UNIVERSITY OF CAMBRIDGE

Modelling Systems Architecture

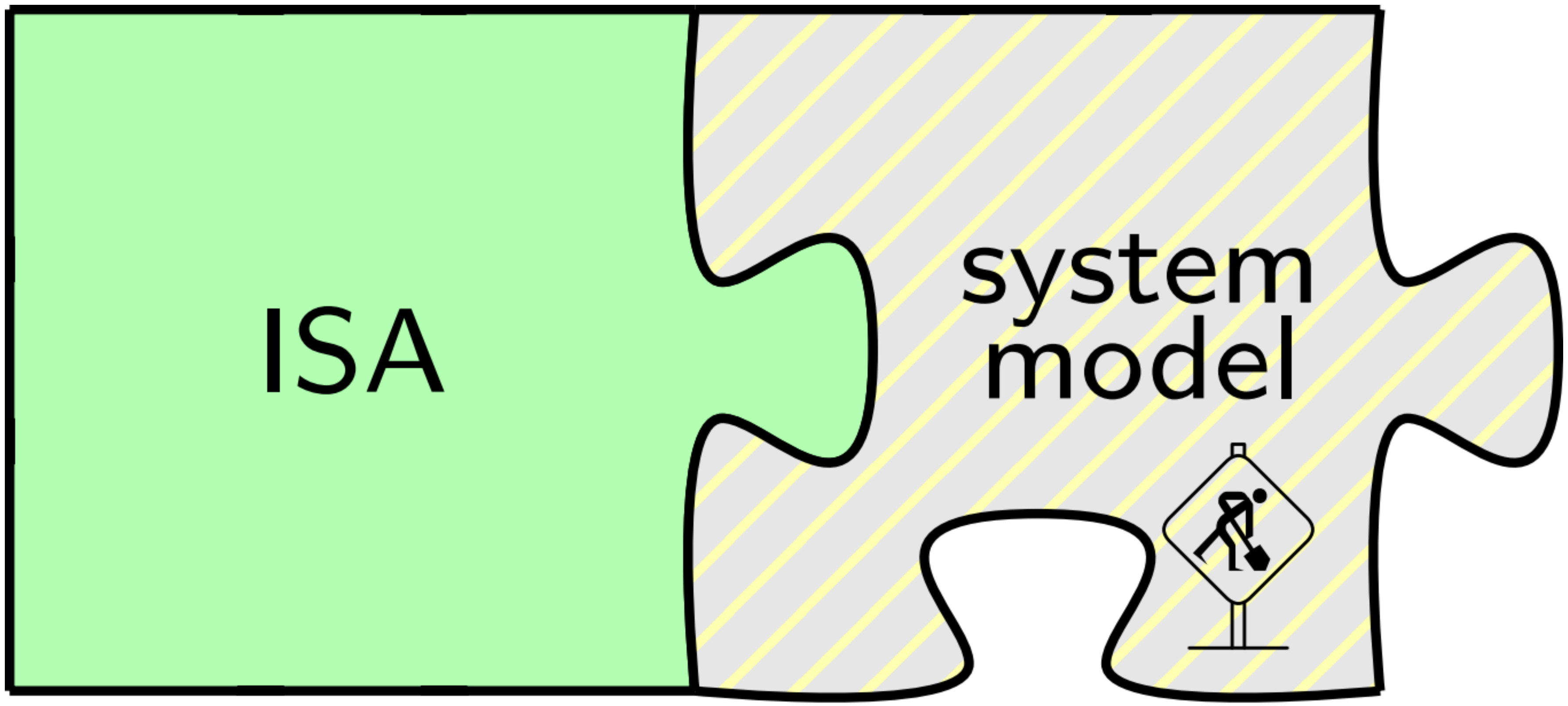
Ben Simner[1]

and

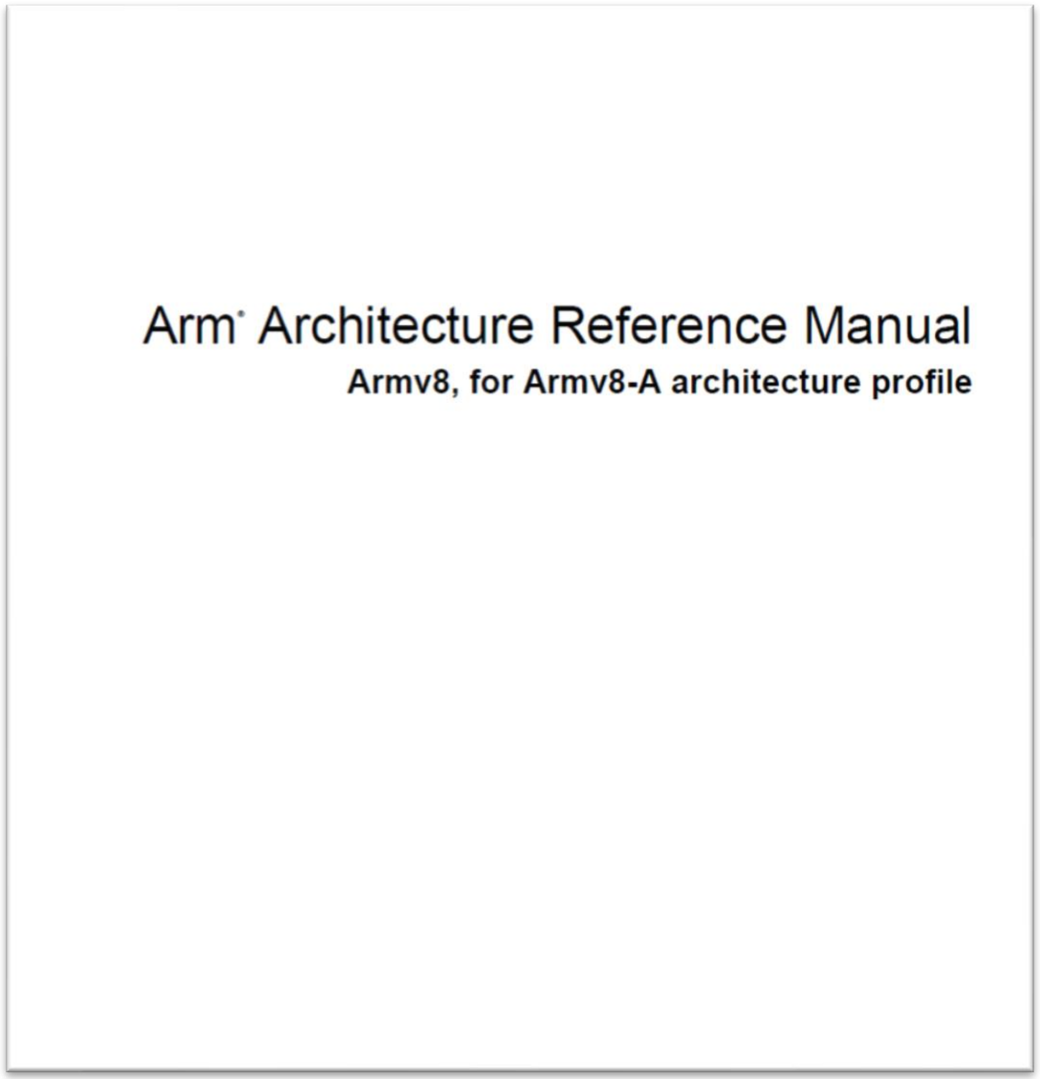
Shaked Flur[1], Christopher Pulte[1], Luc Maranget[2], Jean Pichon-Pharabod[1], Alasdair Armstrong[1], Peter Sewell[1]

[1]University of Cambridge
[2]INRIA Paris

Background: the “architecture”

