CASM
Simulator Synthesis & Model Verification
of the MIPS I architecture

Specification

A specification of a MIPS I instruction set and execution model is given in CASM. The whole specification only needs 700 lines of CASM code. 600 lines are needed to model all instructions, the rest is used to describe the state and the execution model.

Instruction Set Models

```
rule andi(addr:Int) =
let rs = PARG(addr, FV_RS) in
let rt = PARG(addr, FV_RT) in
let imm = PARG(addr, FV_IMM) in
    if rt != 0 then
        GPR(rt) := BVand(32, GPR(rs),
                         BVZeroExtend(imm, 16, 32)))
```

Execution Model

This rule defines one computation step of the CASM program. It is executed until the MIPS program triggers a trap.

```
rule run_program dumps (GPR, LO, HI)->trace  =
let branch = BRANCH in
    seqblock
        debuginfo trace "executing @" + hex(PC)
        call (PMEM(PC)) (PC)
        CYCLES := CYCLES + 1
        if branch = undef then
            PC := PC + 4
        else {
            PC := BRANCH
            BRANCH := undef
        }
        if trapped then {
            print "program stopped (trapped)"
            program(self) := undef
        }
    endseqblock
```

Execution State

This is the relevant state needed to execute MIPS programs. It is also used by the pipelined implementations, although additional state is needed to model pipeline registers. Only this subset is used to perform verification.

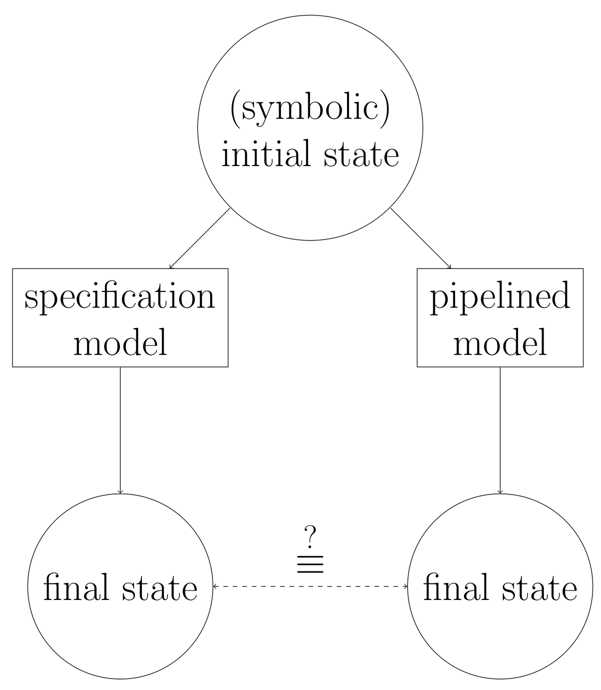
```
function (symbolic) GPR : Int -> Int
function (symbolic) LO : -> Int
function (symbolic) HI : -> Int
function PC : -> Int
function CYCLES : -> Int
function trapped : -> Boolean

function BRANCH : -> Int
```

The CASM language

- based on Abstract State Machine (ASM)
- statements (rules) produce *update sets*
- all rules are *side-effect free*
- update sets are merged either parallel or sequential
- resulting update set applied when computation step concludes
- interleaving: state transformation - rule evaluation - state transformation - rule evaluation - . . .
- efficient compilation
- symbolic execution (generating first-order logic predicates)

Model Verification (using first-order logic)

The Problem: Are the pipelined instruction models combined with the pipeline and execution model *coherent* to the specification (instruction set models and its execution model)?
Solution: For each instruction both models are *symbolically* executed using the same *initial state*. A conjecture stating that the final states are equal is emitted. The fully-automated theorem prover vampire is used to perform the simulation proofs:



Simulator Synthesis (compiling CASM to C++)

- Models compiled to C++
- ELF loader written in C++
- MIPS instruction decoder interfacing CASM written in C++
- MIPS *syscall* interfacing the host C library written in C++
- C++ library implementing arithmetic operations on bit vectors
- need to link MIPS program to simulator specific C library
- able to access host file system and perform terminal IO

Implementation

Two pipeline implementations were developed, one implementing operand forwarding and one stalling on data hazards. Both use the very same pipelined models of the instruction set. The instruction set needs 1500 lines of CASM code, the pipeline models approximately 400.

