Backwards Compatible

~~Bottom-up~~ formalization of the ARM architecture

Alastair Reid

R&D

ARM Ltd

February 2012

What ARM does

ARM Holdings is the world's leading semiconductor intellectual property (IP) supplier and as such is at the heart of the development of digital electronic products. Headquartered in Cambridge UK and employing over 2,000 people, ARM has offices around the world, including design centers in Taiwan, France, India, Sweden, and the US.

Company Highlights

- The world's leading semiconductor IP company
- Founded in 1990
- Over 20 billion ARM based chips shipped to date
- 800 processor licenses sold to more than 250 companies
- Royalties received on all ARM-based chips
- Gaining market share in long-term secular growth markets
- ARM revenues typically grow faster than overall semiconductor industry revenues

Outline

- Challenges in creating a formal ISA specification
  - 2 “Technical” issues
  - 2 “Social/Business” issues
- Bottom-up formalization
  - Process
  - Sketch(es) of semantics
- Work in progress

Challenge #1: Pick a language

Many choices

- Custom ISA specification language
  - E.g., LISA (Ishtiaq)
- General purpose formal specification language
  - E.g., HOL (Fox), Coq (Chong & Ishtiaq)
- Golden Verilog reference
  - E.g., ARM CPU Validation teams
- Frontend for multiple specification languages
  - E.g., LEM (Owens et al.)

#2: ISA Spec is deliberately broad

The ARM architecture supports implementations across a wide range of performance points. The architectural simplicity of ARM processors leads to very small implementations, and small implementations mean devices can have very low power consumption. Implementation size, performance, and very low power consumption are key attributes of the ARM architecture.

UNPREDICTABLE
Means the behavior cannot be relied upon. UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or lower level of privilege using instructions that are not UNPREDICTABLE.

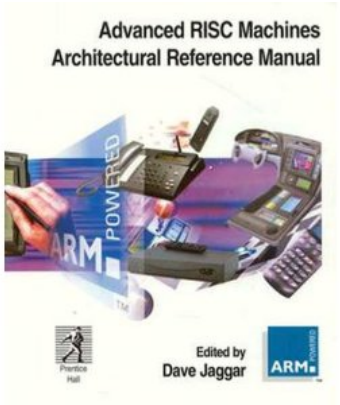
UNKNOWN
An UNKNOWN value does not contain valid data, and can vary from moment to moment, instruction to instruction, and implementation to implementation. An UNKNOWN value must not be a security hole. UNKNOWN values must not be documented or promoted as having a defined value or effect.

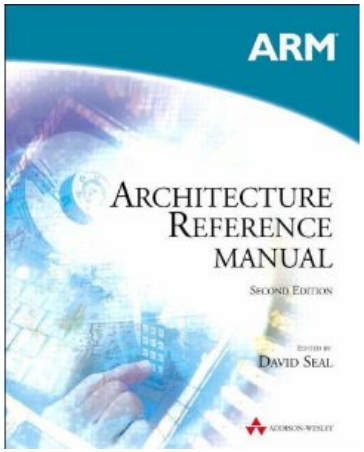
IMPLEMENTATION DEFINED
Means that the behavior is not architecturally defined, but must be defined and documented by individual implementations.

UNDEFINED
Indicates an instruction that generates an Undefined Instruction exception.

