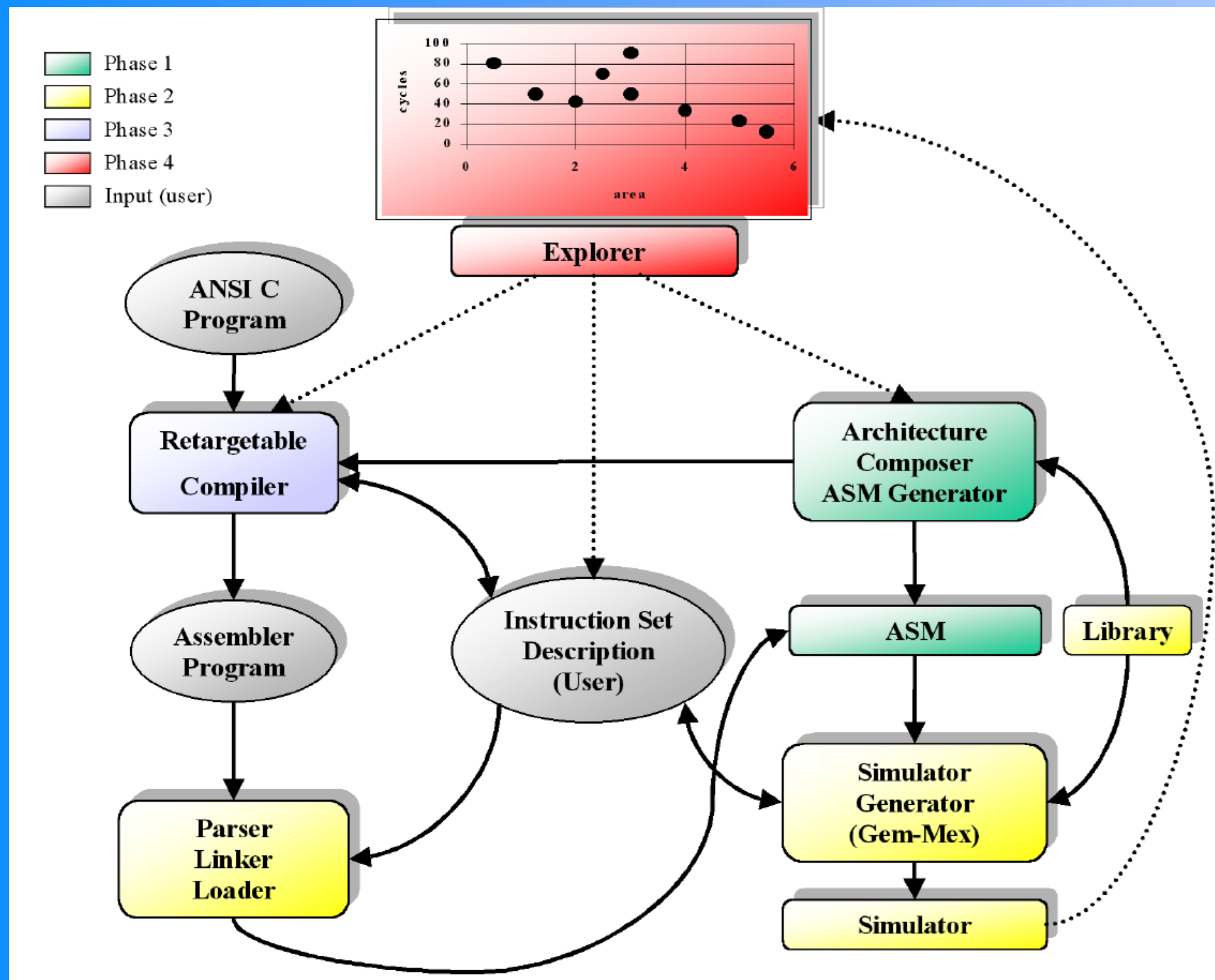


# Modeling and Simulation of Embedded Processors Using Abstract State Machines

Dirk Fischer, Jürgen Teich, Ralph Weper

{fischer,teich,weper}@date.uni-paderborn.de



## The BUILDABONG Project

(Building Special Computer Architectures Based on Architecture and Compiler Co-Generation)

<http://www-date.upb.de/RESEARCH/BUILDABONG/buildabong.html>

### Phase 1: Architecture Entry and Composition

An object-oriented tool for hierarchical graphical entry and composition of a processor architecture (*ArchitectureComposer*) has been developed from which an ASM description of the resulting architecture is generated automatically. For this purpose, a library of common high-level components to compose an architecture (e.g., address generation units, buses, ALUs, memory units, register files, etc.) has been defined being parameterizable in bit width, number of inputs, etc.

