Advanced Topics in Computer Architecture

Testing Processors

Prof. Simon W. Moore
(Presented by Dr. Jonathan Woodruff)

UNIVERSITY OF CAMBRIDGE

Computer Science & Technology

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

Architecture References

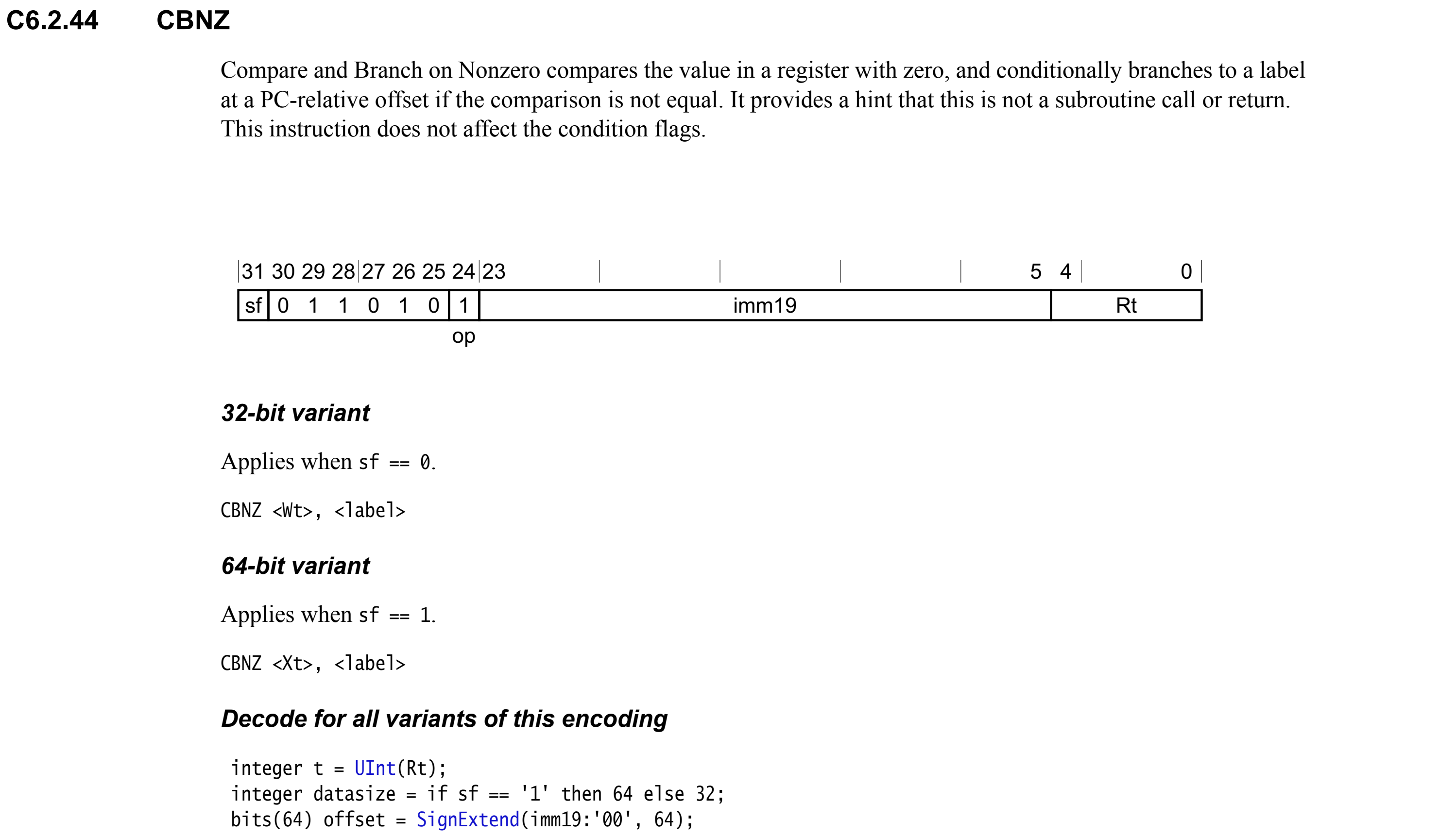
UNIVERSITY OF CAMBRIDGE

Computer Science & Technology

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/
    - 236 pages unprivileged spec, 91 pages privileged spec, 94 pages external debug spec
  - ARM: https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile
    - 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.

