

NexRv Tool (Nexus trace for RISC-V) Description

This file describes NexRv source code and is provided with the following license:

```
/*
 * Copyright (c) 2020 IAR Systems AB.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose with or without fee is hereby granted, provided that the above
 * copyright notice and this permission notice appear in all copies.
 *
 * THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
 * WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
 * ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
 * ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 * OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 */
```

History:

Version, Date	Who	Description
1.00, 2020/05/17	Robert.Chyla@iar.com	Initial version. In PDF format as I have no idea how to make nice diagrams in ADOC format (certainly there is a way ...).
1.10 2020/06/02	Robert.Chyla@iar.com	Making it complete (still published as PDF file)
1.?? ????/??/??	??	TODO: Make nice looking ADOC from it.

Goals and Rationale

This **NexRv** package (short for **Nexus** Trace for **RISC-V**) is open source tools allowing exploration and good understanding of Nexus Trace by RISC-V community. It should be considered as by-product of RISC-V Nexus Trace TG work.

More specifically it is addressing the following topics:

1. Augments definition of Nexus messages (applicable to RISC-V) being elaborated by Nexus Trace TG.
 - Clarification of corner cases as (complex) description in Nexus Standard PDF may not be so obvious – working reference implementation is often best way to clarify all details.
2. Provides quick assessment of RISC-V Nexus trace compression ratio for particular program[s].
 - Logic designers may assess necessary trace bandwidth and related logic complexity quickly.
3. May allow (later) options to customize certain (vendor defined) aspects of Nexus trace encoding (and decoding).
 - This way decision if certain (more complex) options are needed to be implemented can be made early.
 - In certain situation (especially for small silicon) size of trace logic is very important and having tool helping to see difference between simple/complex options is essential.
4. It may be used to create certain corner cases (as small text files) and assess/check intended behavior.
 - Sometimes it is not easy to have ‘true-program’ to see how weird/rare sequence are handled.
 - Good example is handling of exceptions around boundary-crossing of different types of messages.
 - Such set of files may be later changed into Nexus Trace encoder validation suite.
 - As such files are just PC sequences, same test-set may be used to validate Program Trace implementation and protocol as well.
5. Reference encoder in plain-C can be used as guidance to generate Nexus messages from logic.
 - **NexRv** tool allows simpler integration/validation steps without a need to implement encoder in logic.
 - NOTE: Simple (I hope) reference implementation of Nexus encoder in logic language will be later provided.
6. It may (and should) be used as test-bench to compare RISC-V Nexus encoders implemented in logic.
 - Nexus provides different options, but for given program (and given set of options) messages generated should be more or less identical no matter how created.
7. **NaxRv** may be run to see if logic provides good compression by the following simple steps:
 - Run program, create Nexus messages (by encoder implemented in logic) and provide NEXBIN file.
 - Decode that file (using **NexRv -deco ...**) into PC sequence and validate this is correct PC flow.
 - Encode that PC sequence (using **NexRv -enco ...**) into another NEXBIN file.
 - Decode it again (just to display statistics) and compare with compression for original NEXBIN file.
 - Experiment with different **NexRv** encoding options to see which provides best compression and if found good, retro-fit these options back to your ‘Nexus encoder in logic’ implementation and re-iterate.

Overview

1. Reference C code is split into several source files:

makefile	- Allows to compile it by calling 'make'
NexRv.c	- Main executable (option handling, opening files and calling functions)
NexRv.h	- Header with Nexus-related constant definitions (using #define)
NexRvMsg.h	- Array defining RISC-V specific Nexus message types and fields
NexRvDump.c	- Function to dump messages in binary Nexus file (called with -dump option)
NexRvDeco.c	- Function to decode binary Nexus file (called with -deco option)
NexRvEnco.c	- Function to encode PC sequence into Nexus binary (called with -enco option)
NexRvConv.c	- Functions for different conversions (called with -conv option)
NexRvInfo.c	- Common code to handle PCINFO file (used by -conv and -deco)

Examples are provided in *./examples* sub-directory.

