Hardware Assisted Tracing on ARM with CoreSight and OpenCSD

Mathieu Poirier

Linaro

In this Presentation

- End-to-end overview of the technology
- Not an in-depth presentation on CoreSight
- Emphasis on how to use rather than what it is
- Mostly covers the integration with the standard Perf core
- Everything that is needed to get started

- As such
  - Brief introduction on CoreSight
  - Enabling CoreSight on a system
  - OpenCSD library for trace decoding
  - Trace acquisition scenarios
  - Trace decoding scenarios

What is CoreSight

- The name given to an umbrella technology
- Covers all the tracing needs of an SoC, with and without external tools

- Our work concentrate on HW assisted tracing and the decoding of those traces

- What is HW assisted tracing?
  - The ability to trace what is done by a CPU core **without** impact on its performance
  - No external HW need to be connected
  - The CPU core doesn't have to run Linux!

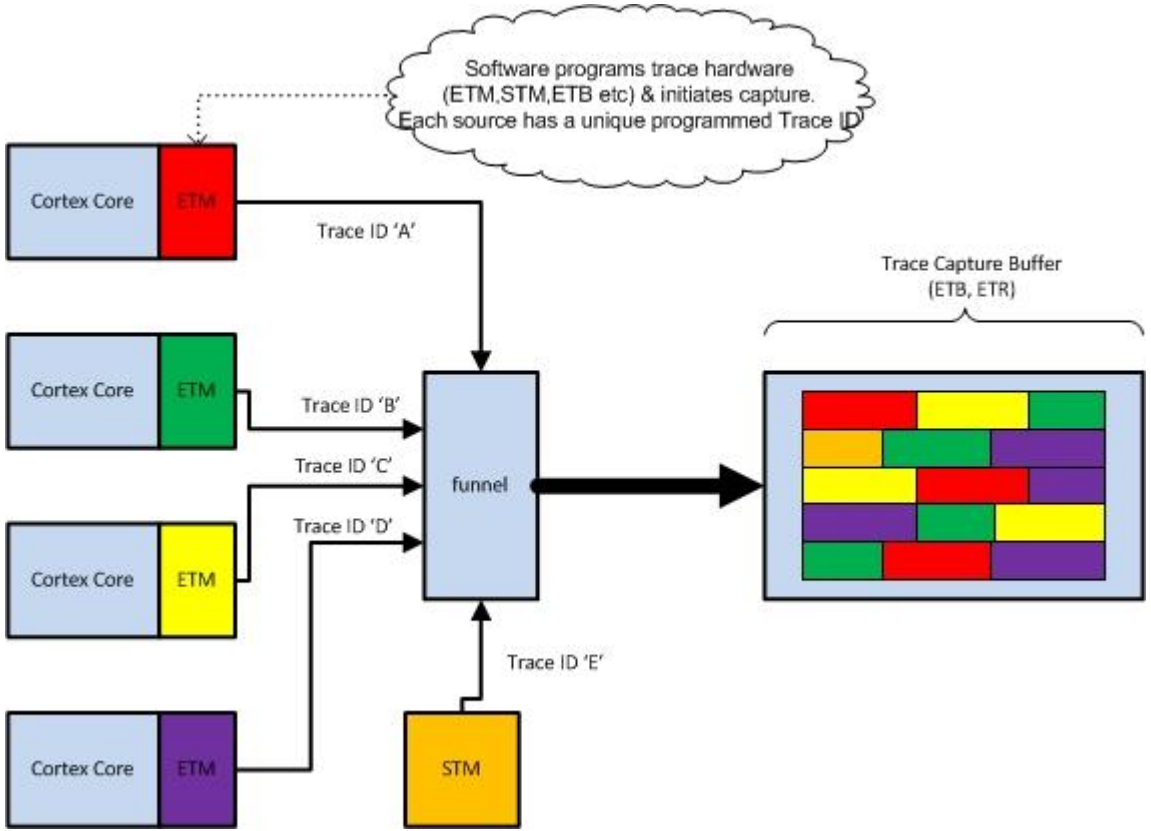
- The CoreSight drivers and framework can be found under

  `drivers/hwtracing/coresight/`

How Does HW Assisted Tracing Work?