31	30 29 28 27 26 25	24	23 5	4 0
sf	0 1 1 0 1 0	1	imm19	Rt
		op		

32-bit variant

Applies when `sf == 0`.

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

64-bit variant

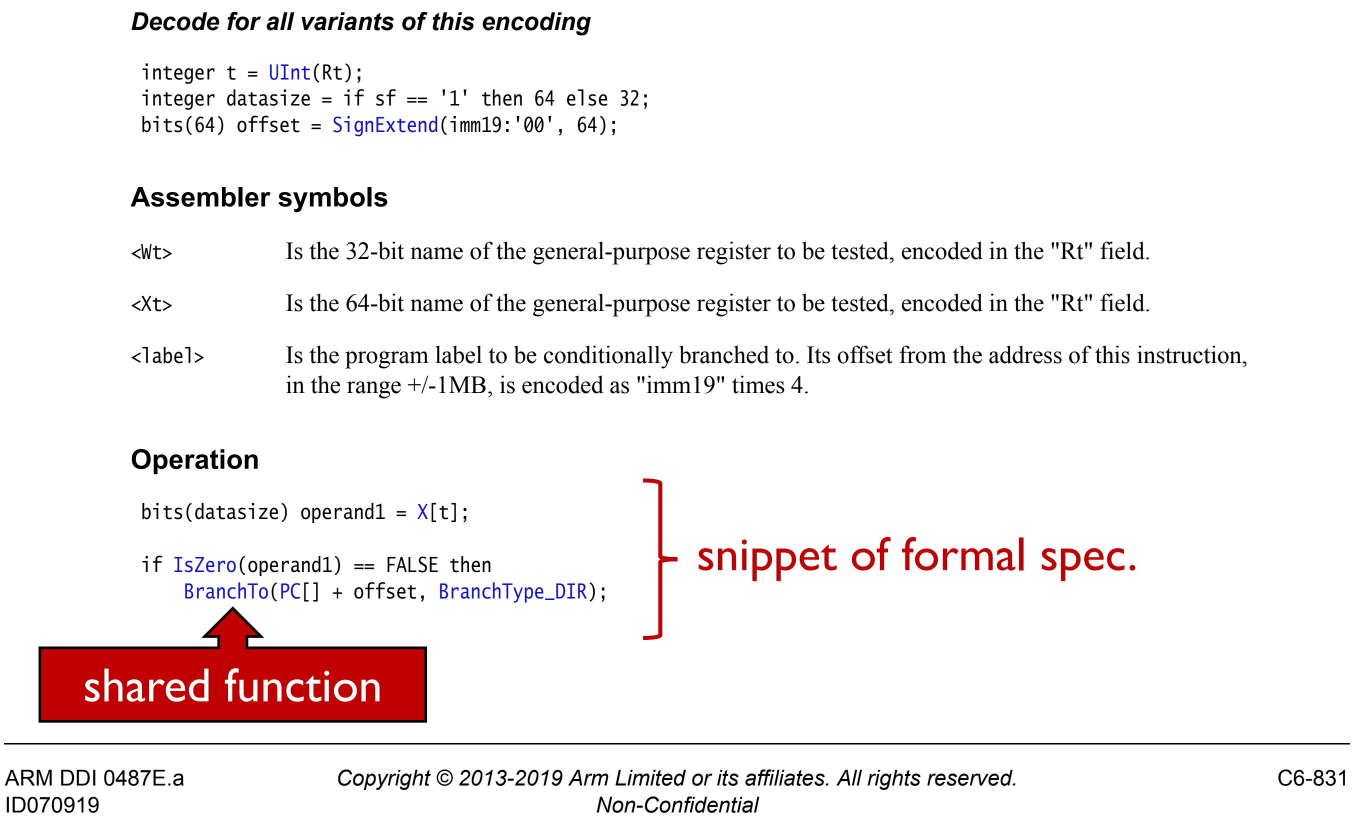
Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
bits(64) offset = SignExtend(imm19:'00', 64);
```

Example from the ARM arch. ref. manual



Decode for all variants of this encoding

