Modelling Systems Architecture

Ben Simner¹
and
Shaked Flur¹, Christopher Pulte¹, Luc Maranget², Jean Pichon-Pharabod¹, Alasdair Armstrong¹, Peter Sewell¹

¹University of Cambridge
²INRIA Paris
Background: the “architecture”
Instruction Set Architecture (ISA)
**Instruction Set Architecture (ISA)**

C6.2.66 CRC32B, CRC32H, CRC32W, CRC32X

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x04C11DB7 is used for the CRC calculation.

In Armv8-A, this is an **OPTIONAL** instruction, and in Armv8.1 it is mandatory for all implementations to implement it.

--- Note ---

ID_A64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

---

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20</th>
<th>16 15 14 13 12 11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf 0 0 0 1 1 0 1 0 1 1 0</td>
<td>Rm 0 1 0 0 sz Rn Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CRC32B variant**

Applies when sf == 0 & sz == 0.

CRC32B <dr>, <im>, <dmi>

**CRC32H variant**
CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial $x^{32} + x^{12} + x^5 + x^3 + 1$ is used for the CRC calculation.

In Armv8-A, this is an OPTIONAL instruction, and in Armv8.1 it is mandatory for all implementations to implement it.

--- Note ---
ID_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16 15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

CRC32B variant
Applies when sf == 0 && sz == 0.
CRC32B <rd>, <lsr>, <lsl>
CRC32H variant
CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial $0x04C11DB7$ is used for the CRC calculation.

In Armv8-A, this is an OPTIONAL instruction, and in Armv8.1 it is mandatory for all implementations to implement it.

--- Note ---

ID_AA64ISARO_EL1.CRC32 indicates whether this instruction is supported.

**CRC32B, CRC32H, CRC32W, CRC32X**

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>16</th>
<th>15 14 13 12</th>
<th>11 10 9</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**CRC32B variant**

Applies when $sf == 0$ & $sz == 0$.

CRC32B $<$dst>, $<$src>, $<$src>

**CRC32H variant**
Instruction Set Architecture (ISA)

C6.2.66  CRC32B, CRC32H, CRC32W, CRC32X

CRC32X variant
Applies when sf == 1 && sz == 11.
CRC32X <Wd>, <Wn>, <Xm>

Decode for all variants of this encoding
if !HaveCRCExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UNDEFINED;
if sf == '0' && sz == '11' then UNDEFINED;
integer size = 8 << UInt(sz);

Assembler symbols
<Wd> Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.
Instruction Set Architecture (ISA)

A64 Base Instruction Descriptions
C6.2 Alphabetical list of A64 base instructions

<hm>  
Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.

Operation

```c
bits(32) acc = X[n];  // accumulator
bits(size) val = X[m]; // input value
bits(32) poly = 0x04C11DB7<31:0>;

bits(32+size) tempacc = BitReverse(acc):Zeros(size);
bits(size+32) tempval = BitReverse(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  — The values of the data supplied in any of its registers.
Instruction Set Architecture (ISA)

Operation

```plaintext
bits(32) acc = X[n];  // accumulator
bits(size) val = X[m]; // input value
bits(32) poly = 0x04C11DB7<31:0>;

bits(32+size) tempacc = BitReverse(acc):Zeros(size);
bits(size+32) tempval = BitReverse(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```
Example of Observable Speculation

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR X2, [X0] //a</td>
<td>LDR X2, [X1] //d</td>
</tr>
<tr>
<td>DMB SY //b</td>
<td>CBZ X2, end</td>
</tr>
<tr>
<td>STR X2, [X1] //c</td>
<td>LDR X3, [X0] //e</td>
</tr>
<tr>
<td>Allowed: 1:X2=1; 1:X3=0;</td>
<td></td>
</tr>
</tbody>
</table>

