



Figure 2 - Chess/Checkers retargetable tool suite.

Chess uses graph-based modelling and optimisation techniques [6], to deliver highly optimised code for specialised architectures exhibiting peculiarities such as complex instruction pipelines, heterogeneous register structures, specialised functional units and instruction-level parallelism. The compiler comes with a retargetable assembler and disassembler, and a retargetable linker.

2) *Checkers*: a retargetable instruction-set simulator (ISS) generator that creates a cycle and bit accurate ISS for the target processor. The ISS can be run in a stand-alone mode or be embedded in a co-simulation environment through an application programming interface (API), optionally including SystemC wrappers. Checkers includes a graphical debugger that can connect both to the ISS, and to the available processor hardware for on-chip debugging. The connection is made via a JTAG interface. Source-level debugging and code profiling are supported.

3) *Go*: a hardware description language (HDL) generator that produces a synthesisable register-transfer level VHDL or Verilog model of the target processor core. Through APIs, users can plug in their own HDL implementations of functional units and of the memory architecture. Go can optimise for low power, e.g. by applying clock gating to selected registers, and operand isolation to functional units. A special debug infrastructure for on-chip debugging via JTAG can be generated.

4) *Risk*: a retargetable test-program generator that can generate assembly-level test-programs for the target processor with a high fault coverage, suited for simultaneous execution in the ISS and in the HDL model.

A unique feature of the Chess/Checkers tool suite is its architectural retargetability, based on the nML processor description language (see Section III). Using nML, an architecture designer can quickly define the instruction-set architecture of a processor. After reading the nML description, the different Chess/Checkers tools are automatically targeted to the specified architecture.

III. ARCHITECTURE MODELLING IN NML

nML is a high-level language that captures a programmer's model of the target processor [7][8].

```
// Start of structural skeleton
mem DM[1024] <num, addr>;
reg R[4] <num>;
pipe C<num>;
trn A<num>; trn B<num>;
fu alu;
...

// Start of instruction-set grammar
opn my_core (alu_inst | mac_inst | shift_inst);
...

opn alu_inst (op:opcod, x:c2u, val:c16s, y:c2u) {
  action {
    stage EX1:
      A = R[x];
      B = val;
      switch (op) {
        case add : C = add(A, B) @alu;
        case sub : C = sub(A, B) @alu;
        case and : C = and(A, B) @alu;
        case or  : C = or(A, B) @alu;
      }
    stage EX2:
      R[y] = C;
  }
  syntax : op " R" y " ", R" x " , " val;
  image  : "0"::op::x::y::val;
}
...
```

Figure 3 – Excerpt from an nML description.

Figure 3 shows an excerpt from an nML description. In a first part, called *structural skeleton*, all storages, connectivity, and functional units are declared. Storages and connections have a *data type*, defined as C++ classes in a user-extensible library. In a second part, the *instruction set* is defined by means of an *attribute grammar*. The grammar specifies a breakdown of the instruction set into classes, in a compact hierarchical way. *Or* rules in the grammar (e.g. `my_core()` in Figure 3) specify alternative choices, while *and* rules (e.g. `alu_inst()`) specify concurrency.

And rules can carry up to three different attributes. The *syntax* attribute defines the assembly language for the instructions. The optional *image* attribute defines a binary encoding for the instructions. The *action* attribute defines the instructions' behaviour. In contrast to more abstract models used in languages like Lisa [9], nML captures the instructions' behaviour in a compact *register-transfer* model, exposing the exact resource utilisation and pipeline of the processor. This formalism, which allows for an accurate description of structural and timing irregularities that are typical of many ASIPs, is at the basis of patented compilation and hardware generation techniques [6] used in the Chess/Checkers tool suite.

Register transfer actions in nML may call *primitive functions* (e.g. `add()` or `sub()` in Figure 3), which are defined in a user-extensible library. A function is called primitive when it is directly supported by the processor hardware. Multiple primitive functions can be grouped into a functional unit, by means of the @ operator in nML.

IV. BROAD ARCHITECTURAL SCOPE

The nML front end of the Chess/Checkers tool suite translates the processor description into an intermediate representation called *instruction set graph (ISG)* [6] that

captures all essential structural and timing information. All optimisations in the retargetable C compiler, instruction-set simulator, and hardware generator directly operate on this detailed hardware representation. As a result, Chess/Checkers can support a very broad range of processor architectures. The tools enable true architectural exploration, beyond many of the limitations of configurable templates offered by intellectual property vendors. Reference [10] illustrates the architectural exploration process with Chess/Checkers, based on profiling information generated by the tools.

