The Ironbark Formal Model

Micah Brown, Brendan Mahony, Jim McCarthy

November 8, 2023

Contents

1 Overview

In this section, we provide an overview of the structure of the Ironbark formal model.

The remainder of this document contains the output generated by Isabelle for the Ironbark formal model. The formal model is built for the 2023 version of Isabelle.

1.1 Design

We provide a brief overview of the design of the Ironbark processor which we model, but this is not intended to be a complete specification. For a complete specification, we refer the interested reader to the implementation of the Ironbark processor in our formal model.

The Ironbark processor can be broken down into the following conceptual components:

- flags
- registers
- memory
- instruction decoding and execution logic.

Flags, registers, and memory provide the state of the processor. Instructions provide the mechanisms for changing that state.

The Ironbark memory architecture takes inspiration from the Harvard architecture, which has separate memory for the executable program and data. In the Ironbark architecture, we have six separate memory spaces, which are:

- program memory
- call memory
- statically allocated memory ("Static memory")
- dynamically allocated memory ("Dynamic memory")
- input memory
- output memory.

Program memory holds the code for execution.

Call memory holds the call stack, which is made of fixed size frames, which include various register backups.

Input memory provides a memory mapped interface to inputs to the processor from the outside world.

Conversely, output memory provides a memory mapped interface for the processor to talk to the outside world.

In contrast with most modern processors, we do not include components for performance optimisation, such as a cache, or a branch predictor. This comes at a performance cost, but greatly simplifies the processor, and makes it much easier to reason about.

Ironbark employs a philosophy of "1 processor, 1 process, 1 thread", and does not support the use of threads, context switching, or other forms of parallelism. If an application benefits from having two processes running in parallel, two processors would be required, with the appropriate mapping of their input and output memories.

1.2 Layers

This document presents the Isabelle theories for the Ironbark processor. In understanding the theories, it is beneficial to understand how we have structured our formal model.

We divide our model into five layers, where each one builds on the one before it:

- preliminary
- state
- state manipulation
- instruction
- execution

The preliminary layer is unique from the other layers, and does not follow much of the pattern or structure of the other layers. It exists solely to provide rules that are not part of the Isabelle libraries that we build on. In particular, the rules included in preliminary are for properties we rely on, but are not particularly related to Ironbark.

The state layer defines the state of the Ironbark processor in Isabelle, and provides some results relating to how it is defined. The results in this layer are trivially derived from the definitions we use.

The state manipulation layer defines how the state can be manipulated, expressed in the underlying Isabelle/HOL semantics. It also provides some basic results on how those definitions work. The results in this layer are trivially derived from the definitions we use.

The instruction layer defines the instructions of the Ironbark architecture in terms of the state manipulation operations. It also provides some results relating to the operation of instructions. The results in this layer are trivially derived from the definitions we use.

The execution layer defines how the execution of the processor develops in terms of instructions and the state manipulation operations. It also proves various properties about the execution of the Ironbark processor. It is at this layer that it becomes possible to describe many of the desired security properties of the Ironbark processor.

On top of the execution layer we have some additional work. This includes variations on showing the correctness of an implementation of the Fibonacci function, and the correctness of a simple program that does $1 + 1$.

1.3 Sublayers

In constructing the definitions and proofs at each layer (other than preliminary), we subdivide the layer into some or all of the following sublayers, in the following order:

- implementations
- auxiliary
- decomposition
- simplification
- reordering
- interface

The implementations sublayer is for providing definitions. It contains no lemmas.

The auxiliary sublayer is included as a pro forma import point for the sections that follow. It is also used to import the results of the layer below. It may also contain auxiliary results that do not fit into any of the other sublayers.

The decomposition (often denoted 'decomp') sublayer provides lemmas that are for decomposing a layer into the layer below it. The rules in this sublayer are typically trivially derived from the definitions of the functions involved.

The simplification (often denoted 'simp') sublayer provides lemmas that are for simplifying the proof state. These are typically formed by enumerating pairings of read and write operations on state components and showing they can be expressed in either one or zero operation(s). In the instruction layer, these rules are aimed at eliminating the instruction from the goal. To aid users in finding simplification rules, they are typically ordered based on the normalised ordering (see 1.3.1). The rules in this sublayer are typically trivially derived from the definitions of the functions involved.

The reordering sublayer provides lemmas that are for changing the order that functions are applied. To avoid creating infinite loops from these rules, we only create reordering rules that lead to a 'normalised' form, which we define below. Any arbitrary ordering can be derived from this by applying the corresponding rule in the appropriate direction. We note that while we define this ordering, we do not have rules which cover all reorderings. This is because we found reordering to rarely be needed, and only created rules when we had a need for them.

The interface sublayer (typically in files named '_top') gathers related lemmas together for easier reference. It also provides a single point for higher layers to import.

Each layer may also contain additional components or divisions as needed, especially when there are noteworthy results, but this is a rough guide for the general structure of the boilerplate results we work from.

1.3.1 Normalised Forms

The normalised form for reordering rules has the highest layer applied last.

Within the state manipulation layer, the following ordering is applied (from applied last, to applied first):

- operators on flags
  - clear (see sub-ordering for set)
  - set
    - end return
    - end call
    - end jump
    - halt
    - error
- operators on registers (ordered by register index; highest index applied first)
- operators on memory
  - program memory
  - call memory
  - static memory
  - dynamic memory
  - input memory
  - output memory
- special operators
  - standard post instruction
  - backup registers before call
  - restore registers after call
  - standard error

Within the instruction layer, reordering is done with highest opcode number applied first.

2 Ironbark Implementation

2.1 Defining the Ironbark Processor State

In this section we give a formal definition of the state of the Ironbark processor. In making these definitions we made some assumptions which are not stated in the definitions themselves, but may be relevant to real-world security applications. The most notable assumptions made is that the hardware is functionally perfect. That is to say that the hardware never produces a fault under any circumstances. We also assume that there are no observable side effects of execution other than timing.

theory *state_implementations*

imports
"Ironbark_prelim.preliminaries"

begin

We begin our definitions with defining the state that the processor has.