- Each core in a system is fitted with a companion IP block called an **Embedded Trace Macrocell (ETM)**
- Typically one embedded trace macrocell per CPU core
- OS drivers program the trace macrocell with specific tracing characteristics
  - There are many examples on doing this in the coming slides

- Once triggered trace macrocells operate independently
- No involvement from the CPU core, hence no impact on performance
- ** Be mindful of the CoreSight topology and the memory bus **


Software programs trace hardware
(ETM,STM,ETB etc) & initiates capture.
Each source has a unique programmed Trace ID
Cortex Core
ETM
Trace ID 'A'
Cortex Core
ETM
Trace ID 'B'
Trace ID 'C'
Cortex Core
ETM
Trace ID 'D'
Cortex Core
ETM
funnel
Trace ID 'E'
STM
Trace Capture Buffer
(ETB, ETR)

Program Flow Trace

- Traces are generated by the HW in a format called **Program flow trace**
- **Program flow traces** are a series of **waypoint** taken by the processor
- **Waypoints** are:
  - Some branch instruction
  - Exceptions
  - Returns
  - Memory barriers

- Using the original program image and the **waypoints**, it is possible to reconstruct the path a processor took through the code.
- **Program flow traces** are decoded into **executed instruction ranges** using the OpenCSD library

CoreSight On A System

- All CoreSight components are supported upstream
- Except for CTI and ITM
  - CTI will be available soon
  - ITM is an older IP - relatively simple to support

- The reference platforms are Vexpress TC2 (ARMv7) and Juno (ARMv8)

- The CoreSight topology for any system is covered in the DT
- The topology is expressed using the generic V4L2 graph bindings
  - The reference platform DTs are upstream and cover pretty much all the cases
  - http://lxr.free-electrons.com/source/Documentation/devicetree/bindings/graph.txt

- With the correct DT additions, CoreSight should just work…

CoreSight - Common Pitfalls

- There is a lot of ground to cover:
  - Like any powerful technology, CoreSight is complex
  - Integration with Perf handles most of the hard stuff
  - OpenCSD library does the rest

- Power Domains and Clock:
  - Most implementation will split CoreSight devices between the core and debug power domains
  - Clocks need to be enabled → the drivers should be taking care of that (if the DT is correct)

- Power Domain management:
  - Trace macrocells often share the same power domain as the CPU they are associated with
  - If CPUidle takes the CPU in a deep sleep state, the power domain is often switched off
  - ***** Don't use CoreSight when CPUidle is enabled *****
  - When developing your own solution, keep the "Power Down Control" register (TRCPDCR:PU) in mind!

Booting with CoreSight Enabled

```
sdhci-pltfm: SDHCI platform and OF driver helper
usbcore: registered new interface driver usbhid
usbhid: USB HID core driver
coresight-etm4x 22040000.etm: ETM 4.0 initialized
coresight-etm4x 22140000.etm: ETM 4.0 initialized
coresight-etm4x 23040000.etm: ETM 4.0 initialized
coresight-etm4x 23140000.etm: ETM 4.0 initialized
coresight-etm4x 23240000.etm: ETM 4.0 initialized
coresight-etm4x 23340000.etm: ETM 4.0 initialized
usb 1-1: new high-speed USB device number 2 using ehci-platform
NET: Registered protocol family 17
9pnet: Installing 9P2000 support
```

```
root@linaro-nano:~# ls /sys/bus/coresight/devices/
20010000.etf              220c0000.cluster0-funnel  23240000.etm
20030000.tpiu             22140000.etm              23340000.etm
20040000.main-funnel      23040000.etm              coresight-replicator
20070000.etr              230c0000.cluster1-funnel
22040000.etm              23140000.etm
root@linaro-nano:~#
```