### Phase 2: Architecture Simulation

The Architecture Composer generates the ASM description of the processor behavior from which subsequently, a cycle-accurate and bit-true instruction set simulator is generated using the *Gem-Mex* environment for ASM prototyping. This tool supports also the customization of a parser for assembler notation.

### Phase 3: Compiler Generation (Retargeting)

In order to allow the compiler to exploit architectural changes, e.g., number of functional units, distributed register files, etc., the necessary information for code generation and instruction scheduling is extracted from the graphical description. Also, the compiler needs a description of the instruction set (syntax, coding, reservation tables, etc.).

### Phase 4: Architecture/Compiler Optimization

The final goal of the project is to provide an exploration framework for joint architecture/compiler co-generation. For this purpose, an exploration tool has to be developed which adds profiling functions to the ASM description that are evaluated during simulation, and is responsible to explore the design space of architecture and compiler changes.

### Recent Publications:

J. Teich, P. Kutter, R. Weper: Description and Simulation of Microprocessor Instruction Sets Using ASMs. In: *International Workshop on Abstract State Machines*. Lecture Notes in Computer Science (LNCS) 1912, pp. 266-286, Springer-Verlag, October 2000.

J. Teich, R. Weper, D. Fischer, S. Trinker: BUILDABONG: A Rapid Prototyping Environment for ASIPs. In: *Proceedings DSP-Deutschland 2000*, pp. 153-162, Munich, Germany. WEKA Fachzeitschriften Verlag, October 2000.

J. Teich, R. Weper, D. Fischer, S. Trinker: A Joined Architecture/Compiler Environment for ASIPs. In: *ACM SIG Proc. International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES 2000)*. San Jose, CA, U.S.A., pp. 26 - 33, November 2000.

## Abstract State Machines (ASM)

An ASM  $M = (V, f_1, \dots, f_r)$  is a first order algebra with:

- $V$  being a finite vocabulary, called *SuperUniverse* and
- $f_i$  being a finite set of  $n$ -ary functions over  $V$ .

States of  $M$  are structures (resp. algebras) over  $V$ .

An ASM  $M$  is defined by an initial state  $S_0$  and a program  $P$  (*Update Rules*) consisting of a finite set of transition rules each of the form:

• if  $\langle \text{Cond} \rangle$  then  $\langle \text{Rule} \rangle$  endif, (Conditional)

with  $\langle \text{Cond} \rangle$ ,  $\langle \text{Rule} \rangle$  being terms over  $V$ .

•  $\text{Rule}_1 \text{ Rule}_2 \text{ Rule}_3 \dots \text{Rule}_n$  (Block)

with  $\langle \text{Rule}_i \rangle$  being terms over  $V$ .

•  $f(t_1, \dots, t_n) := t$  (Update)

with  $f(t_1, \dots, t_n)$ ,  $t$  being terms over  $V$ .

$f$  denotes an arbitrary  $n$ -ary function and the terms  $t_1, \dots, t_n$  denote a sequence of parameters. Then, a (function) update  $f(t_1, \dots, t_n) := t$  sets the value of  $f$  at  $f(t_1, \dots, t_n)$  to  $t$ .

### Operational Semantics of ASMs:

- The effect of a transition rule  $R$  when applied to a state  $S_i$  (which itself is an algebra) is to produce another algebra (or state)  $S_{i+1}$  which differs from  $S_i$  by the new values for those functions at those arguments where the values are updated by the rule  $R$ . If  $\langle \text{Cond} \rangle$  is true, the rule can be executed by simultaneously executing each update in the set of Updates.
- An ASM terminates when it runs into a fixed point  $S_n$  which evokes no further Updates.

