i$ of the available functional units $C_i$. This subset is determined by the configuration of the instruction decoder, and is a way of tuning the instruction set to the application. In the example architecture in Figure 17, each issue slot controls, dependent on the configuration, any 4 out of the 8 functional units connected to the issue slot. The second stage decoder (~60 gates in the example) holds this configuration and expands the (4x) 2 bits from the first decode stage to the (4x) 3 bits that steer the demultiplexors. The first decode stage generates these (4x) 2 bits from the 5 header bits in the example, according to a reconfigurable look-up table. In the example, this look-up table has a cost of about 600 gates. Each decode stage corresponds to a pipeline stage. The first (and most complex) decode stage is optional to allow a tradeoff between the instruction width and the decoding delay.

This instruction decoder fits particularly well with the SRM model of instruction sets; From the main encoding tricks mentioned in section 10, this decoder applies the exact trick efficiently supported by the SRM model for compilation: Restricting opcode bits by allowing only certain combinations of operations to be executed in parallel. Furthermore, the method introduced in section 14 for instruction set design can be used to configure the decoder. The constraints from the decoder architecture affect this design method in the following way. Each issue slot i is connected (via its

---

**Fig. 17. Data-path and reconfigurable instruction-decoder**
demultiplexer) to a limited cluster CL of functional units. Suppose that at most two different multipliers can be reached from different issue slots. This corresponds to a natural bound \( n_{\text{mul}} \leq 2 \), or \( M \leq 2 \) during instruction set design. Similarly, if at most three different multiply or shift units can be reached from different issue slots, then \( n_{\text{mul}} + n_{\text{shift}} \leq 2 \) or \( MS \leq 2 \).

8. Conclusions

In this paper we have discussed, in the context of System-on-Chip (SoC) design, several developments in applying application-specific (reconfigurable) HW at different levels of granularity, as a way to tradeoff hardware efficiency and design effort. Hardware efficiency traditionally regards cost vs. performance (in high-volume electronics), but recently energy consumption emerged as a dominant criterion, even in products without batteries. An effective way to reduce power consumption is hardwired circuitry. More recent ideas focus on tuning the instruction set to an application after fabrication. We have introduced an architecture for a reconfigurable instruction decoder, enabling application tuning of the instruction set after fabrication. The strong phase coupling between instruction selection and scheduling, caused by these highly encoded and irregular instruction sets has hampered the development of efficient compilation tools. In our approach, we replace the instruction set constraints with a set of virtual resources that implement the individual operations contained in the instruction set. With this static resource model, the scheduler has the opportunity to generate better schedules in terms of timing and register requirements. Furthermore, software pipelining can be applied more effectively to exploit the available ILP.

The SRM model also allows instruction set design in terms of allocating (virtual) resources, a practical method used in the HLS of ASICs. We have described an iterative design flow comprising a bottleneck analysis based on fast performance estimation of the instruction set on a set of performance-critical loops. The availability of critical virtual resources is increased to relieve the bottleneck, resulting in an extension of the instruction set.

1. Frontier Design: ART Designer Tutorial.