Linaro

Integration of CoreSight with Perf

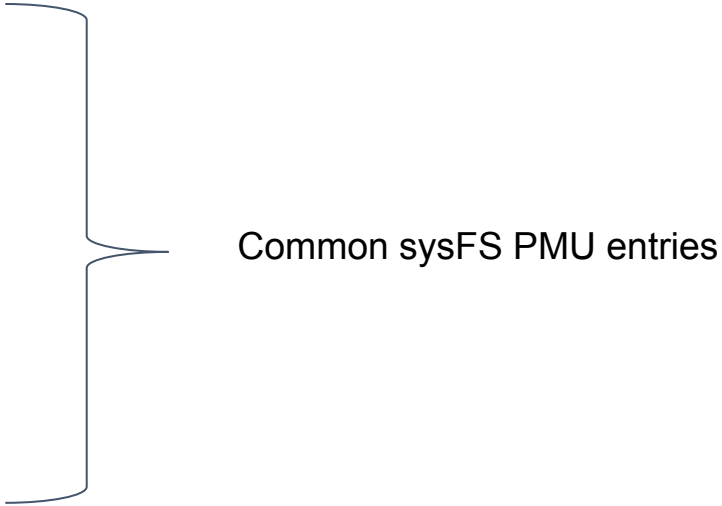
- Perf is ubiquitous, well documented and heavily used by developers
- Offers a framework already geared toward tracing
- Hides most of the complexity inherent to CoreSight
- Provides tools facilitating the integration of trace decoding
  - No need to deal with the “metadata”

- Trace Macrocell are presented as PMUs (Performance Management Unit) to the Perf core
  - Very tight control on when traces are enabled and disabled
  - Zero copy between kernel and user space when rendering data

- PMU registration is done by the CoreSight framework → no intervention needed
- The CoreSight PMU is known as **cs_etm** by the Perf core.

CoreSight Tracers Presented as PMUs

```
linaro@linaro-nano:~$ tree /sys/bus/event_source/devices/cs_etm
/sys/bus/event_source/devices/cs_etm
├── cpu0 -> ../platform/23040000.etm/23040000.etm
├── cpu1 -> ../platform/22040000.etm/22040000.etm
├── cpu2 -> ../platform/22140000.etm/22140000.etm
├── cpu3 -> ../platform/23140000.etm/23140000.etm
├── cpu4 -> ../platform/23240000.etm/23240000.etm
├── cpu5 -> ../platform/23340000.etm/23340000.etm
├── format
│   ├── cycacc
│   └── timestamp
├── nr_addr_filters
├── perf_event_mux_interval_ms
├── power
│   ├── autosuspend_delay_ms
│   ├── control
│   ├── runtime_active_time
│   ├── runtime_status
│   └── runtime_suspended_time
├── subsystem -> ../../bus/event_source
├── type
└── uevent
9 directories, 11 files
linaro@linaro-nano:~$
```



Common sysFS PMU entries

OpenCSD for Trace Decoding

- Open **C**ore**S**ight **D**ecoding library
- A joint development effort between Texas Instrument, ARM and Linaro
- Free and open solution for decompressing Program Flow Traces
- Currently support ETMv3, PTM and ETMv4
- Also has support for MIPI trace decoding (output from STM)
- Fully integrated with Perf
- Available on gitHub**[1]** for anyone to download, integrate and modify
- In-depth presentation in recent CoreDump blog post**[2]**

[1]. https://github.com/Linaro/OpenCSD

[2]. http://www.linaro.org/blog/core-dump/opencsd-operation-use-library/

Putting it all Together

So far we know that….

- We can do HW assisted tracing on ARM using CoreSight IP blocks
- The Linux kernel offers a framework and a set of drivers supporting CoreSight
- The openCSD library is available to anyone who wishes to decode CoreSight traces
- CoreSight and openCSD have been integrated with Perf

- It is now time to see how things fit together and use the technology in real-world scenarios

Getting the Right Tools

- First, the OpenCSD library needs to be downloaded
  - On gitHub**[1]** the master branch carries the OpenCSD code
  - Stable versions are tagged
  - Older version had dedicated branches -- please stick with the latest
  - The “HOWTO.md” tells you which kernel branch will work with the latest version
  - Kernel branches will disappear in a near future

- The kernel branches on gitHub carry the user space functionality
  - There is always a rebase for the latest kernel version
  - perf [record, report, script]
  - Upstreaming of these tools is currently underway
  - Include those patches in a custom tree if CoreSight integration with Perf is to be used