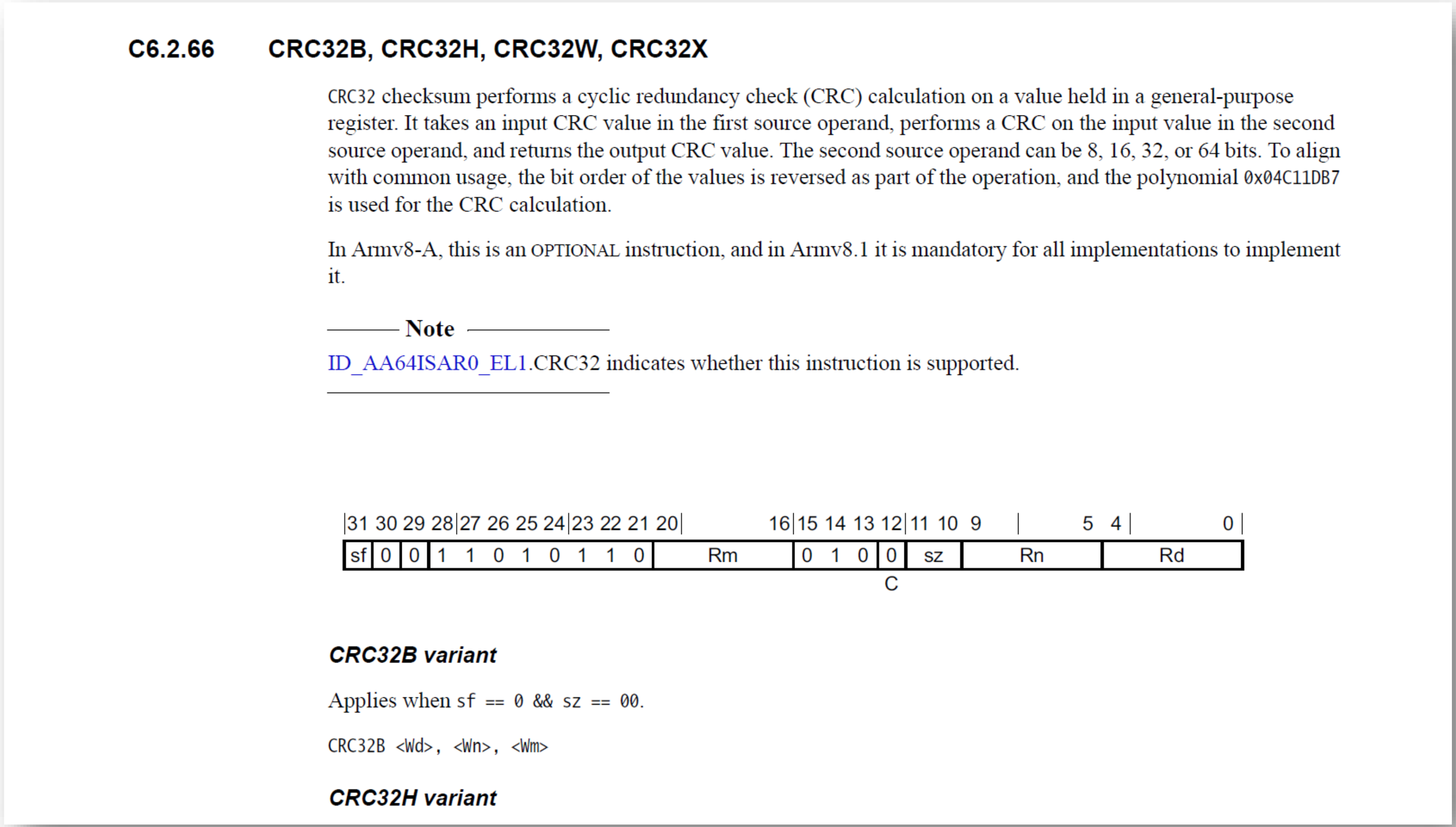


Instruction Set Architecture (ISA)


Arm® Architecture Reference Manual

Armv8, for Armv8-A architecture profile

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_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14	13	12	11 10	9 5	4 0
sf	0	0	1	1	0	1	0	1	1	0	Rm	0	1	0	0	sz	Rn	Rd
															C			

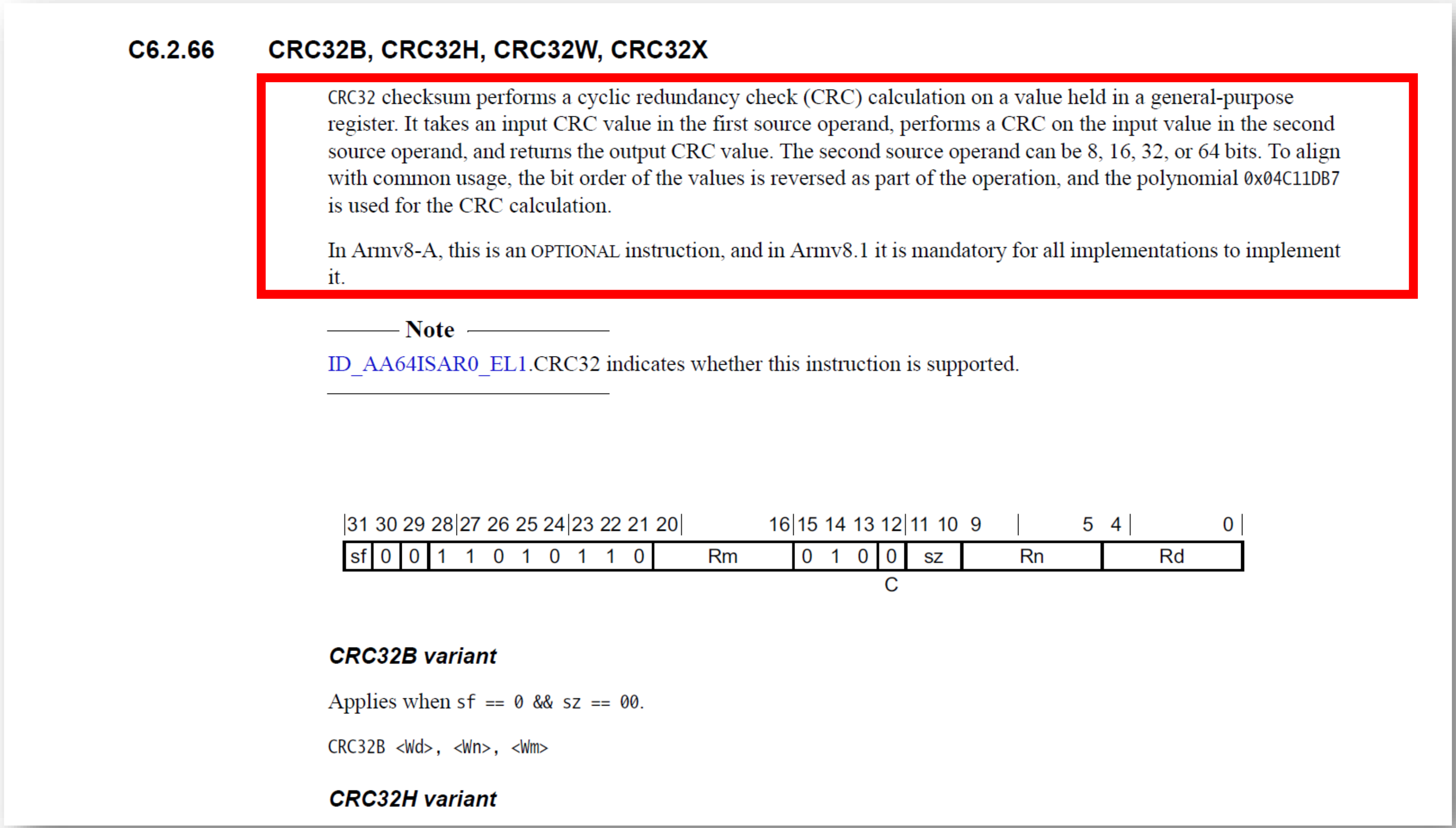
CRC32B variant

Applies when `sf == 0 && sz == 00`.

```
CRC32B <Wd>, <Wn>, <Wm>
```

CRC32H variant

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_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14	13	12	11 10	9 5	4 0
sf	0	0	1	1	0	1	0	1	1	0	Rm	0	1	0	0	sz	Rn	Rd
															C			