The following abbreviations are used to map a number (called a "regID" or "register identifier") to each of the registers in the Ironbark architecture. The allocation can be summarised by the following (where '*' is a wildcard):
r*_ref -> 0x0*
p*_ref -> 0x1*
c*_ref -> 0x2*
arg*_ref -> 0x3*
ret*_ref -> 0x4*

Special registers are allocated/mapped from 0x50 onwards, and are grouped with similar kinds of registers.

abbreviation *(input)* ‹*r00_ref :: 8 word* ≡ *0x0*›
abbreviation *(input)* ‹*r01_ref :: 8 word* ≡ *0x1*›
abbreviation *(input)* ‹*r02_ref :: 8 word* ≡ *0x2*›
abbreviation *(input)* ‹*r03_ref :: 8 word* ≡ *0x3*›
abbreviation *(input)* ‹*r04_ref :: 8 word* ≡ *0x4*›
abbreviation *(input)* ‹*r05_ref :: 8 word* ≡ *0x5*›
abbreviation *(input)* ‹*r06_ref :: 8 word* ≡ *0x6*›
abbreviation *(input)* ‹*r07_ref :: 8 word* ≡ *0x7*›
abbreviation *(input)* ‹*r08_ref :: 8 word* ≡ *0x8*›
abbreviation *(input)* ‹*r09_ref :: 8 word* ≡ *0x9*›
abbreviation *(input)* ‹*r10_ref :: 8 word* ≡ *0xa*›
abbreviation *(input)* ‹*r11_ref :: 8 word* ≡ *0xb*›
abbreviation *(input)* ‹*r12_ref :: 8 word* ≡ *0xc*›
abbreviation *(input)* ‹*r13_ref :: 8 word* ≡ *0xd*›
abbreviation *(input)* ‹*r14_ref :: 8 word* ≡ *0xe*›
abbreviation *(input)* ‹*r15_ref :: 8 word* ≡ *0xf*›

— pointer registers
abbreviation *(input)* ‹*p00_ref :: 8 word* ≡ *0x10*›
abbreviation *(input)* ‹*p01_ref :: 8 word* ≡ *0x11*›
abbreviation *(input)* ‹*p02_ref :: 8 word* ≡ *0x12*›
abbreviation *(input)* ‹*p03_ref :: 8 word* ≡ *0x13*›
abbreviation *(input)* ‹*p04_ref :: 8 word* ≡ *0x14*›
abbreviation *(input)* ‹*p05_ref :: 8 word* ≡ *0x15*›
abbreviation *(input)* ‹*p06_ref :: 8 word* ≡ *0x16*›
abbreviation *(input)* ‹*p07_ref :: 8 word* ≡ *0x17*›
abbreviation *(input)* ‹*p08_ref :: 8 word* ≡ *0x18*›
abbreviation *(input)* ‹*p09_ref :: 8 word* ≡ *0x19*›

abbreviation *(input)* ‹*p10_ref :: 8 word* ≡ *0x1a*›
abbreviation *(input)* ‹*p11_ref :: 8 word* ≡ *0x1b*›
abbreviation *(input)* ‹*p12_ref :: 8 word* ≡ *0x1c*›
abbreviation *(input)* ‹*p13_ref :: 8 word* ≡ *0x1d*›
abbreviation *(input)* ‹*p14_ref :: 8 word* ≡ *0x1e*›
abbreviation *(input)* ‹*p15_ref :: 8 word* ≡ *0x1f*›

— constant registers
abbreviation *(input)* ‹*c00_ref :: 8 word* ≡ *0x20*›
abbreviation *(input)* ‹*c01_ref :: 8 word* ≡ *0x21*›
abbreviation *(input)* ‹*c02_ref :: 8 word* ≡ *0x22*›
abbreviation *(input)* ‹*c03_ref :: 8 word* ≡ *0x23*›
abbreviation *(input)* ‹*c04_ref :: 8 word* ≡ *0x24*›
abbreviation *(input)* ‹*c05_ref :: 8 word* ≡ *0x25*›
abbreviation *(input)* ‹*c06_ref :: 8 word* ≡ *0x26*›
abbreviation *(input)* ‹*c07_ref :: 8 word* ≡ *0x27*›
abbreviation *(input)* ‹*c08_ref :: 8 word* ≡ *0x28*›
abbreviation *(input)* ‹*c09_ref :: 8 word* ≡ *0x29*›
abbreviation *(input)* ‹*c10_ref :: 8 word* ≡ *0x2a*›
abbreviation *(input)* ‹*c11_ref :: 8 word* ≡ *0x2b*›
abbreviation *(input)* ‹*c12_ref :: 8 word* ≡ *0x2c*›
abbreviation *(input)* ‹*c13_ref :: 8 word* ≡ *0x2d*›
abbreviation *(input)* ‹*c14_ref :: 8 word* ≡ *0x2e*›
abbreviation *(input)* ‹*c15_ref :: 8 word* ≡ *0x2f*›

— argument registers
abbreviation *(input)* ‹*arg00_ref :: 8 word* ≡ *0x30*›
abbreviation *(input)* ‹*arg01_ref :: 8 word* ≡ *0x31*›
abbreviation *(input)* ‹*arg02_ref :: 8 word* ≡ *0x32*›
abbreviation *(input)* ‹*arg03_ref :: 8 word* ≡ *0x33*›
abbreviation *(input)* ‹*arg04_ref :: 8 word* ≡ *0x34*›
abbreviation *(input)* ‹*arg05_ref :: 8 word* ≡ *0x35*›
abbreviation *(input)* ‹*arg06_ref :: 8 word* ≡ *0x36*›
abbreviation *(input)* ‹*arg07_ref :: 8 word* ≡ *0x37*›
abbreviation *(input)* ‹*arg08_ref :: 8 word* ≡ *0x38*›
abbreviation *(input)* ‹*arg09_ref :: 8 word* ≡ *0x39*›
abbreviation *(input)* ‹*arg10_ref :: 8 word* ≡ *0x3a*›
abbreviation *(input)* ‹*arg11_ref :: 8 word* ≡ *0x3b*›
abbreviation *(input)* ‹*arg12_ref :: 8 word* ≡ *0x3c*›
abbreviation *(input)* ‹*arg13_ref :: 8 word* ≡ *0x3d*›
abbreviation *(input)* ‹*arg14_ref :: 8 word* ≡ *0x3e*›
abbreviation *(input)* ‹*arg15_ref :: 8 word* ≡ *0x3f*›