```
integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.

`<Xt>` Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.

`<label>` Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(datasize) operand1 = X[t];

if IsZero(operand1) == FALSE then
    BranchTo(PC[] + offset, BranchType_DIR);
```

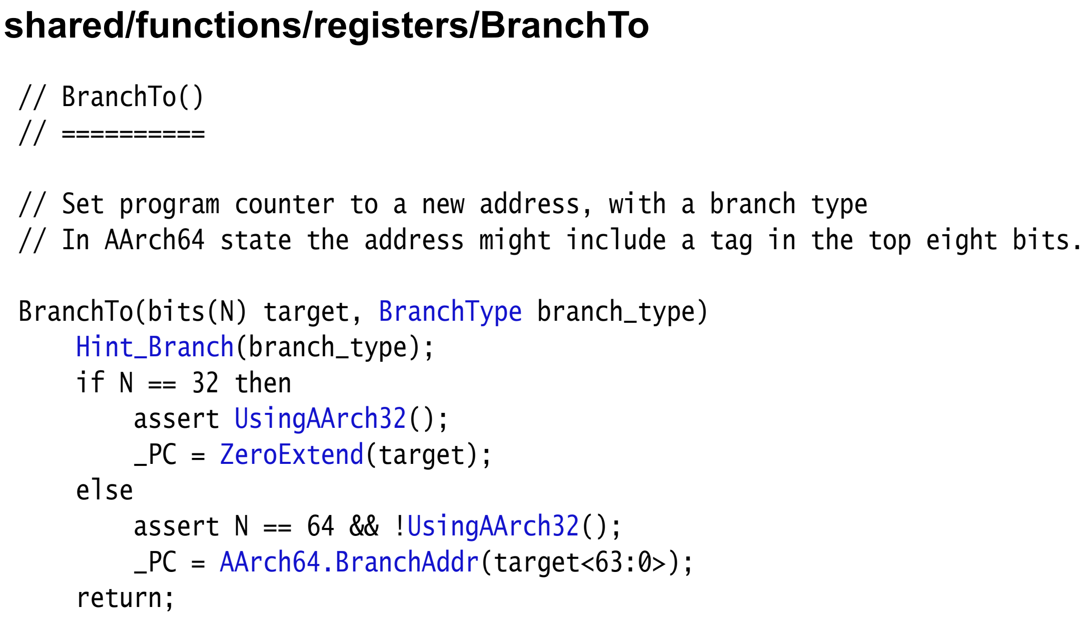
snippet of formal spec.

shared function

ARM DDI 0487E.a
ID070919

C6-831

ARM arch. ref.: shared subfunction example



shared/functions/registers/BranchTo

```
// BranchTo()
// ==========

// Set program counter to a new address, with a branch type
// In AArch64 state the address might include a tag in the top eight bits.

