LLDB: What’s in a register?

David Spickett - Arm / Assigned to Linaro
Disassembly

If it did not exist, someone would invent it.

```
$ ./bin/llvm-mc --triple aarch64-linux-unknown-gnu --disassemble <<< "0xa0 0x01 0x00 0x54"
    .text
    b.eq    #52
```

Do not need to read the manual every time.
Branch If Equal

```
subs x0, x0, x1  // Z = x0 == x1
b.eq #52         // Branch if Z is 1
```

- No explicit operands
- Flags are implicit operands to the branch

Flags also in the “Current Program Status register” (CPSR).
CPSR

(lldb) register read cpsr
cpsr = 0x60001000

13,000 pages of manual await you.

There must be a better way...

(output is from LLDB 17.0.6)
GDB

Register fields included in the XML target description.

https://sourceware.org/gdb/current/onlinedocs/gdb.html/Target-Description-Format.html
Register Fields

- name
- start (least significant bit)
- end (most significant bit)
- Set reg “type” to flags “id”
GDB

(gdb) info registers cpsr
cpsr 0x60001000 [ EL=0 SSBS C Z ]

(single bit fields that are 0 are omitted)
LLDB Catches Up

✔ Uses target XML
✔ Can print rich types for variables

❌ Parses the <flags> elements from target XML
   (not covered here, libXML handles this)
❌ Can shows registers as rich types

```c
struct Node {
    unsigned data;
    struct Node *next;
};
```

```lldb
(lldb) p n
(Node) {
    data = 1
    next = NULL
}
```
The Prototype

- XML parsing works
- Manually building the table of fields

```bash
(lldb) register read cpsr
    cpsr = 0x60001000
| N | Z | C | V | TCO | DIT | UAO | PAN | SS | IL | SSBS | BTYPE | D | ...
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | ...
```
RFC Feedback

“Would it be much harder to read if we treated the cpsr as a “fake structure” and presented the fields as we would any other structure?”

- Jim Ingham [0]

- Reuse the existing type printing code
- Get formatting for free

[0] https://discourse.llvm.org/t/rfc-showing-register-fields-in-lldb/64676/2
Fake Structures

- Each field becomes a struct member.
- Use Clang’s Abstract Syntax Tree (AST) to build the type.
- Print it as if it were a variable.
Build / Debug Cycle

C

struct Node {
    unsigned data;
    struct Node *next;
};

AST

TranslationUnitDecl
  `-RecordDecl ... struct
    Node definition...

DWARF

DW_TAG_structure_type
  DW_AT_name ("Node")
  DW_AT_

Object File

LLDB

DW_TAG_structure_type
  DW_AT_name ("Node")
  DW_AT_

AST

TranslationUnitDecl
  `-RecordDecl ... struct
    Node definition...

(lldb) p n
(Node) {
    data = 1
    next = NULL
}

What we need for registers
Register Printing

Target XML

```
<flags id="cpsr_flags" ...>
  ...
</flags>
```

AST

```
TranslationUnitDecl
`-RecordDecl struct cpsr_flags definition
  |-FieldDecl ...

= (N = 0,
  Z = 1,
  ...
```
Requirements

- Same field order regardless of LLDB’s host endian.
- Match the architecture manual (most significant bit on the left).

Arm® Architecture Reference Manual for A-profile architecture “C5.2.18 SPSR_EL1”
Implementation Defined Behaviour

- Do bitfields straddle the “storage unit” boundary, or move into a new unit?
  (a unit is some number of bytes)
Storage Units

- Solution: each register is 1 storage unit

```c
struct cpsr_flags {
    uint32_t N: 1;
    uint32_t : 30;
    uint32_t SP: 1;
};
```
Implementation Defined Behaviour #2

- What is the order within a unit?

<table>
<thead>
<tr>
<th>Unit 1</th>
<th>F1</th>
<th>F2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>F1</td>
</tr>
</tbody>
</table>
Field Order

- Clang’s pre-codegen order is “big endian” (first member occupies most significant bit)
- Matches architecture manual ✔
- Works for big endian targets ✔
- Works for little endian targets ❌

```c
struct cpsr_flags {
    uint32_t N: 1;
    uint32_t : 30;
    uint32_t SP: 1;
};
```
Swap #1: Field Order

Fit little endian values into the “big endian” struct.

\[
\begin{array}{c}
\text{[ A ] [ B ] [ C ]} \\
0b[10101][10][1]
\end{array}
\]

\[
\begin{array}{c}
\text{[ C ] [ B ] [ A ]} \\
0b[1][10][10101]
\end{array}
\]

Field positions change, field values do not.
Swap #2: Endian

- Registers do not have an endian.
- Pretend the register is in target memory.
- Target memory must be in target endian.

<table>
<thead>
<tr>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
<th>Byte 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 4</td>
<td>Byte 3</td>
<td>Byte 1</td>
<td>Byte 1</td>
</tr>
</tbody>
</table>

Host
(endian: host)

LLDB

Clang Type System
(endian: target)
The Result

(lldb) register read cpsr

cpsr = 0x60001000
    = (N = 0, Z = 1, C = 1, V = 0, SS = 0, IL = 0, SSBS = 1, D = 0, A = 0, I = 0, F = 0, nRW = 0, EL = 0, SP = 0)

Arm® Architecture Reference Manual for A-profile architecture “C5.2.18 SPSR_EL1”

(digits from the manual are for usability reasons)
Back to the Branch

```
subs  x0, x0, x1  // Z = x0 == x1
b.eq  #52         // Branch if Z is 1
```

```
(lldb) register read cpsr
   cpsr = 0x60001000
       = (N = 0, Z = 1, C = 1, ...)
```

The branch will be taken.
Is It Done?

LLDB 18 fully supports this on AArch64 Linux.

Also works with other debug servers:
- gdbserver
- mGBA Gameboy Advance Emulator [0]

Please contribute support for your favourite architecture!

[0] https://github.com/mgba-emu/mgba
Thank you