Dynamic Support of Processor Extensions in Cross Development Tools

V.V. Rubanov

Abstract— **In this paper, we consider the problem of adjusting cross development toolkits (assembler, linker, simulator, profiler, debugger, and IDE) to support processor extensions in the form of accelerators (synchronous coprocessors). Considered approach is based on describing accelerator models in a special specification language and using such descriptions to dynamically reconfigure cross toolkit to support assembling, cycle accurate simulation and debugging of applications containing new instructions implemented by the accelerators. We describe a generalized mathematical model of the supported class of accelerators and interfaces with the main processor. The specification language for defining concrete accelerators is given and our technology for dynamic cross-tools reconfiguration is overviewed.**

Index Terms— **Cross development tools, embedded system design, hardware specification languages, programming environments.**

I. Introduction

In this paper, we consider the problem of reconfiguration of *cross development tools* to support extensions of the target processor in the form of synchronous coprocessors (accelerators). By cross development tools (*cross-system* or *cross toolkit*) we mean a set of software utilities (assembler, linker, simulator, debugger, profiler, and IDE) for software development using a host-machine different from the target hardware (e.g. using PC to develop firmware program for a mobile phone). Supporting processor extensions in the cross tools means ability to assemble, build, simulate and debug application programs that contain instructions implemented by user specific accelerators and unknown when building toolkit for the main processor. Considered approach is based on allowing user to describe accelerator models using developed at ISP RAS specification language. These models are then used for dynamic reconfiguration of the main cross toolkit.

This task appears because many modern hardware architectures are built using a standard processor core and a set of specific extensions in the form of domain specific accelerators (coprocessors). A subset in the core's instruction set is reserved for *accelerator invocation commands* with the semantics of real behavior initiated by such commands defined by concrete accelerators independent of the main processor. Processor core and accelerator vendors may be different companies, and cross development tools from the core's vendor must support core's extensions created by customers. In this article, by core's extensions we mean adding accelerators that may introduce new memory elements (registers, data memories) and new instructions explicitly invoked by the main processor as if they are a part of its instruction set. The entire system consisting of the processor core and a set of accelerators is viewed by an application programmer as a computing system with uniform instruction set and a single program. The memory of such system consists of the main processor memory, shared memory and accelerators' local memories.

To distinguish the class of supported accelerators and interfaces with the main processor as well as to introduce decomposition of the system components, we created a generalized mathematical model that allows to model state and behavior of a wide class of accelerators with cycle true accuracy. To define concrete accelerator models within the generalized one, we developed a specification language ISE (Instruction Set Extension) and implemented tools for visual editing of ISE specifications with automatic inconsistencies/errors analysis and detection. We implemented dynamic cross-system reconfiguration based on the interpretation of user defined accelerator model specifications while assembling/disassembling, simulating and debugging.

The article consists of the introduction, three sections and the conclusion and is organized as follows. Section II presents our accelerator modeling approach, a generalized mathematical model is given and the accelerator specification language ISE is described. Section III contains an introduction to the technology used for dynamic cross system reconfiguration based on the accelerator specifications defined in ISE. Section IV overviews the related work addressing hardware modeling using specification languages targeted at automatic cross tools generation. In the conclusion, practical results are presented and future work is outlined.

II. Modeling Processor Extensions

In this section, we consider modeling of processor extensions in the form of accelerators (coprocessors) using a specification language. A generalized mathematical model for a wide class of supported accelerators is given (see II.A) as well as the interaction protocol with the main processor. In II.B, we present the ISE specification language for concrete

Manuscript received April 20, 2007.
Vladimir V. Rubanov is with the Institute for System Programming of the Russian Academy of Sciences, Moscow, Russia (e-mail: vrub@ispras.ru).

accelerator model specification within the generalized one. Defined in this way accelerator models are used for automatic reconfiguration of the following cross system components: simulator, assembler/disassembler, debugger and profiler (see Section III).

A. Generalized Accelerator Model

We consider accelerators as slave coprocessors with instructions issued by the main processor. All accelerators and the main processor work synchronously using the shared cycle generator. An accelerator can have its own local data memory and also it has access to the memory shared between processor and accelerators. Accelerators do not have their own program memories.

1) Accelerator State

a) Accelerator memory

A *memory cell* is an ordered set of binary variables (*bits*) with possible values 0 or 1. The number of bits in a cell is called cell's *width*. A set of one or more cells with the same width forms a *memory area*. One or more memory areas form a *memory*. We will denote memory by a capital letter *S*. *Cell's state* is defined by the set of particular values of all the cell's bits. *Memory's state* is defined by the state of all the cells in all the areas of the memory. Memory state is denoted by a lower-case letter *s*. Let N to be the sum of all the widths of all cells in a memory *S*, then the memory can be in one of the 2^N states. The set of possible memory states is denoted by {s}. Please note: this set is unambiguously defined by the *memory structure*. *Memory structure descriptor* Σ is a set of the following numbers: the number of areas *A* and a set of *A* pairs (W_i, S_i), defining the width W_i of the area cells and their quantity S_i.

In our model, *system memory* S consists of *main processor local memory* S^P, *shared memory* S^S and a number of *local accelerator memories* S^a:

$$S = \left\{ S^P, S^S, \{S_i^a\}_{i=1}^{N^A} \right\},$$ N^A- the number of accelerators.

Considering local processor memory is not important in this article, because accelerator has no access to it. A pair of shared memory and local accelerator memory forms the full *accelerator memory*, which we will denote as S^A:

$$S^A = \{S^S, S^a\}$$

The set of all possible states of the accelerator memory is denoted by $\{s\}_A = \{s\}_S \times \{s\}_a$. Each area in the local or shared memory is characterized by *access delay* – the number of cycles that should be passed after writing to a cell before the changed value can be read; before that moment the previous value is fetched.