— return registers
abbreviation *(input)* ‹*ret00_ref :: 8 word* ≡ *0x40*›
abbreviation *(input)* ‹*ret01_ref :: 8 word* ≡ *0x41*›
abbreviation *(input)* ‹*ret02_ref :: 8 word* ≡ *0x42*›
abbreviation *(input)* ‹*ret03_ref :: 8 word* ≡ *0x43*›
abbreviation *(input)* ‹*ret04_ref :: 8 word* ≡ *0x44*›

```
abbreviation (input) ‹ret05_ref :: 8 word ≡ 0x45›
abbreviation (input) ‹ret06_ref :: 8 word ≡ 0x46›
abbreviation (input) ‹ret07_ref :: 8 word ≡ 0x47›
abbreviation (input) ‹ret08_ref :: 8 word ≡ 0x48›
abbreviation (input) ‹ret09_ref :: 8 word ≡ 0x49›
abbreviation (input) ‹ret10_ref :: 8 word ≡ 0x4a›
abbreviation (input) ‹ret11_ref :: 8 word ≡ 0x4b›
abbreviation (input) ‹ret12_ref :: 8 word ≡ 0x4c›
abbreviation (input) ‹ret13_ref :: 8 word ≡ 0x4d›
abbreviation (input) ‹ret14_ref :: 8 word ≡ 0x4e›
abbreviation (input) ‹ret15_ref :: 8 word ≡ 0x4f›
```

— special address registers

```
abbreviation (input) ‹arg_frame_pointer_ref           :: 8 word ≡ 0x50›
abbreviation (input) ‹arg_stack_pointer_ref           :: 8 word ≡ 0x51›
abbreviation (input) ‹dynamic_data_frame_pointer_ref  :: 8 word ≡ 0x52›
abbreviation (input) ‹dynamic_data_stack_pointer_ref  :: 8 word ≡ 0x53›
abbreviation (input) ‹static_data_frame_pointer_ref   :: 8 word ≡ 0x54›
abbreviation (input) ‹static_data_stack_pointer_ref   :: 8 word ≡ 0x55›
```

— special purpose registers

```
abbreviation (input) ‹cycles_register_ref           :: 8 word ≡ 0x56›
abbreviation (input) ‹last_instruction_pointer_ref  :: 8 word ≡ 0x57›
abbreviation (input) ‹instruction_pointer_ref       :: 8 word ≡ 0x58›
abbreviation (input) ‹call_frame_pointer_ref        :: 8 word ≡ 0x59›
```

We also explicitly define a few sets of registers, to make groups of registers easier to reference in future lemmas and definitions. These are the definitions for which registers are generic, special purpose, and special address. We also define the set of all registers (which is just the union of those sets). We also define sets for which registers are readable, and writeable.

We start with the set of generic registers. They are 'generic' in that a program is able to freely read and write to them using any instruction. While they can be used for any purpose, it is intended that pointer registers (p00 to p15) are for storing pointers, constant registers (c00 to c15) are for holding constants, argument registers (arg00 to arg15) are for holding arguments to function calls, and return registers (ret00 to ret15) are for holding the return value of functions.

```
definition
  generic_registers :: ‹8 word set›
  where
    ‹generic_registers ≡ {
      r00_ref,   r01_ref,   r02_ref,   r03_ref,
      r04_ref,   r05_ref,   r06_ref,   r07_ref,
      r08_ref,   r09_ref,   r10_ref,   r11_ref,
      r12_ref,   r13_ref,   r14_ref,   r15_ref,

      p00_ref,   p01_ref,   p02_ref,   p03_ref,
      p04_ref,   p05_ref,   p06_ref,   p07_ref,
      p08_ref,   p09_ref,   p10_ref,   p11_ref,
      p12_ref,   p13_ref,   p14_ref,   p15_ref,
```

```
    c00_ref,   c01_ref,   c02_ref,   c03_ref,
    c04_ref,   c05_ref,   c06_ref,   c07_ref,
    c08_ref,   c09_ref,   c10_ref,   c11_ref,
    c12_ref,   c13_ref,   c14_ref,   c15_ref,

    arg00_ref, arg01_ref, arg02_ref, arg03_ref,
    arg04_ref, arg05_ref, arg06_ref, arg07_ref,
    arg08_ref, arg09_ref, arg10_ref, arg11_ref,
    arg12_ref, arg13_ref, arg14_ref, arg15_ref,

    ret00_ref, ret01_ref, ret02_ref, ret03_ref,
    ret04_ref, ret05_ref, ret06_ref, ret07_ref,
    ret08_ref, ret09_ref, ret10_ref, ret11_ref,
    ret12_ref, ret13_ref, ret14_ref, ret15_ref
  }›
```

Next, we define the special purpose registers. They are 'special purpose' in that the processor manages them as a side effect of execution, rather than being directly used by a program.

definition

```
  special_purpose_registers :: ‹8 word set›
  where
    ‹special_purpose_registers ≡ {
      cycles_register_ref,
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      call_frame_pointer_ref
    }›
```

Next, we define the special address registers. They are 'special addresses' in that they point to the boundaries of important pieces of memory, which a program may want to carefully maintain.

definition

```
  special_address_registers :: ‹8 word set›
  where
    ‹special_address_registers ≡ {
      arg_frame_pointer_ref,
      arg_stack_pointer_ref,
      dynamic_data_frame_pointer_ref,
      dynamic_data_stack_pointer_ref,
      static_data_frame_pointer_ref,
      static_data_stack_pointer_ref
    }›
```

Next, we define the set of registers which are readable. Here, readable means that a program is allowed to specify the register as a 'source' for an instruction, and does not refer to its physical ability to be read.

definition

```
  readable_registers :: ‹8 word set›
```

```
  where
    ‹readable_registers ≡
      generic_registers
      ∪ {cycles_register_ref}
      ∪ special_address_registers›
```

Likewise, we define the set of registers which are writeable. Here, writeable means that a program is allowed to specify the register as a 'destination' for an instruction, and does not refer to its physical ability to be written to.

```
definition
  writeable_registers :: ‹8 word set›
  where
    ‹writeable_registers ≡ generic_registers ∪ special_address_registers›
```

Last, we define the set of all registers.

```
definition
  registers :: ‹8 word set›
  where
    ‹registers ≡
      generic_registers
      ∪ special_purpose_registers
      ∪ special_address_registers›
```

The following is a record of all the flags in the Ironbark architecture. While a flag could be thought of conceptually as a one-bit register, we consider them categorically distinct for the purposes of this model and treat them accordingly. This distinction becomes more apparent in future definitions.

```
record flag_record =
  end_return    :: ‹1 word›
  end_call      :: ‹1 word›
  end_jump      :: ‹1 word›
  halt          :: ‹1 word›
  error         :: ‹1 word›
```

The following defines what the complete state of the processor is. The state is made up of the flags ('flag_state'), the registers ('register_state'), and 6 memory components (program, call, static, dynamic, input, and output memory).

