Introduction

**Motivation:** Functional correctness testing is typically >50% of the cost of designing a processor

**Focus of this unit:**
- Exploring research in this space
  - Including published commercial practise
- Comparing against practises in the RISC-V open source community

Types of testing

- **Manufacturing test**
  - Check that the design has been manufactured without defects
  - Important, but not the focus here

- **Functional correctness testing**
  - Does the processor design comply with the Instruction Set Architecture (ISA)?
  - The ISA is the hardware/software interface
    - Violations break software

- **What is “verification”?**
  - Often used to mean “thorough testing”
  - When “formal verification” people use “verification” they mean rigorous (often machine-checked) mathematical proof, or model checking
What is an architecture reference?

- The architecture reference is the contract/interface between hardware and software
  - Includes the ISA
  - Also exception mechanisms (including interrupts), page table definition, etc.
  - May also include a programmable interrupt controller
- Examples:
  - RISC-V: [https://riscv.org/specifications/](https://riscv.org/specifications/)
    - 236 pages unprivileged spec, 91 pages privileged spec, 94 pages external debug spec
    - 7900 pages
- Often the architecture reference is often written in English
  - Can lack rigour; can be ambiguous
  - May include code specifying the instruction

Example from the ARM arch. ref. manual

C6.2.44 CBNZ

Compare and Branch on Nonzero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

32-bit variant

Applies when sf == 0.

```
CBNZ <Wt>, <label>
```

64-bit variant

Applies when sf == 1.

```
CBNZ <Xt>, <label>
```

Decode for all variants of this encoding

```
integer s = sf(1:0);
integer base = (1 if sf = "0" then 0 else 32);
bits(32) offset = signextend(imm19:'00', 64);
```
Example from the ARM arch. ref. manual

**Shared Pseudocode**

**Armv8 Pseudocode**

ARM arch. ref.: shared subfunction example

Example from the ARM arch. ref. manual

**Snippet of formal spec.**

**ARM arch. ref.: shared subfunction example**

**Specifying the RISC-V ISA**

- RISC-V architecture reference manual is just in English
- Spike simulator often used as a “golden reference”
- Recent work in Prof Sewell’s group in Cambridge:
  - Formal RISC-V specification in Sail
  - Now ratified by the RISC-V foundation
  - [https://github.com/rems-project/sail-riscv](https://github.com/rems-project/sail-riscv)
Example: RISC-V - B-type branch instructions

- if(condition) pc = pc + imm
- Note: imm is signed and imm bit zero=0
- BEQ: condition = rf[rs1] == rs2
- BNE: condition = rf[rs1] != rs2
- BLT: condition = rf[rs1] < rs2
- BGE: condition = rf[rs1] >= rs2

RISCV_BLTU => rs1_val <_u rs2_val,
RISCV_BGE  => rs1_val >=_s rs2_val,
RISCV_BNE  => rs1_val != rs2_val

<table>
<thead>
<tr>
<th>funct3 specifies conditional</th>
<th>no rd so use bits for immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>---------------------------------</td>
<td>-----</td>
</tr>
</tbody>
</table>

Sail RISC-V example: branches (1 of 2)

mapping encode_bop : bop <-> bits(3) = {
    RISCV_BEQ  <-  0b000,
    RISCV_BNE  <-  0b001,
    RISCV_BLT  <-  0b100,
    RISCV_BGE  <-  0b101,
    RISCV_BLTU <-  0b110,
    RISCV_BGEU <-  0b111
}

function mapping branch operand (bop) to an enumeration

mapping clause encode = BTYPE(imm7_6 @ imm5_0 @ imm5_4_1 @ 0b0, rs2, rs1, op)  
    <- imm7_6 : bits(1) @ imm5_7_0 : bits(6) @ rs2 @ rs1 @ encode_bop(op) @ imm5_4_1 :  
        bits(4) @ imm5_0 : bits(1) @ 0b100011