Usually, accelerator memory has the following areas:

– A set of one or more *data memories.*

– A set of one or more *register files.*

– *Single registers.*

b) Control state

Accelerator has fixed number of *control slots*, the set of which is denoted by P^A. Each slot has an index, which will be identified with the slot itself. Each slot L_i is a pair of integer variables (fields): *instruction number* field n_f and *instruction state* field *t*. The set of possible values of instruction number field $\{n_f\}$ is finite. Zero always belongs to $\{n_f\}$. There is a one-to-one correspondence between $\{n_f\}\backslash 0$ and instruction descriptors set, which will be defined below. That is why we will identify number $n \in \{n_f\}$ with the corresponding descriptor *f*. Possible values of the instruction set field are natural numbers (including 0). The state of a slot with index n is denoted by $l_n = (n_f, t)$ and is defined by the values of the slot's fields. The set of possible slot states is $\{l\} = \{n_f\} \times \mathrm{N}$. Slot in the state (0, 0) is called *free slot*, otherwise *active slot*. *Accelerator control state p* is a whole of the states of all the accelerator slots. *The set of possible accelerator control states* is denoted by $\{p\}_A = \{l\}^{N_S}$, where N_s is the number of control slots in the accelerator. Active slots correspond to accelerator instructions in progress, thus the number of control slots defines the maximum number of accelerator instructions running in parallel. Instruction number field value of an active slot defines the running instruction for this slot, while the instruction state field value corresponds to the number of cycles passed from this instruction start.

c) Accelerator state

Accelerator state *a* is defined by the pair of memory state and control state: a={s, p}. *The set of possible accelerator states* (accelerator states set) is denoted by $\{a\}_A = \{s\}_A \times \{p\}_A$.

Accelerator *state element* is any accelerator memory cell or any accelerator control slot. State element state is defined by the state of the cell or control slot correspondingly.

Parallel composition of functions $\varphi_1, \varphi_2 ... \varphi_n$ (which are defined on the accelerator states set) is a function $\omega = \varphi_1 \,||\, \varphi_2 \,||\, ... \,||\, \varphi_n$ (also defined on the accelerator states set), specified for accelerator state *a* as follows: let $\mathrm{A} = \{\alpha\} = S^A \cup P^A$ is the set of all state elements of the accelerator, A_i- the set of state elements, which state is changed by function φ_i: $\alpha \in \mathrm{A}_i \Leftrightarrow a_\alpha \neq \varphi_i(a_\alpha)$. If intersection of all $\mathrm{A}_i, i = 1..n$ is not an empty set, then ω is undefined for the accelerator state *a*. Else, the value of ω is specified as follows:

$$a' = \omega = \begin{cases} a'_\alpha = a_\alpha, \alpha \in \mathrm{A} \setminus \bigcup A_i \\ a'_\alpha = \varphi_i(a), \alpha \in \mathrm{A}_i \end{cases}$$, where a'_α and a_α - new and old state of α element correspondingly.

2) *Accelerator Model*

Operation [elementary] is a function ψ defined on the accelerator memory states set $\{s\}_A$ (see 1.a), that calculates the next memory state based on the previous one. Operations correspond to actions that can be done for one cycle (e.g. adding two registers):

$$\psi : \{s\}_A \to \{s\}_A$$

The set of all accelerator operations is denoted by $\Psi^A = \{\psi\}$. For any accelerator this set includes so called empty operation ψ_0 that does not change memory state.

Each elementary operation is characterized by *functional resources* that are necessary for this operation. We denote the set of all accelerator resources by R^A and the set of its subsets (including empty one) by P^A. Function r^A that maps operations set Ψ^A to the set P^A is called *resource function.* This function specifies resources set for each operation in Ψ^A:

$$r^A : \Psi^A \to \mathrm{P}^A$$

Let us denote the set of all subsets in Ψ^A that contains operations with non-intersecting resources as Ω^A:

$$\omega \in \Omega^A \Leftrightarrow \forall \psi_i, \psi_j \in \omega, i \neq j \to r^A(\psi_i) \cap r^A(\psi_j) = \varnothing$$

Thus an element of Ω^A defines a group of operations $\{\psi_i\}$ that can be run in parallel within one cycle. Using parallel composition (see 1.c) for all operations in $\omega \in \Omega^A$ one can assign a function on the accelerator memory states set $\{s\}_A$. Further we will identify elements of Ω^A with corresponding functions defined as above. These functions are called ***complex-operations***. Please note that in this sense $\Psi^A \subset \Omega^A$.

We introduce two ***control actions***: *next* and *end*. Each control action is a parameterized function on the set of accelerator control states. Function parameter is a slot index $n_l \in [1..N_S]$. Action *next* increments by 1 the value of instruction state field *t* for the slot defined by the parameter n_l.

$$p' = next(n_l, p) = \begin{cases} l'_n = l_n, n \neq n_l \\ l'_n = (f_n, t_n + 1), n = n_l \end{cases}$$

Action *end* turns corresponding slot to the idle state.

$$p' = end(n_l, p) = \begin{cases} l'_n = l_n, n \neq n_l \\ l'_n = (0,0), n = n_l \end{cases}$$

The set of these control actions is denoted by $U^A = \{next, end\}$.

Accelerator instruction descriptor is a function f that calculates a pair of complex-operation $\omega \in \Omega^A$ and control action $u \in U^A$ based on the accelerator memory state $s \in \{s\}_A$ and instruction state $t \in \mathrm{N}$ (see 1.b):

$$f : \{s\}_A \times \mathrm{N} \to \Omega^A \times U^A$$

The set of all instruction descriptors of an accelerator is denoted by $I^A = \{f_i\}$. This set is finite and each element f of this set is assigned an index $n_f \geq 1$ (e.g. in order of machine code ascending as defined by the decoding function, see below). For any accelerator instruction descriptor the following is always valid (the only instruction end):

$$f(s,\tau) = \{\omega, end\} \Rightarrow \forall t > \tau : f(\omega(s), t) = undefined$$

An instruction descriptor f unambiguously defines *instruction cycle behavior function* b_f, which is defined on the accelerator states set $\{a\}_A$ and parameterized by a slot number n_l. Function b_f is not defined if instruction number field value in the slot n_l does not match the corresponding to b_f instruction descriptor f. Otherwise this function $b_f(n_l) : \{a\}_A \to \{a\}_A$ is defined as follows:

Given the memory state s and instruction state filed value *t* in the slot n_l, using the instruction descriptor a pair of complex-operation ω and control action u is determined by $\{\omega, u\} = f(s,t)$, where ω defines the resulting memory state $s' = \omega(s)$, while u calculates the resulting control state $p' = u(n_l, p)$. (s', p') defines the values of $b_f(n_l, s, p)$.

