The Ballad of Dependencies: A technical report on the question of dependencies in the Arm memory model

Jade Alglave, Will Deacon, Richard Grisenthwaite and Luc Maranget

July 30, 2021

This document addresses the question of dependencies in the Arm memory model. This document almost certainly will need to be edited and completed as we receive feedback from interested readers; we are distributing it precisely for that reason. Please do reach out to us with questions and comments.

There has been a long-standing question from the Linux community as to whether branches provide order to instructions that come afterwards [4], and the Arm specification [5] has been somewhat unclear on the matter. In this document we answer this question and report on the investigation that led us to take the direction the Arm memory model has now adopted. More precisely:

- the notion and formalisation of *dependencies "down-one-leg"* is given in Section 2. This flavour of dependencies was initially thought to be the direction Arm would be taking. In the end, **dependencies "down-one-leg" were rejected by Arm**. We do present our formalisation of "down-one-leg" dependencies however, because we think it might be useful for certain audiences to see how we formalised those. Indeed long-standing discussions on Linux mailing lists seem to indicate that certain design choices at the Linux level resemble "down-one-leg" dependencies [1]. It might then be possible to reuse or exploit our work somewhat.
- **Arm opted instead for the stronger and more traditional notion of *dependencies "down-two-legs"***, given in Section 3;
- Additionally, **Arm introduced the new notion of *"pick" dependencies***, given in Section 4. Those are **used in the Arm memory model in addition to the notion of dependencies "down-two-legs"**.

In Figure 1, we show the hierarchy of concepts which are needed to define Arm dependencies overall.

All those notions have been implemented as cat files and can be experimented with using the herd tool [3]. We distribute the cat files as well as a collection of interesting litmus tests for "down-one-leg" and "pick" dependencies online [2].

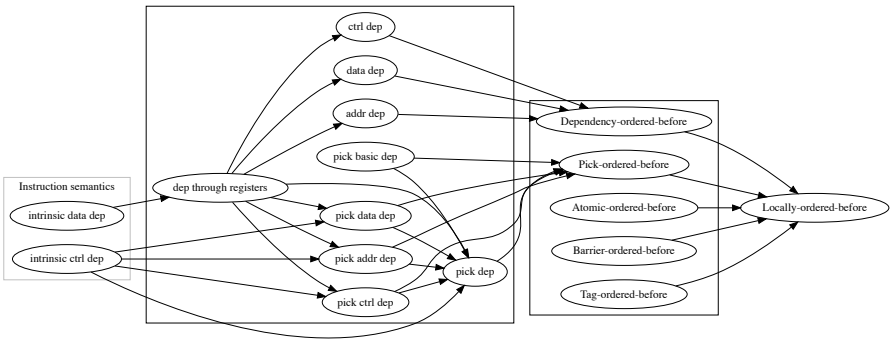

Figure 1: Hierarchy of dependencies notions in the Arm memory model

All three notions of dependencies hinge on the notion of *intrinsic dependencies*. So before presenting any of those dependencies notions, we discuss intrinsic dependencies, in Section 1.

1 Intrinsic dependencies

Intuitively, intrinsic dependencies correspond to the order in which the various effects, or events, generated by the same instruction, are to take place. They correspond to the "iico" arrows in herd diagrams. As illustrations, consider Figures 2 and 3.

In Figure 2, we give the semantics of `LDR X0,[X1]`, where the register `X1` contains the address `x`, which is initialised to the value 1, and the `LDR` instruction loads that value into register `X0`. The diagram shows the following intrinsic dependencies:

- D1: there is an Intrinsic data dependency from the register read of `X1` to the memory read of `x`. Intuitively, we need to obtain the address `x` from reading `X1` before we can get the value contained in `x`.
- D2: there is an Intrinsic data dependency from the memory read of `x` to the register write of `X0`. Intuitively, we need to obtain the value at the address `x` before we can write it to register `X0`.

In Figure 3, we give the semantics of `STR X0,[X1]`, where the register `X1` contains the address `x`, the register `X0` is initialised to 1, and the `STR` instruction stores that value into memory address `x`. The diagram shows the following intrinsic dependencies:

- D1: there is an Intrinsic data dependency from the register read of `X0` to the memory write of `x`. Intuitively, we need to get the value 1 from `X0` before storing it to memory.

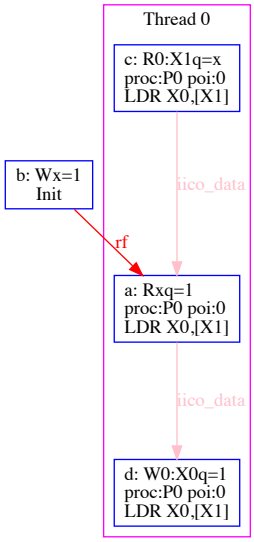

Figure 2: Intrinsic dependencies of `LDR X0,[X1]`

- D2: there is an Intrinsic data dependency from the register read of `X1` to the memory write of `x`. Intuitively, we need to get the address `x` from `X1` before storing a value to that address.

Note that there is no intrinsic dependency between the two register reads: indeed those can happen in parallel and there is no need to mandate ordering between them.

The matter of how to define intrinsic dependencies algorithmically from the semantics of instructions is currently work in progress, which we will not touch upon here but will report on in due course. Instead we provide examples of what those intrinsic dependencies are for a handful of instructions, which we hope will convey enough intuition for the discussions in this document.

1.1 Intrinsic order, data and control dependencies

There is a per-instruction *Intrinsic order dependency relation* that provides a partial order over the effects of that instruction. There is a per-instruction *Intrinsic data dependency relation* that provides a partial order over the effects of that instruction. There is a per-instruction *Intrinsic control dependency relation* that provides a partial order over the effects of that instruction.

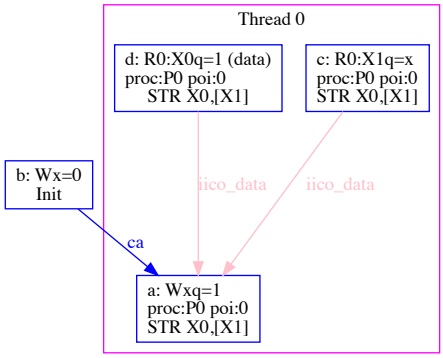

Figure 3: Intrinsic dependencies of `STR X0,[X1]`

1.2 Reads-from-register

The reader will see "rf-reg" arrows in the execution diagrams that we show throughout this report. We define the corresponding Reads-from-register notion as follows:

The Reads-from-register relation couples Read register effects and Write register effects to the same register such that each register read is paired with exactly one Write register effect in the execution of a program. A Read register effect (R2) Reads-from-register a Write register effect (W1) to the same register if and only if R2 takes its data from W1. By construction W1 must be in program order before R2 and there must be no intervening Write register effect to the same register in program order between W1 and R2.

1.3 Dependencies listings per instruction

All instructions provide an Intrinsic data dependency from the Register or Memory effects from which they take their inputs to the Register or Memory effects from which they produce their outputs, except for the following exceptions.