Each register is indexed by its register ID (as defined using the abbreviations above), and stores a 64-bit value. Whilst the map type used here allows for registers which we have not defined to exist, such registers would not exist on a physical board. Additionally, controls in how the state changes are used to prevent reading and writing to non-existent registers.

All memory regions are indexed by 64-bit values, and each index points to a unique 64-bit value, except for program memory, which points to unique 96-bit values. Since it is almost certain that any implementation will not have access to $64 * (2^{64}) * 5 + 96 * (2^{64})$ bits of physical memory, an implementation may produce an out of memory exception where the model does not, potentially creating an exploitable gap between this model and reality. We do not address this gap.

We note that input memory is indexed by both a nat and a 64 bit value. This represents the input memory at a particular time (nat), at a particular address (64 word). The need to parameterise by time is done because we assume input memory is externally controlled.

```
record sequential_state =
  flag_state       ::  ‹flag_record›
  register_state   ::  ‹(8 word,  64 word) map›
  program_memory   ::  ‹(64 word, 96 word) map›
  call_memory      ::  ‹(64 word, 64 word) map›
  static_memory    ::  ‹(64 word, 64 word) map›
  dynamic_memory   ::  ‹(64 word, 64 word) map›
  input_memory     ::  ‹nat ⇒ ((64 word, 64 word) map)›
  output_memory    ::  ‹(64 word, 64 word) map›
```

Next, we introduce a locale to relate the operation of our processor to the 'real' time. This is intended to provide a mechanism to meaningfully talk about sequencing and synchronising events that are occuring in either multiple instances of an Ironbark processor, or an Ironbark processor and some external process. For this, we introduce the time parameter as the 'true' time.

Next, we introduce a locale to create two parameters with a type signature but no definition or value. These are used to model random numbers and the ability for the input memory to arbitrarily change without anything in the model doing anything to it. The input_memory_stream parameter essentially gives us a function from time to the input memory state at that time.

We also create 3 parameters for the duration of various instructions which are used for timing analysis. The memory_instruction_duration parameter parameter is the duration, in clock cycles, to complete a memory access instruction. The call_duration parameter is the duration, in clock cycles, to complete a call or return instruction (which includes various registers being stored to memory). The common_instruction_duration is the duration, in clock cycles, of all other instructions.

In an ideal world, memory_instruction_duration = call_duration = common_instruction_duration = 1. However, we have found that implementing hardware to do this is nontrivial, and results in a complicated hardware implementation that is difficult to understand. Thus, we generalise the design using the three parameters, which may be any fixed finite value.

```
locale Ironbark_world =
  fixes time :: ‹nat›

  fixes random_stream :: ‹nat ⇒ 64 word›
  fixes input_memory_stream :: ‹nat ⇒ (64 word, 64 word) map›

  fixes memory_instruction_duration :: ‹64 word›
  fixes call_duration :: ‹64 word›
  fixes common_instruction_duration :: ‹64 word›

begin
end
```

We note here that the exact treatment of input memory, output memory, and time is a matter of active debate. Future iterations may change their treatment. However, we also note that the

current work is mostly agnostic to the details of their implementation and should not greatly impact anything we show here. Nonetheless, when building on this work, users should be mindful of the potential change.