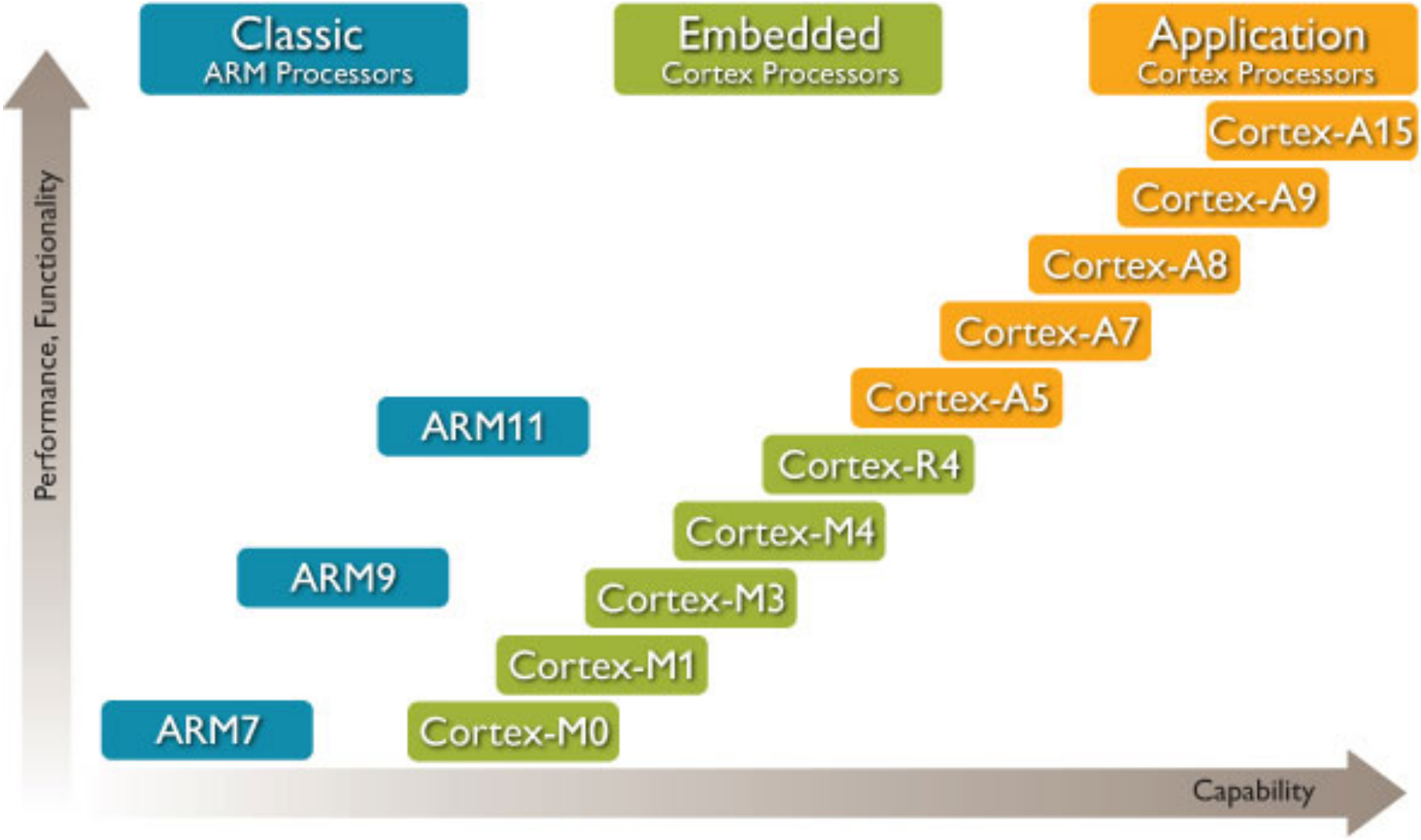
#3: A lot of history

- 1984: Simulator (BBC BASIC)
- 26 April 1985: First Silicon
- 1990: ARM Ltd founded
- Architecture Reference Manual
  - 1996: v4, 328 pages, book
  - 2000: v5, 816 pages, book
  - 2005: v6, 1138 pages, PDF
  - 2007: v7, 2158 pages, PDF
  - 2011: v7, 2668 pages+supplements, PDF
- 2011: ARM Architecture version 8 announced

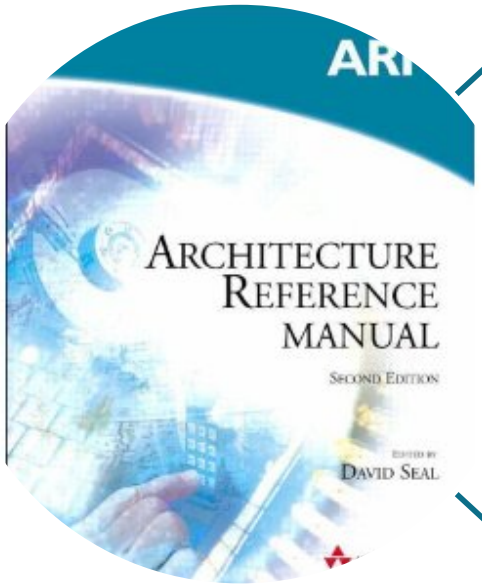




#3: And a lot of processors



#4: What is formal spec used for?