Conditional Selection instructions (e.g. CSEL Xd,Xm,Xn,cond) Conditional Selection instructions (e.g. `CSEL Xd,Xm,Xn,cond`) generate the following intrinsic dependencies:

- if the condition cond is true all of the following apply (see also the litmus test in Figure 4 and the corresponding execution diagram in Figure 5):
  - there is an Intrinsic control dependency from the Read register effect of PSTATE.NZCV to the Read register effect of Xm, and

```
AArch64 CSEL-true
{
0:X0=0;
0:X3=0;
}
P0;
MOV X1,#1       ;
MOV X2,#2       ;
CMP X0,#0       ;
CSEL X3,X2,X1,EQ;
exists(0:X3=2)
```

Figure 4: The `CSEL-true` litmus test

  - there is an Intrinsic data dependency from the Read register effect of Xm to the Write register effect of Xd

- if the condition cond is false all of the following apply (see also the litmus test in Figure 6 and the corresponding execution diagram in Figure 7):
  - there is an Intrinsic control dependency from the Read register effect of PSTATE.NZCV to the Read register effect of Xn, and
  - there is an Intrinsic data dependency from the Read register effect of Xn to the Write register effect of Xd

Compare-and-Swap instructions (e.g. `CAS Xs,Xt,[Xn]`) Compare-and-Swap instructions (e.g. `CAS Xs,Xt,[Xn]`) generate the following intrinsic dependencies:

- in all cases all of the following apply (see also the diagrams in Figures 9 and 11):
  - D1: there is an Intrinsic data dependency from the Read register effect of Xn to the Read memory effect of the Memory Location addressed by Xn
  - C1: there is an Intrinsic control dependency from the Register read effect of Xs to the Write register effect of Xs
  - C2: there is an Intrinsic control dependency from the Read memory effect of the Memory Location addressed by Xn to the Write register effect of Xs

- if the `CAS` instruction fails all of the following apply (see also the litmus test in Figure 8 and the corresponding execution diagram in Figure 9):

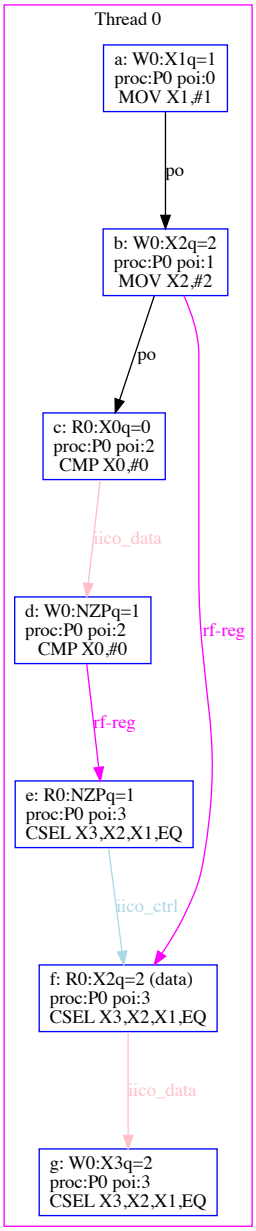

Figure 5: Intrinsic dependencies of `CSEL X3,X2,X1,EQ` if the condition is true

```
AArch64 CSEL-false
{
0:X0=0;
0:X1=0;
0:X2=0;
0:X3=0;
}
P0;
MOV X1,#1       ;
MOV X2,#2       ;
CMP X0,#3       ;
CSEL X3,X2,X1,EQ;
exists(0:X3=1)
```

Figure 6: The `CSEL-false` litmus test

  - Df1: there is an Intrinsic data dependency from the Read memory effect of the Memory Location addressed by Xn to the Write register effect of Xs

- if the `CAS` instruction succeeds all of the following apply (see also the litmus test in Figure 10 and the corresponding execution diagram in Figure 11):
  - Ds1: there is an Intrinsic data dependency from the Read register effect of Xs to the Write register effect of Xs
  - Ds2: there is an Intrinsic data dependency from the Read register effect of Xn to the Write memory effect of the Memory Location addressed by Xn
  - Ds3: there is an Intrinsic data dependency from the Read register effect of Xt to the Write memory effect of the Memory Location addressed by Xn
  - Cs1: there is an Intrinsic control dependency from the Register read effect of Xs to the Write memory effect of the Memory Location addressed by Xn
  - Cs2: there is an Intrinsic control dependency from the Read memory effect of the Memory Location addressed by Xn to the Write memory effect of the Memory Location addressed by Xn

Swap instructions (e.g. SWP Xs,Xt,[Xn]) Swap instructions (e.g. SWP Xs,Xt,[Xn]) generate the following intrinsic dependencies (see also the diagram in Figure 12):

- D1: there is an Intrinsic data dependency from the Read register effect of Xn to the Read memory effect of the Memory Location addressed by Xn

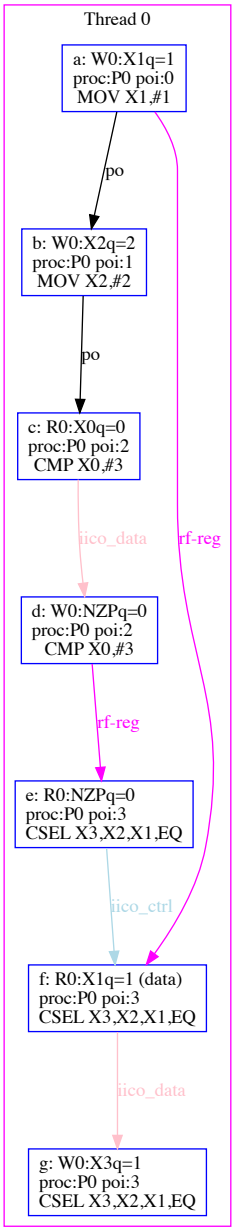

Figure 7: Intrinsic dependencies of `CSEL X3,X2,X1,EQ` if the condition is false

```
AArch64 CAS-failure
{
0:X2=0;
0:X3=0;
0:X1=x;
x=0;
}
P0;
MOV X2,#1     ;
MOV X3,#1     ;
CAS X3,X2,[X1];
exists(x=0)
```

Figure 8: The `CAS-failure` litmus test

- D2: there is an Intrinsic data dependency from the Read register effect of Xn to the Write memory effect of the Memory Location addressed by Xn
- O1: there is an Intrinsic order dependency from the Read memory effect of the Memory Location addressed by Xn to the Write memory effect of the Memory Location addressed by Xn
- D3: there is an Intrinsic data dependency from the Read memory effect of the Memory Location addressed by Xn to the Write register effect of Xt
- D4: there is an Intrinsic data dependency from the Read register effect of Xs to the Write memory effect of the Memory Location addressed by Xn
- O2: there is an Intrinsic order dependency from the Read register effect of Xs to the Write register effect of Xt

2 Dependencies "down-one-leg"

In this section we present a flavour of dependencies which we call informally "down-one-leg" dependencies. This flavour of dependencies was **not retained by Arm**, since it appears to be difficult to exploit outside of some trivial cases yet places a significant burden on the consumers of the memory model, not unlike some of the burdens Arm experienced previously with the Armv7 non-multi-copy-atomic memory model.