2. Code above is written using simple (K&R-style ...) C language constructs, so can be well understood

- It was written having code simplicity and not efficiency in mind.
- 'Advanced' C-feature' are 'for (int x = ...)' and 'int x;' between instructions in few places (these may be converted to pure K&R C if will ever be needed) - these are used just to increase code readability.
- No memory allocations and only few C-library functions are called.
- All files are text (except binary Nexus file which is sequence of raw bytes) for easier understanding.
- It is not intended to process large files, but rather to illustrate key concepts and inspect corner cases.
- Files are processed sequentially except PCINFO file - if this will became bottleneck NexRvInfo.c needs to be improved by reading file once to an array and using it (NexRvInfo.c was extracted to have this in mind).

3. Compilation should be done by running 'make' having 'gcc' compiler available

- It was compiled using the following GCC versions:

gcc (i686-posix-dwarf-rev0, Built by MinGW-W64 project) 8.1.0

gcc (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609 – I had VM with it handy ...

- Code was compiled using Visual Studio compiler as well.
- It would be possible to simplify main code and only compile appropriate sub-module.

4. Complete flow is elaborated in following chapters, but here is simplified usage (as an overview):

NexRv -dump <nexbin>	➔Dump Nexus messages from <nexbin> Nexus file
NexRv -deco <nexbin> ...	➔Decode <nexbin> Nexus file
NexRv -enco -pcseq <pcs> ...	➔Encode PC sequence into Nexus file
NexRv -conv -objd <objd> ...	➔Convert objdump output into INFO-file
NexRv -conv -pconly <pc> ...	➔Convert PCONLY file (plain PC sequence) into

5. Format of all text files is described in dedicated chapter below.

6. Encoder is by default producing Nexus trace with Indirect Branch History and Repeat Branch messages.

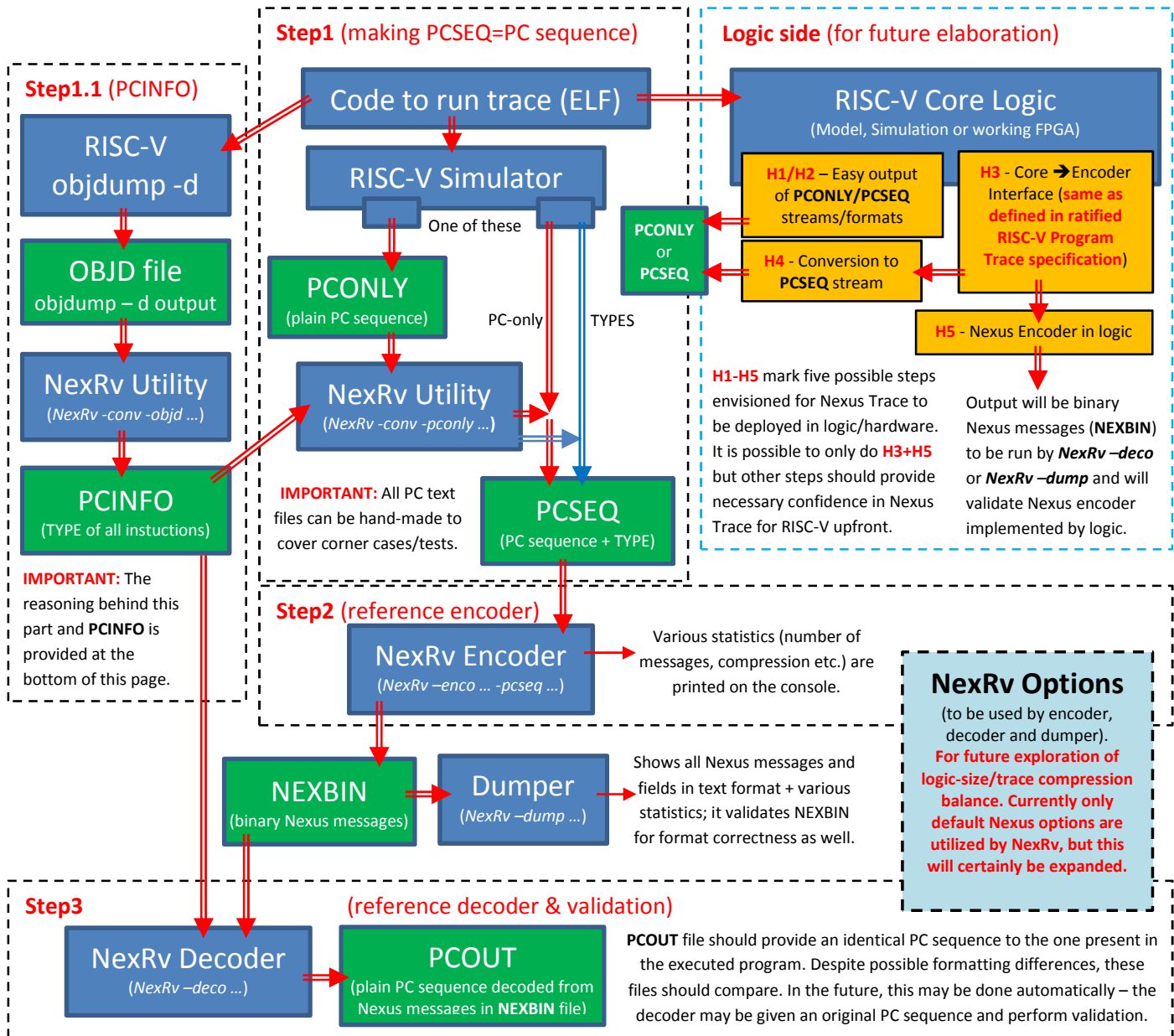
- Encoder can be customized to not generate Repeat Branch and Branch History, but decoder is handling all types of messages.
- More detailed customization of encoder/decoder will be provided later.