Thus the set of instruction descriptors I^A unambiguously defines the set of instruction cycle behavior functions B^A with the one-to-one correspondence.

Accelerator instruction codes set $C^A = \{c_i\}$ is a set of binary numbers with the same width. Each element of this set corresponds to one of the possible values of the *accelerator instruction code* field in the machine code for accelerator invoking command (see 3.a).

Decoding function d^A maps the set of accelerator instruction codes C^A to instructions descriptors set I^A:

$$d^A : C^A \to I^A$$

Decoding function unambiguously defines *accelerator decoder* – function D^A:

$$D^A : C^A \times \{p\}_A \to \{p\}_A :$$

Function D^A is specified as follows: if in p $\exists n_0 : l_{n_0} = (0,0)$ then $p' = D^A(c,p) = \begin{cases} l'_n = (f_n, t_n), & n \neq n_0 \\ l'_n = (d^A(c), 1), n = n_0 \end{cases}$, else D^A is undefined. In other words, in the case of free slot existence decoder initialize the state of this slot to $(f,1)$, where f is determined using accelerator instruction code using decoding function d^A.

Cycle function T^A defined on the accelerator states set $\{a\}_A$ determines accelerator state transition for each cycle:

$$T^A: \ \{a\}_A \to \{a\}_A$$

This function is unambiguously defined by parallel composition (see 1.c) of instruction cycle behavior functions b_f associated with instruction descriptors f corresponding to the instruction number field values n_f for each *active* control slot. In other words, active slots define the set of active instructions characterized by corresponding cycle behavior functions. Parallel composition of these functions defines T^A for the current cycle (accelerator behavior).

Generalized accelerator model M^A is the following set of elements described above:

$$M^A = \left\{S^A, P^A, C^A, D^A, T^A\right\}$$

3) Accelerator Simulation

For simulating of an accelerator specified by model M^A (see p.2 above) we need a *cycle generator* and a concrete accelerator *memory initial state* $s_0 \in \{s\}_A$. In the initial accelerator control state p_0 all slots are always free. In the system under consideration all accelerators and the main processor work cycle-synchronously (cycle generator is the same), that is one accelerator cycle is equal to the main processor cycle. Besides cycle generator, the only external event for accelerator is issuing (feeding) of an instruction to accelerator by the main processor (see 3.a right below).

a) Accelerator instruction invocation

For our system model, the instruction set of the main processor must contain a subset devoted to *accelerator invocation commands*. Such a command initiates an accelerator instruction. Please note that when executing this command the main processor is not responsible for further calculations initiated. In the main processor instruction set, each accelerator invocation command is defined by three fields of the machine code (the fields order is not significant as well as some fields may be non-continuous):

{opcode, accelerator index, accelerator instruction code}

When executing such instruction the main processor determines by `opcode` that it is an accelerator invocation command. Then it extracts `accelerator index` field and passes remaining bits in the `accelerator instruction code` to the corresponding accelerator for further decoding and execution in parallel with the main processor. For the main processor, execution stage of an accelerator invocation command always takes one cycle. In terms of the abstract model, execution of an accelerator invocation command results in issuing accelerator instruction to an accelerator, thus activating accelerator decoder function D^A (C^A is a subset of the possible `accelerator instruction code` field values set). The main processor can issue no more than one accelerator instruction per cycle. Please note that `accelerator instruction code` in its turn consists of *accelerator instruction opcode* and *operands* fields.

Accelerator can run several multi-cycle instructions in parallel, including multiple instructions with the same descriptor. That is the main processor can issue new instruction before the previous ones finish. In terms of the described model this is possible if all operations initiated by all parallel instructions use non-intersecting resources (see p.2). In practice this is possible when accelerator pipeline and functional units allow this. Application programmer (compiler) is responsible for scheduling correct accelerator instructions invocations.

b) Per cycle accelerator behavior

In response to the external cycle generator event in our model accelerator behavior is defined by the state transition according to its cycle function T^A (see p.2). This function defines per cycle behavior of the accelerator.

c) Data exchange and synchronization with the main processor

Data exchange between processor and accelerators is performed via the shared memory (see 1.a). Please note that some information is delivered to accelerator from the processor in the form of accelerator instruction code (see 3.a). Also one should mention that different accelerators do not have access to each other local memories.

Shared memory access rules in our model correspond to CREW (Common Read Exclusive Write) type. This means that processor and accelerator can simultaneously read from the same memory cell (within the current cycle) but simultaneous write is forbidden. In the model, memory areas can have access delay (see 1.a). By default, in the considered implementation all memory areas have access delay 1. This means that changed value of a memory cell can be read on the next cycle only (flip-flop cells model). Please note that if the access delay is more than 0 then it is possible to simultaneously read and write to a cell, old value is fetched when reading.

Accelerator instructions can take fixed or data dependent number of cycles. From the application programmer (compiler) point of view there are three ways of synchronizing the main processor and the results of a running in parallel particular accelerator instruction:

1. If the accelerator instruction takes fixed number of cycles then it is possible to predict in static when the results become available. The main program can be designed to take it into consideration and rely on the synchronous processor and accelerator execution.
2. Accelerator instructions with varying number cycles to execute can use some dedicated flags in the shared memory to indicate to the main processor in run-time when the results of this instruction become available. The main processor program can repeatedly read this flag to check if accelerator result is ready.

3. Raising of an interrupt by accelerator for the main processor when results are ready. Interrupt handler routine can read accelerator results.

B. *Describing Concrete Accelerator Models*

The set of the following parameters, sets and functions defines a **concrete accelerator model**:

$$m^A = \left\{\Sigma^A, N^S, C^A, d^A, I^A, \Psi^A, R^A, r^A\right\}$$

Sections A.1) and A.2) contain definitions for all the symbols in m^A as well as prove that these symbols unambiguously define all the generalized model elements:

- $\Sigma^A \to S^A$;
- $N^S \to P^A$,
- $d^A \to D^A$,
- $\Psi^A, R^A, r^A \to \Omega^A$; $I^A, \Omega^A \to B^A \to T^A$.