[1]. https://github.com/Linaro/OpenCSD

Compiling OpenCSD and the Perf Tools

- OpenCSD is a stand alone library - as such it is not part of the kernel tree
- OpenCSD libraries need to be linked with the Perf Tools
  - If perf tools aren't linked with OpenCSD, trace decoding won't work

- Follow instructions in the **"HOWTO.md"** on gitHub
- Always set environment variable "CSTRACE_PATH"

```
CC       tests/thread-mg-share.o
CC       util/cs-etm-decoder/cs-etm-decoder-stub.o
CC       util/intel-pt-decoder/intel-pt-decoder.o
```

No CS decoding

```
CC       util/auxtrace.o
CC       util/cs-etm-decoder/cs-etm-decoder.o
LD       util/cs-etm-decoder/libperf-in.o
```

With CS decoding

Linaro

Using CoreSight with Perf

- CoreSight PMU works the same way as any other PMU

```
./perf record -e event_name/{options}/ --perf-thread ./main
```

- As such, in its simplest form:

```
./perf record -e cs_etm/@20070000.etr/ --perf-thread ./main
```

- Always specify a **sink** to indicate where to put the trace data
  - A list of all CoreSight devices is available in sysFS

```
linaro@linaro-nano:~$ ls /sys/bus/coresight/devices/
20010000.etf              20040000.main-funnel      22040000.etm              22140000.etm
230c0000.cluster1-funnel  23240000.etm              coresight-replicator      20030000.tpiu
20070000.etr              220c0000.cluster0-funnel  23040000.etm              23140000.etm
23340000.etm
```

Using CoreSight with Perf (Cont'd)

- The default options will often generate too much trace data
- The option 'k' and 'u' can be used to limit tracing to kernel or user space

```
./perf record -e cs_etm/@20070000.etr/u --perf-thread ./main
./perf record -e cs_etm/@20070000.etr/k --perf-thread ./main
```

- Kernel space tracing requires root privileges

- Address filters are provided to limit tracing to specific areas
  - Address range filters → use the "filter" keyword
  - Start/stop filters → user the "start" and "stop" keywords

Using CoreSight Address Range Filters

- Trace between one address and another
- **Exclude** jumps outside of the range

Kernel Space example:

```
$ perf record -e cs_etm/@20010000.etr/k --filter            \
                          'filter 0xffffff8008562d0c/0x48' --per-thread ./main
```

User space example:

```
$ perf record -e cs_etm/@20070000.etr/u --filter            \
                          'filter 0x72c/0x40@/opt/lib/libcstest.so.1.0' --per-thread ./main
```

Using CoreSight Start/Stop Filters

- Start at one address, stops at another
- **Include** jumps outside of the range

Kernel Space example:

```
perf record -e cs_etm/@20070000.etr/k --filter                                  \
                    'start 0xffffff800856bc50,stop 0xffffff800856bcb0' --per-thread ./main

perf record -e cs_etm/@20070000.etr/k --filter                                  \
                    'start 0xffffff800856bc50,stop 0xffffff800856bcb0,          \
                     start 0xffffff8008562d0c,stop 0xffffff8008562d30' --per-thread ./main
```

User space example:

```
perf record -e cs_etm/@20070000.etr/u --filter                              \
                        'start 0x72c@/opt/lib/libcstest.so.1.0'             \
                        'stop 0x26@/main' --per-thread ./main
```

Limitation on CoreSight Filters

- Limited to the amount of address comparator found Trace Macrocells
  - Implementation dependent, currently limited to 8

- Range and start/stop filters can't be combined in the same session

Example that is **not** supported:

```
perf record -e cs_etm/@20070000.etr/k --filter                                  \
                     'start 0xffffff800856bc50,stop 0xffffff800856bcb0',    \   // start/stop
                      filter 0x72c/0x40@/opt/lib/libcstest.so.1.0'          \   // Range
                     --per-thread ./main
```

Using CoreSight with Perf (Cont'd)