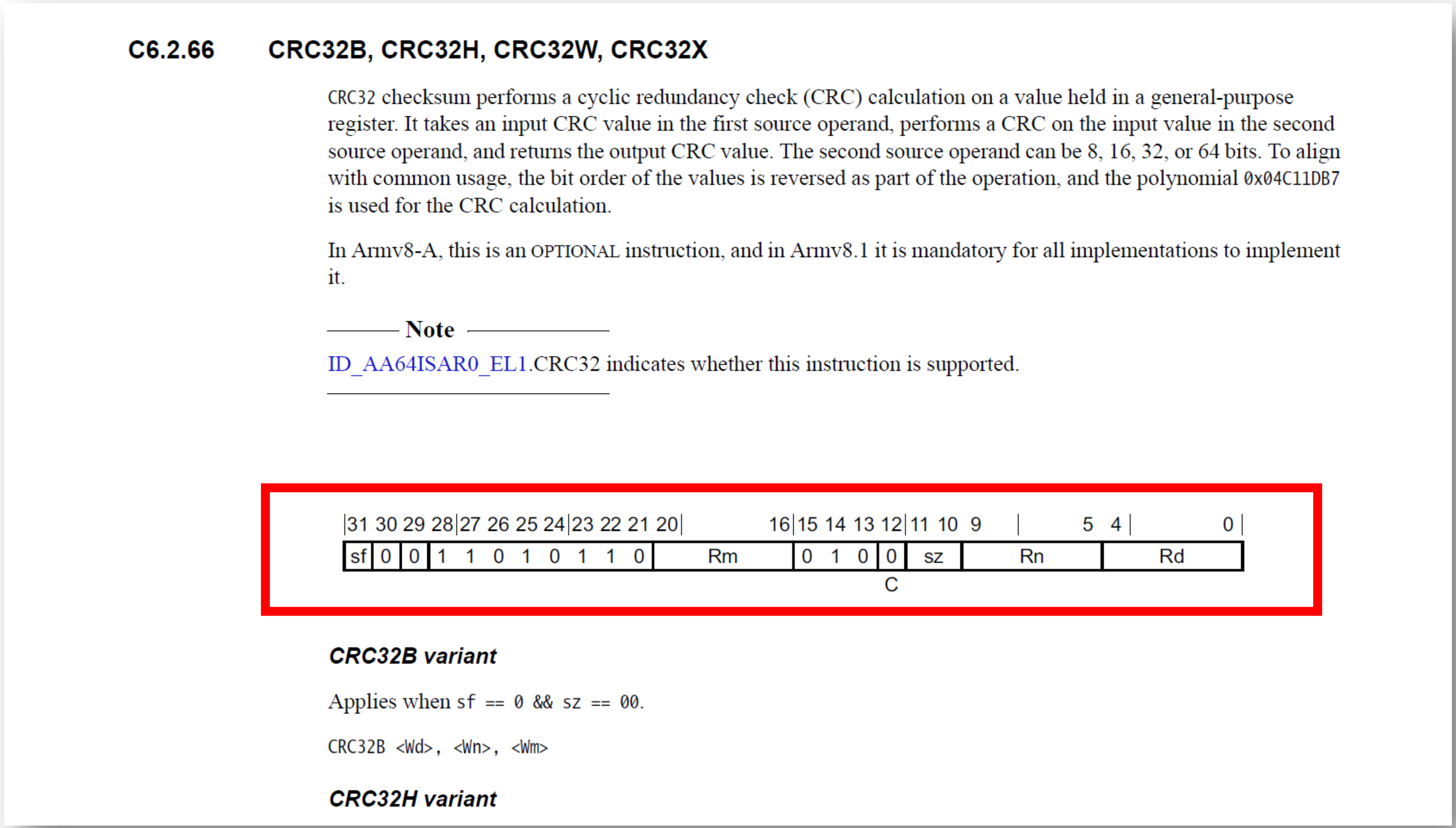
CRC32B variant

Applies when `sf == 0 && sz == 00`.

```
CRC32B <Wd>, <Wn>, <Wm>
```

CRC32H variant

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_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14	13	12	11 10	9 5	4 0
sf	0	0	1	1	0	1	0	1	1	0	Rm	0	1	0	0	sz	Rn	Rd
															C			

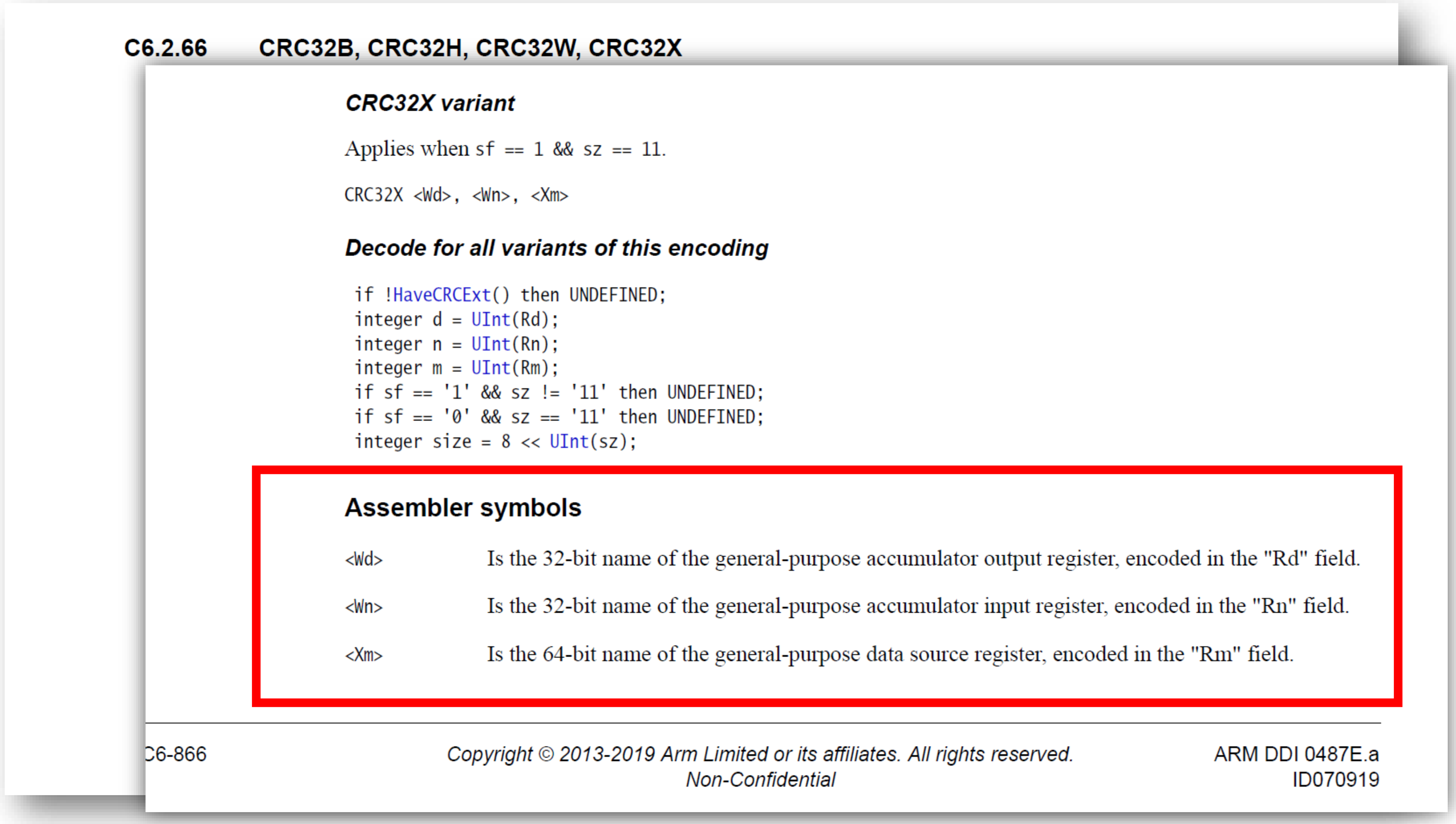
CRC32B variant

Applies when `sf == 0 && sz == 00`.

```
CRC32B <Wd>, <Wn>, <Wm>
```

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

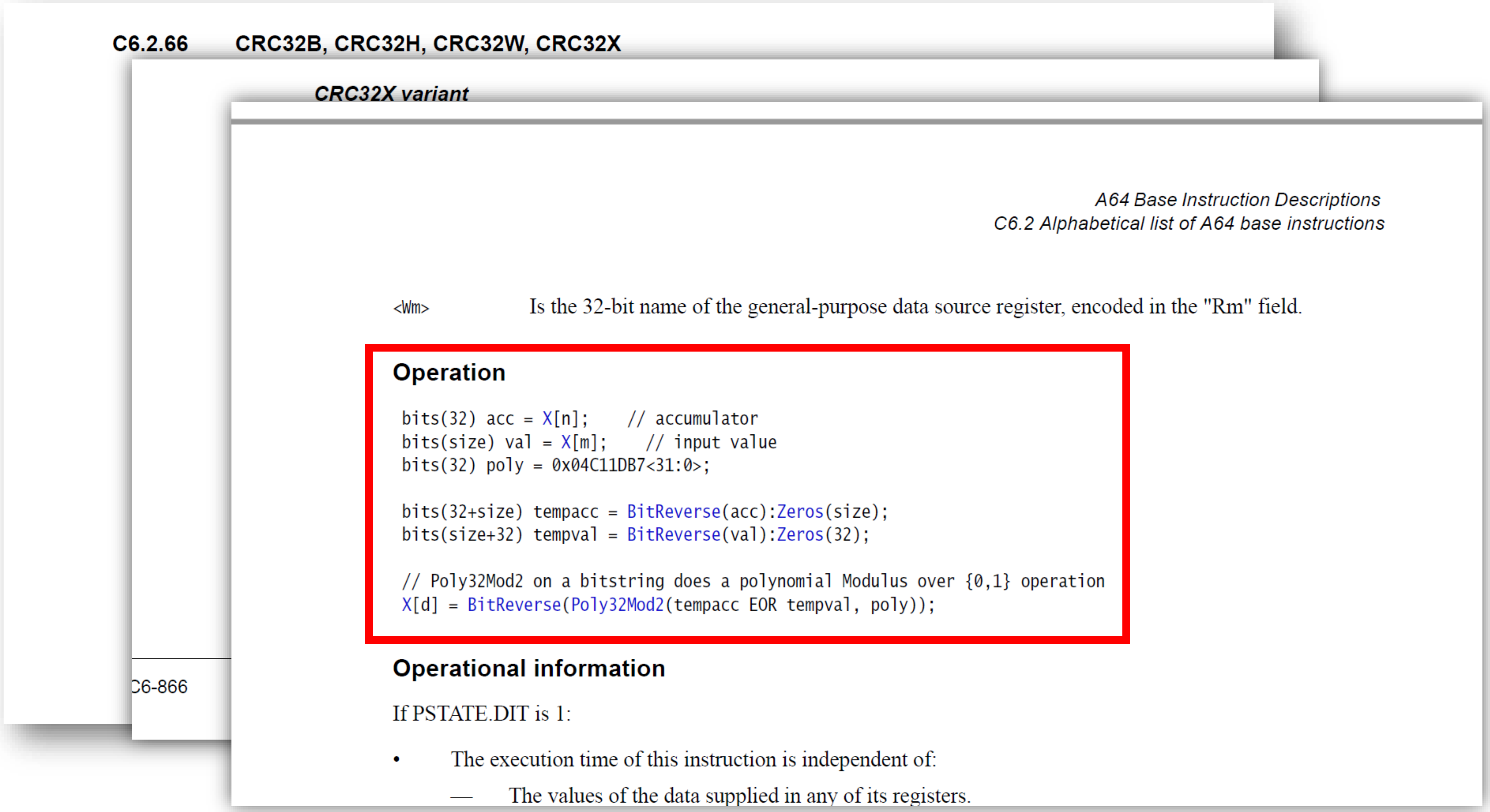
```
if !HaveCRCExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UNDEFINED;
if sf == '0' && sz == '11' then UNDEFINED;
integer size = 8 << UInt(sz);
```