7. Limitations:

- Current implementation is limited to 32-bit PC but as long as someone will run on 64-system in lower 32-bit address range all will work as Nexus is skipping MSB=0. The only difference (in case code runs outside of 32-bit range) would be just longer addresses.
- All sizes/values of Nexus fields in messages are kept as 'recommended' as defined by Nexus Standard.
- Multi-core and timestamps are not considered (these are extra/optional fields in otherwise same messages).

NexRv Processing Flow Diagram

A picture is worth a 1000 words, so below is a complete diagram showing 'code PC flow → core → encoder → decoder → decoded PC' flow options. It explains the **NexRv** tool flow and also touches upon logic side components.



Legend:

- Data (all files but NEXBIN are text).
- Component/tool available now (including NexRv GitHub).
- NexRv part to be elaborated later.
- Envisioned logic side (H1-H5 are elaborated below).

Rationale behind Step 1.1. (PCINFO). Production-level decoders read code from the ELF file and must understand all ISA-encoding variants. For example, the encoded trace only says that jump was executed but the destination address must be found by the decoder – same with instruction sizes. Instead of decoding all of the ISA (and reading the ELF file ...), the code in ELF is dumped by 'riscv-...-objdump -d ...' to a text file. After that addresses are calculated and different types of instructions are clearly visible as plain text. This file is processed by 'NexRv -conv -objd ...' call. The list of PCs in ELF is produced together with an associated instruction type and other extra details. Here is an example output – instruction address is followed by type + size + 'extra' (destination address for direct jumps and branches):

- 0x100, L2 - Linear (plain) instruction, 2 byte (16-bit) – the decoder is not interested in any other details (besides size).
- 0x102, B4, 0x210 - Branch, 4-byte (32-bit), destination address is 0x210 – it would be cumbersome to dig this from opcodes ☹.
- 0x106, R2 - Return instruction, 2-byte (16-bit) – it could be JI2 (Jump, Indirect, 2-byte), but in sophisticated compression options (part of Nexus standard), it is necessary for the decoder (and encoder) to distinguish returns and calls.

This level of information is all that is needed for the decoder to decode PC flow. PCSEQ has the same details but for every instruction executed (branch taken or not). More details about formats are given on the following pages; this roughly explains key PCINFO concept.

NexRv Processing Flow (Details and Example)

Directory `./examples/t1` contains first-test 't1' of NexRv calling. This is done by starting from running example program **test.elf** by RISC-V simulator. PCs for all instructions executed by this program are saved into **test-PCONLY.txt** file as one C-style hex in each line. Format "0x%08X\n" i.e. 0x0000ACD0 (and not 0xabc0) is recommended as it may be later directly compared by 'diff' with decoder output (**test-PCOUT.txt**) file.

Naming convention '**name-TTT**' is used to denote type of a file. However all files are given as full name so this is not mandatory. Type '**TTT**' is always given in upper-case. It is consistent with names used in NexRv flow diagram and easier to spot.

