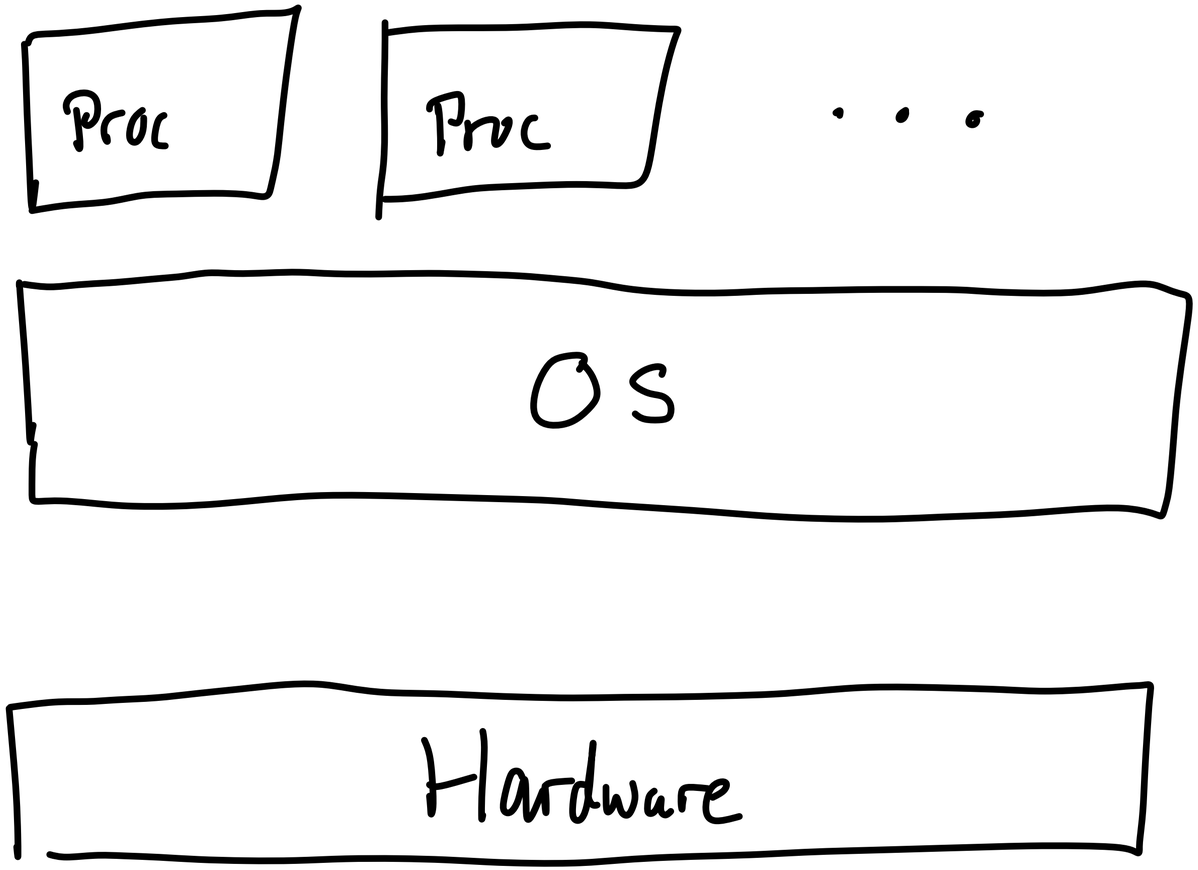
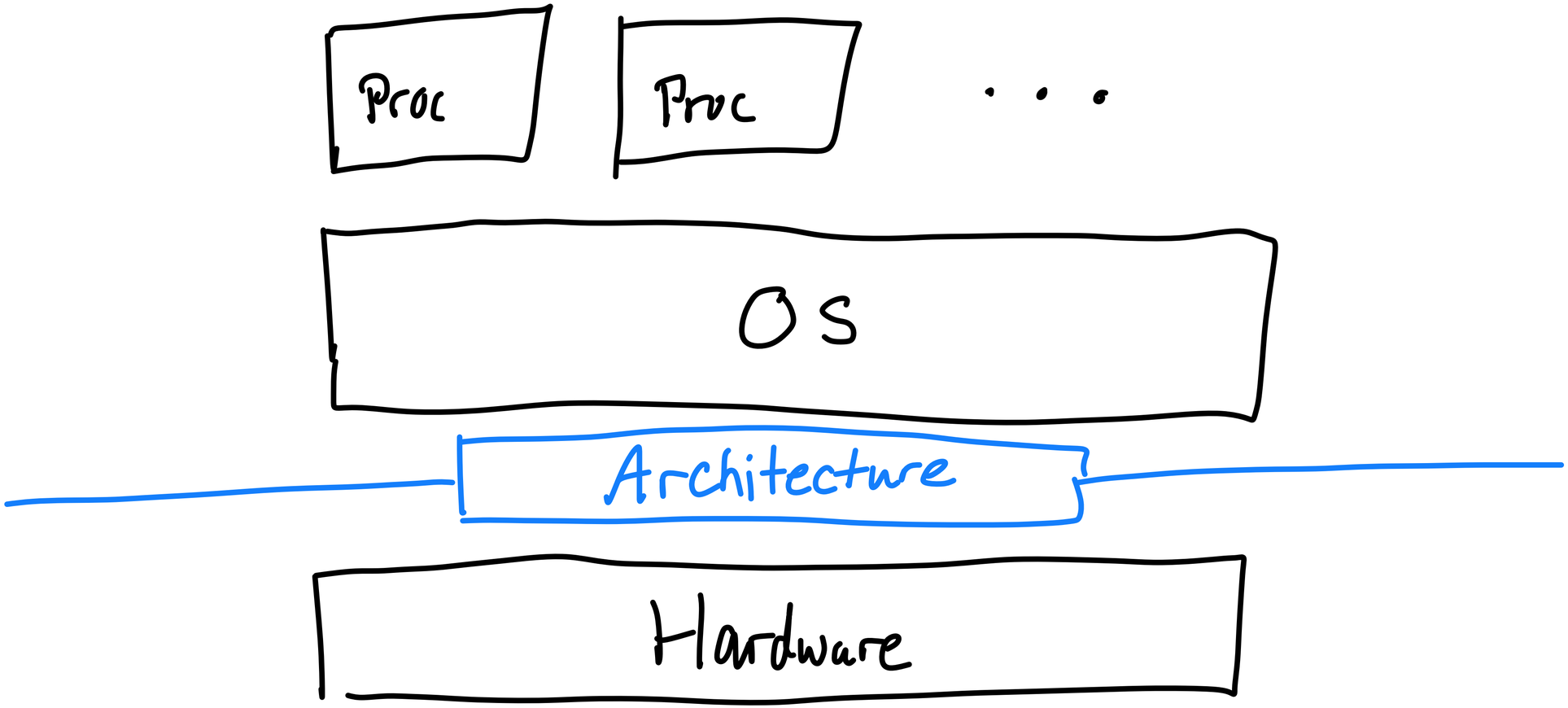
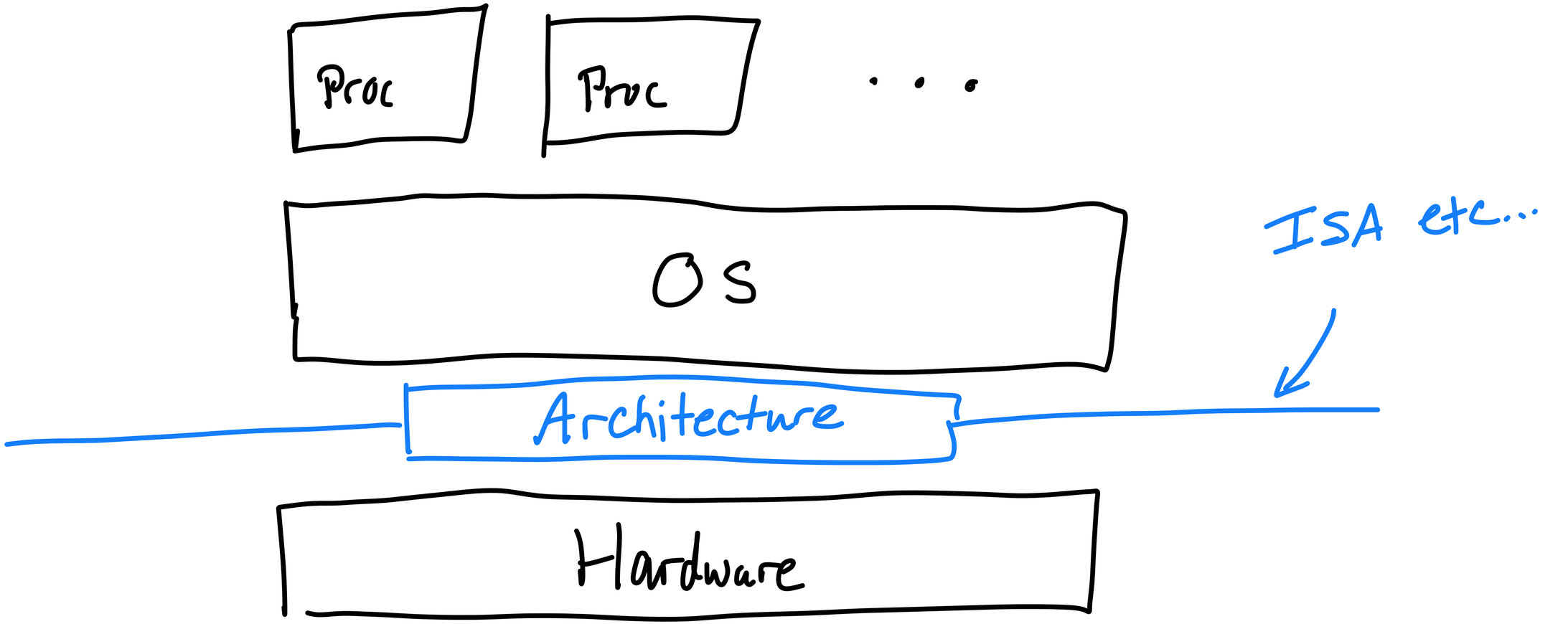
Arm System Semantics

Ben Simner


Proc
Proc
. . .
OS
Hardware


Proc
Proc
. . .
OS
Architecture
Hardware


Proc
Proc
. . .
OS
ISA etc...
Architecture
Hardware

Roadmap

- Recap on ISA
- Memory modelling
- Systems Architecture
- Modelling Process

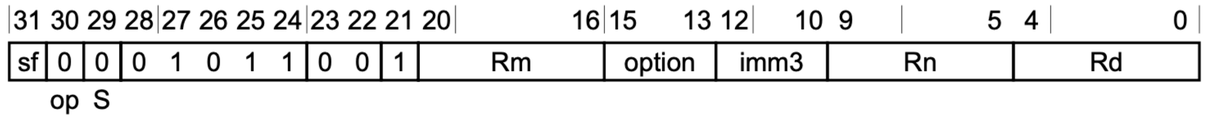
Recap: Arm ISA

Arm® Architecture Reference Manual

Armv8, for Armv8-A architecture profile

C6.2.4 ADD (extended register)

Add (extended register) adds a register value and a sign or zero-extended register value, followed by an op
shift amount, and writes the result to the destination register. The argument that is extended from the <Rm
can be a byte, halfword, word, or doubleword.