A special language ISE (Instruction Set Extension) was developed at ISP RAS for concrete accelerator models specification. In addition to concrete accelerator model elements description it provides the means for defining assembly syntax and binary coding for each instruction plus visualization formats for all memory areas. In the rest of this paper we will identify the concrete accelerator model with its specification in ISE.

1) *Memory Structure Descriptor*

Special constructs are used to declare the following types of memory areas:

- Data memory.
- Register file.
- Stand-alone register.

A combination of such declarations defines a memory structure descriptor Σ^A (see A.1.a).

Example. *Memory structure of a simple accelerator:*

Two data memories: `LDM` with 2048 words of 16 bits width and `TM` with 2048 words of 64 bits width. Access delay for each memory is 3.

Register file `GRF` containing two 16 bit registers `GR0` and `GR1.`

Stand-alone accumulator register `ACR` of 36 bits wide and access delay of 1 (default).

```
DECLARE_MEMORY(INT(16, 3), 2048) LDM;
DECLARE_MEMORY(INT(64, 3), 2048) TM;

DECLARE_REGISTERS_FILE(INT(16), 2) GRF;
DECLARE_REGISTER(UINT(36)) ACR;
// debugging names and registers file
structure
MEMORY(LDM, "Acc LDM");
MEMORY(TM, "Acc TM");
REGFILE_BEGIN(GRF, "General Registers")
  REGISTER(0, "GR0");
  REGISTER(1, "GR1");
REGFILE_END()
```

2) *Behavior Description*

The following elements of m^A are the parts of the accelerator behavior description: the set of resources R^A, the resource function r^A, the set of operations Ψ^A, the set of instruction descriptors I^A and the decoding function d^A (along with C^A).

a) Operations

Each operation from Ψ^A is defined in C++ programming language. All accelerator memory cells are globally accessible from within an accelerator (in particular, register file contents are accessible as indexed arrays). It is allowed to use auxiliary local variables and the means of a special library (for example, N-bit data types `INT<N>`, `UINT<N>`, fixed point data types: `FIXED<I,F>`, bit manipulation subroutines). The set of resources used by the given operation is marked by a call to `UseResources(resources)` function (thus defining the resource function r^A). In the current implementation the set of resources in `UseResources` is indicated as a bit set where each resource corresponds to a specific bit index. The whole set of resources R^A is declared by C++ `enum` with values equal to powers of 2:

```
enum Resources {MAC_ADDER=1,
        MAC_MULTIPLIER=2, ALU_ADDER=4};
```

Example 1. *An addition of two 36 bit integer numbers.*

```
void ADD_36_36
   (INT<36>& res, INT<36> a, INT<36> b)
{
   UseResources(MAC_ADDER);
   res = a + b;
}
```

Example 2. *Multiplication of two 16 bit signed integers.*

```
void SMUL_16_16
   (INT<32>& res, INT<16> a, INT<16> b)
{
   UseResources(MAC_MULTIPLIER);
   res = a * b;
}
```

An operation defined in C++ can be either a separate function (see examples above) or can be embedded directly into the instruction behavior function (see examples in 2.b below).

b) Instruction descriptor

Each accelerator instruction descriptor from I^A is defined by appropriate *instruction behavior function*. The function can have arguments mapped to instruction parameters p_i. Thus each function defines a set of instruction descriptors (each descriptor corresponds to one combination of particular parameter values). The body of the instruction behavior function is defined in C++. The special built-in function `FinishCycle()` implicitly defines the mapping to the set of operations for each instruction state t. Every subsequent call of this function distinguishes operations corresponding to the given state t. Inside the instruction behavior function the dependence of operations performed on the memory state (the functional dependence of $f(s,\tau)$ on s) is achieved by standard C branching and looping constructs (see example 3 below). A call to the `FinishCycle()` means completion of the operations for current cycle and corresponds to the control action `next`. Return from the instruction behavior function corresponds to the control action `end`. This approach allows easy describing of accelerator instructions preserving semantics for cycle-true simulation.

Example 1. Here we define a single-cycle instruction, performing a move between two registers defined by `GRF[greg] = LRF[lreg]` operation:

```
ACC_FUNCTION Move_LREG_GREG
             (INT<4> lreg, INT<4> greg)
{
   GRF[greg] = LRF[lreg];
   FinishCycle();
}
```

Example 2. A double-cycle MAC (multiplication and accumulation) instruction. The multiplication at the first cycle (see SMUL_16_16 in the example 2 from 2.a) is followed by the accumulation at the next cycle (see ADD_36_36 in the example 1 from 2.a):

```
ACC_FUNCTION MAC_LREG_GREG
             (INT<4> grs, INT<4> grt)
{
  SMUL_16_16 (mulres, GRF[grs], GRF[grt]);
  FinishCycle();
  ADD_36_36 (ACC, ACC, mulres);
  FinishCycle();
}
```

If this instruction is issued by the main processor twice at successive cycles then a situation occurs when both stages of this function are executed simultaneously for different instruction instances (multiplication stage of the first instruction and accumulation stage of the second one). This approach allows modeling accelerator pipelines.

Example 3. The convolution of two vectors in different memory areas: DM0 and TM0. Execution time for this instruction depends on the length of input vectors (which should be set in a special register `LOOPREG`). Note that inside the loop body few operations using non-overlapping resources are executed at the same cycle. An interrupt is raised upon completion of convolution actions:

```
ACC_FUNCTION CONV_ACC_DM0_TM0
            (INT<4> dreg, INT<4> treg)
{
  SMUL_16_16 (mulres, DM0[AR[dreg]++],
              TM0[AR[treg]++]);
  FinishCycle();
  while (LOOPREG>0) {
    ADD_36_36 (ACC, ACC, mulres);
    SMUL_16_16 (mulres,
                DM0[AR[dreg]++],
                TM0[AR[dreg]++]);
    LOOPREG--;
    FinishCycle();
  }
  ADD_36_36 (ACC, ACC, mulres);
  InterruptProcessor();
  FinishCycle();
}
```

c) Decoding function

Decoding function d^A is defined by a set of pairs of *machine code format* and a reference to an instruction behavior function:

```
INSTRUCTION(<format_string>,
            <invoker_name>);
```

Machine code format is a string consisting of the following characters alphabet:

1. Bit characters: '0' and '1'
2. Parameter characters: 'A-Z' and 'a-z'
3. Wildcard character: '*'
4. Separator character: '-'

Characters from pp. 1-3 are called *significant characters*. Note, that the number of significant characters in machine code format must be equal to the machine word width.