Finally, we define the initial state of the processor, which is with everything initialised to 0, and memory being 'empty', except for input_memory, which we bind to the input_memory_stream because it is not controlled by Ironbark.

```
definition (in Ironbark_world)
  initial_state :: ‹sequential_state›
  where
    ‹initial_state ≡
    ((|
      flag_state =
      ((|
        end_return  = 0x0,
        end_call    = 0x0,
        end_jump    = 0x0,
        halt        = 0x0,
        error       = 0x0
      |)),
      register_state = Map.empty
      (
        r00_ref   ↦ 0x00,
        r01_ref   ↦ 0x00,
        r02_ref   ↦ 0x00,
        r03_ref   ↦ 0x00,
        r04_ref   ↦ 0x00,
        r05_ref   ↦ 0x00,
        r06_ref   ↦ 0x00,
        r07_ref   ↦ 0x00,
        r08_ref   ↦ 0x00,
        r09_ref   ↦ 0x00,
        r10_ref   ↦ 0x00,
        r11_ref   ↦ 0x00,
        r12_ref   ↦ 0x00,
        r13_ref   ↦ 0x00,
        r14_ref   ↦ 0x00,
        r15_ref   ↦ 0x00,
        p00_ref   ↦ 0x00,
        p01_ref   ↦ 0x00,
        p02_ref   ↦ 0x00,
        p03_ref   ↦ 0x00,
        p04_ref   ↦ 0x00,
        p05_ref   ↦ 0x00,
        p06_ref   ↦ 0x00,
        p07_ref   ↦ 0x00,
        p08_ref   ↦ 0x00,
        p09_ref   ↦ 0x00,
        p10_ref   ↦ 0x00,
```

```
p11_ref   ↦ 0x00,
p12_ref   ↦ 0x00,
p13_ref   ↦ 0x00,
p14_ref   ↦ 0x00,
p15_ref   ↦ 0x00,
c00_ref   ↦ 0x00,
c01_ref   ↦ 0x00,
c02_ref   ↦ 0x00,
c03_ref   ↦ 0x00,
c04_ref   ↦ 0x00,
c05_ref   ↦ 0x00,
c06_ref   ↦ 0x00,
c07_ref   ↦ 0x00,
c08_ref   ↦ 0x00,
c09_ref   ↦ 0x00,
c10_ref   ↦ 0x00,
c11_ref   ↦ 0x00,
c12_ref   ↦ 0x00,
c13_ref   ↦ 0x00,
c14_ref   ↦ 0x00,
c15_ref   ↦ 0x00,
arg00_ref ↦ 0x00,
arg01_ref ↦ 0x00,
arg02_ref ↦ 0x00,
arg03_ref ↦ 0x00,
arg04_ref ↦ 0x00,
arg05_ref ↦ 0x00,
arg06_ref ↦ 0x00,
arg07_ref ↦ 0x00,
arg08_ref ↦ 0x00,
arg09_ref ↦ 0x00,
arg10_ref ↦ 0x00,
arg11_ref ↦ 0x00,
arg12_ref ↦ 0x00,
arg13_ref ↦ 0x00,
arg14_ref ↦ 0x00,
arg15_ref ↦ 0x00,
ret00_ref ↦ 0x00,
ret01_ref ↦ 0x00,
ret02_ref ↦ 0x00,
ret03_ref ↦ 0x00,
ret04_ref ↦ 0x00,
ret05_ref ↦ 0x00,
ret06_ref ↦ 0x00,
ret07_ref ↦ 0x00,
ret08_ref ↦ 0x00,
ret09_ref ↦ 0x00,
ret10_ref ↦ 0x00,
ret11_ref ↦ 0x00,
```

```
    ret12_ref ↦ 0x00,
    ret13_ref ↦ 0x00,
    ret14_ref ↦ 0x00,
    ret15_ref ↦ 0x00,
    — special purpose
    cycles_register_ref           ↦ 0x00,
    instruction_pointer_ref       ↦ 0x00,
    last_instruction_pointer_ref  ↦ 0x00,
    call_frame_pointer_ref        ↦ 0x00,
    — special address
    arg_frame_pointer_ref           ↦ 0x00,
    arg_stack_pointer_ref           ↦ 0x00,
    dynamic_data_frame_pointer_ref  ↦ 0x00,
    dynamic_data_stack_pointer_ref  ↦ 0x00,
    static_data_frame_pointer_ref   ↦ 0x00,
    static_data_stack_pointer_ref   ↦ 0x00
  ),
  program_memory  = Map.empty,
  call_memory     = Map.empty,
  static_memory   = Map.empty,
  dynamic_memory  = Map.empty,
  input_memory    = input_memory_stream,
  output_memory   = Map.empty
|))›
```

end

2.2 Defining the Ways to Manipulate the Ironbark Processor State

In this section we provide definitions for manipulating the state of an Ironbark processor. These are the lowest level functions for altering the state, which all future layers will use.

theory *state_manipulation_implementations*

imports
"Ironbark_state.state_implementations"
HOL.Option

begin

Reading a register takes a register identifier ("regID") and a state to read from as inputs, and returns the 64-bit value in the corresponding register.

Note that there is no check on the regID being a legitimate register, as a map will return 'None' if it doesn't exist by default, which is the closest to the desired behaviour as possible. However, it doesn't actually matter as all instructions check that their register operands are in the register set before execution anyway.

There is also no check if the register is readable here. This is necessary because a register is sometimes indirectly read as part of an instruction. The read-only/write-only restrictions are done

in each instruction based on the 'user' supplied register in an instruction.

definition
```
read_register :: ‹8 word ⇒ sequential_state ⇒ 64 word›
where
  ‹read_register regID state ≡ the (register_state state regID)›
```

The write register operation takes the register to write to, the 64-bit value to write to it, and the (entire) current state as inputs. It returns the (entire) state after the write is complete.

Note that while it checks that the register being written to exists here, it is not necessary, for the same reason given for reading registers. Regardless, the check is made in order to match as close as possible to how we anticipate an implementation would operate. There is also no check if the register is writeable, for the same reasons mentioned for reading registers.

definition
```
write_register :: ‹8 word ⇒ 64 word ⇒ sequential_state ⇒ sequential_state›
where
  ‹write_register regID value state ≡
    if regID ∈ registers then
      state(|register_state := (register_state state)(regID ↦ value)|)
    else
      state(|flag_state := (flag_state state) (|halt := 1, error := 1|)|)›
```

We now provide the mechanism to check if a register is readable and writeable, which is fairly self-explanatory.

definition
```
check_read_permission :: ‹8 word ⇒ bool›
where
  ‹check_read_permission regID ≡ (regID ∈ readable_registers)›
```

definition
```
check_write_permission :: ‹8 word ⇒ bool›
where
  ‹check_write_permission regID ≡ (regID ∈ writeable_registers)›
```

For reading and writing flags, rather than using a general function for all flags, we define individual functions to get, set, or clear each flag.

```
definition ‹get_end_jump    state ≡ end_jump    (flag_state state)›
definition ‹get_end_call    state ≡ end_call    (flag_state state)›
definition ‹get_end_return  state ≡ end_return  (flag_state state)›
definition ‹get_halt        state ≡ halt        (flag_state state)›
definition ‹get_error       state ≡ error       (flag_state state)›
```

— set
definition
```
set_end_jump :: ‹sequential_state ⇒ sequential_state›
where
  ‹set_end_jump state ≡ state(|flag_state := (flag_state state)(|end_jump := 1|)|)›
```

definition
```
  set_end_call :: ‹sequential_state ⇒ sequential_state›
  where
    ‹set_end_call state ≡ state(|flag_state := (flag_state state)(|end_call := 1|)|)›
```

definition
```
  set_end_return :: ‹sequential_state ⇒ sequential_state›
  where
    ‹set_end_return state ≡ state(|flag_state := (flag_state state)(|end_return := 1|)|)›
```
definition
```
  set_halt :: ‹sequential_state ⇒ sequential_state›
  where
    ‹set_halt state ≡ state(|flag_state := (flag_state state)(|halt := 1|)|)›
```

definition
```
  set_error :: ‹sequential_state ⇒ sequential_state›
  where
    ‹set_error state ≡ state(|flag_state := (flag_state state)(|error := 1|)|)›
```