- Trace data are found in the "perf.data" file

  `perf report --dump perf.data`

```
0x728 [0x30]: PERF_RECORD_AUXTRACE size: 0xf0  offset: 0  ref: 0x48b2b5695d22eed5  idx: 0  tid: 1796
cpu: -1

. ... CoreSight ETM Trace data: size 240 bytes
  0: I_ASYNC : Alignment Synchronisation.
  12: I_TRACE_INFO : Trace Info.
  17: I_ADDR_L_64IS0 : Address, Long, 64 bit, IS0.; Addr=0xFFFFFF800857ED08;
  48: I_ASYNC : Alignment Synchronisation.
  60: I_TRACE_INFO : Trace Info.
  65: I_ADDR_L_64IS0 : Address, Long, 64 bit, IS0.; Addr=0xFFFFFF800857ED08;
  96: I_ASYNC : Alignment Synchronisation.
  108: I_TRACE_INFO : Trace Info.
  113: I_ADDR_L_64IS0 : Address, Long, 64 bit, IS0.; Addr=0xFFFFFF800857ED08;
  144: I_ASYNC : Alignment Synchronisation.
  ....
```

A Simple but Real Example

“main.c”

```
#include <stdio.h>

int coresight_test1(int val);

int main(void)
{
     int val;

     val = coresight_test1(10);

     printf("val: %d\n", val);

     return 0;
}
```

“libcstest.c”

```
int coresight_test1(int val)
{
     int i;
     /*
      * A simple loop forcing the
      * instruction pointer to move
      * around.
      */
     for (i = 0; i < 5; i++)
          val += 2;

          return val;
     }
```

Code to trace

Objdump the Code to Trace

```
$ aarch64-linux-gnu-objdump -d libcstest.so.1.0


000000000000072c <coresight_test1>:
 72c:   d10083ff        sub     sp, sp, #0x20
 730:   b9000fe0        str     w0, [sp,#12]
 734:   b9001fff        str     wzr, [sp,#28]
 738:   14000007        b       754 <coresight_test1+0x28>
 73c:   b9400fe0        ldr     w0, [sp,#12]
 740:   11000800        add     w0, w0, #0x2
 744:   b9000fe0        str     w0, [sp,#12]
 748:   b9401fe0        ldr     w0, [sp,#28]
 74c:   11000400        add     w0, w0, #0x1
 750:   b9001fe0        str     w0, [sp,#28]
 754:   b9401fe0        ldr     w0, [sp,#28]
 758:   7100101f        cmp     w0, #0x4
 75c:   54ffff0d        b.le    73c <coresight_test1+0x10>
 760:   b9400fe0        ldr     w0, [sp,#12]
 764:   910083ff        add     sp, sp, #0x20
 768:   d65f03c0        ret
```

Generating Traces on the Target

```
root@linaro-nano:~# date
Wed Sep  7 20:17:36 UTC 2016

root@linaro-nano:~# uname -mr
4.8.0-rc5+ aarch64
root@linaro-nano:~# ls /opt/lib/libcstest.so*
/opt/lib/libcstest.so  /opt/lib/libcstest.so.1  /opt/lib/libcstest.so.1.0


root@linaro-nano:~# perf record -e cs_etm/@20070000.etr/u --filter 'filter  \
                                        0x72c/0x40@/opt/lib/libcstest.so.1.0' --per-thread ./main

val: 20
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.002 MB perf.data ]

root@linaro-nano:~# ls -l perf.data
-rw------- 1 root root 8176 Sep  7 20:17 perf.data
```

Collecting Traces on the Target

```
root@linaro-nano:~# ls -l perf.data
-rw------- 1 root root 8176 Sep  7 20:17 perf.data

root@linaro-nano:~# tar czf cs_example.tgz perf.data ~/.debug
```

- Why do we need the ~/.debug directory?
  - Because it contains a snapshot of all the binaries involved in the traces session
  - Comes for free with Perf
  - Everything is collected on your behalf - except the kernel image