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.

C6-866 *Copyright © 2013-2019 Arm Limited or its affiliates. All rights reserved.* *Non-Confidential* ARM DDI 0487E.a ID070919

Instruction Set Architecture (ISA)



C6.2.66 CRC32B, CRC32H, CRC32W, CRC32X

CRC32X variant

C6-866

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

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

Operation

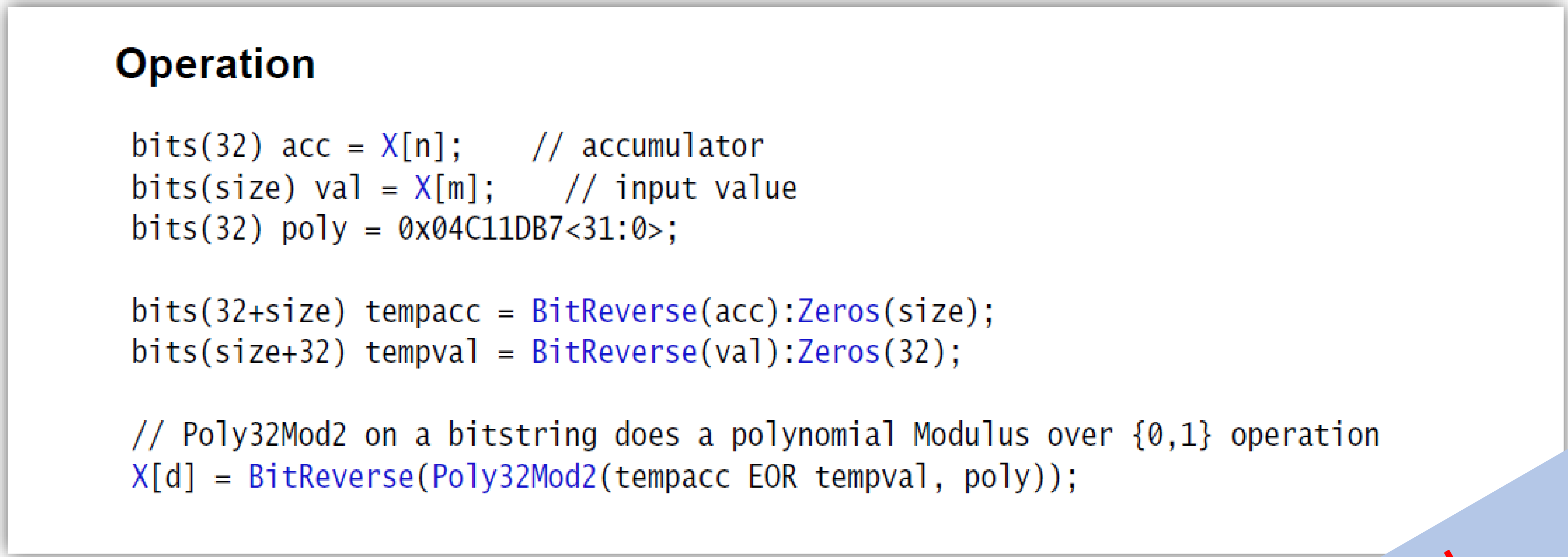
```
bits(32) acc = X[n];     // accumulator
bits(size) val = X[m];     // input value
bits(32) poly = 0x04C11DB7<31:0>;

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