### Advantages of our Approach using ASMs

- Shortness and readability of descriptions (e.g., only 200 lines of XASM code needed to specify an ARM processor)
- Component based library concept
- Full support of the specification environment XASM
- Bit-true simulation of irregular arithmetic operations on arbitrary large word-lengths using a C-based library of configurable standard functions based on arbitrary precision numbers (GNU-MP) since the inclusion of C-libraries is strongly supported by the XASM environment.
- Support of the automatic simulator generator *Gem-Mex*
- Cycle accurate simulation

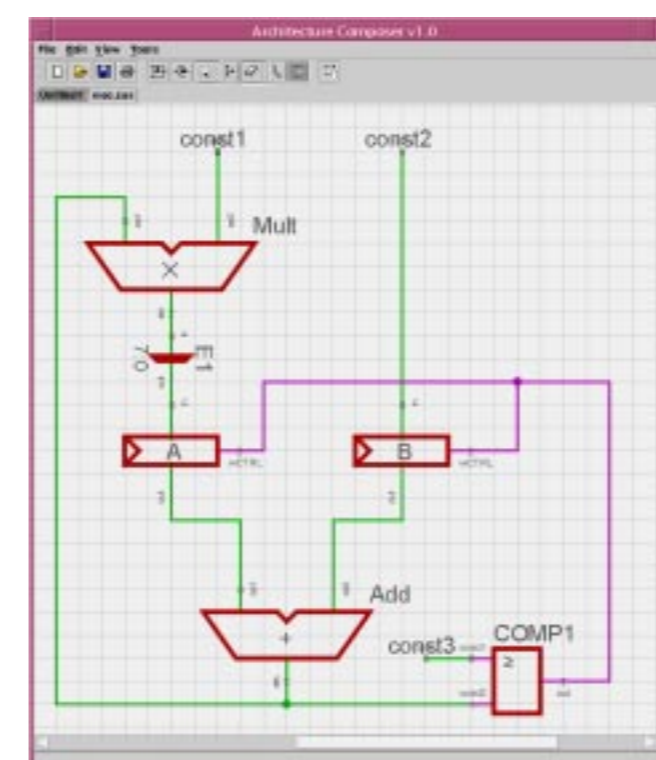
## XASM Code Generation

- From the graphical entry of an architecture's control- and data path which are composed of basic components of an architecture library, the tool *ArchitectureComposer* automatically generates an ASM model of the processor.
- We distinguish two kinds of hardware components: **combinational** elements (e.g. comparators, multipliers, adders, and **sequential** elements (registers, memories, etc.).
- Non-hierarchical combinational elements are modeled by an external library of C-functions providing the necessary arithmetic and logical functions.
- Using the XASM construct *derived function*, an alias for the respective function call is declared.
- For each sequential element, an XASM function is generated which in turn is used in the update rules.

Code generation is performed in three phases:

1. Generation of output signal declarations: For each output signal of each hardware component, a function declaration is generated. A function represents the output signal of a hardware component.
2. The functions corresponding to sequential elements (e.g. memory) are initialized in a separate block encapsulated by the keywords *init* and *endinit*.
3. Memory and register assignments are encoded as guarded update rules.

```
asm MAIN is
use cpucore
derived function Mult_res==c_mult (Add_res, const1_out,8,COMPL_2)
derived function Add_res == c_add (A_out, B_out, 8, COMPL_2)
derived function E1_out == c_extract (Mult_res, 16, 0, 8)
derived function COMPL_out == c_gteq(const3_out,Add_res,8, COMPL_2)
function A_out
function B_out
function const1_out
function const2_out
function const3_out
init
A_out := "00000000"
B_out := "00000000"
const1_out := "00000010"
const2_out := "00000001"
const3_out := "00001000"
endinit
if COMPL_out = "1" then A_out := E1_out
endif
if COMPL_out = "1" then B_out := const2_out
endif
endasm
```