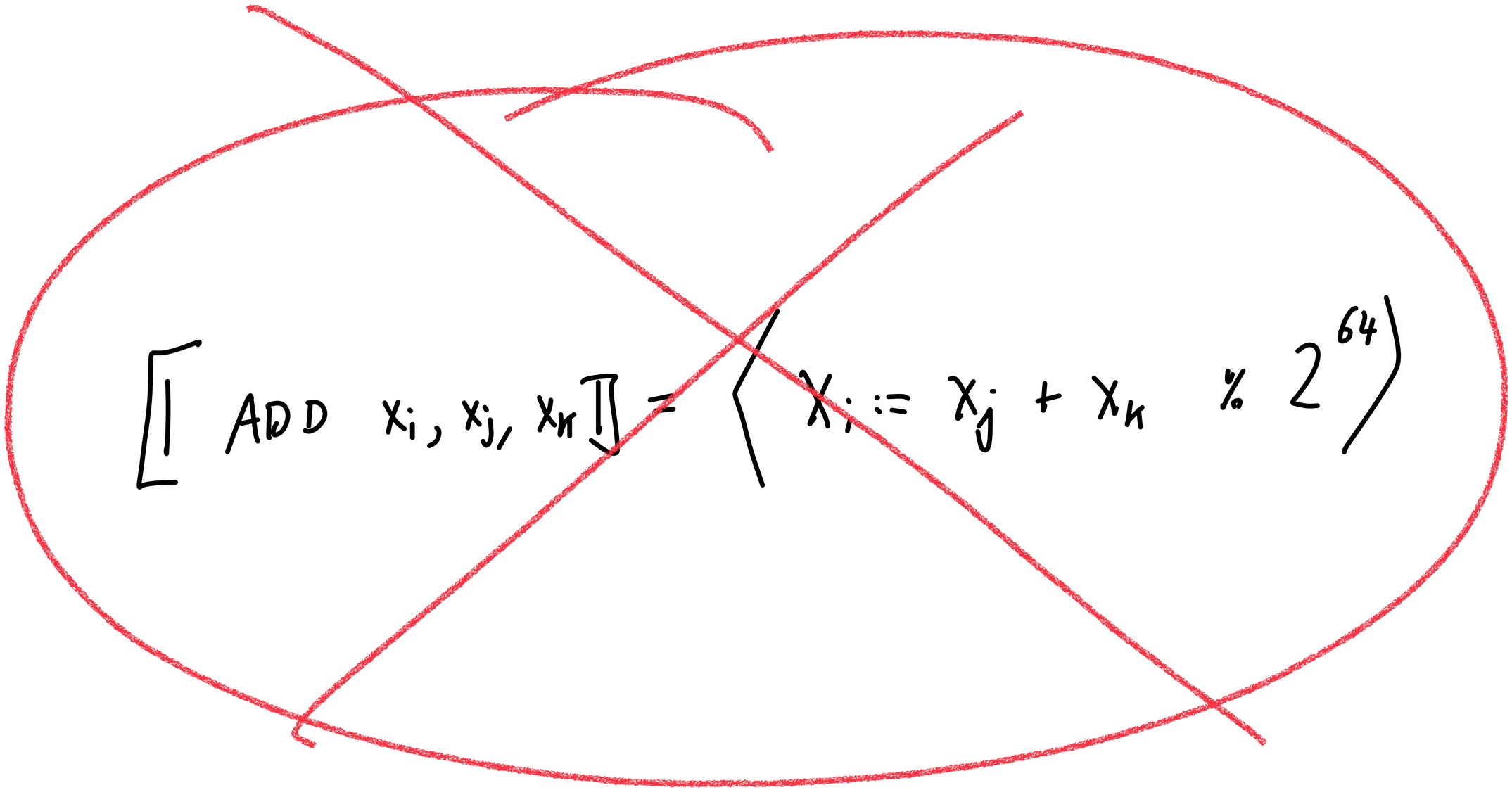


32-bit variant

Applies when `sf == 0`.

```
ADD <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

$$[\![\ \mathrm{ADD}\ x_i, x_j, x_k\ ]\!] = \langle\ x_i := x_j + x_k \ \%\ 2^{64}\ \rangle$$


⟦ ADD x_i, x_j, x_k ⟧ = ⟨ $x_i := x_j + x_k \ \% \ 2^{64}$ ⟩

Decoding 1

```
val decode64 : bits(32) -> unit
   effect {configuration, escape, undef, wreg, rreg, rmem,
       wmem}

function clause decode64
  ((_:bits(1) @ 0b0010001 @ _:bits(24) as op_code) if SEE
      <1066) = {
    SEE = 1066;
    Rd : bits(5) = op_code[4 .. 0];
    Rn : bits(5) = op_code[9 .. 5];
    imm12 : bits(12) = op_code[21 .. 10];
    shift : bits(2) = op_code[23 .. 22];
    S : bits(1) = [op_code[29]];
    op : bits(1) = [op_code[30]];
    sf : bits(1) = [op_code[31]];
    addsub_immediate_decode(Rd, Rn, imm12, shift, S, op, sf)
}
```

Execution

```
function addsub_immediate
      (d, datasize, imm, n, setflags, sub_op) = {
    result : bits('datasize) = undefined : bits('datasize);
    let operand1 : bits('datasize) = if n == 31 then SP()
        else X(n);
    operand2 : bits('datasize) = imm;
    nzcv : bits(4) = undefined : bits(4);
    carry_in : bits(1) = undefined : bits(1);
    if sub_op then {
        operand2 = ~(operand2);
        carry_in = 0b1
    } else {
        carry_in = 0b0
    };
    (result, nzcv) = AddWithCarry(operand1, operand2,
        carry_in);
    if setflags then {
        (PSTATE.N @ PSTATE.Z @ PSTATE.C @ PSTATE.V) = nzcv
    };
    if d == 31 & ~(setflags) then { SP() = result }
    else { X(d) = result }
}
```

Execution auxiliary functions

```
val AddWithCarry : forall ('N : Int), ('N >= 0 & 'N >= 0).
  (bits('N), bits('N), bits(1)) -> (bits('N), bits(4))

function AddWithCarry (x, y, carry_in) = {
  let 'unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
  let 'signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
  let result : bits('N) = __GetSlice_int('N, unsigned_sum,
      0);
  let n : bits(1) = [result['N - 1]];
  let z : bits(1) = if IsZero(result) then 0b1 else 0b0;
  let c : bits(1) = if UInt(result)==unsigned_sum then 0b0
      else 0b1;
  let v : bits(1) = if SInt(result)==signed_sum then 0b0
      else 0b1;
  return((result, ((n @ z) @ c) @ v))
}
```

Decoding 2

```
val addsub_immediate_decode :
   (bits(5), bits(5), bits(12), bits(2), bits(1), bits(1), bits(1))
    -> unit
    effect {escape, rreg, undef, wreg}

function addsub_immediate_decode(Rd, Rn, imm12, shift, S, op, sf) =
     {
    __unconditional = true;
    let 'd = UInt(Rd);  let 'n = UInt(Rn);
    let 'datasize = if sf == 0b1 then 64 else 32;
    let sub_op = op == 0b1;  let setflags = S == 0b1;
    imm : bits('datasize) = undefined : bits('datasize);
    match shift {
      0b00 => { imm = ZeroExtend(imm12, datasize) },
      0b01 => { imm = ZeroExtend(imm12 @ Zeros(12), datasize) },
      0b10 => { throw(Error_See("ADDG, SUBG")) },
      0b11 => { ReservedValue() }
    };
    __PostDecode();
    addsub_immediate(d, datasize, imm, n, setflags, sub_op)
}
```

Auxiliary register-access functions

```
function aset_SP(value) = {
  assert('width == 32 | 'width == 64);
  if PSTATE.SP == 0b0 then {
      SP_EL0 = ZeroExtend(value)
  } else {
      match PSTATE.EL {
        el if el == EL0 => SP_EL0 = ZeroExtend(value),
        el if el == EL1 => SP_EL1 = ZeroExtend(value),
        el if el == EL2 => SP_EL2 = ZeroExtend(value),
        el if el == EL3 => SP_EL3 = ZeroExtend(value)
      }
  }
}
val aget_X : forall 'width 'n, 0 <= 'n <= 31 & 'width in {8, 16,
    32, 64}).
  (implicit('width), int('n)) -> bits('width) effect {rreg}

function aget_X(width, n) =
  if n != 31 then slice(_R[n], 0, width) else Zeros(width)
```

$$\llbracket \text{ADD } x_i, x_j, x_N \rrbracket =$$

Decoding 1

```
val decode64 : bits(32) -> unit
  effect {configuration, escape, undef, wreg, rreg, rmem,
          wmem}

function clause decode64
  (( _:bits(1) @ 0b00010001 @ _:bits(24) as op_code) if SEE
    -(1066) = {
    SEE = 1065;
    Rd : bits(5) = op_code[4 .. 0];
    Rn : bits(5) = op_code[9 .. 5];
    imm12 : bits(12) = op_code[21 .. 10];
    shift : bits(2) = op_code[23 .. 22];
    S : bits(1) = [op_code[29]];
    op : bits(1) = [op_code[30]];
    sf : bits(1) = [op_code[31]];
    aobhuk.immediate.decode(Rd, Rn, imm12, shift, S, op, sf)
  }
```

Decoding 2

```
val aobhuk_immediate_decode :
  (bits(5), bits(5), bits(12), bits(2), bits(1), bits(1), bits(1))
  -> unit
  effect {escape, rreg, undef, wreg}

function aobhuk_immediate_decode(Rd, Rn, imm12, shift, S, op, sf) =
  {
    __unconditional = true;
    let 'd = U(Rd); let 'n = U(Rn);
    let 'datasize = if sf == 0b1 then 64 else 32;
    let sub_op = op == 0b1; let setflags = S == 0b1;
    let 'imm : bits('datasize) = undefined : bits('datasize);
    match shift {
      0b00 => { imm = ZeroExtend(imm12, datasize) },
      0b01 => { imm = ZeroExtend(imm12 @ Zeros(12), datasize) },
      0b10 => { throw(Error_See("AOBHL SUBK")) },
      0b11 => { ReservedValue() }
    };
    __PostDecode();
    aobhuk_immediate(d, datasize, imm, n, setflags, sub_op)
  }
```

Execution