BranchTo(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        _PC = AArch64.BranchAddr(target<63:0>);
    return;
```

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

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

```
mapping encdec_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

matching BTYPE machine code

```
mapping clause encdec = BTYPE(imm7_6 @ imm5_0 @ imm7_5_0 @ imm5_4_1 @ 0b0, rs2, rs1, op)
  <-> imm7_6 : bits(1) @ imm7_5_0 : bits(6) @ rs2 @ rs1 @ encdec_bop(op) @ imm5_4_1 :
bits(4) @ imm5_0 : bits(1) @ 0b1100011
```

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

```
function clause execute (BTYPE(imm, rs2, rs1, op)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let taken : bool = match op {
    RISCV_BEQ  => rs1_val == rs2_val,
    RISCV_BNE  => rs1_val != rs2_val,
    RISCV_BLT  => rs1_val <_s rs2_val,
    RISCV_BGE  => rs1_val >=_s rs2_val,
    RISCV_BLTU => rs1_val <_u rs2_val,
    RISCV_BGEU => rs1_val >=_u rs2_val
  };
  let t : xlenbits = PC + EXTS(imm);
  if taken then {
    /* Extensions get the first checks on the prospective
       target address. */
    match ext_control_check_pc(t) {
      Ext_ControlAddr_Error(e) => {
        ext_handle_control_check_error(e);
        RETIRE_FAIL
      },
      Ext_ControlAddr_OK(target) => {
        if bit_to_bool(target[1]) & (~ (haveRVC())) then {
          handle_mem_exception(target, E_Fetch_Addr_Align());
          RETIRE_FAIL;
        } else {
          set_next_pc(target);
          RETIRE_SUCCESS
        }
      }
    }
  }
  else RETIRE_SUCCESS
}
```

Functional Testing Processors

UNIVERSITY OF CAMBRIDGE

Computer Science & Technology

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)

```
#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         # Should be 42 into $2
        sw      x2, result    # Store result into…
                             # …memory overwriting 1s
        li      x3, 42        # Desired result
        bne     x2, x3, fail  # Fail out if doesn't…
                              # …match
        RVTEST_PASS           # Signal success.
fail:
        RVTEST_FAIL
RVTEST_CODE_END     # End of test code.
```

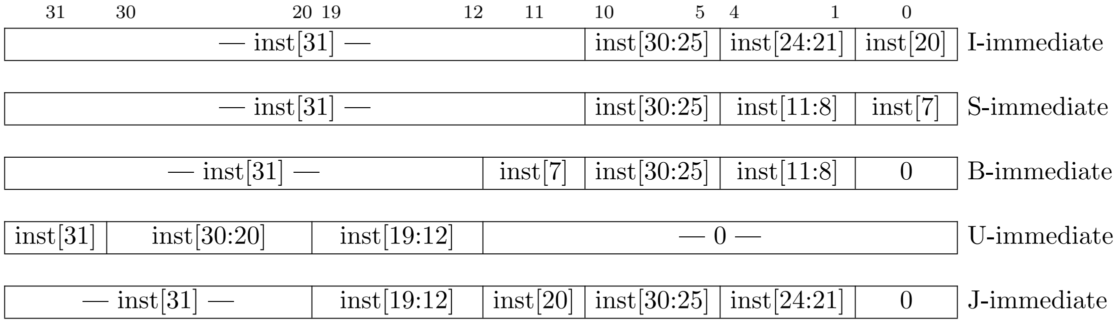
```
# Input data section.
# This section is optional,
# and this data is NOT saved in the output.
.data
        .align 3
testdata:
        .dword 41

# Output data section.
RVTEST_DATA_BEGIN   # Start of test output…
                    # …data region
        .align 3
result:
        .dword -1
RVTEST_DATA_END     # End of test output…
                    # …data region.
```

etc…

Problems with handwritten tests

- Test coverage is often very low
  - Combinatorics is not on our side
  - For 32-bit instructions: 2^{32} 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. immediates are formed from various bits of the instruction and many implementation errors are not found
  - RISC-V compliance group is looking to improve matters



