UNIVERSITY OF ILLINOIS

AT URBANA-CHAMPAIGN

Atmel Micro-controller

and AVR ISA Platform Level Verification

By Joseph F. Girotti

The Idea

- Inspiration: System Level Simplex
  - Toolchain: AADL → Maude → VHDL
- ISA Level
  - Platform matters
    - Ariane 5 Flight 501
      - 64 bit logical → 16 bit Physical
  - Lowest level before Hardware

The Result

- Not as expected
  - ISA level creates a POST synthesis checker
  - Still important: Allows checking of program properties after library linking, and assembly
    - Likely source of Ariane 5 flight 501 bug

The Details

- Maude brief
- The Turing Machine Model
- ISA brief
- ISA Architecture
- ISA Implementation
- Bootloader Details
- The Inductive Theorem Prover (ITP)

Maude

- Declarative language (mathematical axiomatization)
- Rewriting logic
  - Equational logic is a subset of rewriting logic
  - Handles concurrent and deterministic programs

The Turing Machine Model

- a piece of machinery that carries out tasks by successively applying sequences of instructions from a finite set of instructions
- From: “Turing Machines”
  - http://fsl.cs.uiuc.edu/index.php/Main_Page

Stream a sequence of instructions

```
mod STREAM is
   sorts Bit Stream .
   ops 0 1 : -> Bit .
   op _:_ : Bit Stream -> Stream .
   op zeros : -> Stream .
endm
```

Basic Turing Machine

```
mod TURING-MACHINE is including
  STREAM.
   sorts State Configuration .
   op _`(_,_`) : State Stream Stream ->
     Configuration .
   var S : State . var L R : Stream .
   eq S(zeros,R) = S(0 : zeros, R) .
   eq S(L,zeros) = S(L, 0 : zeros) .
endm
```

Turing Successor

```
mod TURING-MACHINE-SUCC is including TURING-
  MACHINE .
   ops qs qh q1 q2 : -> State .
   var L R : Stream . var B : Bit .
   rl qs(L, 0 : R) => q1(0 : L, R) .
   rl q1(L, 0 : R) => q2(L, 1 : R) .
   rl q1(L, 1 : R) => q1(1 : L, R) .
   rl q2( L, 0 : R) => qh(L, 0 : R) .
   rl q2(B : L, 1 : R) => q2(L, B : 1 : R) .
endm
```

The AVR 8-Bit ISA

- RISC architecture with ~131 instructions dependent on implementation
  - 28 arithmetic and logic instructions,
  - 36 branch instructions,
  - 28 bit and bit-test instructions,
  - 35 data transfer instructions,
  - four MCU instructions.

The AVR 8-Bit Architecture

Program Counter

Status & Control

FLASH

CPU

EEPROM

SRAM

Creating the Module

```
fmod ATMEGA is
  protecting ATMEGASTATE .
  op run : ATmegaState -> ATmegaState .
  var A : ATmegaState . var F : Flash .
  var S : SRAM . var E : EEPROM .
  var PC : UInt8 . var N : Nat.
  vars Rd Rr V1 V2 V3 k : UInt8 .
endm
```

The Run Operation

- Each Instruction gets an equation
- ```
  ceq run(pccounter(PC) flash([PC, DEC
  Rd] F) sram([Rd, V1] [63, V3] S)) =
  run(pccounter(PC + 1) flash(F [PC,
  DEC Rd]) sram([Rd, (V1 - 1)] [63,
  setsr10(V3, (V1 - 1))] S))
   if (0 <= Rd and Rd <= 31) .
  ```

  ```

# Opti-Bootloader

- Allows programming of the AVR via the UART vice an ICSP programmer
- Implements the STK500v2 8bit protocol
- 512 Bytes
- Uses 37 instructions

# How it works

- UART Interrupt vector resets micro-controller
- Bootloader starts and checks UART
  - If STK500v2 commands received process them
  - After a certain time with no STK500v2 commands start application

# Simplification

- The watchdog timers job is to determine if we are programming, or booting the application. Therefore since we are testing the programming we can remove the watchdog timer functionality, and it's configuration

# Adding the UART

- Our initial model does not contain the UART. Implement it as two byte arrays with position counters
  - Uartic: the counter for input uart
  - Uarti: byte array for input uart
  - Uartoc: the counter for output uart
  - Uarto: byte array for output uart
- ATMEGASTATE now looks like

```
ATMEGAState=(pccounter(pc)flash(fvals)
   sram(svals)eeprom(eevals)uartic(val)
   uarti(uivals) uartoc(val) uarto(uovals))
```

Maude ITP

- Maude has an Inductive Theorem Prover
  - Similar to PVS
- Define a module for our check functions
  - `Fmod FRAME-RUN-REQUIREMENTS is protecting ATMEGASTATE. endm`
- Define a fuctional theory to prove
  - `Fth RUN-REQUIREMENTS is protecting FRAME-RUN-REQUIREMENTS. endth`

What to Check

- There are two primary things we need to check
  - **Completion:** The program always runs to completion
  - **Correctness:** The loaded program always matches the sent program

Completion

- Find point in code that signals completion
- Make sure there is no legal rewrite rule
- We can then check that we **always** reach

```
run(ATMEGAState)=run(pccounter(INSTR_PC)
  flash([PC, INSTR] fvals) sram(svals)
  eeprom(eevals) uartic(val)
  uarti(uivals) uartoc(val)
  uarto(uovals))
```

Correctness

- Compare end program flash with UART input code

```
ProgFlash(run(ATMEGAState)=
  ProgUART(run(ATMEGAState)
```

Results

- Basic Instruction Tests
  - ~500 rewrites @ 150,000 rewrites/second
  - reduce in ATMEGA : run (pccounter ([0,0]) flash(F [0,0,SBC 1 2]) sram(((S [0,63, 1]) [0,2,1]) [0,1,10])) .
  - rewrites: 613 in 4ms cpu (0ms real) (153250 rewrites/second)
  - result [ATmegaState]: run(flash(F [0,0,SBC 1 2]) sram(S [0,1,10] [0,2,1] [0,63,0]) pccounter([0,1]))
- Completion & Correctness
  - Forthcoming

Future Work

- Add more instructions
- Implement Watchdog Timer functionality
- Implement other asynchronous behavior
- Port to Real Time Maude for Worst Case Execution time verification