A continuous series of parameter characters defines an *operand*. Accelerator decoder extracts designated bits and passes them as a parameter p_i to proper instruction behavior function. Bit, wildcard or separator characters are used to separate different operands.

Bit character specifies particular value for designated position in machine code. Parameter character or wildcard character allows any value (0 or 1) for its position. Separator characters do not correspond to any position and are used to distinguish operands and for cosmetic purposes.

Example:

```
INSTRUCTION("11-**-0000-0000-0001-LREG-
GREG", Move_LREG_GREG);
```

Instruction behavior function `MoveLREG_GREG` (see example 1 in 2.a) has two 4 bit parameters (`LREG[4;7]` and `GREG[0;3]`). Bits [20;21] can take any value for the given instruction (actually they contain an accelerator number used by the decoder of the main processor). The rest of the bits are fixed and form accelerator instruction `opcode`.

The collection of machine code format strings in all the INSTRUCTION pairs defines the set of accelerator instruction codes for the given accelerator (see C^A from A.2).

The number of control slots N^s is defined by using

SLOTS(< N^s >).

C. Assembly Syntax for Accelerator Instructions

There is a separate section in the accelerator specification file that is responsible for assembly syntax definition for the instruction set of this accelerator. More correctly, this section defines syntax for the subset of the main processor's instruction set that corresponds to instructions invoking commands of this accelerator (see A.3.a). This is useful for adequate reflection of the corresponding command semantics at the assembly level because the same (in the sense of machine codes) commands of the main processor (for accelerator invocation) may have different semantics depending on the concrete configuration of accelerators in a specific for each customer system. In addition to semantics definition for accelerator instructions (see B.2), the possibility to adjust assembly syntax for these commands is an important feature of the system in hand. It allows to get a production quality cross system adapted for a specific configuration «processor + user accelerators».

The description of the assembly syntax for the instruction set consists of tree sections:

1. Operand types and aliases.
2. Commands.
3. Constraints.

Detailed specification of the assembly syntax description is out of the scope of this article. Only a concise description of the essential functionality is given below.

1) Mapping Assembly Instructions to Machine Codes

The general template of the allowed assembly syntax for accelerator instructions is defined as follows:

```
command   ::= mnemonic
               [parameter {, parameter}*]
               {|| mnemonic [parameter {,
               parameter}*]}*

mnemonic  ::= const_string

parameter ::= operand
               {[const_string] [operand]}*

operand   ::= const_string

const_string ::= <any text without
                  quotes and spaces>
```

Examples of possible instructions:

```
DMOVE ACR1.h, DM0(DA0--),
      ACR1.L, TM0(TA0++)

MOVE GRA, DM1(TA0+25) || ADD GR3, ACR2.H
```

In the assembly language an instruction consists of a *mnemonic* (several mnemonics for parallel commands) and a set of comma divided *parameters*. Each parameter may contain several parts – *operands* belonging to some of the described types. In the scope of one parameter operands must be divided by non-empty strings of constant symbols. Combination of mnemonics is mapped to opcode field of the corresponding instruction. Each operand is mapped to a field-operand of the machine word. It is possible to map operands to discontinuous fields (when field's bits are interlaced with bits of other fields).

Example:

```
.types
   grn [gr0:0] [gr1:1] [gr2:2] [gr3:3]
   const6b $ -32 31

.mnemonics
    MOVE {grn#GG},{const6b#CCCCCC}
                     % 1010-10CC-11GG-CCCC
```

This description defines the instruction MOVE with two operands. The first operand of type *grn* is a general register. The register code is placed in two bits beginning from the 4th bit. The second operand is a constant in the range [-32; 31] placed in two parts of the machine word: in four bits beginning from the zero bit and two bits beginning from the eighth bit. Opcode equals to 1010-10XX-11XX-XXXX.

2) Constraints

a) Operand type constraints

Each operand has a type that defines imposed by this type single operand constraint. Meanwhile, to make error reporting more sophisticated, one should define a general type for an operand and make use of the general constraints (see the next paragraph) to filter allowed values.

b) Multi-operand predicate constraints

Multi-operand predicate constraints are defined as a set of logical predicates involving arithmetic expressions, which may contain references to any operands used in the instruction. It is assumed that constraints are met if all logical predicates for the given instruction are "true".

Constraint expressions support the following operations:

Logical operations	&&, ||, !
Comparison operations	<, >, <=, >=, ==, <>
Arithmetic operations	+, -, *, /, %
Bit operations	|, &, ^, ~

One can use numerical constants and references to operands in the expressions.

Example. *Operands 1 and 2 must be different:*

```
ADD {GRs#SSSS}, {GRt#TTTT}
           % 1000-1011-SSSS-TTTT
(GRs <> GRt) %
  "Operands must be different for ADD"
```

Note that with the aid of the mechanism described, it is possible to define constraints for single operands. Thus two strategies of working with operand constraints are possible: on the one hand it is possible to create a set of general types and then narrow collections of allowed for particular operand values using logical predicates, on the other hand – one may work with a lot of specific types. The first strategy may result in more specific error messages for instruction operands; the second is simpler for the instruction set description. The user may choose any strategy.

c) Inter-instruction relationship constraints

Each instruction may have a particular *set of properties*. It is possible to assign values and *activation area* for each property. The property value is a constant or a value of one of the instruction operands. The activation area defines a range of the adjacent instructions for which the property is active. The default activation area [1;1] involves only the current instruction. The properties description mechanism (supplemented with the compatibility predicates – see below) actually is a modified description of reservation tables.