Overall Arm concluded that the hardware freedom afforded by the concept of "down-one-leg" dependencies was not sufficient to justify its complexity. Arm's main concern was that specifying control dependencies thus would not be very friendly to software engineers who may end up using them incorrectly or simply

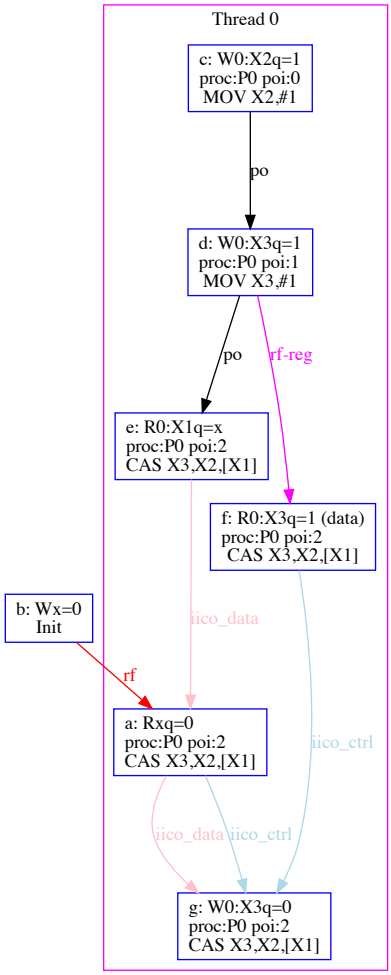

Figure 9: Intrinsic dependencies of `CAS X3,X2,[X1]` in the failure case

```
AArch64 CAS-success
{
0:X2=0;
0:X3=0;
0:X1=x;
x=0;
}
P0;
MOV X2,#1     ;
CAS X3,X2,[X1];
exists(x=1)
```

Figure 10: The `CAS-success` litmus test

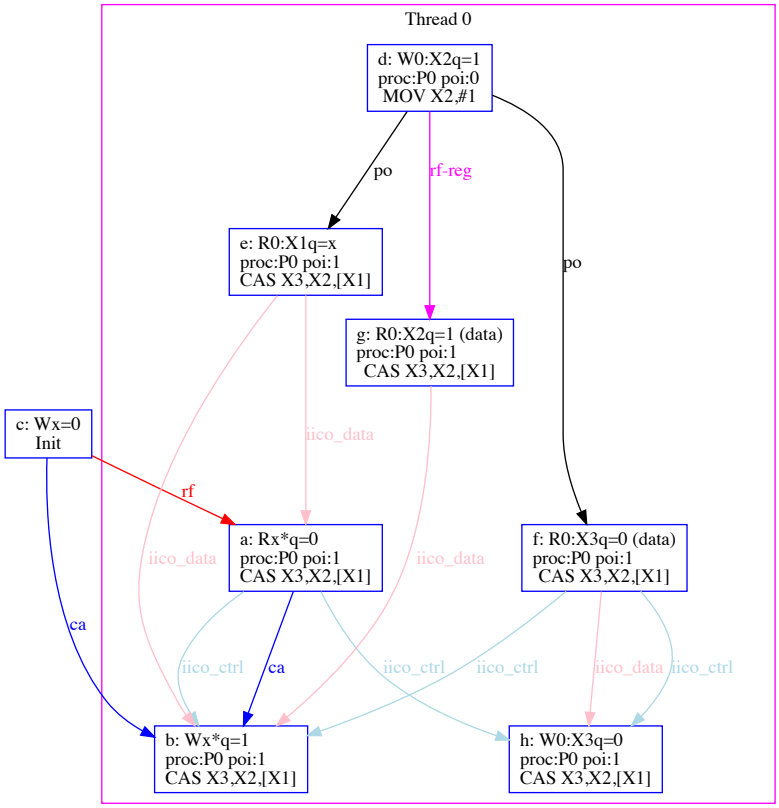

Figure 11: Intrinsic dependencies of `CAS X3,X2,[X1]` in the success case

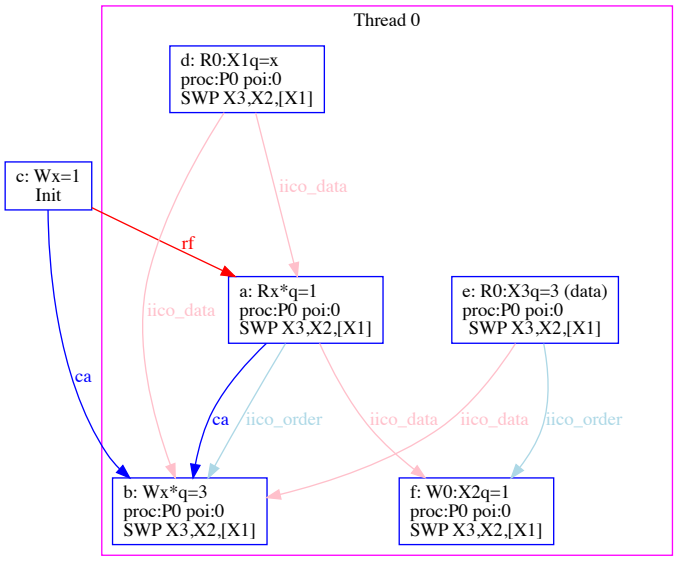

Figure 12: Intrinsic dependencies of `SWP X3,X2,[X1]`

```
A       R0 = Load
B       flags = Compare(R0, 0)
C       R1 = 1
D       conditional branch over
E       R1 = 2
over:
F       Store(R7, [R8])         // completely independent
```

Figure 13: Motivational snippet for "down-one-leg" dependencies

```
A       R0 = Load
B       flags = Compare(R0, 0)
C       R1 = 1
D       conditional branch over
E       R1 = 2
over:
F       Store(R1, [...])
```

Figure 14: A first snippet of code

replacing them with barrier instructions, nor to hardware engineers who may miss the implications for their designs.

We do present our formalisation of "down-one-leg" dependencies however, because we think it might be useful for certain audiences to see how we formalised those. Indeed long-standing discussions on Linux mailing lists seem to indicate that certain design choices at the Linux level resemble "down-one-leg" dependencies [1]. It might then be possible to reuse or exploit our work somewhat.

2.1 Motivation

Consider the snippet given in Figure 13.

There is an argument that the Store (F) can be made visible to other observers before the Load (A) has returned its data, because the Store is "independent" of the conditional branch, so the value returned by the Load (which determines the direction of the branch) does not affect the operation of the Store in any way.

This argument is behind the motivation for "down-one-leg" dependencies. Accurately defining this notion of "independent" stores however is a little convoluted. Let us try to expose why below.