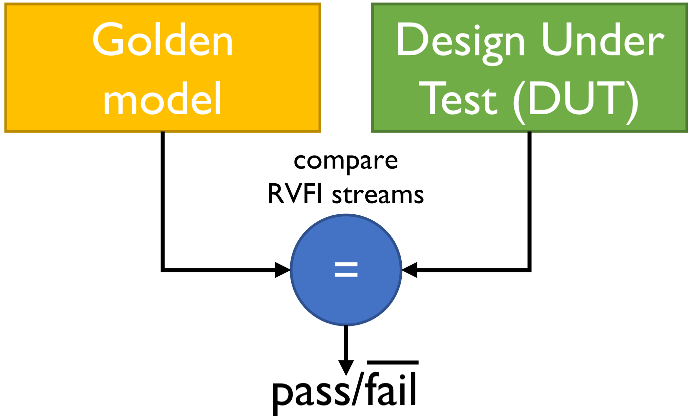
31	30 20	19 12	11	10 5	4 1	0	
— inst[31] —				inst[30:25]	inst[24:21]	inst[20]	I-immediate
— inst[31] —				inst[30:25]	inst[11:8]	inst[7]	S-immediate
— inst[31] —			inst[7]	inst[30:25]	inst[11:8]	0	B-immediate
inst[31]	inst[30:20]	inst[19:12]	— 0 —				U-immediate
— inst[31] —		inst[19:12]	inst[20]	inst[30:25]	inst[24:21]	0	J-immediate

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

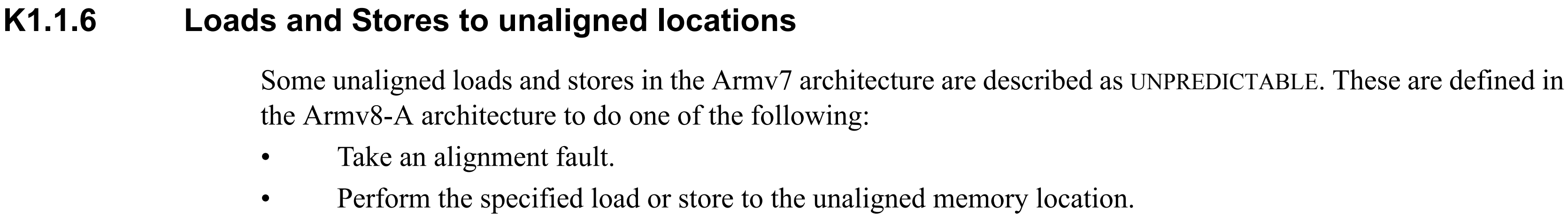
Undefined behaviour

- 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?" https://dl.acm.org/citation.cfm?id=2349905

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)