CPU

- Verification

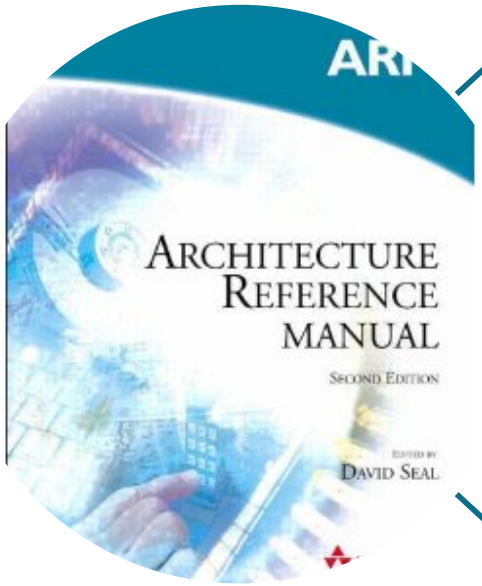
Dev. Tools

- Verify as, ld
- Verify cc

Programs

- Microkernel verification
- OS verification
- App verification

#4: What ARM uses ISA spec for



CPU

- Design
- Licensing
- Validation
  - Test suites
  - Test tools

Models

- Design
- Validation

Dev Tools

- Asm/dasm/ld
- Compiler
- Debugger
- Validation

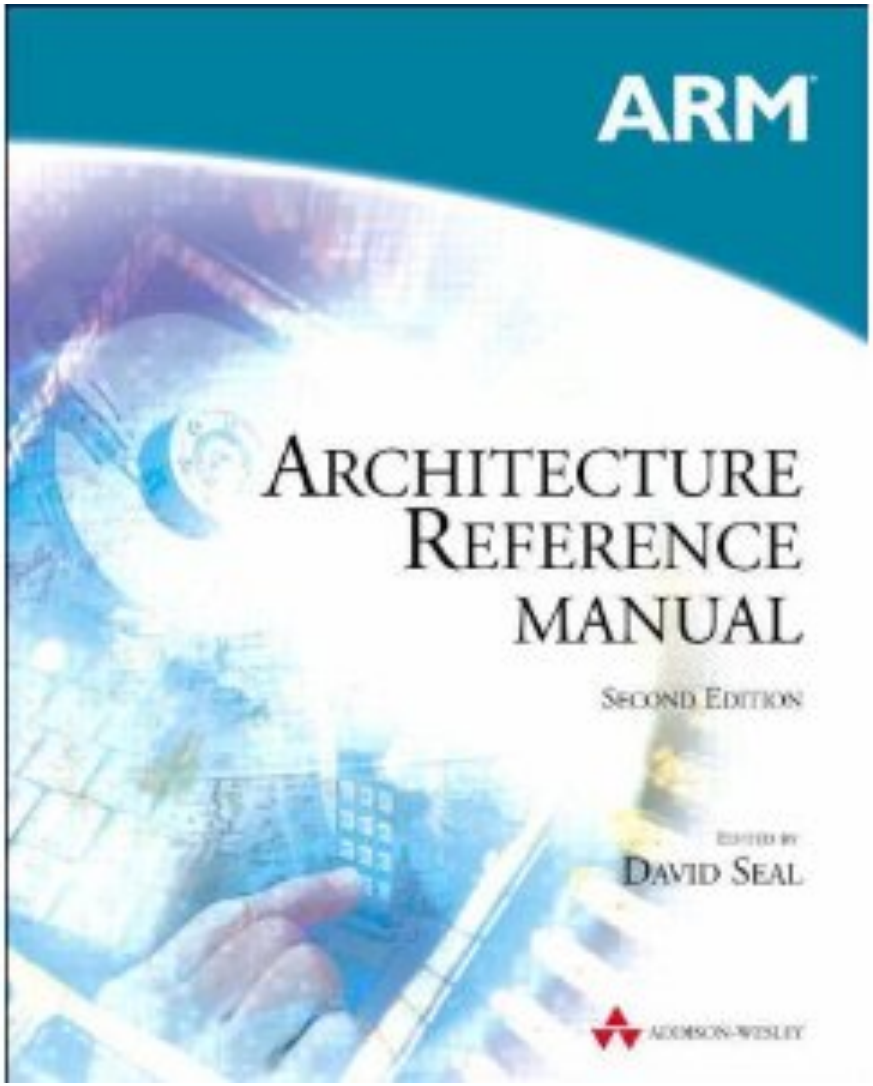
Requirements

- Can be used as a formal specification?
  - Range of languages to choose from
- Broad enough to express full range of legal behaviour?
  - Captures deliberate looseness of specification
- Equivalent to existing spec?
  - Doesn't rule out existing or legitimate implementations
  - Does rule in unacceptable future implementations
- Readable by all the teams inside and outside ARM who need a spec?
  - Hardware engineers, compiler engineers, OS writers, ...

Bottom-up formalization

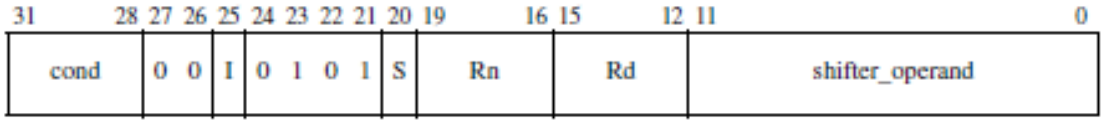
- Start with existing semi-formal specification
- Change (slightly) to make it a formal spec
- Test against existing CPUs, test suites, etc.
  - Prototypes generated from current spec+semantics
- Automatically generate code/data/specs
  - Traditional formal spec (e.g., in Coq/HOL/LEM/...)
  - Reference Verilog
  - Simulators
  - Instrumented interpreters
  - Assembler, disassembler, ...
  - Random Instruction Sequence tester
  - Tables of system registers (for debugger)

The ARM ARM


ARM
ARCHITECTURE
REFERENCE
MANUAL
SECOND EDITION
EDITED BY
DAVID SEAL
ADDISON-WESLEY

ARMv5 pseudocode

A4.1.2 ADC



ADC (Add with Carry) adds two values and the Carry flag. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the addition.

ADC can optionally update the condition code flags, based on the result.

Syntax

```
ADC{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

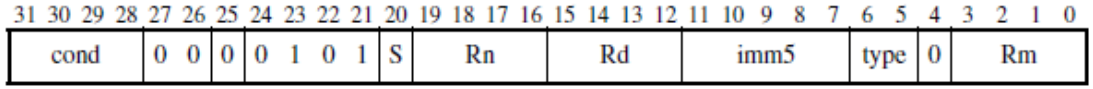
Operation

```
if ConditionPassed(cond) then
    Rd = Rn + shifter_operand + C Flag
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = CarryFrom(Rn + shifter_operand + C Flag)
        V Flag = OverflowFrom(Rn + shifter_operand + C Flag)
```

ARMv7 specification

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ADC{S}<c> <Rd>,<Rn>,<Rm>{,<shift>}
```



31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 0	0	0 1 0 1	S	Rn	Rd	imm5	type	0	Rm

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

Assembler syntax

```
ADC{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, APSR.C);
    if d == 15 then          // Can only occur for ARM encoding
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

ARMv7 support functions

```
(SRType, integer) DecodeImmShift(bits(2) type, bits(5) imm5)

    case type of
        when '00'
            shift_t = SRType_LSL;  shift_n = UInt(imm5);
        when '01'
            shift_t = SRType_LSR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '10'
            shift_t = SRType_ASR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '11'
            if imm5 == '00000' then
                shift_t = SRType_RRX;  shift_n = 1;
            else
                shift_t = SRType_ROR;  shift_n = UInt(imm5);

    return (shift_t, shift_n);


(bits(N), bit) Shift_C(bits(N) value, SRType type, integer amount, bit carry_in)
    assert !(type == SRType_RRX && amount != 1);

    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case type of
            when SRType_LSL
                (result, carry_out) = LSL_C(value, amount);
            when SRType_LSR
                (result, carry_out) = LSR_C(value, amount);
            when SRType_ASR
                (result, carry_out) = ASR_C(value, amount);
            when SRType_ROR
                (result, carry_out) = ROR_C(value, amount);
            when SRType_RRX
                (result, carry_out) = RRX_C(value, carry_in);