2.2 Complexities of "down-one-leg" dependencies

Let us start with the snippet given in Figure 14. In this snippet, it is important that the Store (F) cannot be observed by another CPU before the initial Load

```
A       R0 = Load
B       flags = Compare(R0, 0)
C       R1 = 1
D       R2 = &y
E       conditional branch over
F       R2 = &z
over:
G       Store(R1, [R2])
```

Figure 15: A second snippet of code

```
A       R0 = Load
B       flags = Compare(R0, 0)
C       R2 = 1
D       conditional branch over
E       Store(R2, [R3])
F       R4 = Load([R3])
G       R5 = R4 & 0
H       R6 = Load([R7, R5])
over:
```

Figure 16: Half of the `PPOCA` litmus test

(A) has returned its data. This is because the value of R1 as written by the Store (F) changes depending on the value of R0, so it might be tempting to classify this as a data dependency.

Similarly for the version of the same snippet given in Figure 15, we require order between the Load (A) and the Store (G) and it is tempting to classify this as an address dependency because the address accessed by the Store (G) changes depending on the value of R0.

However, we can express both of these tests as forms of `PPOCA` (see Figure 17) which shows that they actually both behave like control dependencies, despite the Store occuring on "both legs" of the conditional branch. To illustrate, Figure 16 gives part of the `PPOCA` litmus test.

In this case, the final Load (H) can return its data before the initial Load (A) because the Store (E) can be speculatively placed into a local store buffer and forwarded within the CPU to (F), which resolves the address dependency between (F) and (H) before (A) has returned its data.

In the standard litmus test format, the full test looks like what is given in Figure 17, which is permitted and observed on existing silicon.

Now let us replace P1's code with new instruction sequences based on the snippets in Figures 14 and 15 above. For example, something based on Figure 14 is given in Figure 18 which, again, allows the final Load (J) to return its data before the initial Load (A) for similar reasons to the previous example:

```
AArch64 PPOCA
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=z; 1:X7=x;
}
P0            | P1                    ;
MOV W0,#1     | LDR W0,[X1]           ;
STR W0,[X1]   | CBZ W0,over           ;
MOV W2,#1     | MOV W2,#1             ;
STLR W2,[X3]  | STR W2,[X3]           ;
              | LDR W4,[X3]           ;
              | EOR W5,W4,W4          ;
              | LDR W6,[X7,W5,SXTW]   ;
              | over:                 ;
exists(1:X0=1 /\ 1:X4=1 /\ 1:X6=0)
```

Figure 17: The PPOCA test in standard litmus test format

```
A       R0 = Load
B       flags = Compare(R0, 0)
C       R2 = 1
D       conditional branch over
E       R2 = 2
over:
F       Store(R2, [R3])
G       R4 = Load([R3])
H       R5 = R4 $\&$ 0
J       R6 = Load([R7, R5])
```

Figure 18: A variant of half-PPOCA based on the snippet in Figure 14

the conditional branch (D) can be predicted so that the Store (F) can speculatively write to a local store buffer which is read by (G), allowing the address dependency between (G) and (J) to resolve early.

For completeness, the litmus test is given in Figure 19, which is also permitted and observed on existing silicon.

If we say that there is a data dependency between P1's initial Load (A) and its subsequent Store (F), as was tempting when analysing the snippet in Figure 14 in isolation, then this presents us with a problem because of the snippet given in Figure 20, where the data dependency between the initial Load (A) and the subsequent Store (D) ensures that the final Load (G) cannot return data before (A). The full litmus test, given in Figure 21, is forbidden.

The realisation then, is that in both Figure 14 and Figure 15, the dependency between the Load and the Store is a control dependency, despite the Store being

```
AArch64 PPOCA-variant
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=z; 1:X7=x;
}
P0            | P1                    ;
MOV W0,#1     | LDR W0,[X1]           ;
STR W0,[X1]   | MOV W2,#1             ;
MOV W2,#1     | CBZ W0,over           ;
STLR W2,[X3]  | MOV W2,#2             ;
              | over:                 ;
              | STR W2,[X3]           ;
              | LDR W4,[X3]           ;
              | EOR W5,W4,W4          ;
              | LDR W6,[X7,W5,SXTW]   ;
exists(1:X0=1 /\ 1:X4=2 /\ 1:X6=0)
```

Figure 19: A variant of PPOCA based on the snippet in Figure 14

```
A       R0 = Load
B       R2 = R0 ^ R0
C       R2 = R2 + 1
D       Store(R2, [R3])
E       R4 = Load([R3])
F       R5 = R4 ^ R4
G       R6 = Load([R7, R5])
```

Figure 20: A third snippet

```
AArch64 ThirdSnippet
{
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=z; 1:X7=x;
}
P0            | P1                    ;
MOV W0,#1     | LDR W0,[X1]           ;
STR W0,[X1]   | EOR W2,W0,W0          ;
MOV W2,#1     | ADD W2,W2,#1          ;
STLR W2,[X3]  | STR W2,[X3]           ;
              | LDR W4,[X3]           ;
              | EOR W5,W4,W4          ;
              | LDR W6,[X7,W5,SXTW]   ;
exists(1:X0=1 /\ 1:X4=1 /\ 1:X6=0)
```

Figure 21: A litmus test based on the third snippet of Figure 20

executed on "both legs" of the conditional branch. This makes it very tempting to define a control dependency so that it applies to any instructions following a conditional branch. The drawback of this approach is that it would require order for the "independent" case, such as the snippet given in Figure 13.

2.3 Concepts for defining dependencies "down-one-leg"

Points of divergence The Points of divergence of an instruction are effects which correspond to a branching decision being taken. At each point of divergence, the program order is split into two distinct branches, called executed branch and speculated branch respectively. The minimal Points of divergence in a program execution are the points of divergence which have no other point of divergence before them in program order.

Dependency through registers A dependency through registers from a first effect (E1) to a second effect (E2) exists within a PE if and only if any of the following applies:

- E1 is a register write W1 which has not been generated by a Store Exclusive, E2 is a register read R2 and R2 reads-from-register W1, or
- E1 and E2 have been generated by the same instruction and E1 is before E2 in the intrinsic dependencies of that instruction, or
- there is a dependency through registers from E1 to a third effect E3, and there is a dependency through register from E3 to E2.

Address dependency An Address dependency from a Memory read R1 to a subsequent Memory effect RW2 exists if and only if there is a Dependency through registers from R1 to a Register effect E3 generated by the same instruction as RW2, and E3 affects the address part of RW2, and one of the following applies:

- RW2 is a Memory write effect W2, or
- RW2 is a Memory read effect R2, and there is no Point of divergence D4 such that there is a Dependency through registers from R1 to D4 and from D4 to R2.

Data dependency A data dependency from a memory read (R1) to a subsequent memory write (W2) exists if and only if there is a dependency through registers from R1 to a register effect E3 generated by W2, and E3 affects the data part of W2.

Pre-control dependency A pre-control dependency from a memory read (R1) to a subsequent memory effect (RW2) exists if and only if there is a dependency through register from R1 to a point of divergence D3, and D3 is program-order-before RW2.