```
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));
```

ASL

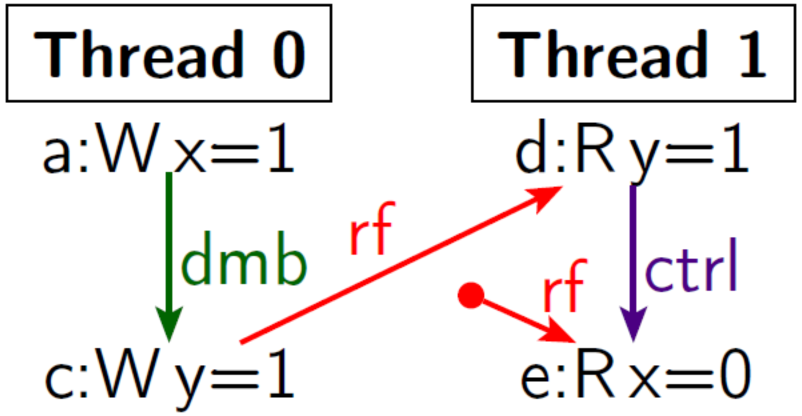
Example of Observable Speculation

MP+dmb.sy+ctrl AArch64

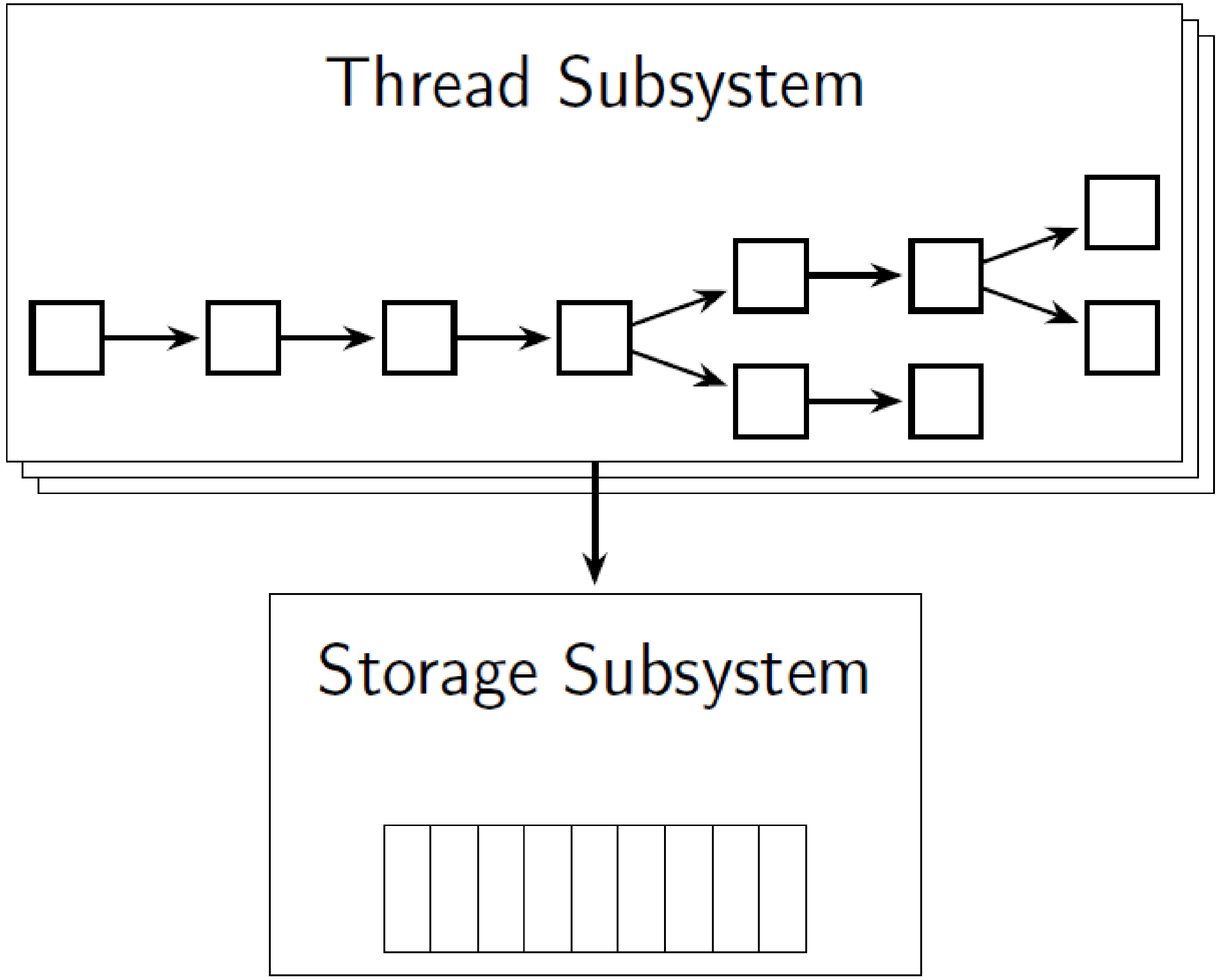
Initial state: x=0; y=0; X0=x; X1=y; 0:X2=1;

Thread 0	Thread 1
`STR X2,[X0]       //a`	`LDR X2,[X1]       //d`
`DMB SY            //b`	`CBZ X2,end`
`STR X2,[X1]       //c`	`LDR X3,[X0]       //e`
	`end:`

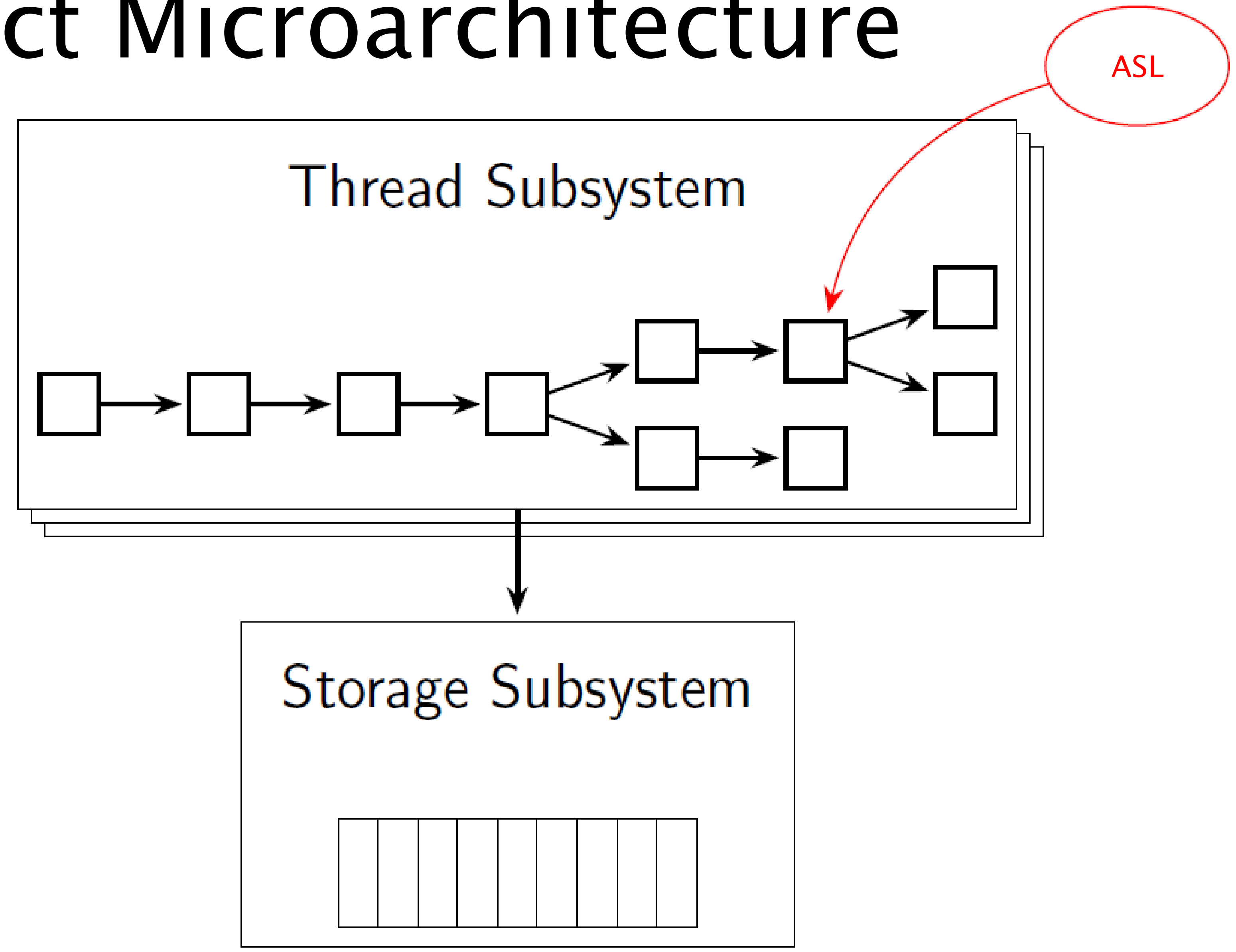
Allowed: 1:X2=1; 1:X3=0;



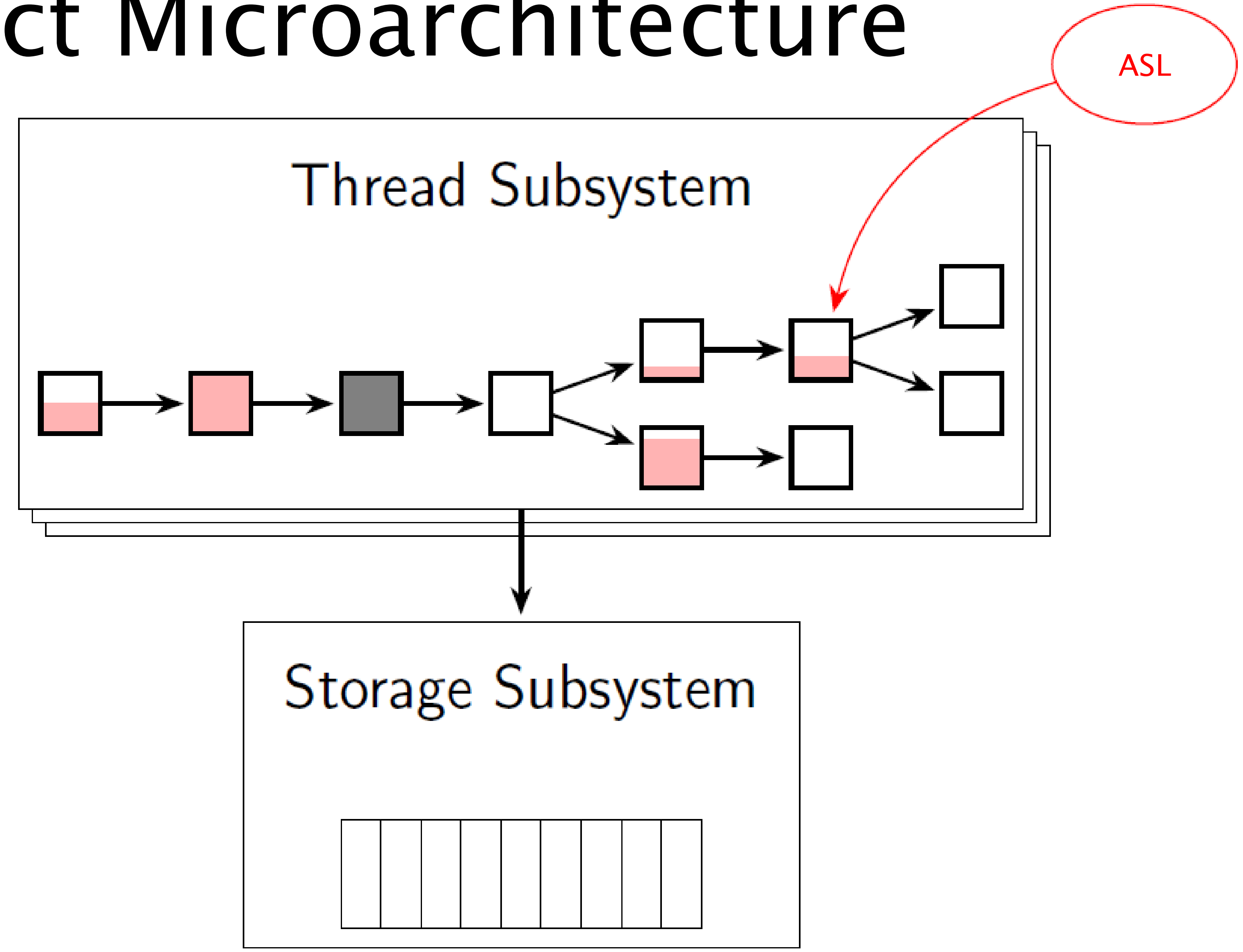
Abstract Microarchitecture



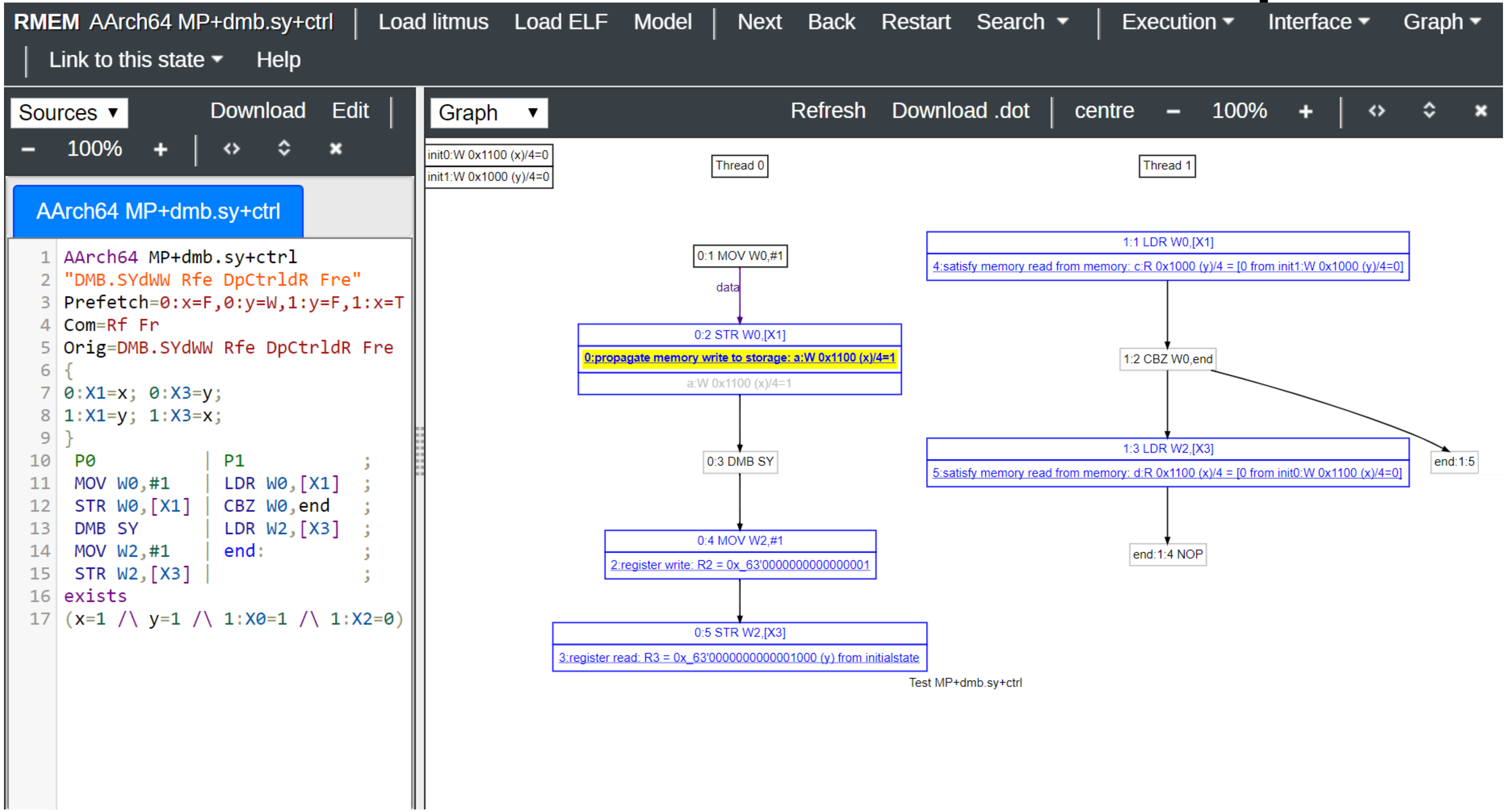
Abstract Microarchitecture



Abstract Microarchitecture



RMEM
Exhaustive Architecture Explorer