Sail RISC-V example: branches (2 of 2)

function clause execute (BTYPE(imm, rs2, rs1, op)) = {
    if taken then {
        /* Extensions get the first checks on the prospective target address. */
        match control_check_pc(r);
        Ext_ControlAddr_error(o) = {
            ext_handle_control_check_error();
            RETIRE_FAIL
        },
        Ext_ControlAddr_OK(target) -> {
            if bit_to_bool(target[1]) & (~ (haveVU))) then {
                handle_mem_exception(target, E_Fetch_Addr_Align());
                RETIRE_FAIL;
            } else {
                set_next_pc(target);
                RETIRE_SUCCESS
            }
        }
    } else {
        RETIRE_SUCCESS
    }
}
Challenge and approaches

- **Challenge**
  - Processors have lots of internal state
  - Some programmer visible…
  - …some less so
    - e.g. register colouring maps programmable visible register names onto a larger register file
    - allows data hazards due to false sharing to be avoided
    - helps with exception handling (preserve old register state in case for roll-back)

- **Tests often in the form of instruction sequences**
  - Handwritten
  - Templated/generated/constrained-random

---

Example RISC-V test (from https://github.com/riscv/riscv-tests)

```c
#include "riscv_test.h"
RVTEST_RV64U        # Define TVM used by program.

# Test code region.
RVTEST_CODE_BEGIN   # Start of test code.
lw x2, testdata
addi x2, 1
sw x2, result
li x3, 42
bne x2, x3, fail
RVTEST_PASS        # Signal success.

fail: RVTEST_FAIL
RVTEST_CODE_END     # End of test code.
```

```c
# Input data section.
# Define TVM used by program.
RVTEST_RV64U

# Test code region.
RVTEST_CODE_BEGIN   # Start of test code.
lw x2, testdata
addi x2, 1  # Should be 42 into $2
sw x2, result
li x3, 42   # Desired result
bne x2, x3, fail # Fail out if doesn’t...
RVTEST_PASS # Signal success.

fail: RVTEST_FAIL
RVTEST_CODE_END     # End of test code.
```

---

```c
# Output data section.
# This section is optional,
# and this data is NOT saved in the output.
.data .align 3

testdata:
.dword 41
RVTEST_DATA_BEGIN   # Start of test output...

result: .dword -1

RVTEST_DATA_END     # End of test output...
```

---

etc…
Problems with handwritten tests

- Test coverage is often very low
  - Combinatorics is not on our side
  - For 32-bit instructions: $2^{12}$ possibilities, though not all are valid instructions
  - Need sequences of instructions to probe internal pipeline state and forwarding paths
- The current RISC-V test suite is woeful
  - e.g. imitates are formed from various bits of the instruction and many implementation errors are not found
- RISC-V compliance group is looking to improve matters

Semi-automatic instruction test generation

- Aim to create large test sets with better coverage
- Templating
  - Generate sequences of instruction classes with knowledge of possible pipeline bugs
- Directed random generation
  - Constrained random instruction testing (e.g. constrain to class of instruction or even simply valid instructions)
  - Random distribution, e.g. on registers used
    - May want to work on a small number of registers to reduce combinatorics
    - But throw in a smaller proportion of other registers
- Example: RISC-V torture tests: https://github.com/ucb-bar/riscv-torture
- Sometimes SAT solvers are used

Tandem verification

- Compare the processor design under test (DUT) with a model by checking committed instructions
- Execute test or real code (e.g. OS boot and application launch)
- For RISC-V there is a standard instruction trace interface
  - riscv-formal interface (RVFI)
- Advantages:
  - Great for debugging issues with large code bases
- Disadvantages:
  - Test coverage is only as good as code run
    - e.g. OS boot writes (initialises) many data structures but reads and checks little; also uses a subset of instructions, e.g. little or no floating-point
Formal verification