Antecedent An effect E1 is an antecedent of an effect E2 if and only any of the following applies:

- there is an intrinsic dependency from E1 to E2, or
- there is a read-from-register from E1 to E2, or
- E1 is a memory write effect W1, E2 is a memory read effect R2, and R2 is a Local read successor of W1,
- E1 is memory write effect W1, E2 is a memory write effect W2, and W2 is the immediate Local write successor of W1.

Pre-equivalent effects Two effects E1 and E2 are pre-equivalent if and only if all of the following applies:

- E1 and E2 are generated by the same instruction on different branches of the same Point of divergence, with the same identifier
- One of the following applies:
  - E1 is a memory write effect
  - E1 is a memory read effect and either:
    - E1 reads from a write on the same observer
    - E1 reads-from a write effect W1 on a different observer, E2 is a memory read effect (R2) and R2 reads-from W1

Equivalent effects Two effects E1 and E2 are equivalent if and only if all of the following applies:

- E1 and E2 are pre-equivalent
- If an effect E1' is an antecedent of E1, then there exists an antecedent E2' of E2, and E1' and E2' are equivalent
- If an effect E2' is an antecedent of E2, then there exists an antecedent E1' of E1, and E2' and E1' are equivalent

Always executed memory write effects A memory write effect W1 is always executed with respect to a point of divergence D2 if and only if all of the following apply:

- W1 is in program order after D2 on one of the branches stemming from D2
- there exists a memory write effect W1' equivalent to W1 on the other branch stemming from D2.

Control dependency A control dependency from a memory read (R1) to a subsequent memory effect (RW2) exists if and only if all of the following applies:

- there is a pre-control dependency from R1 to RW2 via a minimal point of divergence D3, and
- RW2 is not always executed with respect to D3.

2.4 Formalisation of "down-one-leg" dependencies in cat

In Figures 22, 23 and 24 we give the cat code for computing always executed write effects. In Figure 25, we give the cat code for computing address and data dependencies, as well as control dependencies "down-one-leg".

```
(* Utilities *)
let roots(r) = domain(r) \ range(r)
let succs (e,r) = range ([e];r)

let mapset2rel f =
  let rec map_rec xs = match xs with
  || {} -> 0
  || x ++ xs ->
     f x | map_rec xs
  end in
  map_rec

let mapset2set f =
  let rec map_rec xs = match xs with
  || {} -> {}
  || x ++ xs ->
     f x | map_rec xs
  end in
  map_rec
```

Figure 22: Cat file computing always executed write effects, cont'd in Figure 23

```
(* Basic definitions *)
let poDorW = [PoD|W];po;[PoD|W]
let nextDorW = poDorW \ (poDorW;poDorW)
let sibling =
  let inv = nextDorW^-1 in
  fun s -> ([s];inv;nextDorW;[s])\(id|inv|nextDorW)
let DpoDorW = [PoD];poDorW

(* Add pair dw for w in ws. More precisely:
   addD d (ws,dw) = dw U {(d,w) | w in ws} *)
let addD d =
  let rec do_rec ws =  match ws with
  || {} -> fun k -> k
  || w ++ ws ->
       let kws = do_rec ws in
       fun k -> (d,w) ++ kws k
  end in do_rec

(* Set combination: union of equiv-related sets *)
let addifequiv (equiv,ws0) = mapset2set (fun w -> succs (w,equiv) & ws0)
let combine (equiv,ws1,ws2) =
  addifequiv (equiv,ws1) ws2 | addifequiv (equiv,ws2) ws1
```

Figure 23: Cat file computing always executed write effects, cont'd in Figure 24

```
(* Tree scan: returns ws,dw, where ws is the set of writes always executed at
this level and dw is the relation from PoD to writes always executed *)
let rec case_disjunction (equiv,e) = match { e } & PoD with
|| {} ->  (* e is not a PoD, hence is a write *)
    let (ws,dw) = write_acc (equiv, (succs (e,nextDorW))) in
    (e ++ ws,dw)
|| d ++ es ->   (* e is a PoD *)
   let nexts = (succs (d,nextDorW)) in
   match sibling(nexts) with
   || {} -> (* Only one branch, nothing equivalent *)
      ({},0)
   || _e ++ _es ->
      let (ws,dw) = keep_equiv_writes (equiv,nexts) in
      (ws,addD d ws dw)
   end
end

and write_acc (equiv,es) = match es with
|| {} -> ({},0)
|| e ++ es ->
    let (ws_e,dw_e) = case_disjunction (equiv, e)
    and (ws_es,dw_es) = write_acc (equiv, es) in
    (ws_e|ws_es, dw_e|dw_es)
end

and keep_equiv_writes (equiv,es) = match es with
  || {} -> ({},0)  (* No successors, ie no W po-after, special case *)
  || e ++ es ->
        match es with
        || {} -> case_disjunction(equiv,e) (* Singleton = base case *)
        || f ++ fs  ->
          let (ws_e,dw_e) = case_disjunction(equiv,e)
          and (ws_es,dw_es) = keep_equiv_writes(equiv,es) in
          (combine (equiv,ws_e,ws_es), dw_e|dw_es)
        end
end

(* Final call *)
let zyva equiv =
  mapset2rel
    (fun r -> let (ws,dw) = case_disjunction(equiv,r) in dw)
let Dmins = roots(DpoDorW)
let always-executed (to-PoD,equiv) =
  let DW = zyva equiv Dmins in
  to-PoD; DW
```

Figure 24: Cat computing always executed write effects, cont'd from Figure 23

```
(* Intrinsic dependencies *)
let intrinsic = (iico_data|iico_ctrl)+

(* Dependency through register *)
let generated-by-stxr = udr(same-instr;[range(lxsx)])
let rf-reg-no-stxr =
  let NW = ~generated-by-stxr in
  [NW];rf-reg
let dd = (rf-reg-no-stxr | intrinsic)+

(* The relation to-PoD is a dependency through register
   to a point of divergence*)
let to-PoD = [R]; dd; [PoD]

(** Data and Address dependencies *)
let data = [R]; dd; [DATA]; intrinsic; [W]
let addrW = [R]; dd; [NDATA]; intrinsic; [W]
let addrR =
 let dd-no-interv-PoD = dd \ (to-PoD;dd) in
 [R]; dd-no-interv-PoD; [NDATA]; intrinsic; [R]
let addr = addrW | addrR

(** Control dependency *)
(* Pre-control dependency *)
let pre-ctrl = to-PoD; po

(* Equivalent writes *)
let antecedent =
  let immediate-lws = singlestep([W];po-loc;[W]) in
  (intrinsic|rf-reg|lrs|immediate-lws)^-1

let pre-equiv-evts =
  let rdw-ext =
    let wwloc = ((W * W) & loc) \ id in
    ((rfe^-1;wwloc;rf)|(rf^-1;wwloc;rfe))
  in same-static\rdw-ext
show pre-equiv-evts \ id as pre-equiv-evts

let equivalent-writes = (W*W) & bisimulation(antecedent,pre-equiv-evts)
show equivalent-writes \ id as equivalent-writes

(* Control dependency *)
let ctrl = pre-ctrl \ (always-executed(to-PoD,equivalent-writes))
```

Figure 25: Cat file to compute address and data dependencies, as well as control dependencies "down-one-leg"

3 Dependencies "down-two-legs"

In this section we present a more traditional notion of dependencies which we call informally "down-two-legs" dependencies. This flavour of dependencies was **the one retained by Arm**. Essentially, "down-two-legs" dependencies guarantee ordering from the read which controls a branch to any write which appears syntactically after the branch, no matter where the control flow goes.