```
function aobhuk_immediate
  (d, datasize, imm, n, setflags, sub_op) = {
  result : bits('datasize) = undefined : bits('datasize);
  let operand1 : bits('datasize) = if n == 31 then SP()
                                   else X(n);
  operand2 : bits('datasize) = imm;
  nzcv : bits(4) = undefined : bits(4);
  carry_in : bits(1) = undefined : bits(1);
  if sub_op then {
    operand2 = ~(operand2);
    carry_in = 0b1
  } else {
    carry_in = 0b0
  };
  (result, nzcv) = AddWithCarry(operand1, operand2,
                                carry_in);
  if setflags then {
    (PSTATE.N @ PSTATE.Z @ PSTATE.C @ PSTATE.V) = nzcv
  };
  if d == 31 & ~(setflags) then { SP() = result }
  else { X(d) = result }
}
```

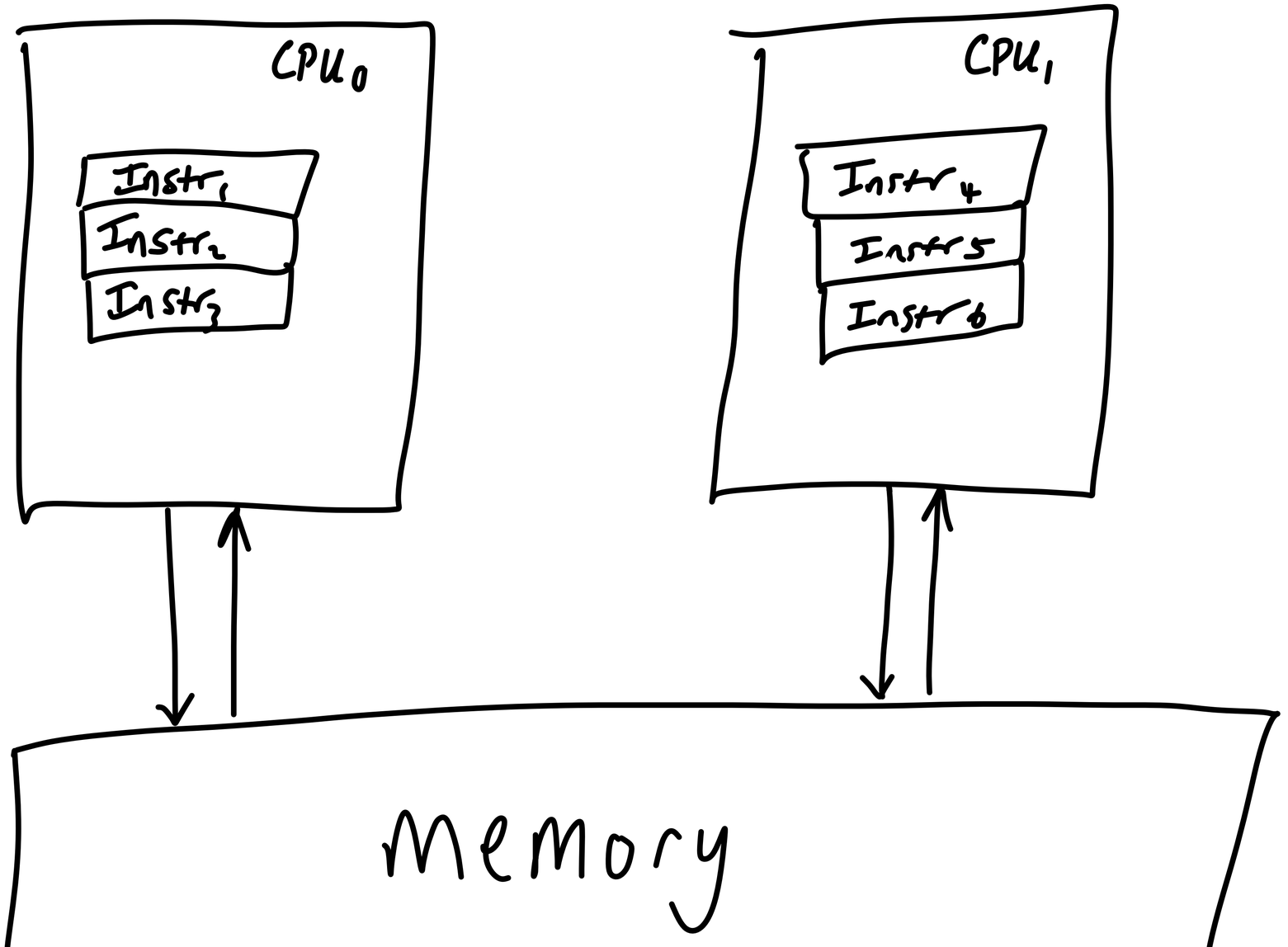
Auxiliary register-access functions

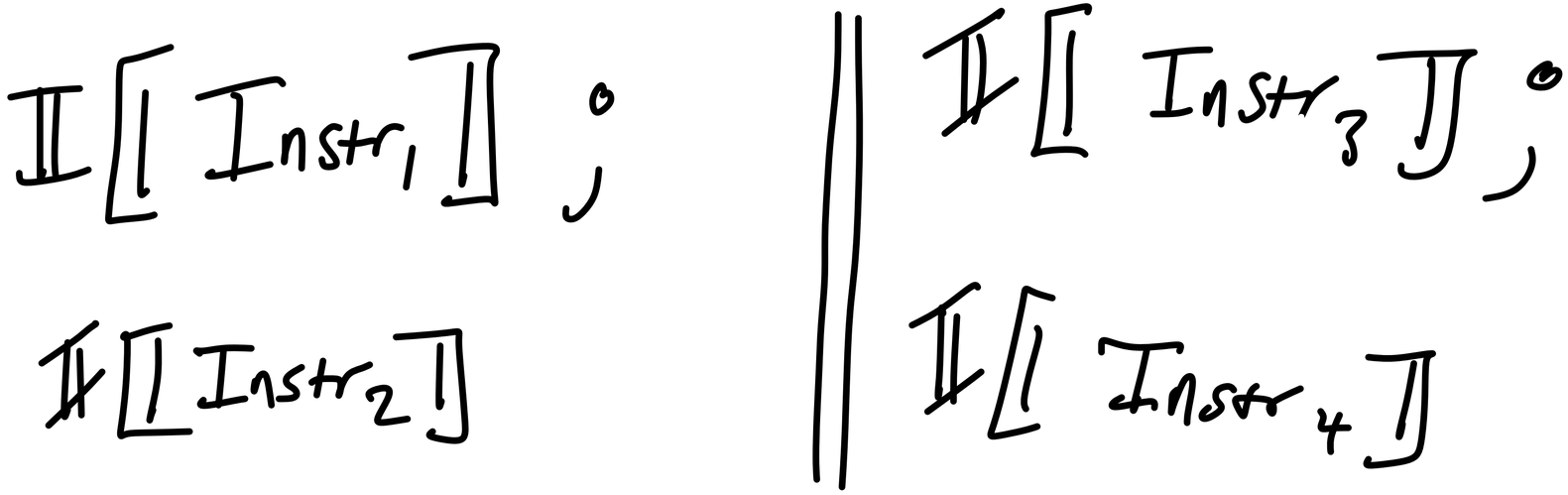
```
function aset_SP(value) = {
  assert('width == 32 | 'width == 64);
  if PSTATE.SP == 0b0 then {
      SP_EL0 = ZeroExtend(value)
  } else {
      match PSTATE.EL {
        0b00 if n1 == EL0 => SP_EL0 = ZeroExtend(value),
        0b01 if n1 == EL1 => SP_EL1 = ZeroExtend(value),
        0b10 if n1 == EL2 => SP_EL2 = ZeroExtend(value),
        0b11 if n1 == EL3 => SP_EL3 = ZeroExtend(value)
      }
  }
}
val sget_X : forall 'width 'n, 0 <= 'n <= 31 & 'width in {8, 16,
    32, 64}.
  (implicit('width), int('n)) -> bits('width) effect {rreg}