- Gold standard
  - Machine checked proof of all properties
  - e.g. mathematically proving equivalence between an implementation and a model
  - Very expensive
  - Only as good as the golden model
- More practical uses:
  - Check a few key properties that are hard to test
  - e.g. floating-point arithmetic
  - Still very challenging
- Verilog model checking tools
  - ISA model and processor implementation can both be written in Verilog
  - Verilog model checking tools can then be used to check equivalence, e.g. for short sequences of instructions

Undefined behaviour

- How should we handle undefined/unimplemented instructions?
  - NOP?
  - Undefined behaviour? (e.g. for old 6502 used by the NES)
  - Raise “undefined instruction exception”? – preferred option these days
- Design choice
  - Integer division by zero
    - Raises an exception on x86 and ARM; is silently ignored on MIPS and PowerPC
    - Is defined to be “undefined behaviour” for C
  - Signed integer overflow
    - Wraps on x86; raises an exception on MIPS
  - n-bit left shift on n-bit values
    - x86: no shift, PowerPC: result is zero
- Ref: “Undefined behavior: What happened to my code?”
  
Constrained unpredictable behaviour

- Unused/reserved bits on control/status/configuration registers
  - Leave “undefined”, so ignored by processor implementations that don’t use the bits?
  - Or define to be “zero” for this ISA version and checked by current processors that the bits are zero?

- ARM v8 – unaligned loads/stores (from ARM Arch Ref Manual)
  K1.1.6 Loads and Stores to unaligned locations
  Some unaligned loads and stores in the Armv7 architecture are described as UNPREDICTABLE. These are defined in the Armv8-A architecture to do one of the following:
  - Take an alignment fault.
  - Perform the specified load or store to the unaligned memory location.

- ARM v8 – unaligned branches – see next slide
### K1.1.5 Branching to an unaligned PC

In A32 state, when branching to an address that is not word aligned and is defined to be **CONSTRAINED UNPREDICTABLE**, one of the following behaviors must occur:

- The unaligned location is forced to be aligned.
- The unaligned address generates an exception on the first instruction using the unaligned PC value.
  - If that instruction is executed at EL0 and either of the following applies, the exception is taken to EL2:
    - EL2 is using AArch32 and the value of HCR.TGE is 1.
    - EL2 is using AArch64 and the value of HCR.TGE is 1.
  - If the instruction is executed at EL0 when the applicable TGE bit is 0 the exception is taken to EL1.
- If the instruction is executed at an exception level that is higher than EL0 the exception is taken to the
  Exception level at which the instruction was executed.
- In all cases, the exception is generated only if the first instruction using the unaligned PC value is
  architecturally executed.

If the exception that results from a branch to an unaligned PC value:

- Is taken to an Exception level that is using AArch64, it is reported as a PC alignment fault exception, see ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault on page D13-2835.
- Is taken to an Exception level that is using AArch32, it is reported as a Prefetch Abort exception reporting a PC alignment fault exception on page G1-5544.

**Note**

Because bit[0] is used for interworking, it is impossible to specify a branch to A32 state when the bottom bit of the target address is 1. Therefore the bottom bit of IFAR, IFAR, or FAR, ELx is 0 for all these cases.

---

### Further reading

- **ISA specification & verification:**
  - **Mandatory:** “Who Guards the Guards? Formal validation of the Arm ‘v8-m architecture specification”, OOPS LA 2017
    https://dl.acm.org/citation.cfm?id=3152784.3133912
  - “ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS”, POPL 2019
  - Sail RISC-V docs: https://github.com/remsi-project/sail-riscv/tree/master/doc

- **Instruction test generation:**
  - **Mandatory:** “Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification”, IBM Research, IEEE Design and Test 2004
    http://dx.doi.org/10.1109/MDT.2004.1277900
  - RISC-V torture tests: https://github.com/ucb-bar/riscv-torture

- **Additional material:**
  - RISC-V tests: https://github.com/riscv/riscv-tests
  - RISC-V formal framework: https://github.com/SymbioticEDA/riscv-formal

---

Copyright © Simon W. Moore, 2020