    return (result, carry_out);
```

- Bounded Precision Ints
- Unbounded Precision Ints (and Rationals)
- Type Inference
- Enumerations
- Indentation-based Syntax
- Dependent Types
- Imperative
- Exceptions

Revised Goal

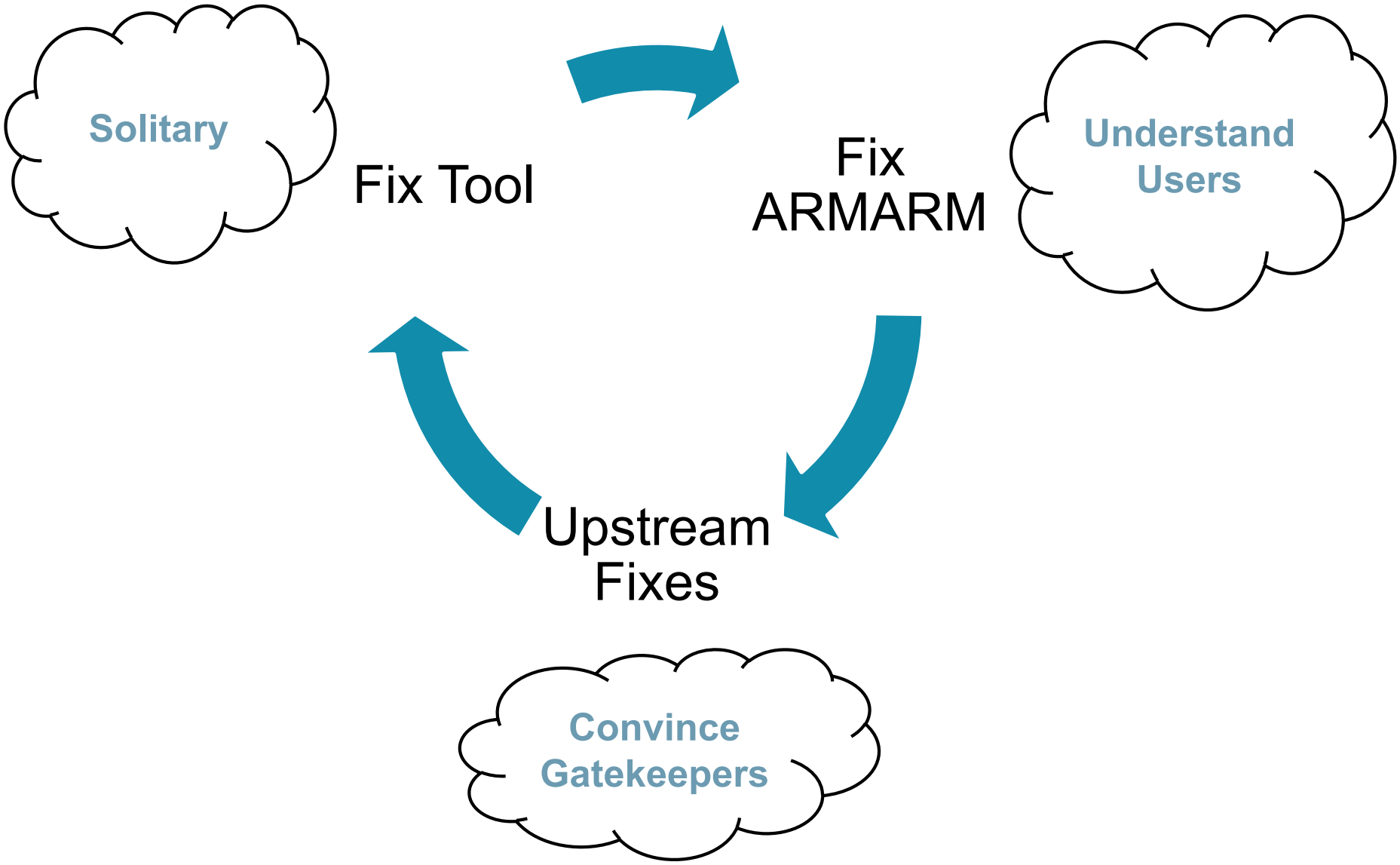
Evolve existing specification into formal specification

- With a precise (but non-deterministic) meaning
- Without excluding existing interpretations
- Without losing readability
- Without making too many changes

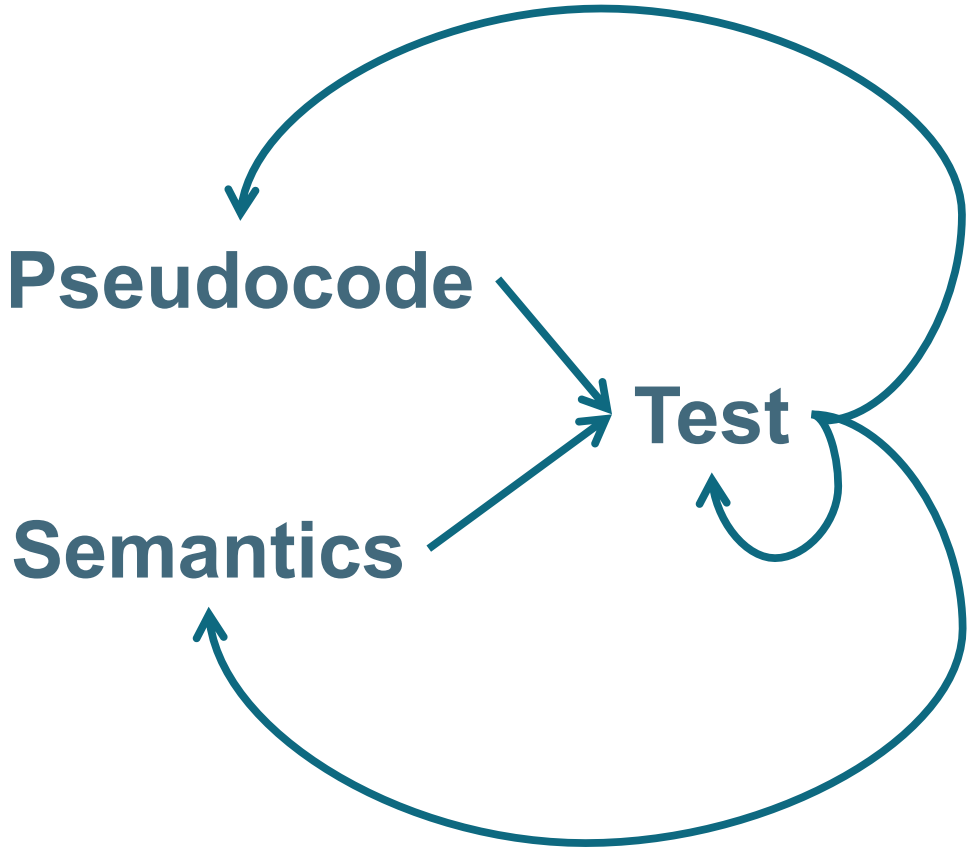
Formalizing Existing Specification

1. Write a parser
   - ➔ Fix syntax errors in specification
   - ➔ Fix specification of language syntax
2. Write a typechecker
   - ➔ Fix typing errors in specification
   - ➔ Fix specification of language typesystem
3. Write a compiler/interpreter
   - ➔ Fix semantic errors in specification
   - ➔ Fix specification of language semantics

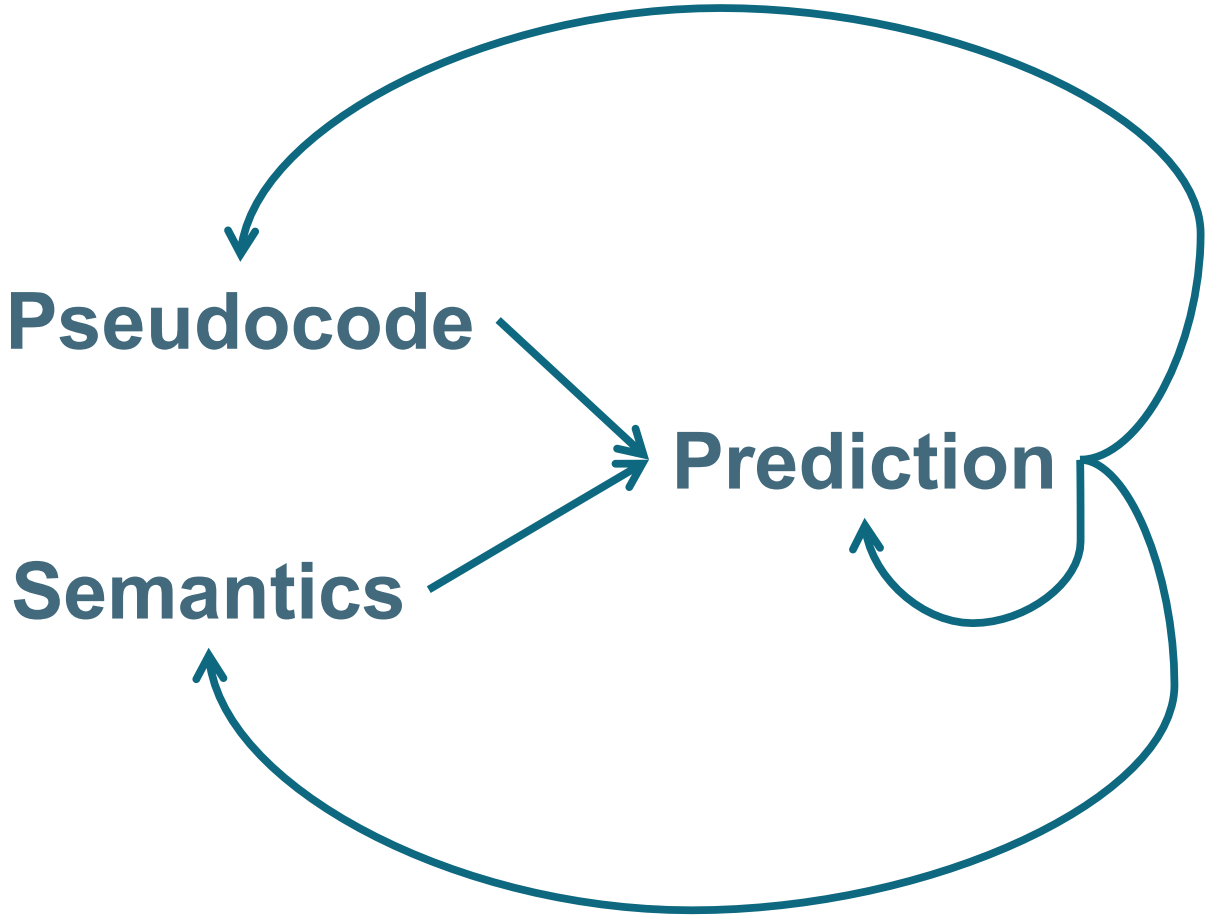
Highly iterative (and social) process



Iterative process



Iterative process

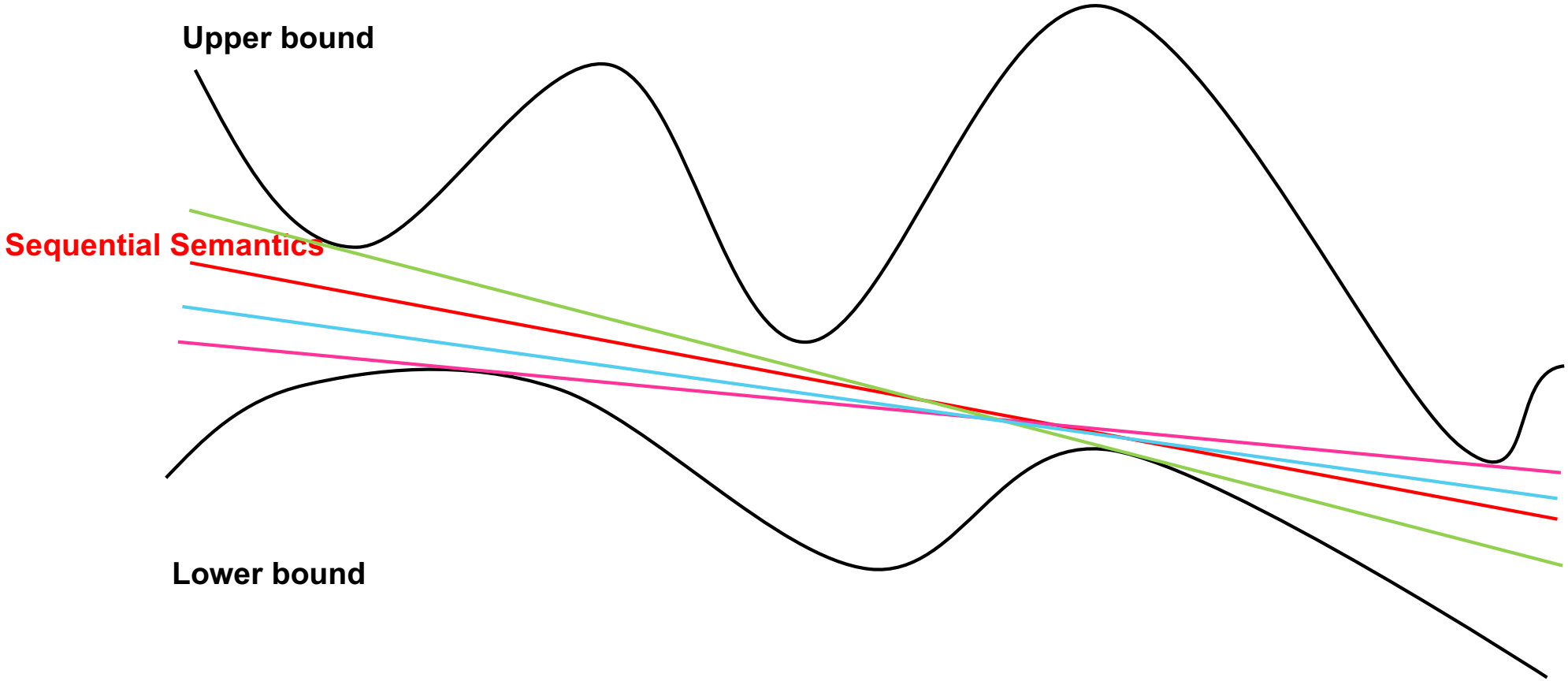


Initial version of semantics