- a: W x=1
- b: dmb
- c: W y=1
- d: R y=1
- e: R x=0

MP+dmb.sy+ctrl AArch64

Initial state: x=0; y=0; X0=x; X1=y; 0:X2=1;
Abstract Microarchitecture
Abstract Microarchitecture

Thread Subsystem

Storage Subsystem

ASL
Abstract Microarchitecture

Thread Subsystem

Storage Subsystem
RMEM
Exhaustive Architecture Explorer
Systems Software

• Self-modifying Code (Completed)
• Exceptions and Interrupts (Partial)
• TLB Maintenance (Soon …)
• Devices and System MMU (Eventually …)
Systems Software

Self Modifying Code:

- Hypervisors
- Linux
- JITs
Example: Observable Instruction Hazard

SM

<table>
<thead>
<tr>
<th>Initial state: 0:W0=&quot;MOV X30,#42&quot;, 0:X1=main</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
</tr>
<tr>
<td>1. STR W0,[X1] // a: overwrite main</td>
</tr>
<tr>
<td>2. BL main // b: call main</td>
</tr>
<tr>
<td>3. ...</td>
</tr>
<tr>
<td>4. main: MOV X0,#1 // c: fetch and execute main</td>
</tr>
<tr>
<td>Allowed: execute &quot;MOV X0,#1&quot;</td>
</tr>
</tbody>
</table>

AArch64

<table>
<thead>
<tr>
<th>Thread 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: write main=&quot;MOV X32,#42&quot;</td>
</tr>
<tr>
<td>b: jump to main</td>
</tr>
<tr>
<td>c: fetch main=&quot;MOV X0,#1&quot;</td>
</tr>
</tbody>
</table>

irf
Example: cache maintenance

SM+cachesync          AArch64

Initial state: 0:W0="MOV X30,#42", 0:X1=main

<table>
<thead>
<tr>
<th>Thread 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
</tr>
<tr>
<td>2.</td>
</tr>
<tr>
<td>3.</td>
</tr>
<tr>
<td>4.</td>
</tr>
<tr>
<td>5.</td>
</tr>
<tr>
<td>6.</td>
</tr>
<tr>
<td>7.</td>
</tr>
<tr>
<td>8.</td>
</tr>
<tr>
<td>9.</td>
</tr>
</tbody>
</table>

Forbidden: execute "MOV X0,#1"

Thread 0

- a: write main="MOV X32,#42"
- b: jump to main
- c: fetch main="MOV X0,#1"
Example: cache maintenance

| Initial state: \( 0:W0 = \text{"MOV X30,#42"}, 0:X1 = \text{main} \) |
|-----------------|-----------------|
| Thread 0        |                 |
| 1. STR W0,[X1]  | a: overwrite main |
| 2. DC CVAU, X1  | clean d-cache   |
| 3. DSB SY       | wait            |
| 4. IC IVAU, X1  | invl i-cache    |
| 5. DSB SY       | wait            |
| 6. ISB          | pipeline flush  |
| 7. BL main      | b: call main    |
| 8. ...          |                 |
| 9. main: MOV X0,#1 | c: fetch and execute main |

Forbidden: execute "MOV X0,#1"
Modelling Process

Create Test → Run on hardware → Discuss with architects → Find Model → Create Test
Models from Models

let obs = rfe | fre | coe
let dob = addr | data | ctrl;[W] …
let bob = po; [dmb]; po …
let ob = obs | dob | aob | bob

**Axiom**: ob acyclic

…
Conclusion

• [https://cl.cam.ac.uk/~bs630/](https://cl.cam.ac.uk/~bs630/) Ben.Simner@cl.cam.ac.uk

• Architecture made up of **ISA** and **System model**

• Arm have precisely specified the ISA in their **ASL** language.

• System models describe concurrent execution of many instructions.

• Models help programmers understand the architecture and check correctness of their programs.

• Systems software rely on parts of the architecture the ISA and system model do not cover (yet):
  • Instruction Fetch
  • Exceptions & Interrupts
  • Pagetables