The following list of architectural options illustrates the broad architectural scope of the Chess/Checkers tool suite:

- *Data types:* Using the power of C++, any user-defined data type can be supported, including e.g. integer, fractional, floating-point, Boolean, complex, and vector (SIMD) data types.
- *Arithmetic functions:* In addition to the built-in functions and operators of the C language, user-defined (primitive) functions are supported. The compiler supports function overloading, as well as calling user-defined functions as *intrinsics* in the C source code.
- *Registers and interconnect:* Special purpose register sets and irregular interconnection schemes can be specified in nML, and are well supported by the compiler thanks to innovative register allocation techniques.
- *Memory and I/O:* Both Von Neumann and Harvard architectures are supported. There is no limitation to the number of data memories or memory ports that can be specified. External memory and communication models with data-dependent latencies can be included via a memory interface API. In the ISS, this API is based on SystemC. The compiler automatically exploits a wide variety of addressing modes.
- *Instruction word:* Both orthogonal (VLIW) and encoded instruction sets can be specified and are supported by all tools. Also, variable-length instructions are supported. Powerful data flow analysis, scheduling, and software pipelining techniques in the compiler aim at exploiting the available instruction-level parallelism.
- *Instruction pipeline:* The instruction pipeline is accurately described in nML, including pipeline hazards and register bypasses. There is no limitation on the pipeline depth. The compiler ensures that the generated code is hazard free. Alternatively, the tools support hardware stalling to avoid pipeline hazards. In this case the required interlocking hardware is automatically generated in the ISS and HDL code.
- *Program control:* Supported program control features include: subroutines, interrupts, hardware do-loops, residual control (i.e. mode-dependent behaviour), and predication (i.e. conditional execution of instructions). Subroutines can be implemented even in the absence of a software stack. The compiler supports optimised

context switching, including inter-procedural code optimisations.

V. DESIGN EXAMPLES

The Chess/Checkers tool-suite has been applied by Target's customers to design and program processors for a variety of applications. This section presents a selection of designs made with Chess/Checkers, illustrating typical design tradeoffs made in each application domain.

1) *Portable audio and hearing instruments.* NXP Semiconductors' CoolFlux DSP [11] is a representative example from this application domain.

While typical data rates are low enough to permit the use of general-purpose DSPs, a major challenge comes from the ultra-low power requirement of the targeted battery-powered systems, combined with the flexibility requirement imposed by multiple coding standards and a growing variety of advanced audio algorithms.

The best compromise is an ASIP that has a general-purpose instruction set, augmented with domain-specific instructions with a high degree of instruction-level parallelism. CoolFlux DSP supports up to 8 parallel operations in its most parallel instructions. As a result, most audio codecs can be executed in significantly fewer instruction cycles than on a general-purpose DSP, which provides the necessary headroom for reducing the processor's clock frequency and supply voltage, thus reducing dynamic power consumption. Additionally, a compact architecture with a small register set and an encoded instruction word, as well as an optimised HDL design contribute to CoolFlux DSP's ultra-low power consumption of 60 $\mu\text{W}/\text{MHz}$ at 1.2V (130 nm CMOS).

CoolFlux DSP is available with a rich software library of audio and telecom applications, developed by NXP using the Chess compiler.

2) *Wireline modems.* The discrete multi-tone (DMT) engine in STMicroelectronics' Astra+ chip set for ADSL2+ customer-premises equipment modems [12], is an example from this application domain.

This application uses advanced equalisation algorithms, requiring specialised computations on complex numbers, that must sustain the system's high data-rate of 24 Mbps. Because of this high throughput and the non-conventional data types, such an application would traditionally be implemented as a fixed-function ASIC core, in which the required algorithmic data-flow patterns are directly mapped into hardwired data paths controlled by a finite-state machine (FSM). However, by adding a few registers and multiplexers, and by replacing the FSM by a microcoded engine, a programmable ASIP was derived that offered more flexibility at almost the same cost as a fixed-function core. Such an ASIP does not offer general-purpose programmability, but can support restricted algorithmic variations within the same application domain. The DMT engine ASIP was designed in nML and programmed using