3.1 Concepts for defining dependencies "down-two-legs"

Branching effect Conditional branch instructions generate a Branching Effect, whether the branch is taken or not.

Note: Conditional instructions or Compare-and-Swap instructions do not generate a Branching effect.

Dependency through registers A dependency through registers from a first effect (E1) to a second effect (E2) exists within a PE if and only if any of the following applies:

- E1 is a register write W1 which has not been generated by a Store Exclusive, E2 is a register read R2 and R2 reads-from-register W1, or
- E1 and E2 have been generated by the same instruction and E1 is before E2 in the intrinsic order of that instruction, or
- there is a dependency through registers from E1 to a third effect E3, and there is a dependency through register from E3 to E2.

Address dependency An Address dependency from a Memory read R1 to a subsequent Memory effect RW2 exists if and only if all of the following applies:

- there is a Dependency through registers from R1 to a Register effect E3 generated by the same instruction as RW2, and E3 affects the address part of RW2, and
- one of the following applies:
  - RW2 is a Memory write effect W2, or
  - RW2 is a Memory read effect R2, and there is no Branching Effect B4 such that there is a Dependency through registers from R1 to B4 and from B4 to R2.

Data dependency A Data dependency from a memory read (R1) to a subsequent memory write (W2) exists if and only if there is a dependency through registers from R1 to a register effect E3 generated by W2, and E3 affects the data part of W2.

Control dependency A control dependency from a memory read (R1) to a subsequent memory effect (RW2) exists if and only if there is a dependency through register from R1 to a Branching effect B3, and B3 is program-order-before RW2.

3.2 Formalisation of "down-two-legs" dependencies in cat

In Figure 26, we give the cat code for computing address and data dependencies, as well as control dependencies "down-two-legs".

```
(* Intrinsic dependencies *)
let intrinsic = (iico_data|iico_ctrl)+

(* Dependency through register *)
let rf-reg-no-stxr =
  rf-reg \ ([W & range(lxsx)];rf-reg)

let dd =
  let r = (rf-reg-no-stxr | intrinsic)* in
  r;rf-reg-no-stxr;r

(* The relation to-BR is a dependency through register
   to a Branching effect *)
let to-BR = [R]; dd; [BR]

(** Data and Address dependencies *)
let data = ([R]; dd; [DATA]; intrinsic; [W]) \ same-instance
let addrW = [R]; dd; [NDATA]; intrinsic; [W]
let addrR =
 let dd-no-interv-BR = dd \ (to-BR;dd) in
 [R]; dd-no-interv-BR; [NDATA]; intrinsic; [R]
let addr = (addrW | addrR) \ same-instance

(** Control dependencies *)
let ctrl = to-BR; po
```

Figure 26: Cat file to compute address and data dependencies, as well as control dependencies "down-two-legs"

4 Pick dependencies

As discussed in the previous sections, Arm has investigated the weakening of its dependency definitions, in particular the notion of control dependency to dependencies "down-one-leg" as described in Section 2. In light of this investigation,

Arm has decided to keep its notion of control dependency strong, and retained the notion of control dependencies "down-two-legs" as described in Section 3.

In addition to control dependencies "down-two-legs", Arm also introduced a new notion of dependency, called "pick dependencies". Pick dependencies allow for very targetted ordering, and it is hoped that this will enable hardware optimisations.

4.1 Motivation

Originally, the behaviour of `MP+rel+CAS-addr`, shown in Figure 27 was Forbidden by the architecture.

```
AArch64 MP+rel+CAS-addr
{
z=1;
0:X1=x; 0:X3=y;
1:X1=x; 1:X3=y; 1:X8=z;
}
P0              | P1                ;
MOV W0, #1      | LDR W0, [X1]      ;
STR W0, [X3]    | MOV W5, W0        ;
STLR W0, [X1]   | CAS W0, W6, [X8]  ;
                | LDR W0, [X8]      ;
                | EOR X0, X0, X0    ;
                | ADD X3, X3, X0    ;
                | LDR W4, [X3]      ;
exists (1:X5=1 /\ 1:X4=0)
```

Figure 27: The litmus test `MP+rel+CAS-addr`

Arm chose to weaken the architecture to allow the behaviour. The reason for `MP+rel+CAS-addr` being Forbidden was as follows, referring to the events' labels given in Figure 28:

1. there is a Data dependency on P1 from the memory read of `LDR W0,[X1]` (event c) to the memory write of `CAS W0,W6,[X8]` (event e)

2. there is a read-from on P1 from the Memory write Effect of `CAS W0,W6,[X8]` (event e) to the Memory read Effect of `LDR W0,[X8]` (event f). In other words the Memory read Effect of `LDR W0,[X8]` (event f) is a Local read successor of the Memory write Effect of `CAS W0,W6,[X8]` (event e)

3. there is an Address dependency on P1 from the Memory read Effect of `LDR W0,[X8]` (event f) to the Memory read Effect of `LDR W4,[X3]` (event g).

Points 1 and 2 combined impose order between the Memory read Effect of `LDR W0,[X1]` (event c) to the Memory read Effect of `LDR W0,[X8]` (event f),