Example:

```
MAC {acr#A},{grs#SSSS},{grt#TTTT}
                    % 1010-A001-SSSS-TTTT
[read_grn:grs, read_grn:grt,
 write_acr:acr:2;2]
```

The given instruction has the following properties:

– `read_grn` – double property with values equal to values of the operands *grs* and *grt*. The activation area by default involves only the current instruction (this means that values for the registers *grs* and *grt* are read at the first cycle of the execution stage).
– `write_acr` – the property value equals to the value of the operand *acr*. The activation area [2;2] affects the next instruction (this means that the value of the *acr* register will be written at the second cycle of the execution stage).

Definition of the resource usage constraints is based on the property description mechanism. It is necessary to define a list of *property compatibility predicates*. The property compatibility predicate specifies a collection of property pairs in square brackets (pairs are divided by commas, properties in a pair by the '=' sign). Predicate is "true" for an instruction pair if the following conditions are met: the first instruction has all the properties from the left side of the pair, the second has all the properties from the right side of the pair; properties' values are equal within their activation areas intersection for each pair. The assembler evaluates compatibility predicates for each instruction pair during program assembling. Thus it discovers conflicting instructions. Note that evaluation works correctly for sure only for linear sequence of instructions.

Example:

```
[write_acr=read_acr] % warning:
   "Write after read conflict for ACRs"
```

This predicate is "true" if the *write_acr* property value of the first instruction equals to the *read_acr* property value of the second instruction within their activation areas intersection. The given example reflects the WRITE AFTER READ data conflict (for accumulator registers).

There is a reserved property "any" that is assigned to every instruction by default. [any=X] gives a true predicate if the second instruction has the property X (with no regard to its value). The similar rule is for [X=any].

3) Error Messages

a) Simulation errors

The number N^S (see A.1.b) defines the limit for instructions running by accelerator in parallel. The simulator generates an error if the main processor tries to issue an instruction if no free slots are available.

Another mechanism for discovering run-time simulation errors is based on the *functional resource* usage (see A.2). If two parallel instructions try to use the same resource simultaneously (during the same cycle) then the simulator generates a run-time error.

To support the CREW data access model the simulator discovers situations when separate processes perform writes to the same memory cell and generates a run-time error.

b) Assembly errors

Users can define custom error message and severity (warning or error) for each constraint of the types 2.b and 2.c. It is possible to define either individual text messages for each constraint or refer to the shared messages table. This approach allows users to customize assembly errors diagnostics with detailed descriptions, which is very important for production use of the cross system.

III. Dynamic Cross-System Reconfiguration

This section contains a brief description of our dynamically reconfigurable cross-system. Reconfiguration is based on accelerator specifications in ISE defined as according to II.B and II.C.

A. System Configuration

By a *system configuration* we mean a particular set of accelerators with defined type and index for each accelerator. Accelerator type is defined by a specification file in ISE format. Development environment GUI allows specifying an ordered list (indexed from 0) of accelerator specification files. This list defines a system configuration and is used by cross-system components for their dynamic reconfiguration.

System configuration may be changed multiple times within a single user session including changing the order of attached accelerators (indexing) and their ISE descriptions. However no change to system configuration is possible during debugging and simulation. The debugger must be stopped to change the system configuration. A visual front-end is available to simplify the process of editing accelerator specifications in ISE with analytical and validation features (e.g. for detecting binary conflicts for different instructions).

B. *Simulator Reconfiguration*

The simulator is configured by extracting information corresponding to the model parts described in II.B from accelerator description files.

For simulator reconfiguration two different approaches are possible in the current implementation:

– Compiling accelerator specification in ISE using an external C++ compiler.
– Accelerator specification run-time interpretation.

In the first case, an accelerator description file is translated by an external C++ compiler to a dynamic link library (DLL). Please note that the syntax described in II.B can be treated as a C++ program with C macros. The macros are expanded to variable definitions for II.B.1 or to C++ functions for II.B.2. The decoding function is generated as a table of correspondence between machine codes and instruction behavior functions (see II.B.2.c). Assembly syntax related information (II.C) is ignored by simulator and is translated by the compiler to a single text string initializer for a global variable that is used by assembler/disassembler.

The API of resulting DLL contains a collection of interface functions. Using these functions it is possible to get all the information about corresponding accelerator in run-time. Accelerator memory areas are represented by a set of variables and arrays inside the DLL address space. Accelerator instruction behavior functions and operations are translated to host-executable subroutines. The simulator of the core processor accesses the DLL API functions to issue a next accelerator instruction and perform cycle synchronization (the clock generator is managed by the debugger). Cycle driven synchronization between processor and accelerators modules is implemented by means of user manageable threads. For this purpose, we used Fibers API in MS Windows and qt library (QuickThreads library by David Keppel, 1993) in Unix. The `FinishCycle()` in this case is translated to an explicit switch between fibers.

If external C++ compiler is not available then the second approach is used where accelerator instruction behavior functions are interpreted by an internal virtual machine taking an ISE file as input. However this approach implies certain restrictions on the usage of C++ constructs and external libraries inside the accelerator description file, because not all of them are supported by the interpreter.

C. *Assembler/Disassembler Reconfiguration*

To provide a mechanism for dynamic assembler reconfiguration, a universal assembler was developed. This assembler interprets the description of assembly instructions syntax and binary mapping, defined according to II.B, in run-time during assembling of an application program. Note that accelerator independent part of the instruction set of the main processor is also defined in the form of II.B. Thus, besides the source of the application program, assembler input contains a set of the instruction set description files for the accelerator independent part of the instruction set of the main processor and instruction sets of all accelerators in the system. A union of these descriptions defines full assembly syntax for all instructions of the «processor + accelerators» system.

In the current implementation, the assembler extracts syntax specification either from the specified in the command line DLL libraries or directly from the accelerator description files. The description of the main processor instruction set is embedded into the assembler because it is not modified by user.

Note that, in the current implementation, only syntax for individual instructions is dynamically configured. General syntax of the assembly file structure is fixed:

– Sections
– Variables declaration
– Expressions
– Macro processor constructs
– C debug information

The disassembler also extracts instruction set information from the set of description files dynamically in run-time.