function sget_X(width, n) =
  if n != 31 then slice(R(n), 0, width) else Zeros(width)
```

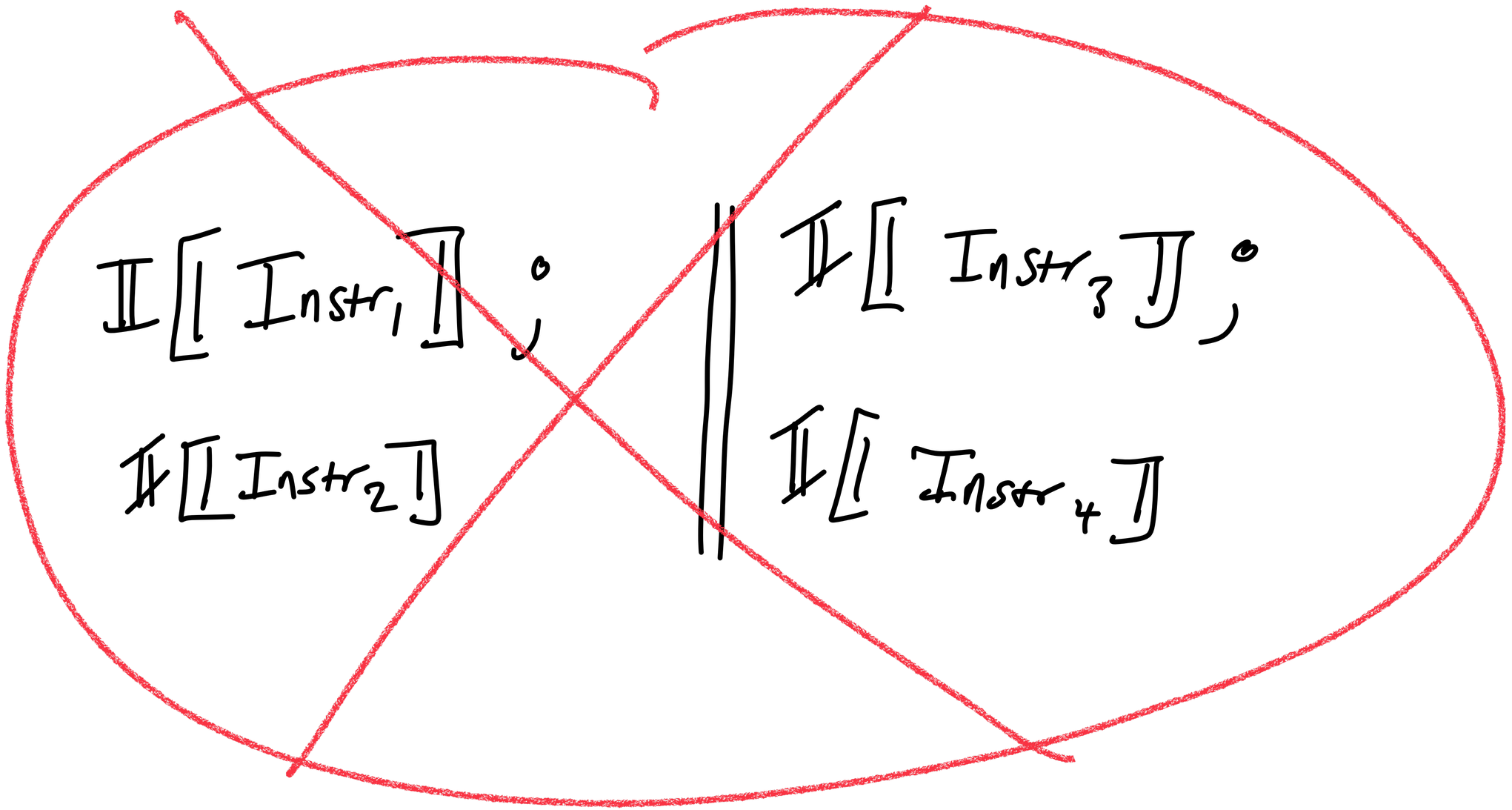
Execution auxiliary functions

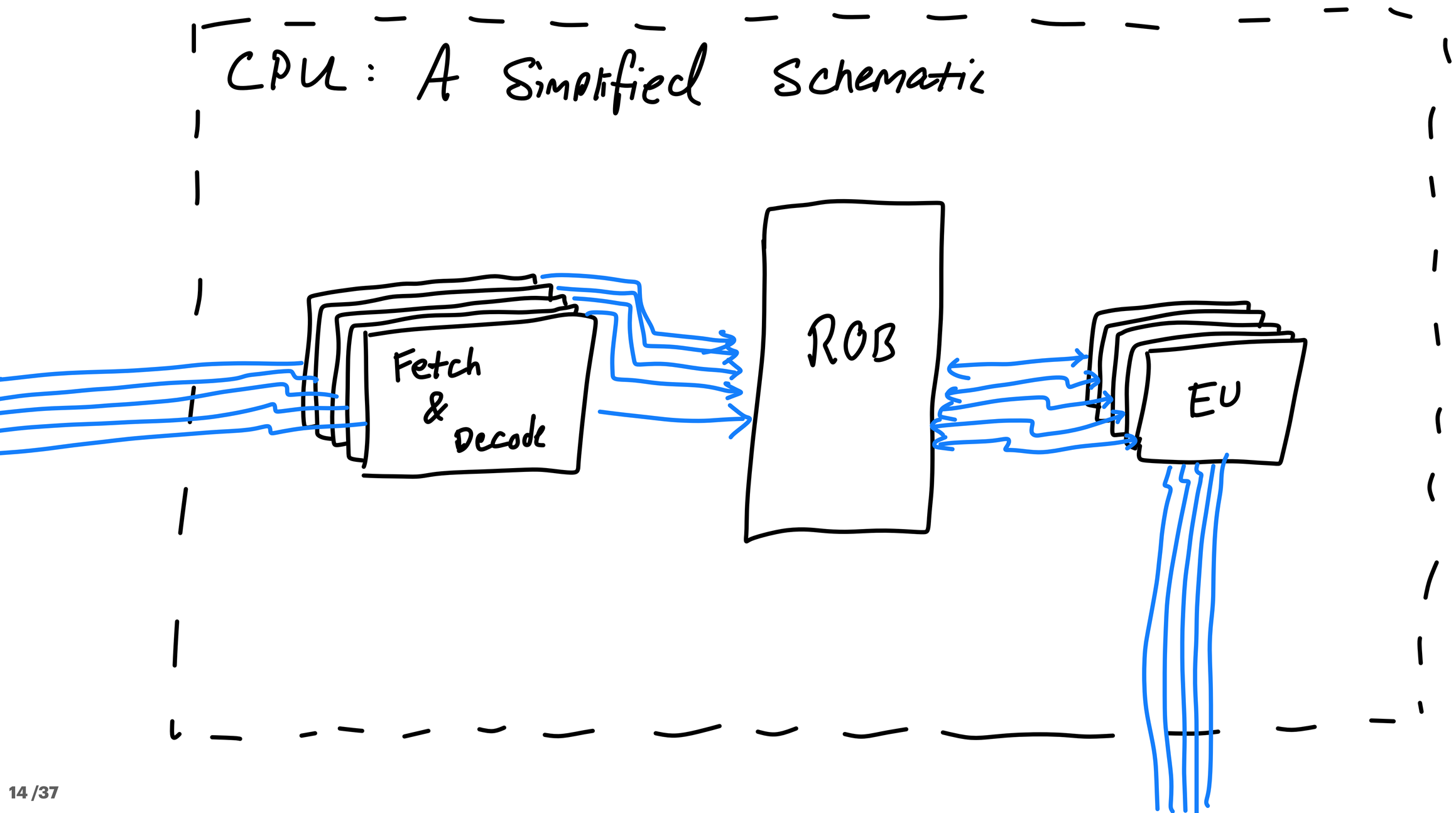
```
val AddWithCarry : forall 'N : Int, 'N >= 0 & 'N <= 0.
  (bits('N), bits('N), bits(1)) -> (bits('N), bits(4))