```
AArch64 MP+dmb.sy+ctrl
"DMB.SYdWW Rfe DpCtrldR Fre"
Prefetch=0:x=F,0:y=W,1:y=F,1:x=T
Com=Rf Fr
Orig=DMB.SYdWW Rfe DpCtrldR Fre
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
 P0          | P1          ;
 MOV W0,#1   | LDR W0,[X1] ;
 STR W0,[X1] | CBZ W0,end  ;
 DMB SY      | LDR W2,[X3] ;
 MOV W2,#1   | end:        ;
 STR W2,[X3] |             ;
exists
(x=1 /\ y=1 /\ 1:X0=1 /\ 1:X2=0)
```

Systems Software

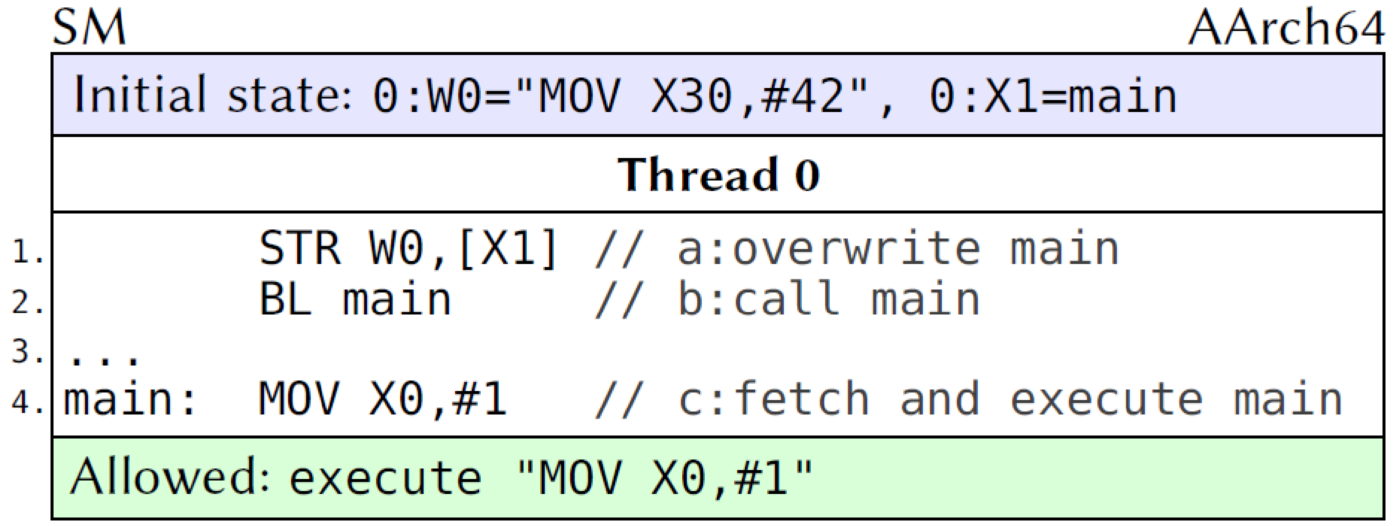
- Self-modifying Code (Completed)
- Exceptions and Interrupts (Partial)
- TLB Maintenance (Soon …)
- Devices and System MMU (Eventually …)

Systems Software

Self Modifying Code:

- Hypervisors
- Linux
- JITs

Example: Observable Instruction Hazard



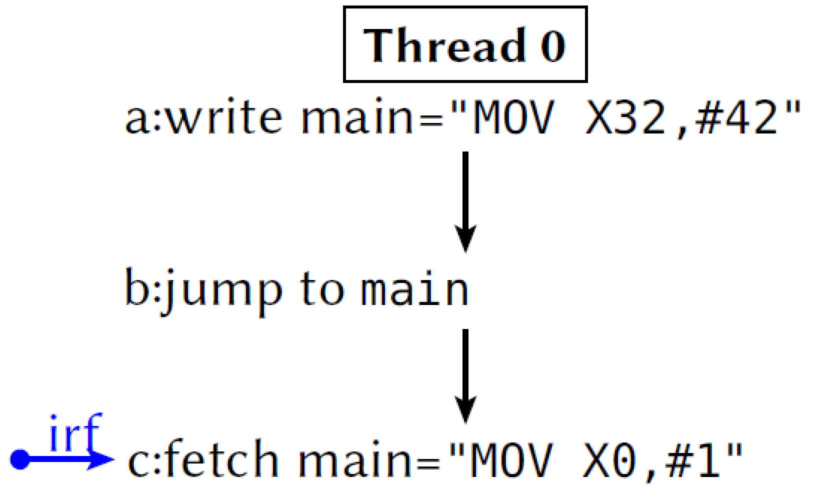
SM — AArch64

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

Thread 0