D. *Debugger Reconfiguration*

Debugger reconfiguration implies adjusting corresponding windows that display the state of accelerator memories during debugging including partitioning into named areas. The debugger allows users to change values of individual cells manually.

During initialization of a debug session the debugger either reads information about accelerator memories structure (see II.A.1.a) from the corresponding DLL libraries or directly from the accelerator description files. Also IDE editor dynamically uses syntax information for assembly instructions highlighting (see II.B). One of the profilers (namely the instruction set coverage profiler) extracts information from the same source too.

IV. Related Work

To specify hardware at the detailed level, *hardware description languages* (HDL) are used, most famous of which are **Verilog** [6] и **VHDL** [7]. The purpose of these languages is creation of specifications ready for real HW synthesis for silicon production. That is why hardware specification at this level of abstraction is laborious and is not suitable for fast *design space exploration* (DSE). Also automatic cross development tools construction based on HDL specifications is problematic because explicit instruction set description is missing (see [1]).

An interesting approach to model hardware structure is being developed under the Open SystemC Initiative [8], initially presented in 2000. Currently all activities on

SystemC are sponsored and controlled by a committee involving industrial companies such as: ARM, Cadence, CoWare, Fujitsu, Mentor, Motorola, NEC, Sony, ST, Synopsys. SystemC is a C++ class library that simplifies creation of hardware system models by using library macros and classes that implement constructs similar to those in HDL. With these constructs (see [9]) it is possible to specify a hardware system structure using the notions such as modules, processes, ports, signals, interfaces, events, etc. The library also provides a set of data types simplifying hardware elements modeling such as bit strings, fixed point numbers, logical (four state) numbers, etc. However SystemC models are used only for simulation and further HW synthesis, instruction set is not explicitly defined so automatic cross toolkit construction is problematic based on SystemC (see [5]). In this sense SystemC is closer to HDL languages, automatic converters exist from Verilog and VHDL to SystemC ([10] и [11]). Simulation speed for SystemC models is slow because of the too low level hardware details description that is inessential for cross system. However one should note that HW behavior description primitives in SystemC are quite convenient to describe operations behavior, similar constructs are used in the considered approach for operations and instruction behavior functions descriptions (see II.B.2)). In particular, we provide similar data types for accelerator designers as in SystemC; also our cycle synchronization is very similar to the SystemC processes synchronization (`FinishCycle()` is like `wait()` in SystemC).

To address the task of automatic cross development environment generation based on high level hardware specifications, architecture description languages (ADLs) are being researched in various academic and industrial organizations. Additional information and full overview of existing ADL languages can be found in [1] - [5]. Below only most famous languages are described.

One of the first ADL languages was **nML** [12], initially developed at Technical University in Berlin, Germany (1991). nML was used as a specification language for SIGH/SIM simulator and CBC compiler generation. In nML processor instruction set is described using attribute grammars. Attributes include behavior (action), assembly syntax (syntax) and mapping to machine codes (image). Original nML does not support multi-cycle functional units and multi-word instructions. However nML took further research at IMEC, where under the subsidiary company Target Compiler Technologies a commercial environment was created [13]-[14], oriented to DSP architectures. This environment includes CHESS compiler, simulator CHECKERS, assembler, disassembler and linker. VHDL synthesis is also supported. nML was improved by Target Compiler Technologies to support more complex hardware modeling (in particular pipeline specification introduced, although from available marketing reports (technical specification are proprietary) it is not clear what specification mechanisms are supported. Also nML supports only fixed duration instructions and simulator performance reported in [14] is not high compared to other approaches.

A follower of nML is **Sim-nML** [15], developed since 1998 at Indian Institute of Technology Kanpur under Cadence company support. The main new feature of Sim-nML is a new attribute for resources usage (uses) in the instruction set specification grammar. This allows describing resource conflicts between instructions. In this project they developed compiler code generator, simulator, assembler and disassembler. Unfortunately no results on debugging environment are available.

ISDL was developed at MIT, USA [16] and presented at DAC [17] in 1997. The main ISDL focus is on specification of VLIW architectures. Similar to nML ISDL mainly specifies processor instruction set including behavior semantics, assembly syntax, mapping to binary codes and also resource conflicts using an attribute grammar. Explicit pipeline description is missing but conflicts and delays can be specified in the form of logical rules. Unfortunately no real tools supporting ISDL are available. Project team limited tools implementation only to assembler and compiler code generator under some PhD work at MIT.

EXPRESSION [18]-[19] was developed at University of California, Irvine, USA and initially presented at DATE in 1999. This language supports a wide class of embedded systems with ILP and memory hierarchies: RISC, DSP, ASIP, VLIW. EXPRESSION contains an integrated description for structure and behavior of processor-memory subsystem. An EXPRESSION hardware model specification contains six sections (the first three are for behavior; the latter three are for structure description):

– Operations (set of atomic commands with codes, operands and behavior specification)
– Instruction formats (instruction consists of slots responsible for particular functional unit; each slot can be filled with atomic operations for parallel execution)
– Mapping of general compiler operations to machine operations. This section is used by compiler code generator.
– Components (functional units, buses, ports, etc.).
– Pipeline and components links.
– Memory hierarchy (register memory, cache, SRAM, DRAM).

Based on EXPRESSION description compiler EXPRESS and simulator SYMPRESS are automatically generated. Unfortunately simulation performance is not good, also due to the rather detailed structural description it is not easy to quickly specify hardware using EXPRESSION. In this sense EXPRESSION is in the middle between behavioral ADLS (like nML) and HDL languages.

LISA [21]-[22] language was developed at RWTH Aachen (Germany) and was initially designed for hardware specification for automatic simulator generation. The first results were reported in 1996. The initial target architectures were DSP processors. Key characteristic of LISA is detailed pipeline specification at operations level with dependencies and latencies definition. Pipeline conflicts are explicitly specified. Each instruction is defined as a set of operations that do register transfers during one synchronization period. Description in LISA consists of two parts: resources specification and operations specification. Operation specification in its turn contains the following sections:

– DECLARE (objects and groups definitions based on other objects – actually like grammar rules)
– CODING (binary coding for operations)
– SYNTAX (assembly syntax and operands)
– BEHAVIOR и EXPRESSION (behavior description using C/C++ like script language)
– ACTIVATION (timings and pipeline behavior specification)

Unfortunately full public LISA language reference is not available.