The Importance of the ".debug" Directory

```
root@linaro-nano:~# tree .debug
.debug
├── [kernel.kallsyms]
│   └── 942a60ae69427f5dbaa1c3541671e504509bd5db
│       └── kallsyms
├── [vdso]
│   └── f1e1d7c7f2c709fb14ee135018417767eecbc0dd
│       └── vdso
├── home
│   └── linaro
│       └── main
│           └── 9a6850fab2ebbe386d3619bce3674a55622f2872
│               └── elf
├── lib
│   └── aarch64-linux-gnu
│       ├── ld-2.21.so
│       │   └── 94912dc5a1dc8c7ef2c4e4649d4b1639b6ebc8b7
│       │       └── elf
│       └── libc-2.21.so
│           └── 169a143e9c40cfd9d09695333e45fd67743cd2d6
│               └── elf
└── opt
    └── lib
        └── libcstest.so.1.0
            └── 3b3051b8a67f212a66e383fc90db3c2bde8f936f
                └── elf

18 directories, 6 files
```

....

Linaro

Off Target Trace Decoding: “perf report”

```
$ tar xf cs_example.tgz
$ rm -rf ~/.debug                // remove previous trace data
$ cp -dpR .debug ~/              // copy the current trace data
$ perf report --stdio            // by default file “perf.data” is used

# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#

# Samples: 8  of event 'instructions:u'
# Event count (approx.): 55
#
# Children      Self  Command  Shared Object     Symbol
# ........  ........  .......  .................  ......................
#
    81.82%    81.82%  main     libcstest.so.1.0  [.] 0x000000000000073c
     7.27%     7.27%  main     libcstest.so.1.0  [.] 0x000000000000072c
     5.45%     5.45%  main     libcstest.so.1.0  [.] 0x0000000000000754
     5.45%     5.45%  main     libcstest.so.1.0  [.] 0x0000000000000760
```

Off Target Trace Decoding: “perf script”

```
$ perf script
     main  1796          4   instructions:u:        7fb19c972c [unknown] (/opt/lib/libcstest.so.1.0)
     main  1796          3   instructions:u:        7fb19c9754 [unknown] (/opt/lib/libcstest.so.1.0)
     main  1796          9   instructions:u:        7fb19c973c [unknown] (/opt/lib/libcstest.so.1.0)
     main  1796          9   instructions:u:        7fb19c973c [unknown] (/opt/lib/libcstest.so.1.0)
     main  1796          9   instructions:u:        7fb19c973c [unknown] (/opt/lib/libcstest.so.1.0)
     main  1796          9   instructions:u:        7fb19c973c [unknown] (/opt/lib/libcstest.so.1.0)
     main  1796          9   instructions:u:        7fb19c973c [unknown] (/opt/lib/libcstest.so.1.0)
     main  1796          3   instructions:u:        7fb19c9760 [unknown] (/opt/lib/libcstest.so.1.0)
```

Off Target Trace Decoding: “perf script”

```
FILE: /opt/lib/libcstest.so.1.0  CPU: 3
        7fb19c972c:d10083ff     sub    sp, sp, #0x20
        7fb19c9730:b9000fe0     str    w0, [sp,#12]
        7fb19c9734:b9001fff     str    wzr, [sp,#28]
        7fb19c9738:14000007     b      7fb19c9754 <__gmon_start__@plt+0x134>
```

- Where does the first part of the address come from?

```
$ perf script --show-mmap-events | grep PERF_RECORD_MMAP2
       main  1796 PERF_RECORD_MMAP2 1796/1796: [0x400000(0x1000) @ 0 08:02 33169 1522333852]: r-xp /home/linaro/main
       main  1796 PERF_RECORD_MMAP2 1796/1796: [0x7fb19db000(0x2f000) @ 0 08:02 574 1811179601]: r-xp
/lib/aarch64-linux-gnu/ld-2.21.so
       main  1796 PERF_RECORD_MMAP2 1796/1796: [0x7fb19c9000(0x12000) @ 0 08:02 38308 4289568329]: r-xp
/opt/lib/libcstest.so.1.0
       main  1796 PERF_RECORD_MMAP2 1796/1796: [0x7fb1880000(0x149000) @ 0 08:02 543 1811179570]: r-xp
/lib/aarch64-linux-gnu/libc-2.21.so
```