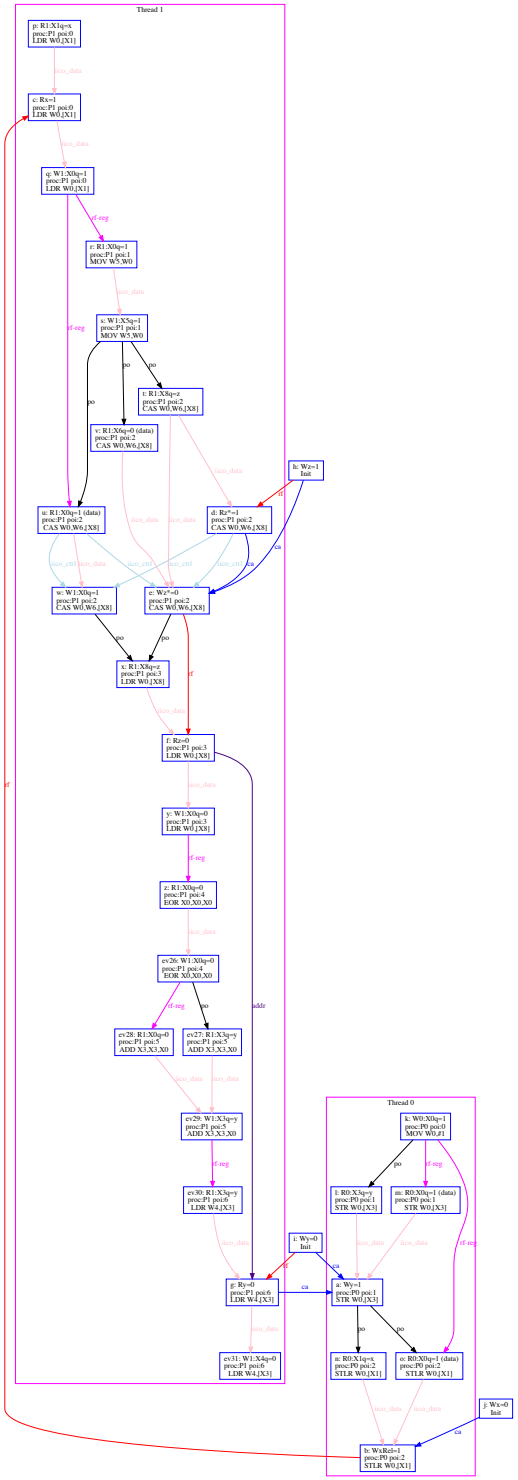

Figure 28: Execution of `MP+rel+CAS-addr`

because `data; rfi` is in `dob`. In the text of the Arm ARM [5] this corresponds to the clause:

> *RW2 is a Local read successor R2 of a write W3 and there is an Address dependency or a Data dependency from R1 to W3.*

A way forward to allow this behaviour would be to decide that there is no ordering altogether between the Memory read Effect of `LDR W0,[X1]` (event c) to the Memory write Effect of `CAS W0,W6,[X8]` (event e). However, this would lead to allowing `LB+rel+CAS`, given in Figure 29, which must be Forbidden.

```
AArch64 LB+rel+CAS
{
0:X1=x; 0:X3=y;
1:X1=x; 1:X3=y;
}
P0             | P1                 ;
MOV W9, #1     | MOV W4, #1         ;
LDR W0, [X3]   | LDR W5, [X1]       ;
STLR W9, [X1]  | AND W10, W5, #2    ;
               | CAS W10, W4, [X3]  ;
exists (1:X5=1 /\ 0:X0=1)
```

Figure 29: The test `LB+rel+CAS`

To allow `MP+rel+CAS-addr` as desired, and keep `LB+rel+CAS` forbidden, Arm created a new type of dependency (called "Pick dependency"), such that:

- Pick dependencies are not subjected to the clause above, hence will not impose order in the case of `MP+rel+CAS-addr`
- Pick dependencies provide order in the case of `LB+rel+CAS`

Interestingly, this notion of "pick dependency" appears useful in resolving the following. The proposal to strengthen Control dependencies to be "down-two-legs" initially included `CSEL` instructions. However, this is undesirable as this would prescribe ordering in the litmus test `MP+rel+CSEL` given in Figure 30 for example.

Therefore `CSEL` instructions should not provide similar ordering to branch instructions. However this highlights again the need for a new dependency notion, as the litmus test `S+rel+CSEL-data` given in Figure 31 must be ordered.

As in the `CAS` case, the nature of that new dependency notion is interesting. Indeed the test `MP+rel+CSEL-addr` given in Figure 32 should be allowed, which in turn entails that this new dependency cannot be a Data dependency, as otherwise the `LDR W1,[X2]` and the `LDR W9,[X10,X8,SXTW]` would be ordered.

Therefore Arm decided that `CAS` and `CSEL` create Pick dependencies. The changes to the text and the cat file, as well as a collection of interesting tests, follow in the next sections.

```
AArch64 MP+rel+CSEL
{
0:X10=x; 1:X10=x;
0:X2=y; 1:X2=y;
}
P0             | P1                  ;
MOV X9,#1      | LDR X1, [X2]        ;
STR X9,[X10]   | CMP X1, #1          ;
MOV X11,#1     | CSEL X3, X4, X5, EQ;
STLR X11,[X2]  | MOV X13,#2          ;
               | STR X13, [X10]      ;
exists(1:X1=1 /\ x=1)
```

Figure 30: The test `MP+rel+CSEL`

```
AArch64 S+rel+CSEL-data
{
0:X10=x; 1:X10=x;
0:X2=y; 1:X2=y;
1:X6=z;
}
P0             | P1                  ;
MOV W9,#1      | LDR W1, [X2]        ;
STR W9,[X10]   | CMP W1, #1          ;
MOV W11,#1     | CSEL W3, W4, W5, EQ;
STLR W11,[X2]  | STR W3, [X6]        ;
               | LDR W9, [X6]        ;
               | EOR X8, X9, X9      ;
               | ADD W8,W8,#2        ;
               | STR W8,[X10]        ;
exists(1:X1=1 /\ x=1)
```

Figure 31: The test `S+rel+CSEL-data`

```
AArch64 MP+rel+CSEL-addr
{
0:X10=x; 1:X10=x;
0:X2=y; 1:X2=y;
1:X6=z;
}
P0             | P1                   ;
MOV W9,#1      | LDR W1, [X2]         ;
STR W9,[X10]   | CMP W1, #1           ;
MOV W11,#1     | CSEL W3, W4, W5, EQ;
STLR W11,[X2]  | STR W3, [X6]         ;
               | LDR W7, [X6]         ;
               | EOR X8, X7,X7        ;
               | LDR W9,[X10,X8,SXTW]  ;
exists(1:X1=1 /\ 1:X9=0)
```

Figure 32: The test `MP+rel+CSEL-addr`

4.2 Concepts for defining "pick dependencies"

Dependency through registers and memory A Dependency through registers from a first Register or Memory effect E1 to a second Register or Memory effect E2 exists within a PE if and only if at least one of the following applies:

- E1 is a Write register effect which has not been generated by a Store Exclusive, E2 is a Read register effect and E2 Reads-from-register E1.
- E2 is a Local read successor of E1
- E1 and E2 have been generated by the same instruction and there is an Intrinsic data dependency from E1 to E2.
- There is a Dependency through registers from E1 to a third effect E3, and there is a Dependency through registers from E3 to E2.

Pick Basic dependency A Pick Basic dependency from a Read register or memory effect R1 to a Register or Memory effect E2 exists if and only if all of the following apply:

- there is a Dependency through registers and memory from R1 to a Register effect E3, or R1 and E3 are the same effect, and one of the following applies:
- there is an Intrinsic control dependency from the Register effect E3 to a Register effect E4, and there is a Dependency through registers and memory from E4 to E2, or
- there is an Intrinsic control dependency from the Register effect E3 to E2.

Pick Address dependency A Pick Address dependency from a Read register or memory effect R1 to a Read or write memory effect RW2 exists if and only all of the following apply:

- there is a Pick Basic dependency from R1 to a Register effect E3
- there is an Intrinsic data dependency from the Register effect E3 to RW2
- the Register effect E3 affects the address of the location accessed by RW2.

Pick Data dependency A Pick Data dependency from a Read register or memory effect R1 to a Write memory effect W2 exists if and only all of the following apply:

- there is a Pick Basic dependency from R1 to a Register effect E3
- there is an Intrinsic data dependency from the Register effect E3 to W2
  • the Register effect E3 affects the value written by W2.