According to [20] among all the mentioned approaches, only the system based on EXPRESSION supports explicit capturing of processor–coprocessors architecture. Although this system is aimed at design space exploration only and does not support such production components as assembler, disassembler and debugger. Also, none of the systems above support dynamic system retargetability, because all the cross system components are created using special generators in the form of high level programming language programs requiring subsequent external compilers usage to be built. Customizable error reporting is also not in the focus of the systems above, though this is very important for production use.

V. Conclusion

This article presented a technology for dynamic reconfiguration of cross development tools to support user defined processor extensions. In real hardware, such extensions are implemented by special accelerators (coprocessors) designed while constructing an application specific configuration of the “processor + accelerators” system.

To solve this task, we developed a generalized model for a wide class of supported accelerator architectures and defined an interface with the main processor. We proposed a specification language (ISE) that allows describing exact accelerator models within the generalized one. Accelerator description in ISE covers memory structure and accelerator instruction set including cycle accurate behavior, assembly syntax and mapping to binary codes. Sophisticated mechanisms are available for customizing error detection rules both statically during assembling (e.g. resource and pipeline hazards detection for accelerator instructions by the assembler) and in simulation run-time. Special tools for visual editing, validation and analysis of ISE specifications have been developed. A set of ISE description files reflects a custom configuration of the “processor + accelerators” system. Given this configuration, cross development tools are dynamically (in run-time) reconfigured to accept new instructions and accelerator memories during assembling/disassembling, simulation, debugging and profiling. In result, it is possible to interactively (within one working session) adapt cross development tools for user specific accelerator models without using external compilers. This is a key feature for using the cross tools for prototyping various design alternatives of “processor + accelerators” system. At the same time, when adding extensions, the resulting cross tools remain of production quality suitable for real life application development.

Based on the introduced technology, 3 commercial cross development systems for proprietary DSP core processors (with accelerator interface support) have been developed at ISP RAS. The systems are being successfully used in production and the following real accelerator models have been developed by customers using our approach:

– Fast Fourier Transform (FFT).
– Echo cancellation algorithms.
– Complex (imaginary) arithmetic operations.
– Image processing operations (JPEG accelerator).
– Digital voice filtering operations (FIR, IIR).
– Voice coding/decoding (AMR).
– MP3 music decoding.

Peak performance of the core processor simulator (on Intel Core 2 Duo 2000MHz host machine) is about 50 million cycles per second. In the presence of an accelerator (in particular echo cancellation) the system simulation performance is about 5 million cycles per second for processor and accelerator running in parallel, which is due to the big overhead for synchronization of processor and accelerator fibers.

The ongoing work targets at extending ISE language and developing a method to support full system modeling including main processor specification.

References

[1] *Hiroyuki Tomiyama, Ashok Halambi, Peter Grun*. Architecture Description Languages for Systems-on-Chip Design. Center for Embedded Computer Systems, Univertsity of California. 2000.

[2] *Wei Qin, Sharad Malik*. Architecture Description Languages for Retargetable Compilation. The Compiler Design Handbook, CRC Press, 2003.

[3] *Clifford Liem, Pierre G. Paulin, Ahmed A.Jerraya*. Retargetable Compilers for Embedded Core Processors. Kluwer Academic Publishers, 1997.

[4] *Rainer Leupers*. Retargetable Code Generation for Digital Signal Processors. Kluwer Academic Publishers, 1997.

[5] *Lin Yung-Chia*. Hardware/Software Co-design with Architecture Description Language. Programming Language Lab. NTHU. 2003.

[6] IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language, IEEE Std 1364-1995.

[7] IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987

[8] Open SystemC Initiative. http://www.systemc.org/

[9] SystemC User’s Guide. Version 2.0.1. http://www.systemc.org/

[10] *N.Agliada, A.Fin, F.Fummi, M.Martignano, G.Pravadelli*. On the Reuse of VHDL Modules into SystemC Designs, FDL 2001

[11] *Leila Mahmoudi Ayough Ali Haj Abutalebi Omid F. Nadjarbashi Shaahin Hessabi*. Verilog2SC: A Methodology for Converting Verilog® HDL to SystemC. HDLCon 2002

[12] *A. Fauth, J. Van Praet, M. Freericks*. Describing Instruction Set Processors Using nML. Proc European Design and Test Conf., Paris, March 1995.

[13] Chess/Checkers Products. Target Compiler Technology. http://www.retarget.com/

[14] *Mark R. Hartoog, James A. Rowson, Prakash D. Reddy*. Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign. Alta Group of Cadence Design Systems, Inc. DAC 1997.

[15] Sim-nML Homepage. http://www.cse.iitk.ac.in/sim-nml/

[16] ISDL Project Homepage. http://caa.lcs.mit.edu/caa/home.html

[17] *George Hadjiyannis, Silvina Hanono*. ISDL: An Instruction Set Description Language for Retargetability. Srinivas Devadas. Department of EECS, MIT. DAC 1997.

[18] EXPRESSION Homepage. http://www.cecs.uci.edu/~aces/index.html

[19] *Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt and Alex Nicolau*. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability, DATE 99.

[20] *Prabhat Mishra, Frederic Rousseau, Nikil Dutt, Alex Nicolau*. Architecture Description Language Driven Design Space Exploration in the Presence of Coprocessors. SASIMI 2001.

[21] *V. Zivojnovic, S. Pees, and H. Meyr*. LISA: Machine description language and generic machine model for HW/SW co-design. In Proc. of Workshop on VLSI Signal Processing, 1996.

[22] *Andreas Hoffmann, Achim Nohl, Stefan Pees, Gunnar Braun, Heinrich Meyr*. Generating Production Quality Software Development Tools Using a Machine Description Language. DATE 2001.

[23] *N. Cutland*. Computability: an introduction to recursive function theory. Cambridge University Press, 1980.