$$\text{expr: State} \rightarrow \text{State x Value}_{\perp} + \text{Exception}$$

$$\text{stmt: State} \rightarrow \text{State x Exception}$$

Overly narrow specification



Note: Don’t yet have a definitive semantics – this is a sketch of one direction it might go.

Limitations of pseudocode

“The pseudocode descriptions of instructions have a number of limitations.

These are mainly due to the fact that, for clarity and brevity, the pseudocode is a sequential and mostly deterministic language.

These limitations include: ... ”

Limitation 1a: Memory access order

Pseudocode does not describe the ordering requirements when an instruction generates multiple memory accesses, except in the case of SWP and SWPB instructions where the two accesses are to the same memory location. For a description of the ordering requirements on memory accesses see *Memory access order* on page A3-143.

Mem[i] = a; Mem[j] = b;

==

Mem[j] = b; Mem[i] = a;

Note: i==j case discussed later

Limitation 1b: Register access order

A processor exception can be taken during execution of the pseudocode for an instruction, either explicitly as a result of the execution of a pseudocode function such as `DataAbort()`, or implicitly, for example if an interrupt is taken during execution of an `LDM` instruction. If this happens, the pseudocode does not describe the extent to which the normal behavior of the instruction occurs. To determine that, see the descriptions of the processor exceptions in *Exception handling* on page B1-1164.

R[i] = a; R[j] = b;

==

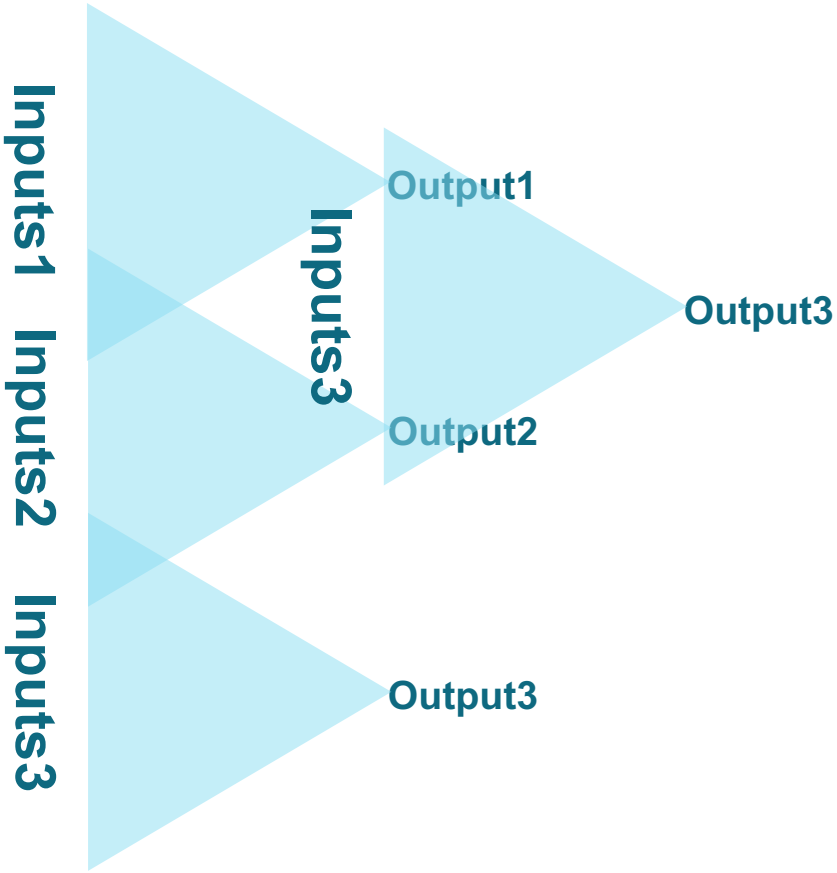
R[j] = b; R[i] = a;

Note: i==j case discussed later

Pseudocode is not entirely sequential

- Language designed by and for hardware engineers
- Hardware engineers ‘think parallel’

- Logic cones



Revised semantics

- Each value is tagged with its logic cone
  - i.e., global variables that the value depends on
- Well defined if
  - At most one value assigned to each global variable
  - No value depends on a global variable that is assigned to

$$R[i] = a;\ R[j] = b; = \begin{cases} R[i] = a; \quad R[j] = b; & \text{if } i!=j \\ R[i] = \bot;\ R[j] = \bot; & \text{if } i==j \end{cases}$$

Revised semantics

- Each value is tagged with its logic cone
  - i.e., global variables that the value depends on