```
1.        STR W0,[X1] // a:overwrite main
2.        BL main     // b:call main
3. ...
4. main:  MOV X0,#1   // c:fetch and execute main
```

Allowed: execute `"MOV X0,#1"`



Example: cache maintenance

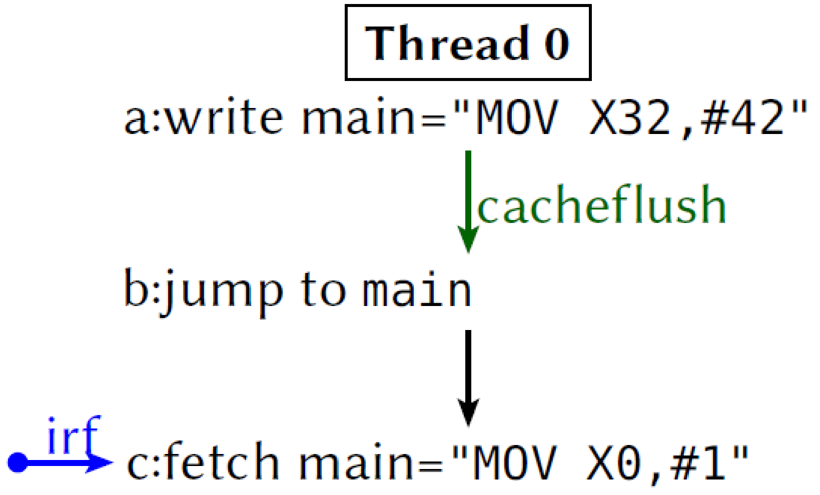
SM+cachesync AArch64

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

Thread 0

```
      STR W0,[X1] // a:overwrite main
      DC CVAU, X1 // clean d-cache
      DSB SY      // wait
      IC IVAU, X1 // invl i-cache
      DSB SY      // wait
      ISB         // pipeline flush
      BL main     // b:call main
...
main: MOV X0,#1   // c:fetch and execute main
```

Forbidden: execute `"MOV X0,#1"`



Example: cache maintenance

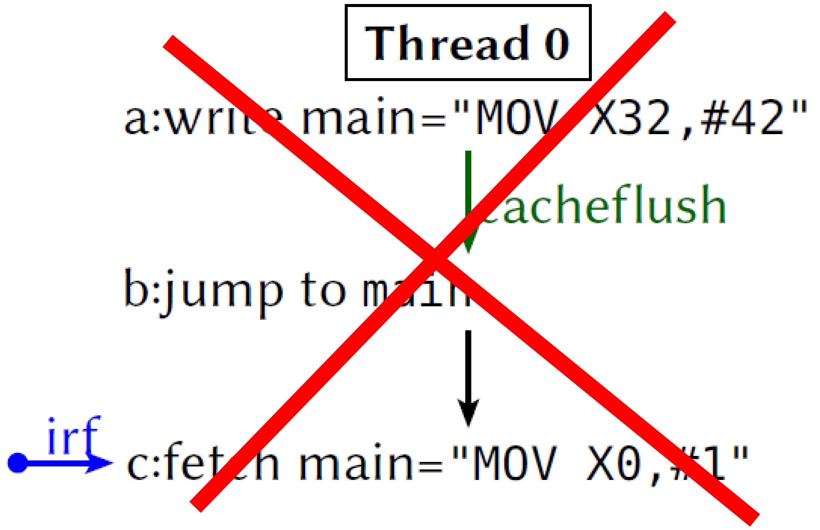
SM+cachesync AArch64

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

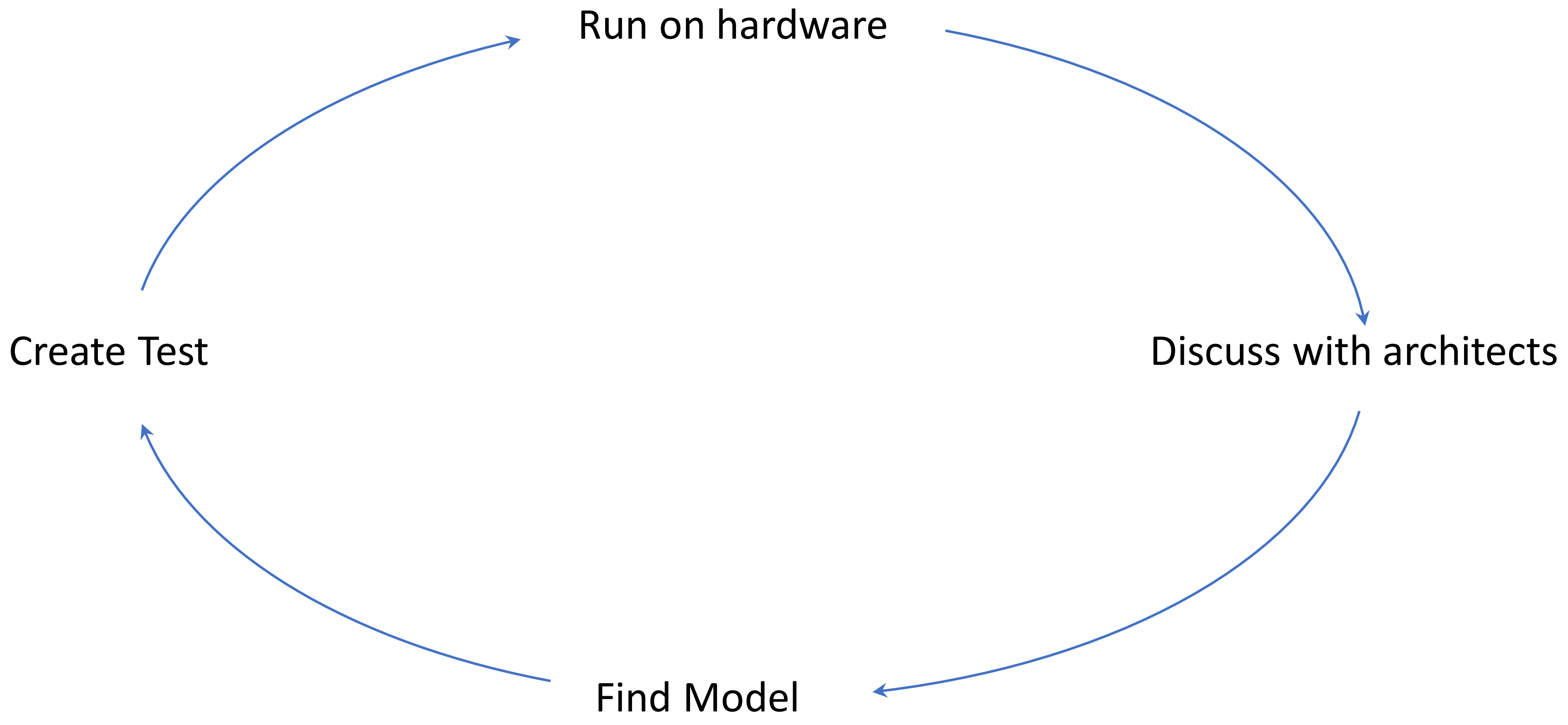
Thread 0