— clear
definition
```
  clear_end_jump :: ‹sequential_state ⇒ sequential_state›
  where
    ‹clear_end_jump state ≡ state(|flag_state := (flag_state state)(|end_jump := 0|)|)›
```

definition
```
  clear_end_call :: ‹sequential_state ⇒ sequential_state›
  where
    ‹clear_end_call state ≡ state(|flag_state := (flag_state state)(|end_call := 0|)|)›
```

definition
```
  clear_end_return :: ‹sequential_state ⇒ sequential_state›
  where
    ‹clear_end_return state ≡ state(|flag_state := (flag_state state)(|end_return := 0|)|)›
```

definition
```
  clear_halt :: ‹sequential_state ⇒ sequential_state›
  where
    ‹clear_halt state ≡ state(|flag_state := (flag_state state)(|halt := 0|)|)›
```

definition
```
  clear_error :: ‹sequential_state ⇒ sequential_state›
  where
    ‹clear_error state ≡ state(|flag_state := (flag_state state)(|error := 0|)|)›
```

When reasoning about the state, it is sometimes useful to be able to describe certain special states, which are given below.

definition
```
  halted :: ‹sequential_state ⇒ bool›
```

where
  ‹*halted state* ≡ (*get_halt state* = *1*)›

definition
  errored :: ‹*sequential_state* ⇒ *bool*›
  where
    ‹*errored state* ≡ (*get_error state* = *1*)›

definition
  returned :: ‹*sequential_state* ⇒ *bool*›
  where
    ‹*returned state* ≡ (*get_end_return state* = *1*)›

definition
  called :: ‹*sequential_state* ⇒ *bool*›
  where
    ‹*called state* ≡ (*get_end_call state* = *1*)›

For brevity, it is often convenient to use the following definitions to describe the state of all the flag registers. These definitions describe the most common sets of flag states encountered in normal execution.

abbreviation
  ‹*typical_flags state* ≡
    get_end_return state = *0*
    ∧ *get_end_call state* = *0*
    ∧ *get_end_jump state* = *0*
    ∧ *get_halt state* = *0*
    ∧ *get_error state* = *0*›

abbreviation
  ‹*end_return_flags state* ≡
    get_end_return state = *1*
    ∧ *get_end_call state* = *0*
    ∧ *get_end_jump state* = *0*
    ∧ *get_halt state* = *0*
    ∧ *get_error state* = *0*›

abbreviation
  ‹*end_call_flags state* ≡
    get_end_return state = *0*
    ∧ *get_end_call state* = *1*
    ∧ *get_end_jump state* = *0*
    ∧ *get_halt state* = *0*
    ∧ *get_error state* = *0*›

abbreviation
  ‹*end_jump_flags state* ≡
    get_end_return state = *0*
    ∧ *get_end_call state* = *0*

```
    ∧ get_end_jump  state = 1
    ∧ get_halt      state = 0
    ∧ get_error     state = 0›
```

abbreviation

```
  ‹halt_flags state ≡
    get_end_return  state = 0
    ∧ get_end_call  state = 0
    ∧ get_end_jump  state = 0
    ∧ get_halt      state = 1
    ∧ get_error     state = 0›
```

abbreviation

```
  ‹error_flags state ≡
    get_end_return  state = 0
    ∧ get_end_call  state = 0
    ∧ get_end_jump  state = 0
    ∧ get_halt      state = 1
    ∧ get_error     state = 1›
```

Next, we describe reading and writing to memory. We assume that memory is perfect, and that all addressable spaces exist. As noted earlier, this is almost certainly not the case in reality, and is a gap between our model and reality.

Note that even though program memory region is not writeable, we still define the function that would write to it, because we expect that the implementation of it would use a media which can physically be written to. However, we will later prove that there is no sequence of instructions which can result in an Ironbark processor writing to program memory.

definition

```
  read_program_memory :: ‹64 word ⇒ sequential_state ⇒ 96 word›
  where
    ‹read_program_memory address state ≡ the (program_memory state address)›
```

definition

```
  write_program_memory :: ‹64 word ⇒ 96 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹write_program_memory address value state
    ≡ state⦇program_memory := (program_memory state)(address ↦ value)⦈›
```

definition

```
  read_call_memory :: ‹64 word ⇒ sequential_state ⇒ 64 word›
  where
    ‹read_call_memory address state ≡ the (call_memory state address)›
```

definition

```
  write_call_memory :: ‹64 word ⇒ 64 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹write_call_memory address value state
    ≡ state⦇call_memory := (call_memory state)(address ↦ value)⦈›
```

definition
```
read_static_memory :: ‹64 word ⇒ sequential_state ⇒ 64 word›
where
  ‹read_static_memory address state ≡ the (static_memory state address)›
```

definition
```
write_static_memory :: ‹64 word ⇒ 64 word ⇒ sequential_state ⇒ sequential_state›
where
  ‹write_static_memory address value state
  ≡ state⦇static_memory := (static_memory state)(address ↦ value)⦈›
```

definition
```
read_dynamic_memory :: ‹64 word ⇒ sequential_state ⇒ 64 word›
where
  ‹read_dynamic_memory address state ≡ the (dynamic_memory state address)›
```

definition
```
write_dynamic_memory :: ‹64 word ⇒ 64 word ⇒ sequential_state ⇒ sequential_state›
where
  ‹write_dynamic_memory address value state
  ≡ state⦇dynamic_memory := (dynamic_memory state)(address ↦ value)⦈›
```

We only provide definitions for reading input memory, and writing output memory, as we expect these to actually be physically implemented in a read-only and write-only manner.

definition *(in Ironbark_world)*
```
read_input_memory :: ‹64 word ⇒ sequential_state ⇒ 64 word›
where
  ‹read_input_memory address state ≡ the (input_memory state time address)›
```

definition
```
write_output_memory :: ‹64 word ⇒ 64 word ⇒ sequential_state ⇒ sequential_state›
where
  ‹write_output_memory address value state
  ≡ state⦇output_memory := (output_memory state)(address ↦ value)⦈›
```

For program memory, we also define a few helper definitions for extracting the different parts of an instruction.

definition
```
get_opcode :: ‹96 word ⇒ 8 word›
where
  ‹get_opcode instr ≡ slice 88 instr›
```

definition
```
get_immediate :: ‹96 word ⇒ 64 word›
where
  ‹get_immediate instr ≡ ucast (take_bit 64 (slice 0 instr)::64 word)›
```

definition

```
  get_reg1 :: ‹96 word ⇒ 8 word›
  where
    ‹get_reg1 instr ≡ ucast (take_bit 8 (slice 80 instr)::16 word)›
```

definition

```
  get_reg2 :: ‹96 word ⇒ 8 word›
  where
    ‹get_reg2 instr ≡ ucast (take_bit 8 (slice 72 instr)::24 word)›
```

definition

```
  get_reg3 :: ‹96 word ⇒ 8 word›
  where
    ‹get_reg3 instr ≡ ucast (take_bit 8 (slice 64 instr)::32 word)›
```

The following two definitions are specific state manipulations that are used across many instructions, and so are placed here for re-usability across instructions.