function AddWithCarry (x, y, carry_in) = {
  let 'unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
  let 'signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
  let result : bits('N) = __GetSlice_int('N, unsigned_sum,
                          0);
  let n : bits(1) = [result['N - 1]];
  let z : bits(1) = if IsZero(result) then 0b1 else 0b0;
  let c : bits(1) = if UInt(result)==unsigned_sum then 0b0
                    else 0b1;
  let v : bits(1) = if SInt(result)==signed_sum then 0b0
                    else 0b1;
  return((result, (n @ z) @ c) @ v))
}
```

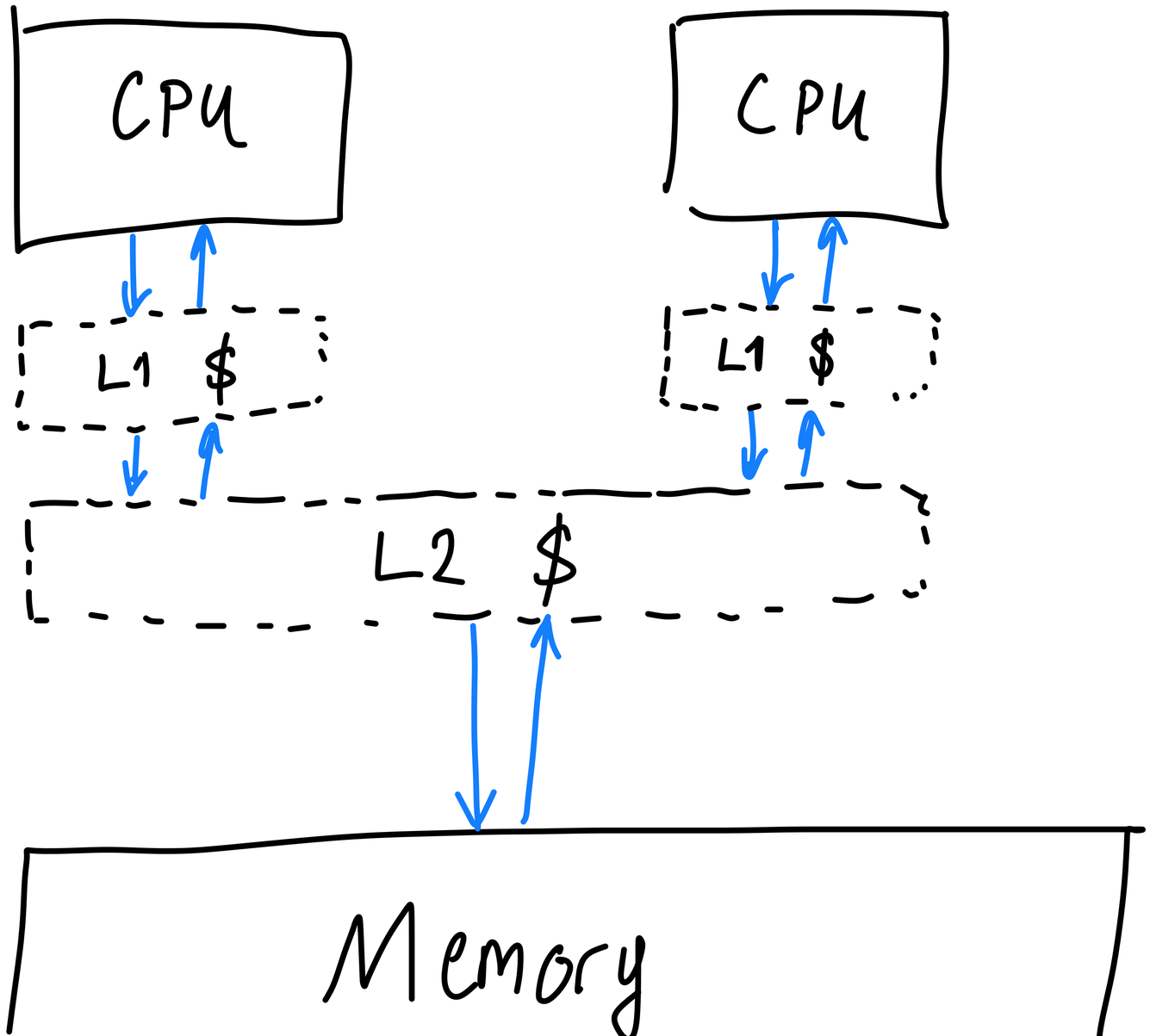
Multicore


CPU0
Instr1
Instr2
Instr3
CPU1
Instr4
Instr5
Instr6
Memory


Instr1 ;
Instr2
||
Instr3 ;
Instr4


⟦ Instr1 ⟧ ;
⟦ Instr2 ⟧
⟦ Instr3 ⟧ ;
⟦ Instr4 ⟧


CPU: A Simplified Schematic
Fetch & Decode
ROB
EU


CPU
CPU
L1 $
L1 $
L2 $
Memory

Not invisible to Software

↳ Not just side-channels !

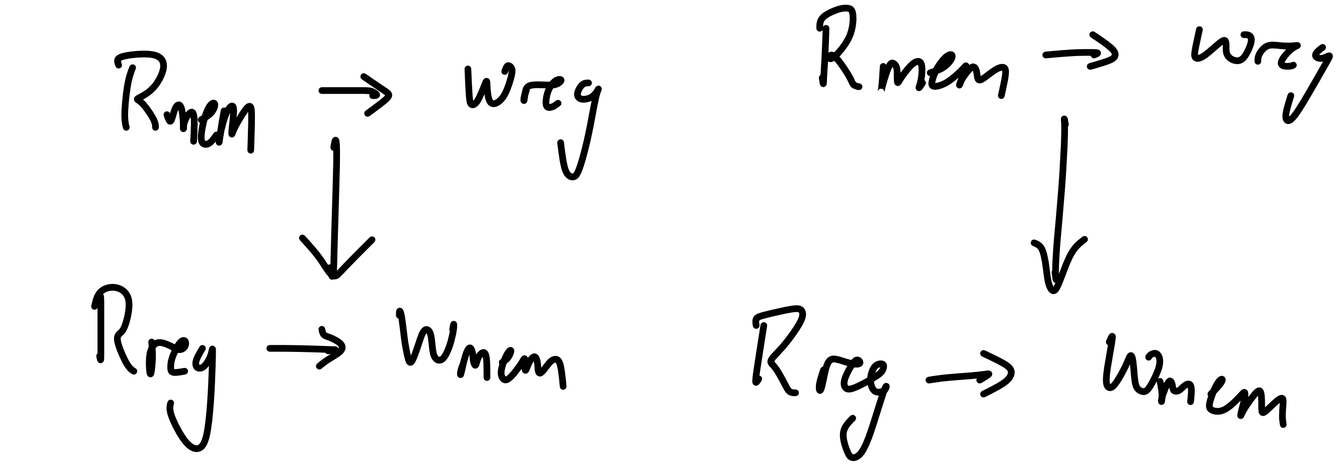
How to give Semantics?

Insight #1: Instructions are effectful programs

$$[\![\texttt{ADD}\ x_i, x_j, x_k]\!] = \mathsf{Rreg}\ j, \lambda v_1.\ \mathsf{Rreg}\ k, \lambda v_2.\ \mathsf{Wreg}\ i, v_1 \oplus v_2.$$

Insight #2: Executions as graphs of events

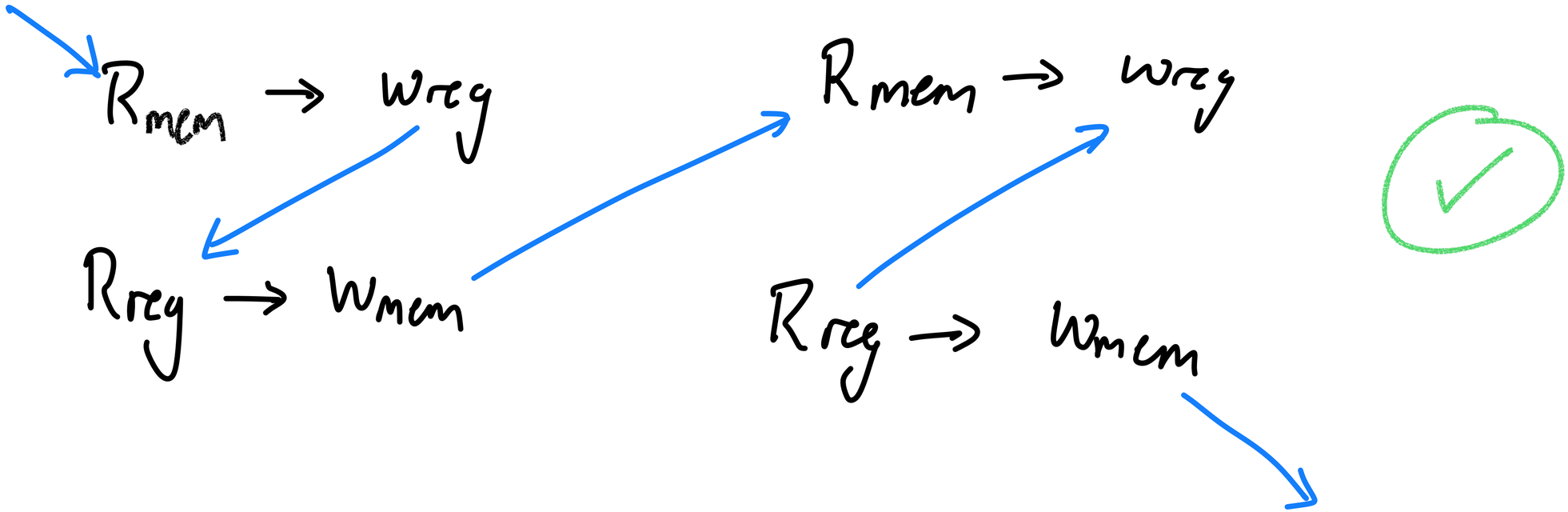
$$\mathcal{G}[\![\, I_1; I_2 \parallel I_3; I_4 \,]\!] =$$



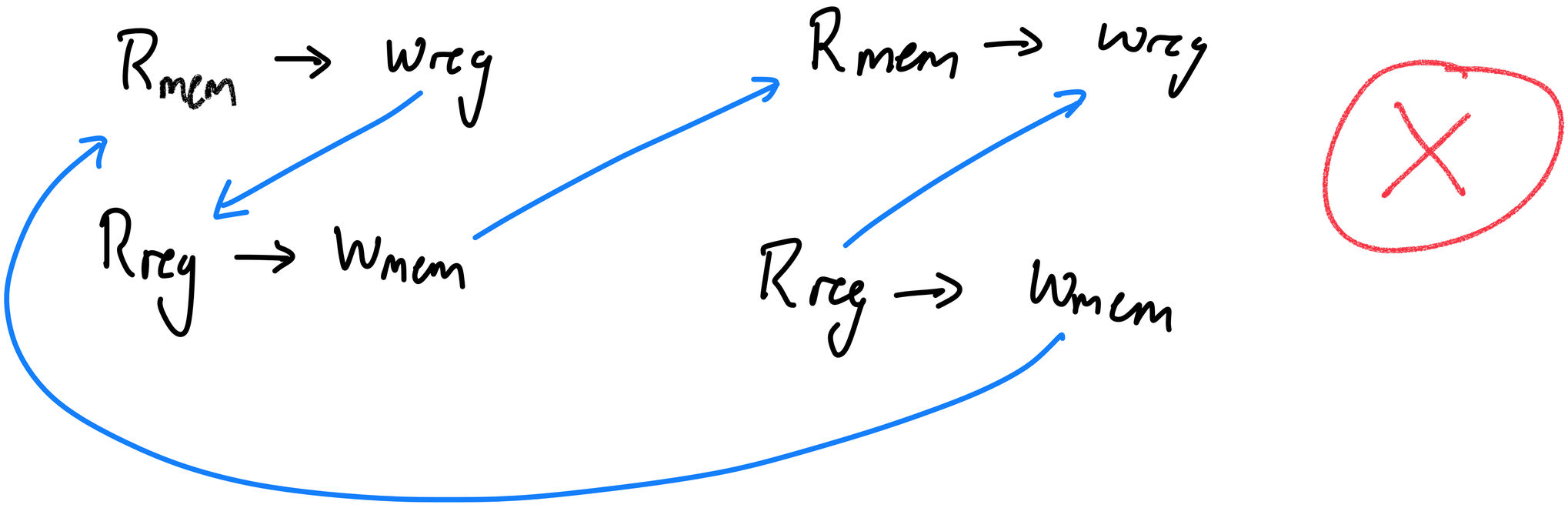
$$\mathcal{T}[\![\, i_1 ; i_2 \,]\!] = \mathcal{I}[\![i_1]\!] \fatsemi \mathcal{I}[\![i_2]\!]$$

$$\mathcal{G}[\![\, P_1 \parallel P_2 \,]\!] = \mathcal{T}[\![P_1]\!] \bowtie \mathcal{T}[\![P_2]\!]$$

Insight #3: Only some graphs are valid



Insight #3: Only some graphs are valid



Declarative Models

↳ a.k.a. "Axiomatic" models

$$\mathcal{V}(X) = \text{acyclic}(X, \hat{r})$$

$$[\![P]\!] = \{ X \mid X \in G[\![P]\!] \wedge \mathcal{V}(X) \}$$

aarch64.cat

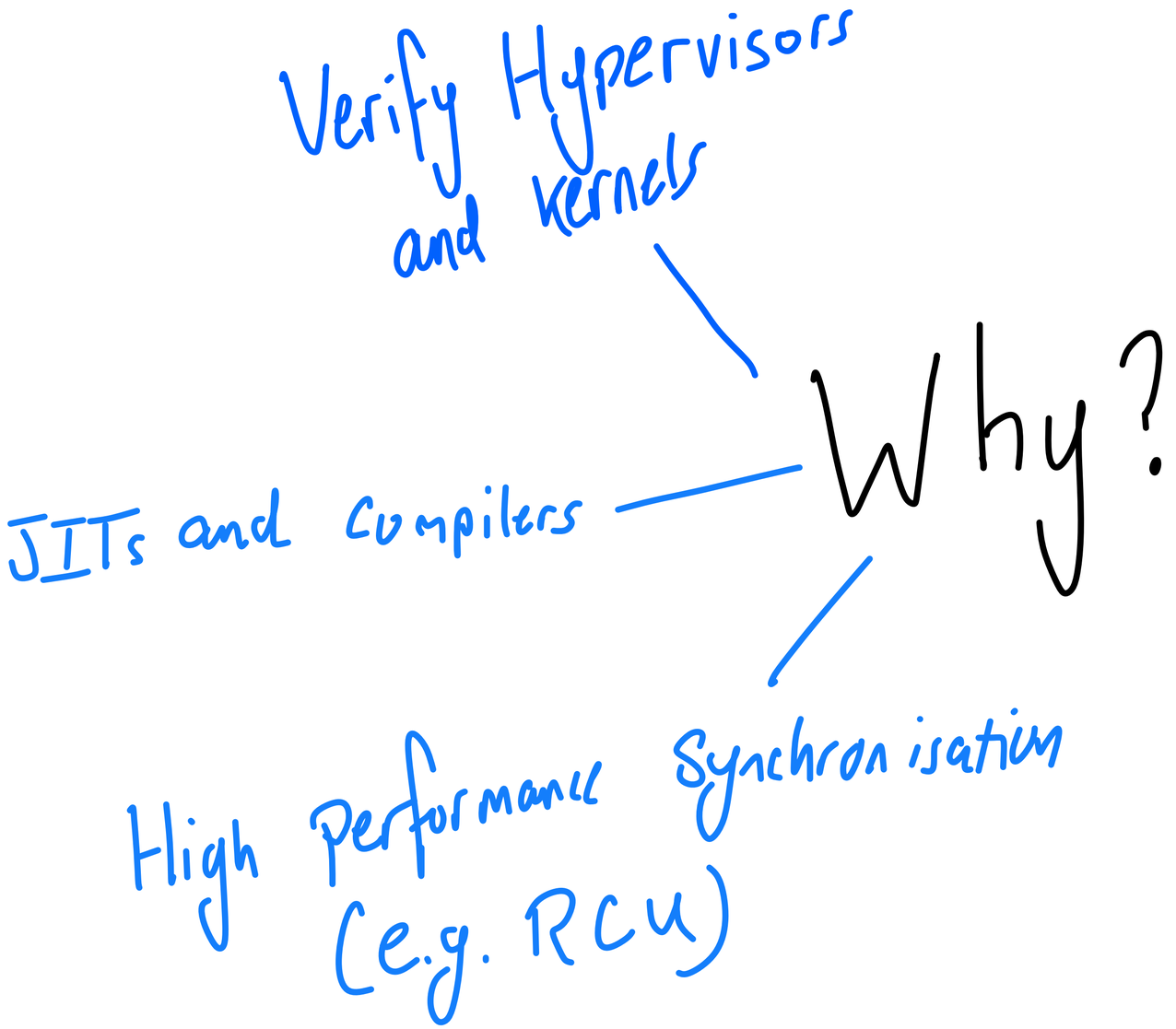
```
(* Coherence-after *)
let ca = fr | co