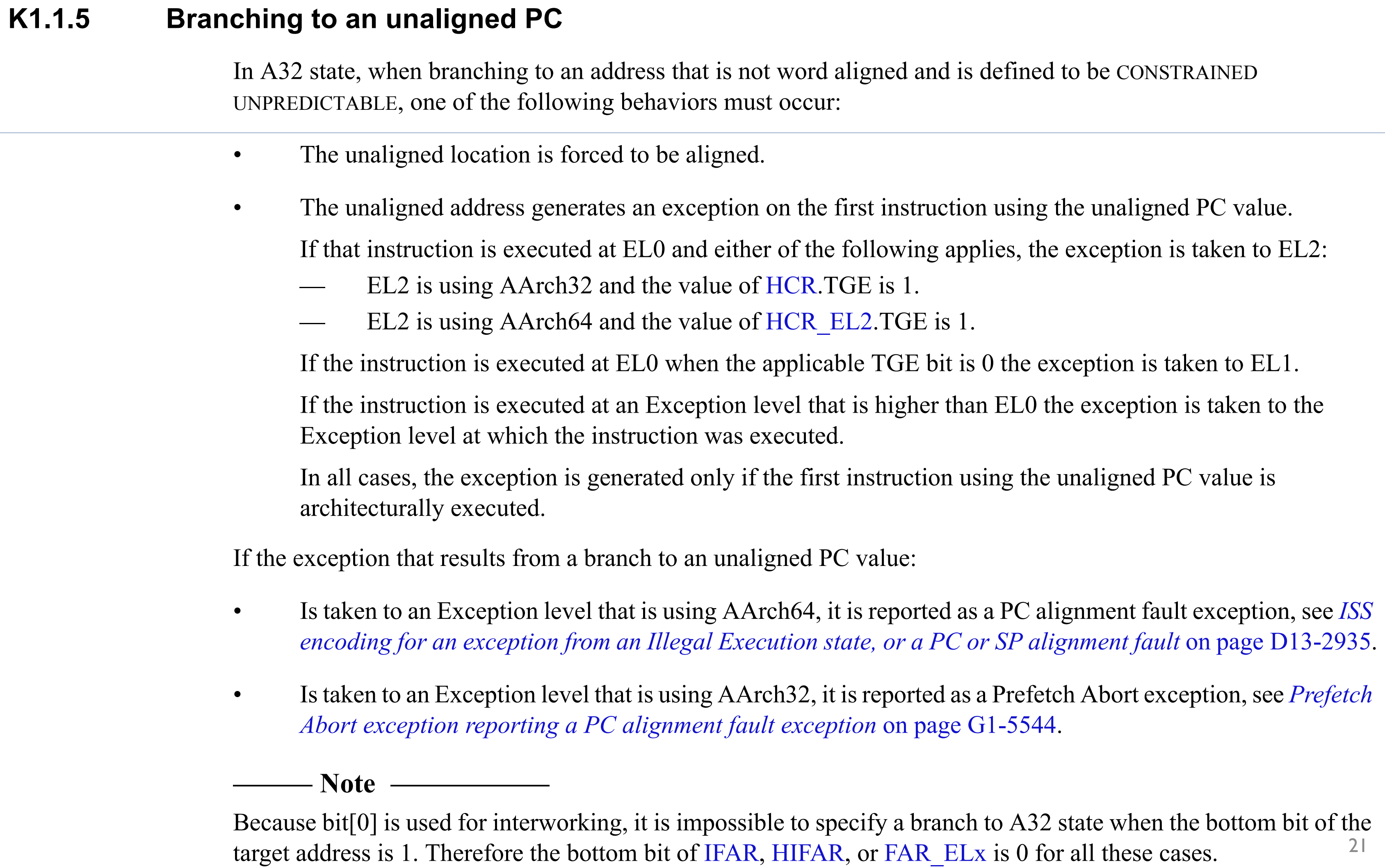


K1.1.6 Loads and Stores to unaligned locations

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_EL2.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 D13-2935.
- Is taken to an Exception level that is using AArch32, it is reported as a Prefetch Abort exception, see *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, HIFAR, or FAR_ELx is 0 for all these cases.

From the ARM v8 Architecture Reference Manual

Further reading

- ISA specification & verification:
  - “Who Guards the Guards? Formal validation of the Arm v8-m architecture specification”, OOPSLA 2017
    https://dl.acm.org/citation.cfm?id=3152284.3133912
- Instruction test generation:
  - “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
- Formal Verification:
  - ”End-to-end verification of processors with ISA-Formal”, ARM Ltd, CAV 2016
    https://alastairreid.github.io/papers/cav2016_isa_formal.pdf
- Additional material:
  - “ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS”, POPL 2019
    https://www.cl.cam.ac.uk/~pes20/sail/sail-popl2019.pdf
  - Sail RISC-V docs: https://github.com/rems-project/sail-riscv/tree/master/doc
  - “Randomised testing of a microprocessor model using SMT-solver state generation”, 2015.
    http://dx.doi.org/10.1016/j.scico.2015.10.012
  - RISC-V torture tests: https://github.com/ucb-bar/riscv-torture
  - RISC-V tests: https://github.com/riscv/riscv-tests
  - RISC-V formal framework: https://github.com/SymbioticEDA/riscv-formal
    http://www.clifford.at/papers/2017/riscv-formal/slides.pdf