Pick Control dependency A Pick Control dependency from a Read register or memory effect R1 to an Effect E2 exists if and only all of the following apply:

- there is a Pick Basic dependency from R1 to a Branching effect BR3
- the Branching effect BR3 is in program order before E2.

Pick dependency A Pick dependency from a Read register or memory effect R1 to an effect E2 exists if and only if one of the following applies:

- there is a Pick Basic dependency from R1 to E2, or
- there is a Pick Address dependency from R1 to E2, or
- there is a Pick Data dependency from R1 to E2, or
- there is a Pick Control dependency from R1 to E2.

4.3 Formalisation of "pick" dependencies in cat

In Figure 33 we give the cat code to compute address, data, control dependencies "down-two-legs" as well as "pick" dependencies.

```
(* Dependency through registers and memory *)
let rec dtrm =
  rf-reg \ ([W & range(lxsx)];rf-reg)
  | rfi
  | iico_data
  | dtrm; dtrm

let Reg=~M | ~BR
(** Data, Address and Control dependencies *)
let ADDR=NDATA
let basic-dep =
   [R|Rreg]; dtrm?
let data = basic-dep; [DATA]; iico_data; [W]
let addr = basic-dep; [ADDR]; iico_data; [M]
let ctrl = basic-dep; [BR]; po

(** Pick dependencies *)
let pick-basic-dep =
   [R|Rreg]; dtrm?; [Reg]; iico_ctrl; [Reg]; dtrm?
let pick-addr-dep =
   pick-basic-dep; [ADDR]; iico_data; [M]
let pick-data-dep =
   pick-basic-dep; [DATA]; iico_data; [W]
let pick-ctrl-dep =
   pick-basic-dep; [BR]; po
let pick-dep =
(
   pick-basic-dep |
   pick-addr-dep  |
   pick-data-dep  |
   pick-ctrl-dep
)
```

Figure 33: Cat file to compute address, data, control dependencies "down-two-legs", as well as "pick" dependencies

5 Arm dependencies overall

5.1 Concepts for defining Arm dependencies overall

Pick-ordered-before A Read register or memory effect (R1) is pick-ordered-before a Read or a Write memory effect (RW2) from the same observer if and only if R1 appears in program order before RW2 and any of the following cases apply:

- RW2 is a Write memory effect W2 and there is a Pick-dependency from R1 to W2.
- RW2 is a Read memory effect R2 generated by an instruction appearing in program order after an instruction that generates a Context synchronization event E3, and there is a Pick-control dependency from R1 to E3.
- RW2 is a Memory effect generated by an instruction appearing in program order after an instruction that generates a Context synchronization event E3, there is a Pick Address dependency from R1 to an effect E4 and E4 is in program order before E3.
- RW2 is a Write memory effect W2, there is a Pick-address-dependency from R1 to a Read or Write memory effect RW3 and W2 is program-order-after RW3.

Locally-ordered-before Local-write-successor, dependencies, Load/Store-Exclusive, atomic and barrier instructions can be composed within an observer to create externally-visible order. A Read or a Write Memory effect (RW1) is locally-ordered- before a Read or a Write memory effect (RW2) from the same observer if and only if any of the following cases apply:

- RW1 is a Write memory effect (W1) and RW2 is a Write memory effect (W2) that is either a Local write successor of RW1, or which belongs to the same single-copy-atomic class as a Local write successor of RW1
- RW1 is dependency-ordered-before RW2
- RW1 is pick-ordered-before RW2
- RW1 is atomic-ordered-before RW2
- RW1 is barrier-ordered-before RW2
- RW1 is tag-ordered-before RW2
- RW1 is locally-ordered-before a Read or a Write memory effect RW3 that is locally-ordered-before RW2

Pick-locally-ordered-before A Read or a Write Memory effect (RW1) is Pick-locally-ordered-before a Read or a Write memory effect (RW2) from the same observer if and only if all of the following apply:

- there is Pick-basic-dependency from RW1 to an effect E3 • RW2 is a Write memory effect W2
- E3 is Locally-ordered-before W2

5.2 Formalisation of Arm dependencies in cat

In Figures 34 and 35 we give the cat code to compute Arm dependencies overall.

```
(* Intrinsic order *)
let intrinsic = (iico_data|iico_ctrl)+

(* Coherence-after *)
let ca = fr | co

(* Local read successor *)
let lrs = [W]; (po-loc \ (po-loc;[W];po-loc)) ; [R]

(* Local write successor *)
let lws = po-loc; [W]

(* Observed-by *)
let obs = rfe | fre | coe

(* Read-modify-write *)
let rmw = lxsx | amo

(* Dependency-ordered-before *)
let dob = addr | data | ctrl; [W]
        | (ctrl | (addr; po)); [ISB]; po; [M]
        | addr; po; [W]
        | (addr | data); lrs

(* Pick-ordered-before *)
let pob = pick-dep; [W]
        | (pick-ctrl-dep | pick-addr-dep; po); [ISB]; po; [M]
        | pick-addr-dep; [M]; po; [W]

(* Atomic-ordered-before *)
let aob = rmw
        | rmw; lrs; [A | Q]

(* Barrier-ordered-before *)
let bob = po; [dmb.full]; po
        | po; ([A];amo;[L]); po
        | [L]; po; [A]
        | [R\NoRet]; po; [dmb.ld]; po
        | [A | Q]; po
        | [W]; po; [dmb.st]; po; [W]
        | po; [L]

(* Tag-ordered-before *)
let tob = [R & T]; iico_data; [PoD]; iico_ctrl; [M \ T]
```

Figure 34: Overall Arm cat file, cont'd in Figure 35

```
(* Locally-ordered-before *)
let rec lob = lws; si
            | dob
            | pob
            | aob
            | bob
            | tob
            | lob; lob
let pick-lob = pick-basic-dep; lob; [W]

(* Ordered-before *)
let rec ob = obs; si
        | lob
        | pick-lob
        | ob; ob

(* Tag-location-ordered *)
let tlo = [R & T]; tob; loc; tob^-1; [R & T]

(* Internal visibility requirement *)
let po-loc =
  let Exp = M\domain(tob) in
  (po-loc & Exp*Exp) | (po & tlo)
acyclic po-loc | ca | rf as internal

(* External visibility requirement *)
irreflexive ob as external

(* Atomic: Basic LDXR/STXR constraint to forbid intervening writes. *)
empty rmw & (fre; coe) as atomic
```

Figure 35: Overall Arm cat file, cont'd from Figure 34

References

[1] [RFC] LKMM: Add volatile_if(). 2021.

[2] Jade Alglave, Will Deacon, Richard Grisenthwaite, and Luc Maranget. The Ballad of Dependencies - cat files, kinds and litmus tests. 2021.

[3] Jade Alglave and Luc Maranget. The herd+diy toolsuite. Since 2010.

[4] Will Deacon, Peter Zijlstra, Paul McKenney, and Jade Alglave. The never-ending saga of control dependencies. 2021.

[5] Arm Ltd. Armv8, for armv8-a architecture profile. In *Arm Architecture Reference Manual*, volume ARM DDI 0487G.b, 2021.