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

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

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

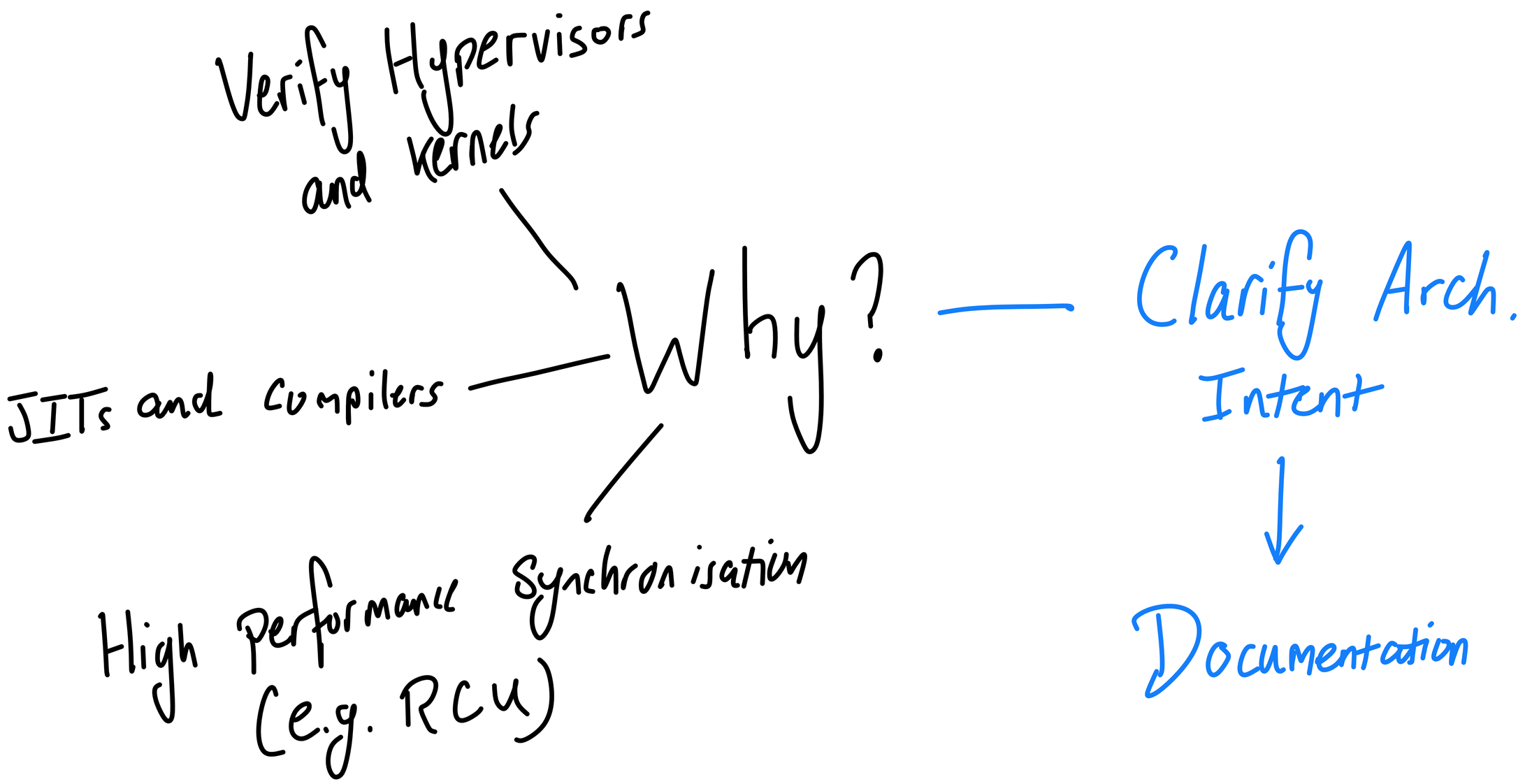
```
(* Barrier-ordered-before *)
let bob = po; [dmb.full]; po
    | [L]; po; [A]
    | [R]; po; [dmb.ld]; po
    | [A | Q]; po
    | [W]; po; [dmb.st]; po; [W]
    | po; [L]
    | po; [L]; coi

(* Ordered-before *)
let rec ob = obs
    | dob
    | aob
    | bob
    | ob; ob

(* Internal visibility requirement *)
acyclic po-loc | ca | rf as internal

(* External visibility requirement *)
irreflexive ob as external
```

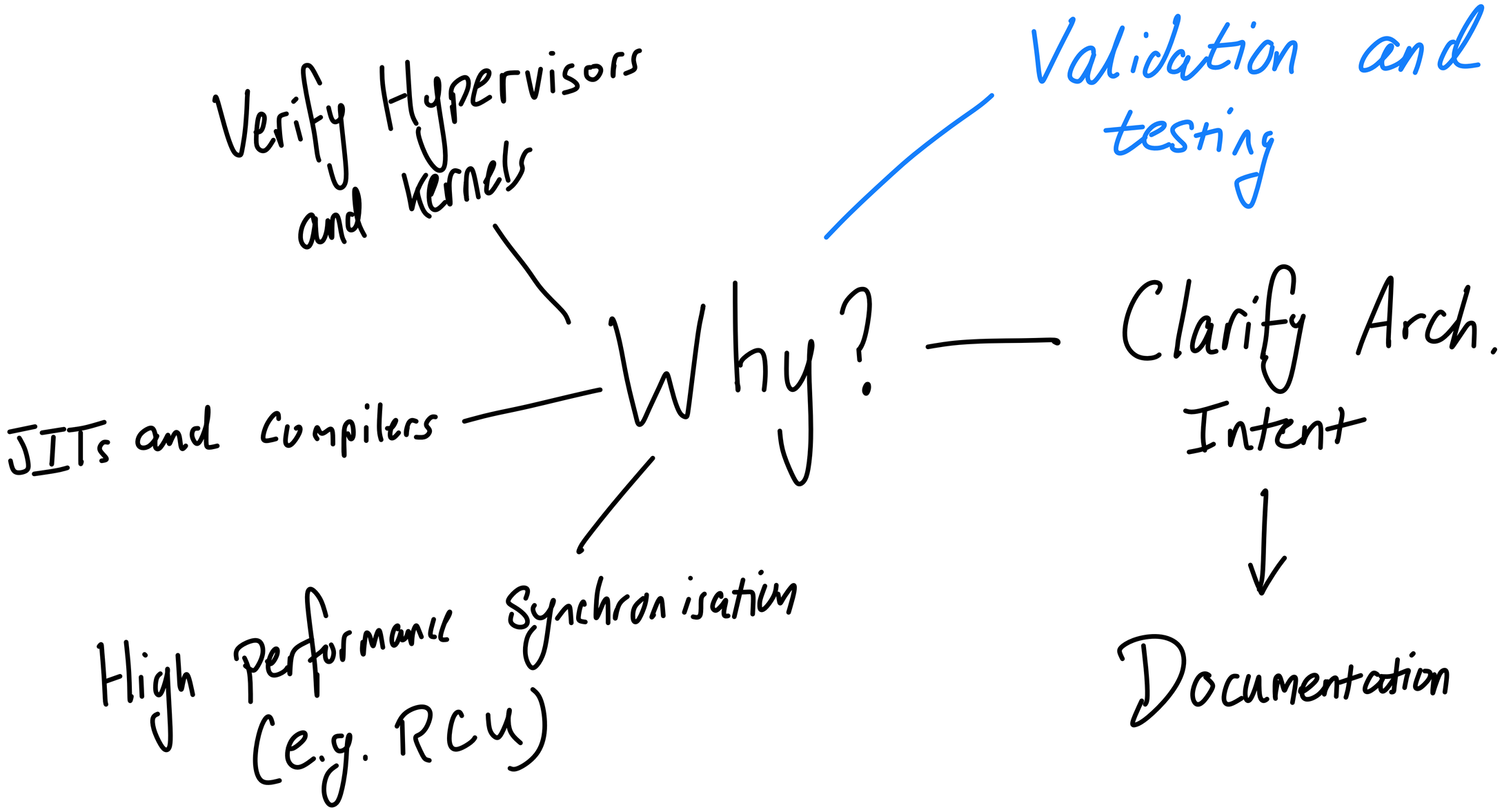

[Deacon, 2017]