- Well defined if
  - At most one value assigned to each global variable
  - No value depends on a global variable that is assigned to

$$\text{Mem[i] = a; Mem[j] = b; } = \begin{cases} \text{Mem[i] = a;} \quad \text{Mem[j] = b;} & \text{if i!=j} \\ \text{Mem[i] = } \bot\text{;} & \text{if i==j} \end{cases}$$

Revised semantics

- Each value is tagged with its logic cone
  - i.e., global variables that the value depends on
- Well defined if
  - At most one value assigned to each global variable
  - No value depends on a global variable that is assigned to

Mem[i] = a; i = Mem[j]=

- Mem[i] = a; i = Mem[j]; if i!=j
- UNPREDICTABLE; if i==j

Revised version of semantics

$$CState = Var \rightarrow Cone[Value_{\perp}]$$

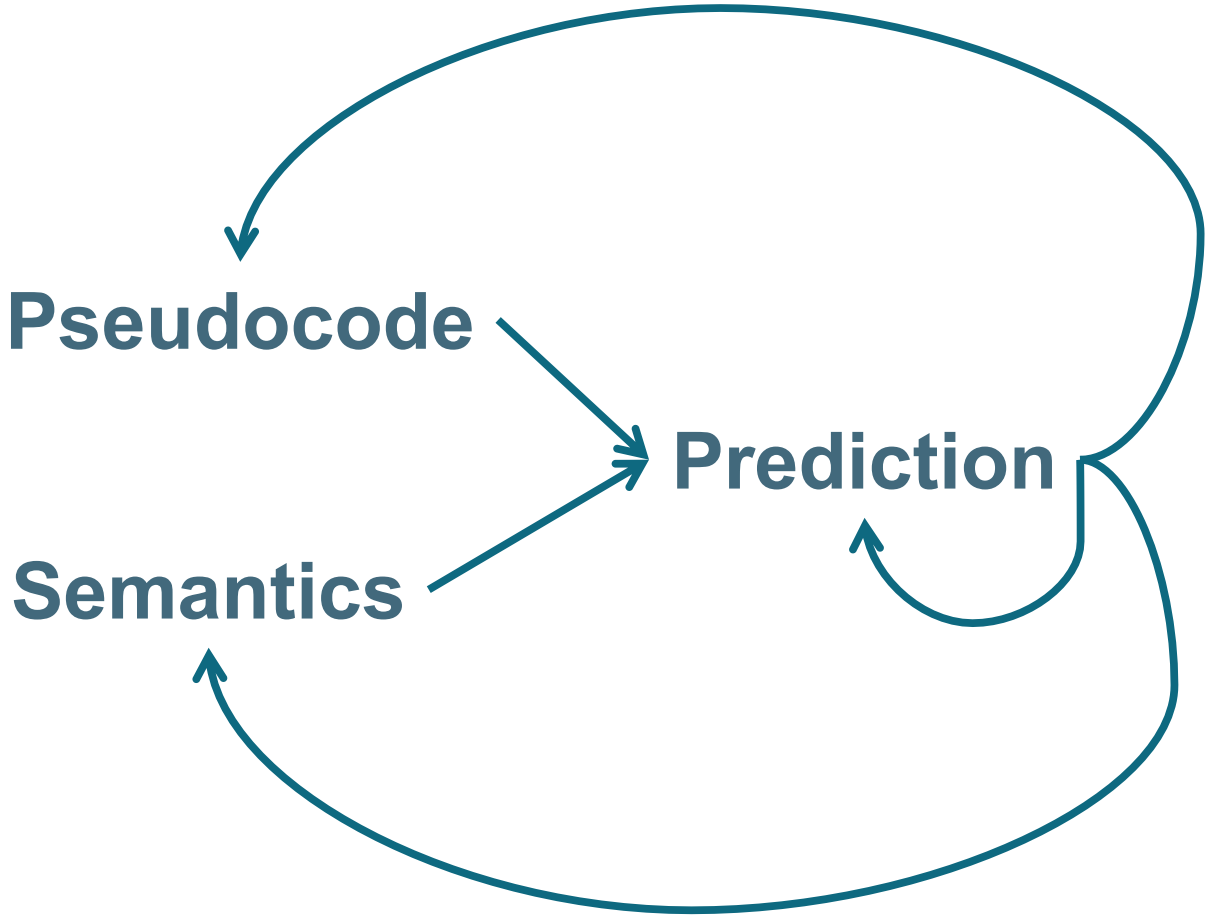
$$expr: State \rightarrow CState \times Cone[Value_{\perp}] \times Exception$$

$$stmt: State \rightarrow CState \times Exception$$

$$instr: State \rightarrow State \times Exception$$

Note: expression and statement composition left as an exercise...

Iterative process



Testing this semantics #1

LDM r1!, {r1,r2}

Testing this semantics #1

LDM Rn!, {registers}