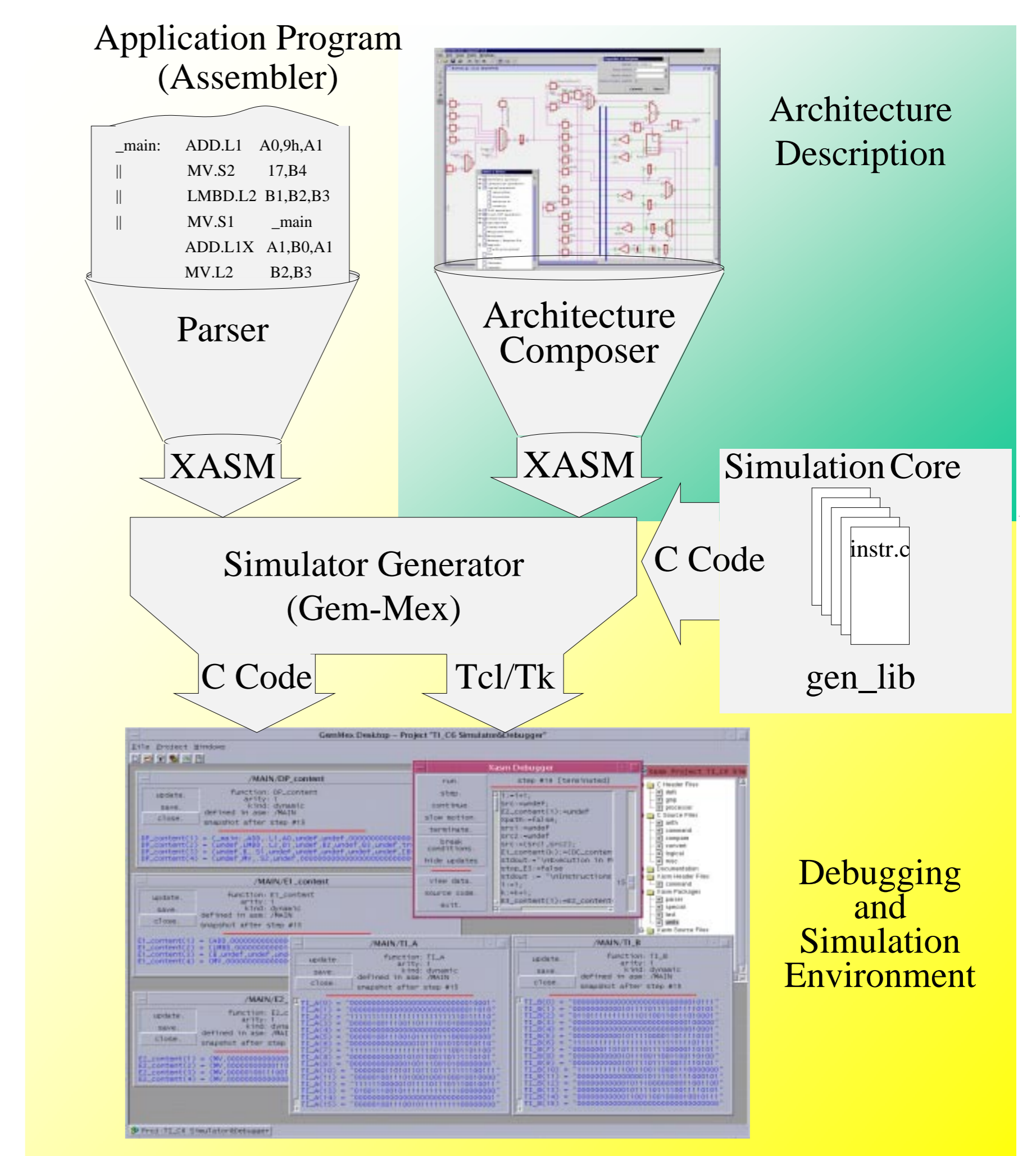
## Simulation Workflow

### ArchitectureComposer

- Graphical entry of the architecture
- Automatic generation of XASM code

### Gem-Mex

- Call of the XASM compiler, linking of a simulation core consisting of the library *gen\_lib* of generic C-functions => Generation of C-code
- Call of the C-compiler, linking of Tcl/Tk library => Automatic generation of an interactive debugger and simulator environment
- Input of the application program (assembler notation)
- Parsing of the input to XASM-objects
- Program start



The current implementation state of the project BUILDABONG: workflow of the simulation of a DSP of the Texas Instruments TMS 320C6201 series. Note, that in this case study, the ASM model was still handwritten.