Systems Architecture

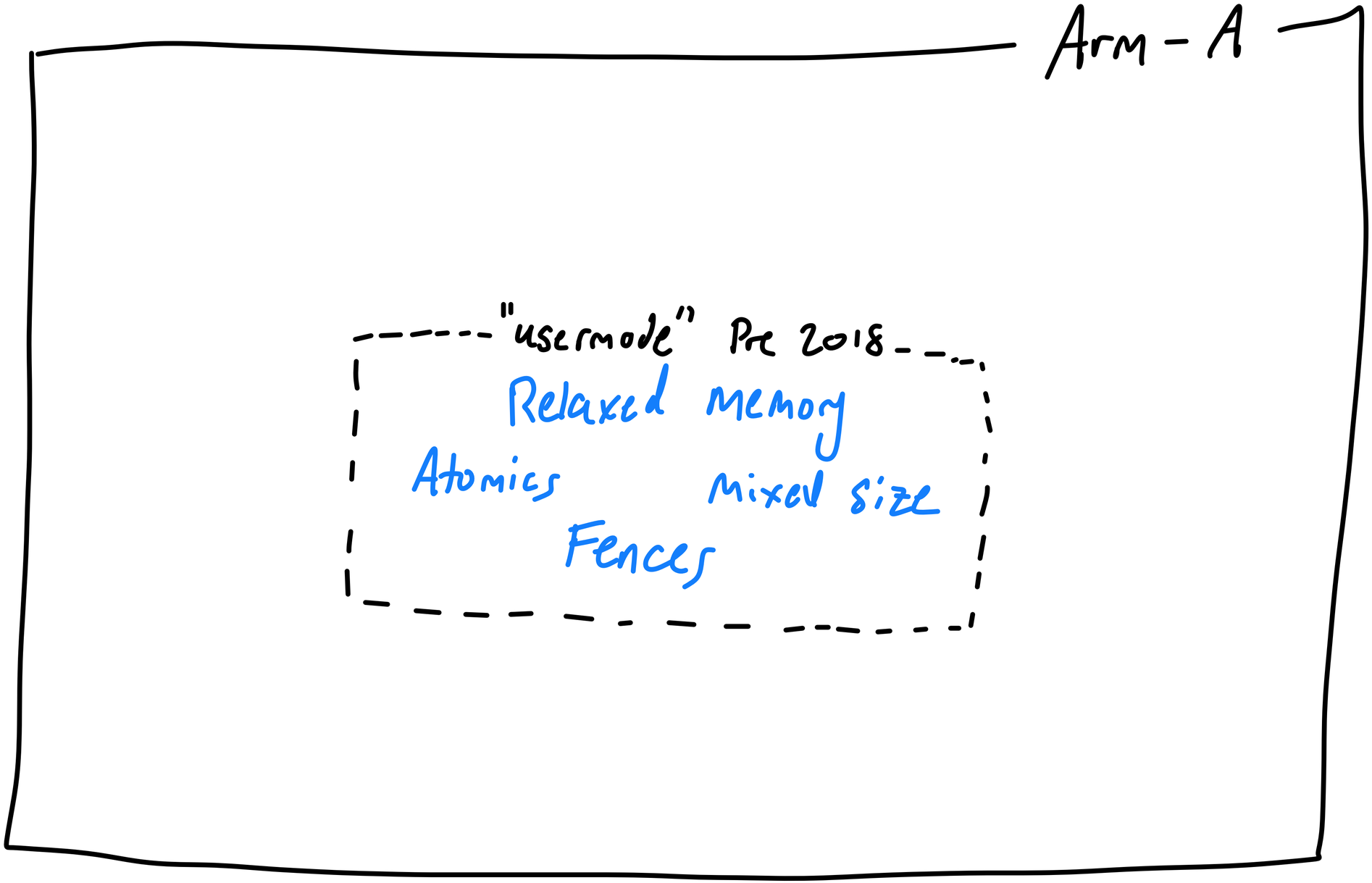
Why?

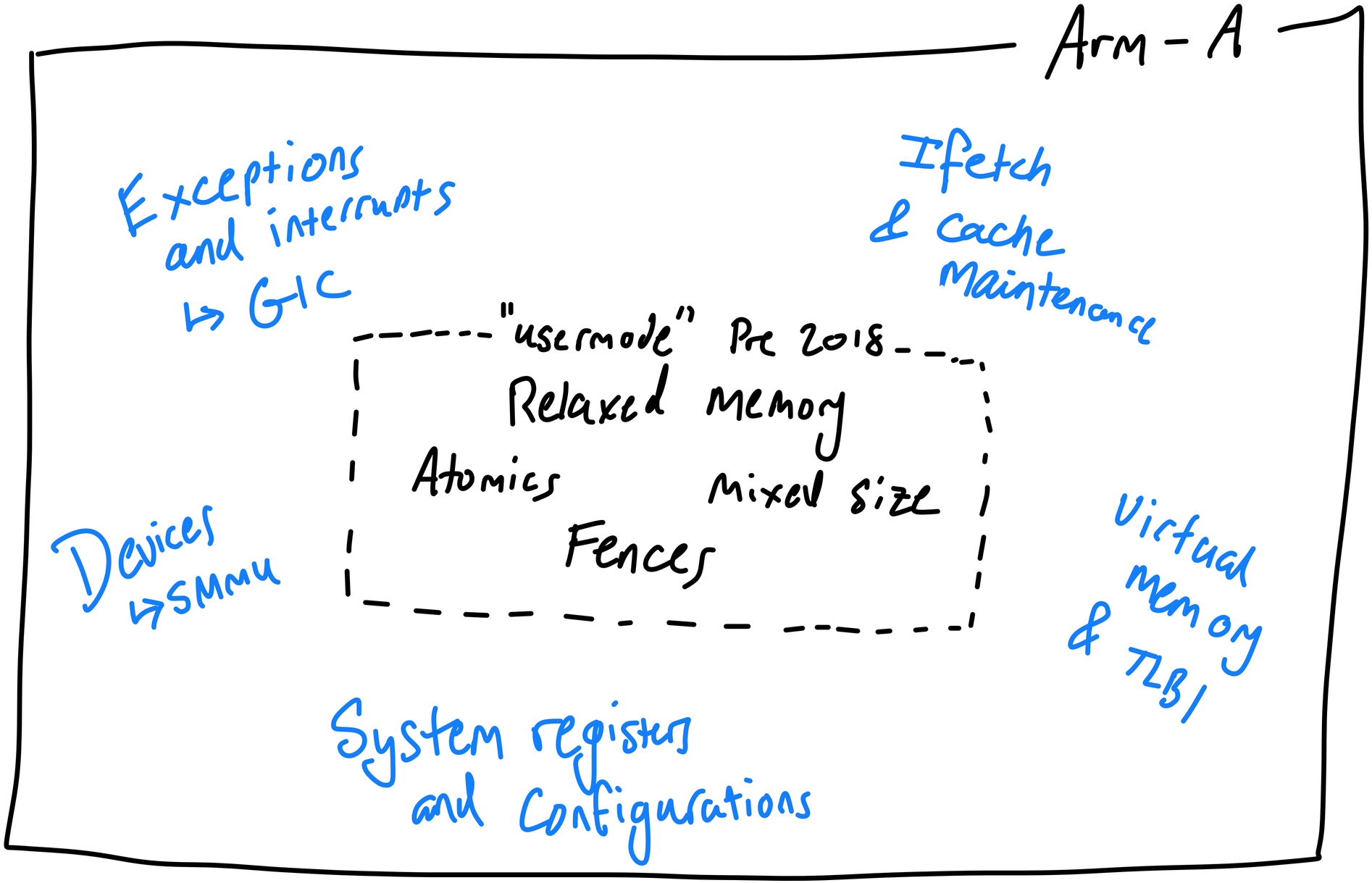

Verify Hypervisors and kernels
JITs and Compilers
Why?
High Performance Synchronisation (e.g. RCU)

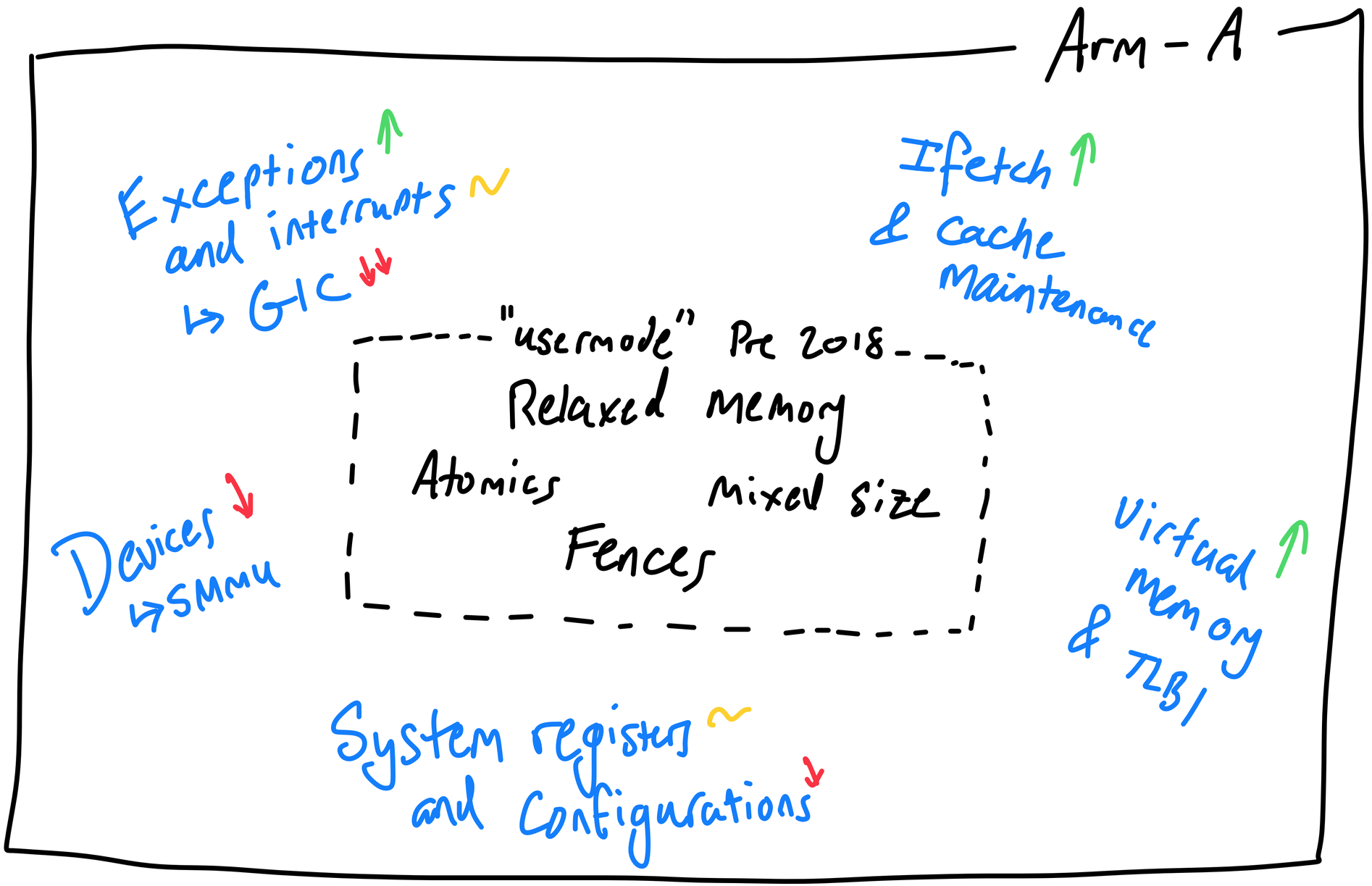

Verify Hypervisors and kernels
JITs and Compilers
Why?
High Performance Synchronisation (e.g. RCU)
Clarify Arch. Intent
Documentation