Input files:

test.elf	- Binary ELF file with program which was run.
test-PCONLY.txt	- Text file with all instruction run in this program (PCONLY file).

Steps needed to exercise full-flow (conversions, encode, dump, decode and validate):

1. Run RISC-V objdump utility from GNU package and create **OBJD** file (via redirection):

```
riscv64-unknown-elf-objdump -d test.elf >test-OBJD.txt
```

2. Convert above **OBJD** file to **PCINFO** file:

```
NexRv -conv -objd test-OBJD.txt -pcinfo test-PCINFO.txt
```

3. Create **PCSEQ** file from **PCONLY** file (using **PCINFO** file):

```
NexRv -conv -pcinfo test-PCINFO.txt -pconly test-PCONLY.txt -pcseq test-PCSEQ.txt
```

4. Encode **PCSEQ** file into **NEXBIN** file (binary with Nexus messages as sequence of bytes):

```
NexRv.exe -enco test-PCSEQ.txt -nex test-NEX.bin
```

IMPORTANT: Encoding is only using **PCSEQ** (logic-side encoder will only know 'PC+type' as input from traced core).

5. Dump created **NEXBIN** file (optional step, but allows to understand and validate Nexus messages):

```
NexRv -dump test-NEX.bin test-DUMP.txt
```

6. Decode trace from **NEXBIN** file to **PCOUT** text file:

```
NexRv -deco test-NEX.bin -pcinfo test-PCINFO.txt -pcout test-PCOUT.txt
```

7. Verify that **PCONLY** (input) and **PCOUT** (output) files has same PC sequence:

```
diff -s -q test-PCONLY.txt test-PCOUT.txt
```

NOTE: Hex numbers as text in 0x format may not compare byte-to-byte. In future **NexRv** may be extended to allow say '**NexRv -comp test-PCONLY.txt test-PCOUT.txt**'. Used **PCONLY** file has format same as **PCOUT** file which is created by `fprintf(f, "0x%08X\n", pc)` in C code.

Example files **test-OBJD.txt** and **test-PCONLY.txt** are provided on GitHub and all above can be quickly run as follows:

```
make          - Compile NexRv tool
cd examples/t1
make          - Run all commands from steps 2. to 7. Above (it runs encoder/decoder 3 times)
```

NOTE: Running '**make**' does not run any RISC-V tools. These may have different name and may not be in the path on your system. However '**make ELF**' and '**make OBJD**' can be used to re-create **test.elf** and **test-OBJD.txt** files.

Format of Files Explained

This chapter describes format of files using **test*.txt** files from **./examples/t1[/output]** directories.

OBJD file – created by running RISC-V **riscv64-unknown-elf-objdump** from ELF file

```
test.elf:      file format elf32-littleriscv

-- skipped startup (trace was stated starts from 'main')

Disassembly of section .text:
-- skipped
20010510:      4785          li      a5,1          ← 16-bit linear instruction
20010512:      fef718e3     bne     a4,a5,20010502 <myExit+0x16> ← 32-bit branch
20010516:      fec42783     lw      a5,-20(s0)    ← 32-bit linear instruction
2001051a:      853e          mv      a0,a5
2001051c:      4472          lw      s0,28(sp)
2001051e:      6105          addi    sp,sp,32
20010520:      8082          ret                     ← 16-bit return

20010522 <main>:
20010522:      1141          addi    sp,sp,-16
20010524:      c606          sw      ra,12(sp)
20010526:      c422          sw      s0,8(sp)
20010528:      0800          addi    s0,sp,16
2001052a:      800107b7     lui     a5,0x80010
-- skipped
20010552:      391d          jal     20010188 <xrle_compress> ← 16-bit call (jump and link)
20010554:      872a          mv      a4,a0
-- skipped
```

INFO file (address,type<,dest>) – it is created by scanning above OBJD file and parsing some specific **instruction names**. Binary opcodes (**4-byte** or **2-byte**) are used to determine size of instructions. Direct jumps and branches have **addresses**.

```
-- skipped
0x20010510, L2          ← 2-byte (16-bit) linear instruction
0x20010512, BD4, 0x20010502 ← 4-byte (32-bit) branch to address 0x20010502
0x20010516, L4          ← 4-byte (32-bit) linear instruction
0x2001051A, L2
0x2001051C, L2
0x2001051E, L2
0x20010520, R2          ← 2-byte (16-bit) return
0x20010522, L2
0x20010524, L2
0x20010526, L2
0x20010528, L2
0x2001052A, L4
-- skipped
0x20010552, BD2, 0x20010188 ← 4-byte (32-bit) call to address 0x20010188
0x20010554, L2
```

NOTE: Colors are used to highlight corresponding items.