Pipelined Instruction Models

An implementation of the instruction using the classic 5-stage pipeline. Each stage consists of 2 phases, begin (latching input signals) and end (outputs are available), which allows to model inter-stage concurrency.

```
rule andi(addr:Int, stage:Int, phase:Int) = {
    if stage = ID and phase = end then
    let rs = PARG(addr, FV_RS) in
    let rt = PARG(addr, FV_RT) in
    let imm = PARG(addr, FV_IMM) in {
        call(ID_READ_OP1)(rs)
        IDOP2 := BVZeroExtend(imm, 16, 32)
        IDRESREG := rt
    }
    if stage = EX and phase = begin then {
        EXRES := BVand(32, EXOP1, EXOP2)
    }
    if stage = WB and phase = begin then
        call (WRITE_REGISTER)(WBRESREG, WBRES)
}
```

Operand Forwarding

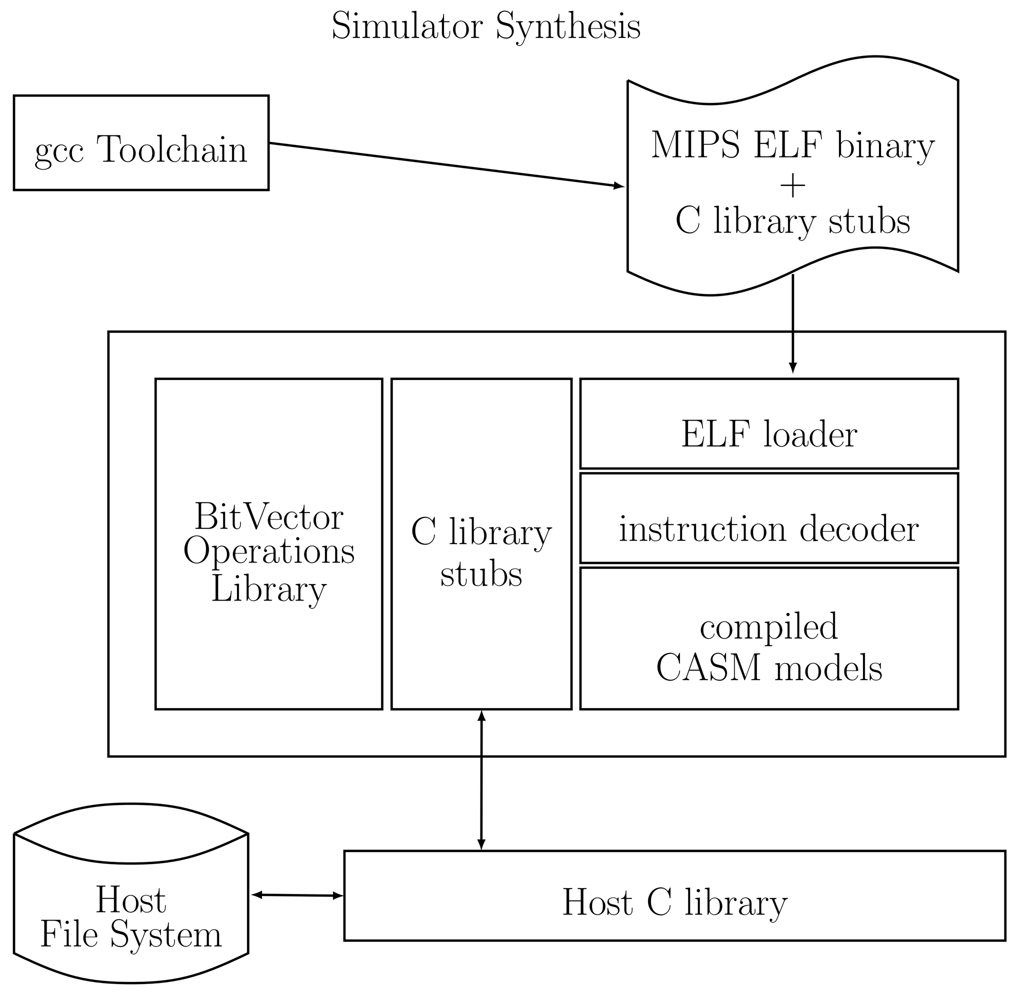
```
rule ID_READ_OP1(reg : Int) =
    if EXRESREG = reg then {
        IDOP1 := EXRES
    } else if MEMRESREG = reg then {
        IDOP1 := MEMRES
    } else
        IDOP1 := GPR(reg)
```

Atomically execution of pipelined models

Execution of pipelined instruction models, the 2 phases (begin, end) are executed sequential using an intermediate (temporary) state. The *seqblock* makes them to appear as a combined atomic state transformation.

```
rule execute_pipeline =
seqblock
    forall s in PipelineStages do
        let op = pipeline(s) in
            if op != undef then
                call (PMEM(op))(op, s, begin)
    forall s in PipelineStages do
        let op = pipeline(s) in
            if op != undef then
                call (PMEM(op))(op, s, end)
endseqblock
```

Simulator Synthesis



Simulator Performance (MiBench and SpecInt)

This work is partially supported by the Austrian Research Promotion Agency (FFG) under contract 827485, Correct Compilers for Correct Application Specific Processors and Catena DSP GmbH.

Roland Lezuo <rlezuo@complang.tuwien.ac.at>
Andreas Krall <andi@complang.tuwien.ac.at>

Vienna University of Technology
Institute of Computer Languages (E185)
Argentinierstraße 8
1040 Vienna, Austria