Verify Hypervisors and kernels
Validation and testing
JITs and Compilers
Why?
Clarify Arch. Intent
High Performance Synchronisation (e.g. RCU)
Documentation


Arm-A

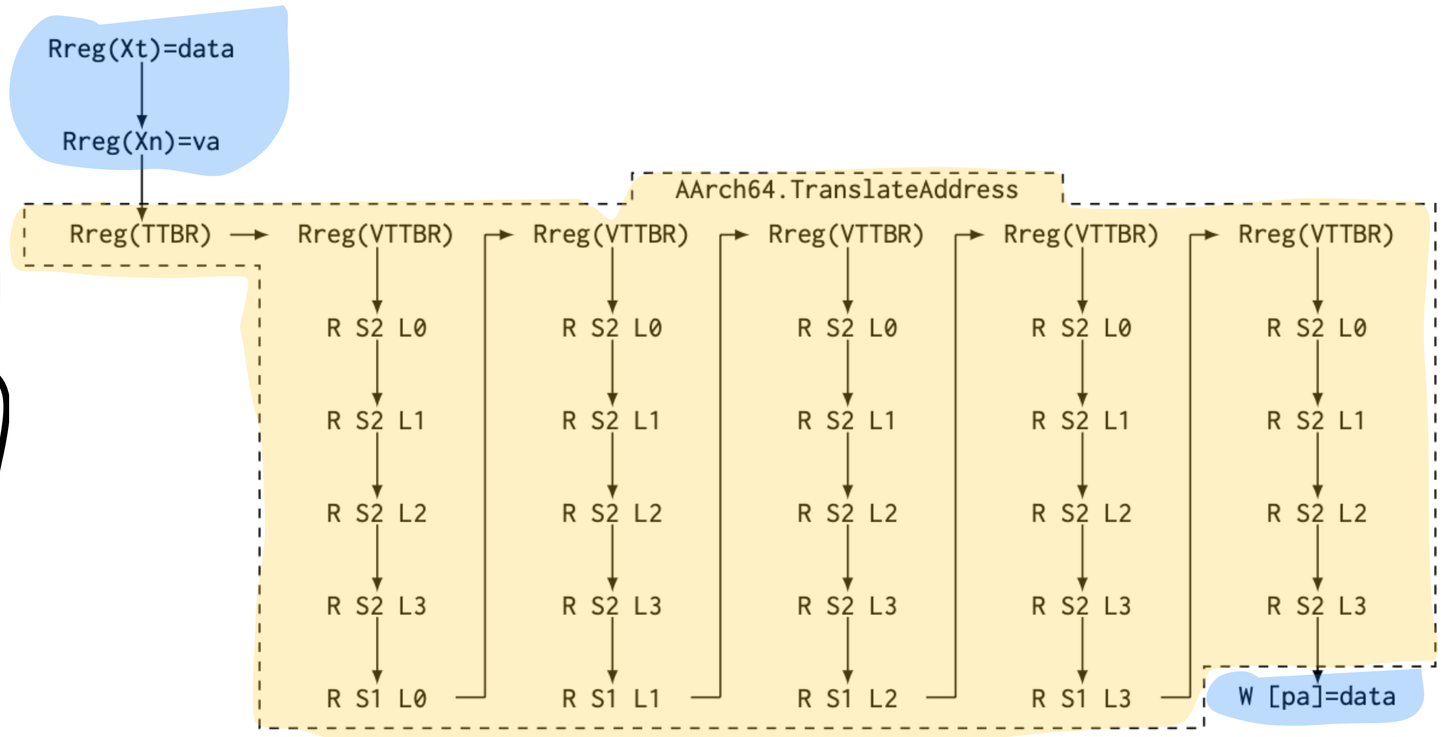

Arm-A
"usermode" Pre 2018
Relaxed memory
Atomics
Mixed size
Fences


Arm-A
Exceptions and interrupts ↦ GIC
Ifetch & cache Maintenance
"usermode" Pre 2018
Relaxed memory
Atomics
Mixed size
Fences
Devices ↦SMMu
Virtual memory & TLBI
System registers and Configurations


Arm-A
Exceptions and interrupts
↳ GIC
Ifetch & cache Maintenance
"usermode" Pre 2018
Relaxed memory
Atomics
Mixed size
Fences
Devices
↳SMMu
Virtual memory & TLBI
System registers and Configurations

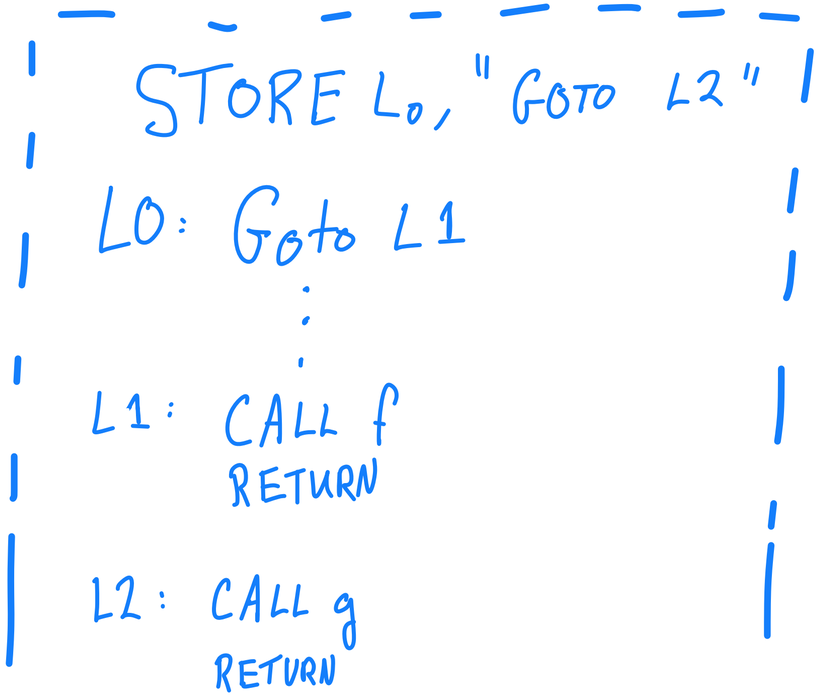
Systems Behaviours are implicit

$[\![\text{STR} \quad X_t, [X_n]]\!] =$



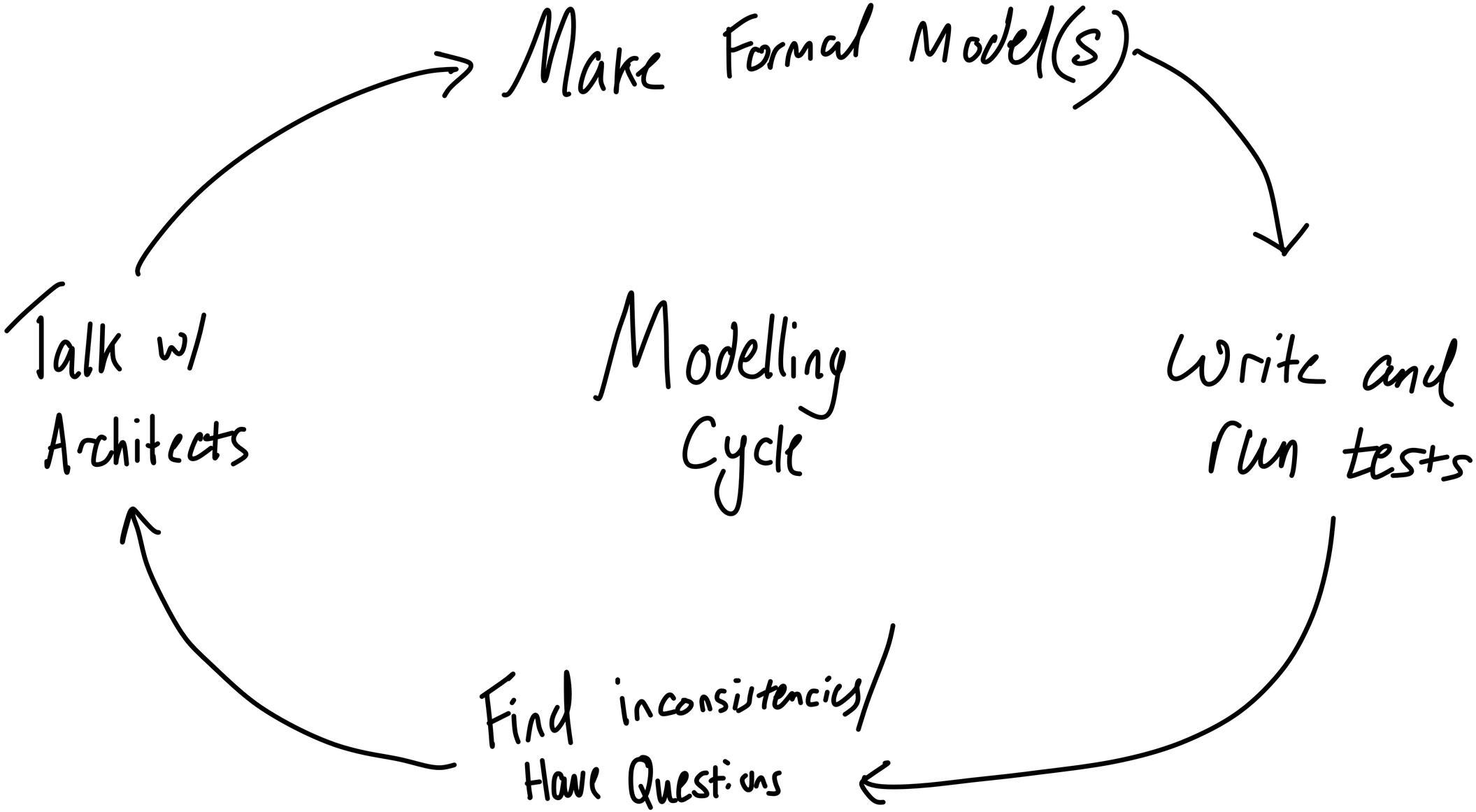
Systems Problems:

— Concurrency within a thread



Expected:
Always call g

Reality:
Non-det call f or g.


Make Formal Model(s)
Write and run tests
Find inconsistencies/ Have Questions
Talk w/ Architects
Modelling Cycle

Conclusion

- Architectures like Arm underpin all software
- Have tooling to build models integrated with ISA
- Systems features important and under-explored
- Talking with architects works

Thanks!

Jean Pichon-Pharabod

Shaked Flur

Christopher Pulte

Peter Sewell

Luc Maranget

Yeji Han

Alasdair Armstrong

Brian Campbell

Thomas Fawcier