```
address = R[n] - 4*BitCount(registers) + 4;
for i = 0 to 14
  if registers<i> == '1' then
    R[i] = Mem[address,4];
    address = address + 4;

if registers<15> == '1' then LoadWritePC(MemA[address,4]);

if wback && registers<n> == '0' then R[n] = R[n] - 4*BitCount(registers);

if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

Testing this semantics #2

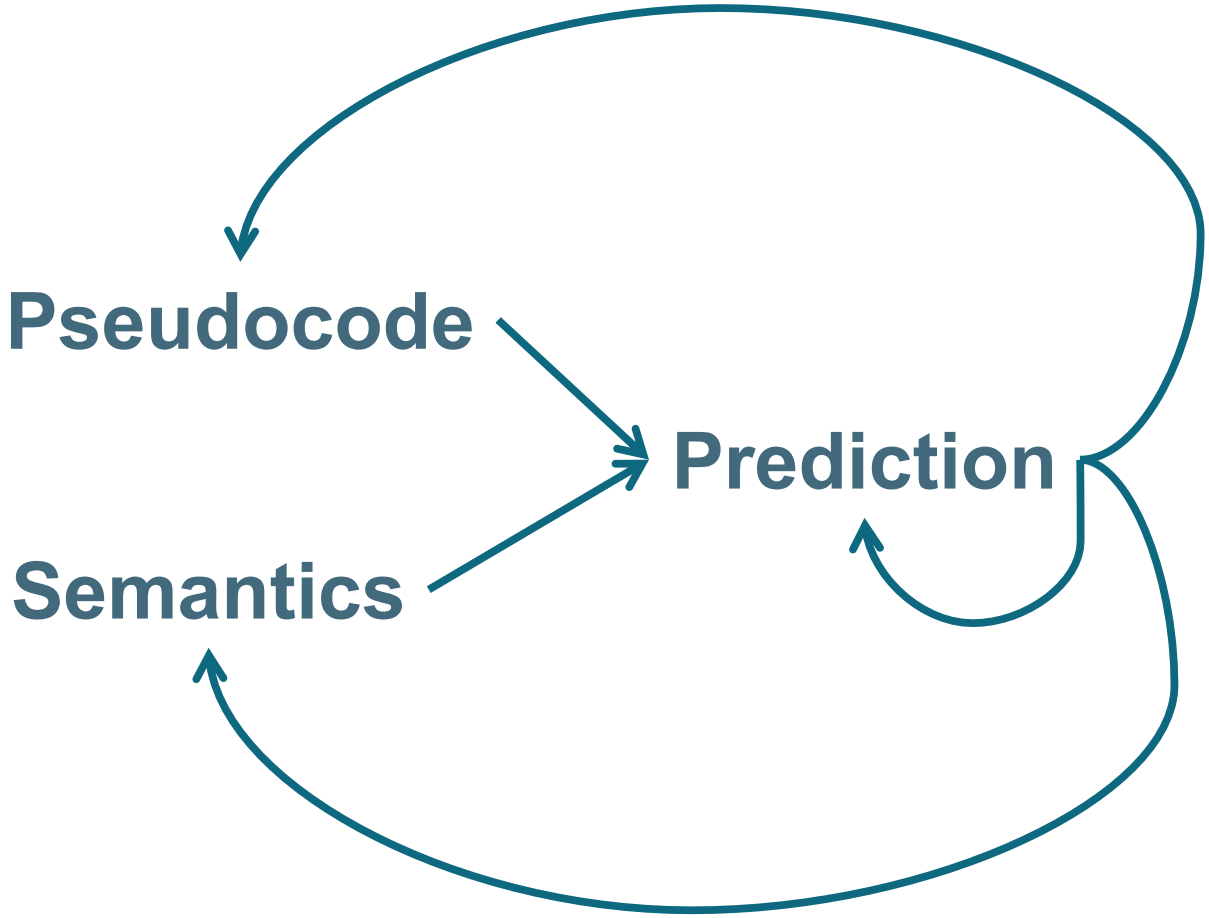
STR R0,[R0]!

Testing this semantics #2

STR Rn,[Rm,offset]!

```
offset = Shift(R[m], shift_t, shift_n, APSR.C);
offset_addr = if add then (R[n] + offset)
                     else (R[n] - offset);
address = if index then offset_addr else R[n];
if t == 15 then data = PCStoreValue();
           else data = R[t];
MemU[address,4] = data;
if wback then R[n] = offset_addr;
```

Iterative process



Summary of semantics

- ‘Parallel’ semantics of sequential language
  - Based on data dependencies
  - Multiple writes to same piece of global state UNKNOWN/UNPREDICTABLE

- Iterative development process
  - Continually test against existing codebase and architecture team
    - ➔Change spec
    - ➔Change semantics
  - About to start testing against test suites and CPUs

Generating tools from ARMARM

- Translate to C (Simulator)
  - ARMv6-M (Microcontroller)
  - ARMv7-R (Real Time, Protected Memory)
  - ~~ARMv7-A (Applications, Virtual Memory)~~
  - ~~ARMv8 (64-bit)~~
- Translate to Verilog (Validation Reference)
  - ARMv6-M (Microcontroller)
- Generate Assembler/Disassembler
  - ARMv8 (64-bit)

Current focus: testing existing tools and processors

(Validating our approach in the process)

Conclusion/Status

- Evolving existing semi-formal spec into a formal spec
  - Avoid large discontinuities
  - Focus on acceptability to various communities
- Focus until now has been on syntax+typesystem
  - Iterative process: test on codebase + users
- Finding semantics will take time and experimentation
  - Current semantics ‘correct’ but excludes many legal implementations
  - Iterative process: test on ARM validation suites + CPUs + users
- Some initial experience of building tools
  - (but no formal specification yet)

Fin

ARMv7 specification language

- Syntax
  - Algol-like
  - Indentation based
- Types
  - Simple type inference
  - Dependent types (integer additive expressions)
  - First order, bits(N), integer, real (== rational), enumerations, records
- Semantics
  - Imperative, *mostly* sequential
  - Exceptions: UNDEFINED, etc.
  - *Supplemented by natural language descriptions*