Standard error just sets the halt and error flags in the state.

definition

```
  standard_error :: ‹sequential_state ⇒ sequential_state›
  where
  ‹standard_error state
  ≡ set_halt (set_error state)›
```

Standard post instruction updates the cycles register, last instruction pointer, and the instruction pointer. Note that branch instructions will not use the standard post instruction because it would overwrite the instruction pointer that they set.

definition

```
  standard_post_instruction :: ‹64 word ⇒ sequential_state ⇒ sequential_state›
  where
  ‹standard_post_instruction instruction_duration state ≡
    let
      — update cycles register
      state1 =
        write_register
          cycles_register_ref
          ((read_register cycles_register_ref state) + instruction_duration)
          state;
      — update that this was the last instruction
      state2 =
        write_register
          last_instruction_pointer_ref
          (read_register instruction_pointer_ref state1)
          state1
    in
      — move instruction pointer to the next instruction
      write_register
        instruction_pointer_ref
        ((read_register instruction_pointer_ref state2) + 0x1)
```

```
    state2›
```

These last two definitions are provided as a matter of convenience for the definitions of the call and return instructions, respectively. While their definitions are long, their operations can be described quite simply. The backup_registers_before_call function backs up the state of various registers to call memory, and restore_registers_after_return restores those same registers from call memory.