```
       STR W0,[X1] // a:overwrite main
       DC CVAU, X1 // clean d-cache
       DSB SY      // wait
       IC IVAU, X1 // invl i-cache
       DSB SY      // wait
       ISB         // pipeline flush
       BL main     // b:call main
...
main:  MOV X0,#1   // c:fetch and execute main
```

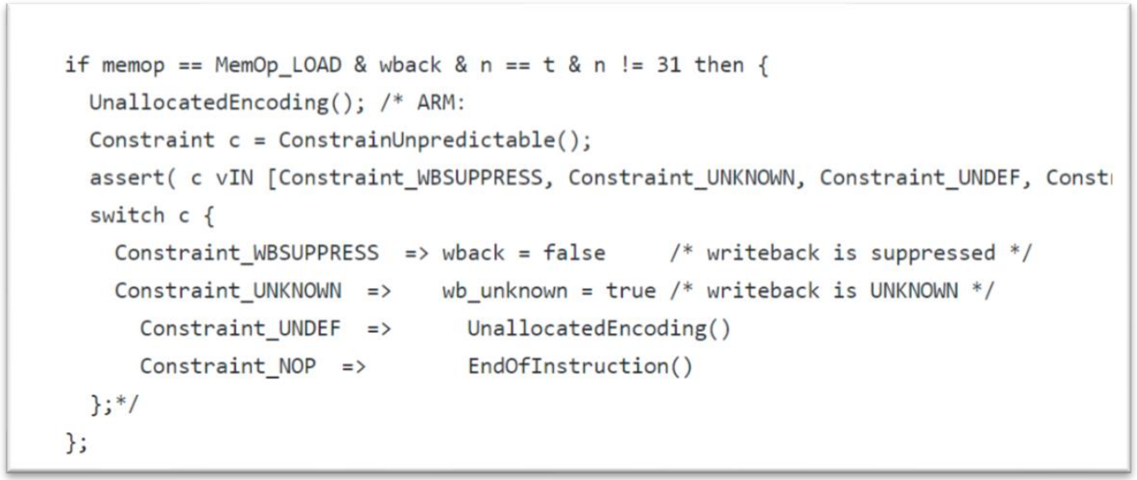
Forbidden: execute "MOV X0,#1"



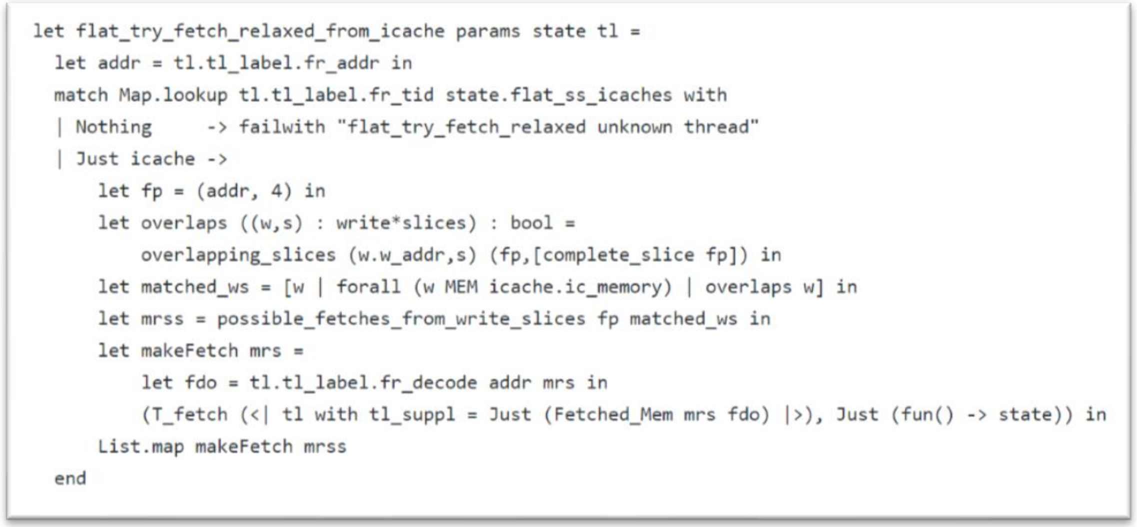
Modelling Process



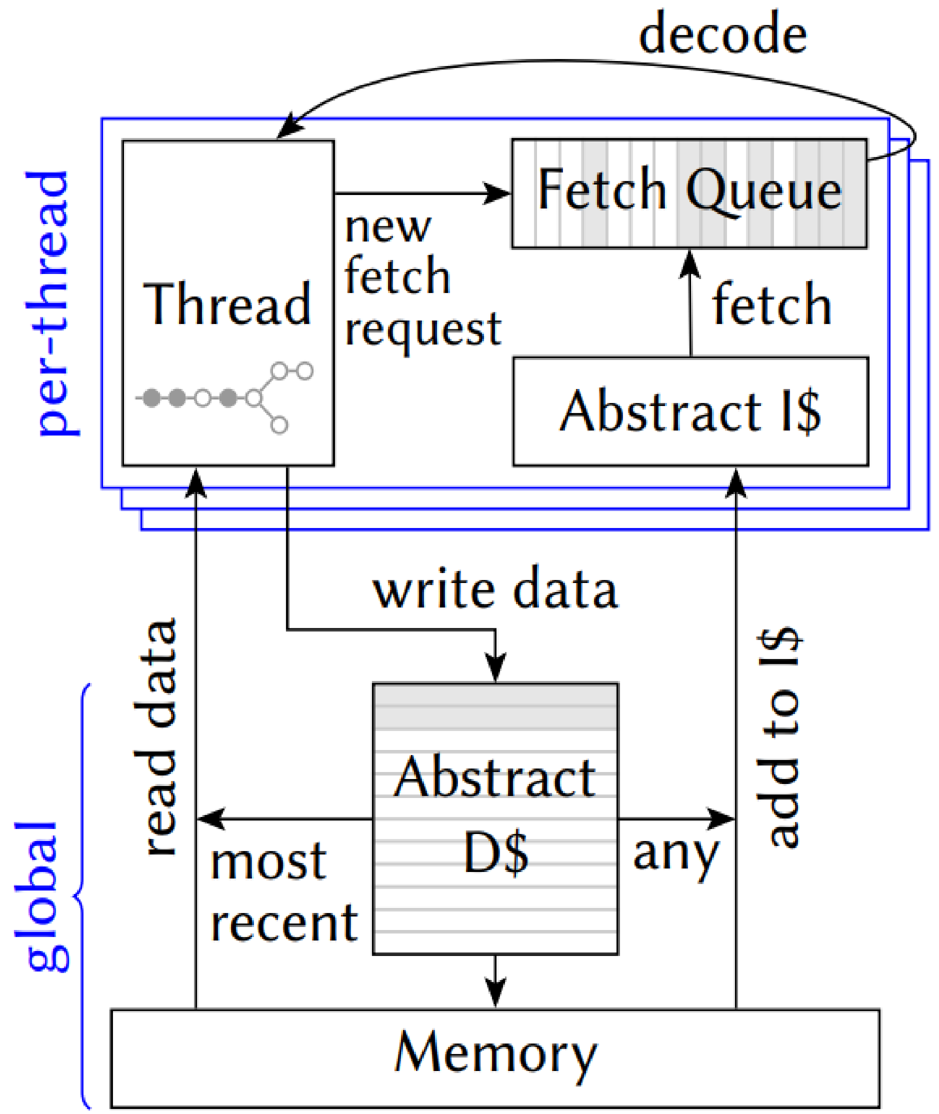
System Model



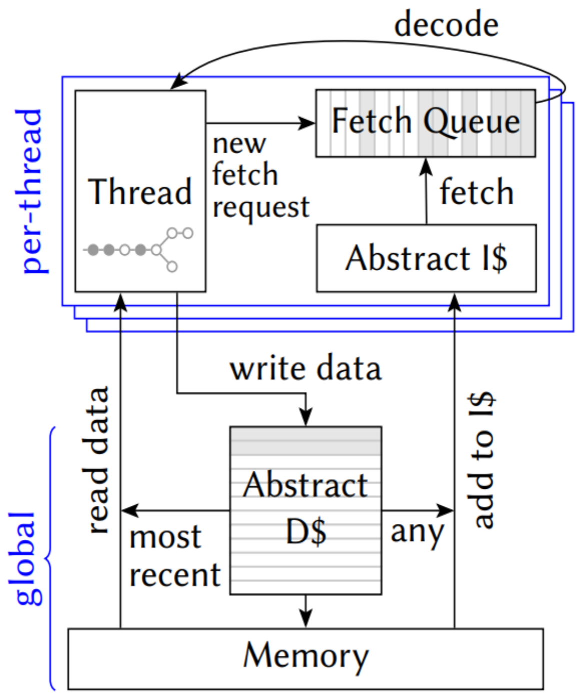
```
if memop == MemOp_LOAD & wback & n == t & n != 31 then {
  UnallocatedEncoding(); /* ARM:
  Constraint c = ConstrainUnpredictable();
  assert( c vIN [Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Const
  switch c {
    Constraint_WBSUPPRESS  => wback = false     /* writeback is suppressed */
    Constraint_UNKNOWN  =>    wb_unknown = true /* writeback is UNKNOWN */
      Constraint_UNDEF  =>      UnallocatedEncoding()
      Constraint_NOP  =>        EndOfInstruction()
  };*/
};
```



```
let flat_try_fetch_relaxed_from_icache params state tl =
  let addr = tl.tl_label.fr_addr in
  match Map.lookup tl.tl_label.fr_tid state.flat_ss_icaches with
  | Nothing     -> failwith "flat_try_fetch_relaxed unknown thread"
  | Just icache ->
      let fp = (addr, 4) in
      let overlaps ((w,s) : write*slices) : bool =
          overlapping_slices (w.w_addr,s) (fp,[complete_slice fp]) in
      let matched_ws = [w | forall (w MEM icache.ic_memory) | overlaps w] in
      let mrss = possible_fetches_from_write_slices fp matched_ws in
      let makeFetch mrs =
          let fdo = tl.tl_label.fr_decode addr mrs in
          (T_fetch (<| tl with tl_suppl = Just (Fetched_Mem mrs fdo) |>), Just (fun() -> state)) in
      List.map makeFetch mrss
  end
```



Models from Models



Operational

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
. . .

Axiomatic-Style

Conclusion

- 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