Off Target Trace Decoding: “perf script”

```
$ cat range.sh
#!/bin/bash

EXEC_PATH=${HOME}/work/linaro/coresight/kernel-stm/tools/perf/
SCRIPT_PATH=${EXEC_PATH}/scripts/python/
XTOOLS_PATH=${HOME}/work/linaro/coresight/toolchain/gcc-linaro-aarch64-linux-gnu-4.8-2013.11_linux/bin/

perf --exec-path=${EXEC_PATH} script --script=python:${SCRIPT_PATH}/cs-trace-ranges.py

$ ./range.sh
range: 7fb19c972c - 7fb19c973c
range: 7fb19c9754 - 7fb19c9760
range: 7fb19c973c - 7fb19c9760
range: 7fb19c973c - 7fb19c9760
range: 7fb19c973c - 7fb19c9760
range: 7fb19c973c - 7fb19c9760
range: 7fb19c973c - 7fb19c9760
range: 7fb19c9760 - 7fb19c976c
```

Off Target Trace Decoding: “perf script”

```
$ cat disasm.py
#!/bin/bash

EXEC_PATH=${HOME}/work/linaro/coresight/kernel-stm/tools/perf/
SCRIPT_PATH=${EXEC_PATH}/scripts/python/
XTOOLS_PATH=${HOME}/work/linaro/coresight/toolchain/gcc-linaro-aarch64-linux-gnu-4.8-2013.11_linux/bin/

perf --exec-path=${EXEC_PATH}                                                    \
                          script --script=python:${SCRIPT_PATH}/cs-trace-disasm.py --    \
                          -d ${XTOOLS_PATH}/aarch64-linux-gnu-objdump
```

Off Target Trace Decoding: “perf script”

```
FILE: /opt/lib/libcstest.so.1.0  CPU: 3
        7fb19c972c:d10083ff     sub    sp, sp, #0x20
        7fb19c9730:b9000fe0     str    w0, [sp,#12]
        7fb19c9734:b9001fff     str    wzr, [sp,#28]
        7fb19c9738:14000007     b      7fb19c9754 <__gmon_start__@plt+0x134>
FILE: /opt/lib/libcstest.so.1.0  CPU: 3
        7fb19c9754:b9401fe0     ldr    w0, [sp,#28]
        7fb19c9758:7100101f     cmp    w0, #0x4
        7fb19c975c:54ffff0d     b.le   7fb19c973c <__gmon_start__@plt+0x11c>
FILE: /opt/lib/libcstest.so.1.0  CPU: 3
        7fb19c973c:b9400fe0     ldr    w0, [sp,#12]
        7fb19c9740:11000800     add    w0, w0, #0x2
        7fb19c9744:b9000fe0     str    w0, [sp,#12]
        7fb19c9748:b9401fe0     ldr    w0, [sp,#28]
        7fb19c974c:11000400     add    w0, w0, #0x1
        7fb19c9750:b9001fe0     str    w0, [sp,#28]
        7fb19c9754:b9401fe0     ldr    w0, [sp,#28]
        7fb19c9758:7100101f     cmp    w0, #0x4
        7fb19c975c:54ffff0d     b.le   7fb19c973c <__gmon_start__@plt+0x11c>
...
...
```

Linaro

Things I Haven't Talked About

- Integration with Perf:
  - When collecting traces in kernel space, the "vmlinux" file doesn't end up in the .debug directory
  - Supports "snapshot mode" letting users do trace acquisition endlessly
  - At this point only ARMv8 is integrated with perf → fairly easy to do for ARMv7
- By design, things work the same way on Intel PT
- CoreSight framework and drivers can be used from sysFS
- Upstreaming
  - All the kernel space part of the solution will be present in the 4.9 cycle
  - The user space, i.e "perf tools" are actively being upstreamed
- Support for "Cross Trigger Interface" (CTI) is coming

Thank You for Attending

The Linaro CoreSight Team:

Chunyan Zhang

Tor Jeremiassen

Mike Leach

Serge Broslavsky

Mathieu Poirier

Thank You

For further information: www.linaro.org

Linaro