The following types are allowed (each followed by 2 or 4 of instruction size expressed in bytes):

- J[I|D] - Jump Indirect or Jump Direct
- BD - Branch (it is always Direct on RISC-V)
- C[I|D] - Call Indirect or Call Direct
- R - Return (it is always Indirect)
- L - Linear (none of the above)

For direct branch of jump addresses destination address (not an offset!) is provided.

PCONLY file (just sequence of PC from running a program as hex numbers)

```
0x20010522    ← This is 'main' (simulator output was started at 'main')
0x20010524
0x20010526
-- skipped    ← Bunch of 16-bit and 32-bit instructions
0x2001054A
0x2001054E
0x20010552    ← This is 'jal' instruction (visible in PCINFO file above)
0x20010188    ← This is destination address of that 'jal' (also visible in PCINFO above)
0x2001018A
-- skipped
```

PCSEQ file (this is PCONLY file with details from PCINFO file added for each instruction). This is sub-set of PCINFO file format (type + size, but NOT destination addresses):

```
0x20010522,L2    ← Linear 2-byte (16-bit) instruction at start
0x20010524,L2
0x20010526,L2
-- skipped    ← Bunch of 16-bit and 32-bit instructions
0x2001054A,L4
0x2001054E,L4
0x20010552,C2    ← Here we have 'C2', what means this is 2-byte (16-bit call). 'C2' is same as 'CD2'.
0x20010188,L2
0x2001018A,L2
```

As it can be see above, PCSEQ file has same records as PCONLY file, but each line is augmented with type and size of particular instruction - destination address (for direct branches) is not provided as encoder will not know it – it also mimics condition when trace encoder ingress port only provides 32-bit address + type.

Non taken branches are marked as '**BN**' – this will allow encoder to treat taken/non-taken branches differently.

DUMP file (dump of binary Nexus trace file as plain text file for human-eye analysis)

Each packet is displayed as byte (first column) – it is split into 6+2 bits (MDO+MSEO) and then fields are encoded. MSEO bits are encoded **01** (end of variable size field) and **11** (end of message – which is end of field as well).

```
0x24 001001_00: TCODE[6]=9 (MSG #0) - ProgramTraceSynchronization    ← TCODE is always 6-bit
0x05 000001_01: SYNC[4]=0x1 ICNT[2]=0x0    ← Fixed 4-bit SYNC field followed by 2-bit ICNT field
0x44 010001_00:    ← First bits of FADDR (Full Address) field
0x28 001010_00:
0x20 001000_00:
0x00 000000_00:
0x43 010000_11: FADDR[30]=0x10008291    ← End of FADDR field (5*6=30 bits) - end of message.
```

Above packet is encoding full PC address. As RISC-V cannot execute code from odd addresses, LSB bit is skipped. Value 0x1000_8291 shifter 1 bit to the left is 0x2001_0522, what is starting address of trace capture. **SYNC** field provides a reason (which is 'leave debug mode').

```
0x70 011100_00: TCODE[6]=28 (MSG #1) - IndirectBranchHistory    ← TCODE is always 6-bit
0x80 100000_00: BTYPE[2]=0x0    ← Fixed 2-bit BTYPE field followed by beginning of ICNT field
0x95 100101_01: ICNT[10]=0x258    ← More bits for ICNT field (4+6=10 bits total)
0x3C 001111_00:    ← First 6-bits of UADDR field
0x3D 001111_01: UADDR[12]=0x3CF    ← More UADDR field bits (6+6=12 bits total)
0x00 000000_00:    ← LSB part of HIST field
0x00 000000_00:
0xA0 101000_00:
0x50 010100_00:
0x54 010101_00:
0x0F 000011_11: HIST[36]=0xD5528000    ← Last part of HIST field (6*6=36 bits total) - end of message.
```

This packet is more complex – it has 4 fields. **BTYPE** is branch type. **ICNT** encodes number of 16-bit instructions executed (0x258=600). **UADDR** field encodes address update value. **HIST** field shows branch history (bit sequence encoded from MSB side). This 11-byte packet encodes 600 instructions. It is then $(11*8)/600 = \sim 0.147$ bits/instruction (it is very efficient).

Basic Nexus Encoding Principle

This chapter is using files from `./example/t1` directory (used above) to illustrate Nexus encoding principle. In general encoder is looking at PC discontinuity points as linear code is not so interesting ...

Let's look at `test-PCSEQ.txt` file with skipped linear instructions as example of simple trace:

```
0x20010522,L2 (*A)
-- skipped linear instructions between *A and *B
0x2001054E,L4
0x20010552,C2 (*B)
0x20010188,L2 (*C)
-- skipped linear instructions between *C and *D
0x200101D0,L2
0x200101D2,B4 (*D)
0x200101EA,L2 (*E)
-- skipped linear instructions following *E
```

Let's see how above PC sequence is encoded using **Direct Branch** messages (taken from `test-DUMP.txt` file).

1. Program starts at address `0x2001_0522` ***A** and **Program Trace Synchronization** message is emitted:

```
0x24 001001_00: TCODE[6]=9 (MSG #0) - ProgramTraceSynchronization
0x05 000001_01: SYNC[4]=0x1 ICNT[2]=0x0 ← ICNT=0 means, that no instruction we executed before
0x44 010001_00:
0x28 001010_00:
0x20 001000_00:
0x00 000000_00:
0x43 010000_11: FADDR[30]=0x10008291 ← Full address without LSB bit, so it corresponds to 0x2001_0522
```

2. Encoder starts counting all linear instructions till call (at ***B**). Since this is direct call (destination address ***C** is fixed), we do not emit anything and we keep counting (but we count call instruction as executed as well).
3. At point ***D** encoder encounter taken branch. In this case flow is changing. But by looking at code we cannot predict if this jump was taken or not. So encoder emits the following **Direct Branch** message:

```
0x0C 000011_00: TCODE[6]=3 (MSG #1) - DirectBranch
0x00 000000_00:
0x07 000001_11: ICNT[12]=0x40 ← Here we have 0x40 (=64) 16-bit instruction units executed
```

4. Let's see from where this `0x40` is taken – it involves some address calculations, but I encourage reader to follow it – these are key details to be understood when implementing own Nexus encoder in logic:

- First linear code from ***A** to ***B** is executed – it will count as $(0x2001_0552 - 0x2001_0522)/2 = 0x18$.
- Then 16-bit call instruction at ***B** is executed – it will count as `0x1` (as this is 16-bit instruction).
- Then linear code from ***C** to ***D** is executed– it will count as $(0x2001_01D2 - 0x2001_0188)/2 = 0x25$.
- Finally 32-bit branch at ***D** is executed, - it will count as `0x2` (as this is 32-bit instruction).
- Total number of instruction units will be than $0x18 + 0x1 + 0x25 + 0x2 = 0x40$ (what is emitted ICNT value).

Decoder starts from full address defined by '**Program Trace Synchronization**' packet (***A**=`0x2001_0522`) and follows all `0x40` 16-bit instruction units the same way as encoder did (changing address at call at ***B**). It will finally end-up at branch instruction at ***D**=`0x2001_01D2`. And as this is direct branch, it will continue at destination address of that branch (taken from ELF file, what is in our case converted into PCINFO file).

Example provided above is very simple and not showing all important cases. Take this into consideration:

1. Non taken branches are considered as 'linear instructions' – these just advance ICNT accordingly (+1 or +2).
2. Linear instructions between PC discontinuity points do not matter. Code which does 'something' (except jumping like crazy) will compress better. That's very reason to judge trace compression by running 'real-life' code. In our example 64 instructions were encoded in 3 byte message. This is $(3*8)/64 = 0.375$ bits/instruction.
3. In case of '**Branch History Messages**' encoder is NOT emitting any packets at branches. Instead it keeps branch-history buffer by adding single bit (0=not taken, 1=taken) at every branch. It provides very good compression.