definition

```
  backup_registers_before_call :: ‹sequential_state ⇒ sequential_state›
  where
    ‹backup_registers_before_call state ≡
      state
      (|
        call_memory :=
          (call_memory state)
          (
            (read_register call_frame_pointer_ref state) + 66
              ↦ (read_register r00_ref  state),
            (read_register call_frame_pointer_ref state) + 65
              ↦ (read_register r01_ref  state),
            (read_register call_frame_pointer_ref state) + 64
              ↦ (read_register r02_ref  state),
            (read_register call_frame_pointer_ref state) + 63
              ↦ (read_register r03_ref  state),
            (read_register call_frame_pointer_ref state) + 62
              ↦ (read_register r04_ref  state),
            (read_register call_frame_pointer_ref state) + 61
              ↦ (read_register r05_ref  state),
            (read_register call_frame_pointer_ref state) + 60
              ↦ (read_register r06_ref  state),
            (read_register call_frame_pointer_ref state) + 59
              ↦ (read_register r07_ref  state),
            (read_register call_frame_pointer_ref state) + 58
              ↦ (read_register r08_ref  state),
            (read_register call_frame_pointer_ref state) + 57
              ↦ (read_register r09_ref  state),
            (read_register call_frame_pointer_ref state) + 56
              ↦ (read_register r10_ref  state),
            (read_register call_frame_pointer_ref state) + 55
              ↦ (read_register r11_ref  state),
            (read_register call_frame_pointer_ref state) + 54
              ↦ (read_register r12_ref  state),
            (read_register call_frame_pointer_ref state) + 53
              ↦ (read_register r13_ref  state),
            (read_register call_frame_pointer_ref state) + 52
              ↦ (read_register r14_ref  state),
            (read_register call_frame_pointer_ref state) + 51
              ↦ (read_register r15_ref  state),
            (read_register call_frame_pointer_ref state) + 50
              ↦ (read_register p00_ref  state),
```

```
(read_register call_frame_pointer_ref state) + 49
  ↦ (read_register p01_ref  state),
(read_register call_frame_pointer_ref state) + 48
  ↦ (read_register p02_ref  state),
(read_register call_frame_pointer_ref state) + 47
  ↦ (read_register p03_ref  state),
(read_register call_frame_pointer_ref state) + 46
  ↦ (read_register p04_ref  state),
(read_register call_frame_pointer_ref state) + 45
  ↦ (read_register p05_ref  state),
(read_register call_frame_pointer_ref state) + 44
  ↦ (read_register p06_ref  state),
(read_register call_frame_pointer_ref state) + 43
  ↦ (read_register p07_ref  state),
(read_register call_frame_pointer_ref state) + 42
  ↦ (read_register p08_ref  state),
(read_register call_frame_pointer_ref state) + 41
  ↦ (read_register p09_ref  state),
(read_register call_frame_pointer_ref state) + 40
  ↦ (read_register p10_ref  state),
(read_register call_frame_pointer_ref state) + 39
  ↦ (read_register p11_ref  state),
(read_register call_frame_pointer_ref state) + 38
  ↦ (read_register p12_ref  state),
(read_register call_frame_pointer_ref state) + 37
  ↦ (read_register p13_ref  state),
(read_register call_frame_pointer_ref state) + 36
  ↦ (read_register p14_ref  state),
(read_register call_frame_pointer_ref state) + 35
  ↦ (read_register p15_ref  state),
(read_register call_frame_pointer_ref state) + 34
  ↦ (read_register c00_ref  state),
(read_register call_frame_pointer_ref state) + 33
  ↦ (read_register c01_ref  state),
(read_register call_frame_pointer_ref state) + 32
  ↦ (read_register c02_ref  state),
(read_register call_frame_pointer_ref state) + 31
  ↦ (read_register c03_ref  state),
(read_register call_frame_pointer_ref state) + 30
  ↦ (read_register c04_ref  state),
(read_register call_frame_pointer_ref state) + 29
  ↦ (read_register c05_ref  state),
(read_register call_frame_pointer_ref state) + 28
  ↦ (read_register c06_ref  state),
(read_register call_frame_pointer_ref state) + 27
  ↦ (read_register c07_ref  state),
(read_register call_frame_pointer_ref state) + 26
  ↦ (read_register c08_ref  state),
(read_register call_frame_pointer_ref state) + 25
```

```
  ↦ (read_register c09_ref  state),
(read_register call_frame_pointer_ref state) + 24
  ↦ (read_register c10_ref  state),
(read_register call_frame_pointer_ref state) + 23
  ↦ (read_register c11_ref  state),
(read_register call_frame_pointer_ref state) + 22
  ↦ (read_register c12_ref  state),
(read_register call_frame_pointer_ref state) + 21
  ↦ (read_register c13_ref  state),
(read_register call_frame_pointer_ref state) + 20
  ↦ (read_register c14_ref  state),
(read_register call_frame_pointer_ref state) + 19
  ↦ (read_register c15_ref  state),
(read_register call_frame_pointer_ref state) + 18
  ↦ (read_register arg00_ref  state),
(read_register call_frame_pointer_ref state) + 17
  ↦ (read_register arg01_ref  state),
(read_register call_frame_pointer_ref state) + 16
  ↦ (read_register arg02_ref  state),
(read_register call_frame_pointer_ref state) + 15
  ↦ (read_register arg03_ref  state),
(read_register call_frame_pointer_ref state) + 14
  ↦ (read_register arg04_ref  state),
(read_register call_frame_pointer_ref state) + 13
  ↦ (read_register arg05_ref  state),
(read_register call_frame_pointer_ref state) + 12
  ↦ (read_register arg06_ref  state),
(read_register call_frame_pointer_ref state) + 11
  ↦ (read_register arg07_ref  state),
(read_register call_frame_pointer_ref state) + 10
  ↦ (read_register arg08_ref  state),
(read_register call_frame_pointer_ref state) +  9
  ↦ (read_register arg09_ref  state),
(read_register call_frame_pointer_ref state) +  8
  ↦ (read_register arg10_ref  state),
(read_register call_frame_pointer_ref state) +  7
  ↦ (read_register arg11_ref  state),
(read_register call_frame_pointer_ref state) +  6
  ↦ (read_register arg12_ref  state),
(read_register call_frame_pointer_ref state) +  5
  ↦ (read_register arg13_ref  state),
(read_register call_frame_pointer_ref state) +  4
  ↦ (read_register arg14_ref  state),
(read_register call_frame_pointer_ref state) +  3
  ↦ (read_register arg15_ref  state),
(read_register call_frame_pointer_ref state) +  2
  ↦ (read_register static_data_frame_pointer_ref state),
(read_register call_frame_pointer_ref state) +  1
  ↦ (read_register static_data_stack_pointer_ref state),
```

```
        (read_register call_frame_pointer_ref state) +  0
          ↦ (read_register instruction_pointer_ref state)
      )
  ⦆›

definition
  restore_registers_after_return :: ‹sequential_state ⇒ sequential_state›
  where
    ‹restore_registers_after_return state ≡
      state
      (|
        register_state :=
          (register_state state)
          (
            r00_ref ↦
              read_call_memory
                ((read_register call_frame_pointer_ref state) -  1)
                state,
            r01_ref ↦
              read_call_memory
                ((read_register call_frame_pointer_ref state) -  2)
                state,
            r02_ref ↦
              read_call_memory
                ((read_register call_frame_pointer_ref state) -  3)
                state,
            r03_ref ↦
              read_call_memory
                ((read_register call_frame_pointer_ref state) -  4)
                state,
            r04_ref ↦
              read_call_memory
                ((read_register call_frame_pointer_ref state) -  5)
                state,
            r05_ref ↦
              read_call_memory
                ((read_register call_frame_pointer_ref state) -  6)
                state,
            r06_ref ↦
              read_call_memory
                ((read_register call_frame_pointer_ref state) -  7)
                state,
            r07_ref ↦
              read_call_memory
                ((read_register call_frame_pointer_ref state) -  8)
                state,
            r08_ref ↦
              read_call_memory
                ((read_register call_frame_pointer_ref state) -  9)
```

```
    state,
r09_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 10)
    state,
r10_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 11)
    state,
r11_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 12)
    state,
r12_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 13)
    state,
r13_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 14)
    state,
r14_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 15)
    state,
r15_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 16)
    state,
p00_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 17)
    state,
p01_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 18)
    state,
p02_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 19)
    state,
p03_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 20)
    state,
p04_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 21)
    state,
```

```
p05_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 22)
    state,
p06_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 23)
    state,
p07_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 24)
    state,
p08_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 25)
    state,
p09_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 26)
    state,
p10_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 27)
    state,
p11_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 28)
    state,
p12_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 29)
    state,
p13_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 30)
    state,
p14_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 31)
    state,
p15_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 32)
    state,
c00_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 33)
    state,
c01_ref ↦
```

```
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 34)
    state,
c02_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 35)
    state,
c03_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 36)
    state,
c04_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 37)
    state,
c05_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 38)
    state,
c06_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 39)
    state,
c07_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 40)
    state,
c08_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 41)
    state,
c09_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 42)
    state,
c10_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 43)
    state,
c11_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 44)
    state,
c12_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 45)
    state,
c13_ref ↦
  read_call_memory
```

```
    ((read_register call_frame_pointer_ref state) - 46)
    state,
c14_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 47)
    state,
c15_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 48)
    state,
arg00_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 49)
    state,
arg01_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 50)
    state,
arg02_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 51)
    state,
arg03_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 52)
    state,
arg04_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 53)
    state,
arg05_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 54)
    state,
arg06_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 55)
    state,
arg07_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 56)
    state,
arg08_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 57)
    state,
arg09_ref ↦
  read_call_memory
    ((read_register call_frame_pointer_ref state) - 58)
```

```
        state,
      arg10_ref ↦
        read_call_memory
          ((read_register call_frame_pointer_ref state) - 59)
          state,
      arg11_ref ↦
        read_call_memory
          ((read_register call_frame_pointer_ref state) - 60)
          state,
      arg12_ref ↦
        read_call_memory
          ((read_register call_frame_pointer_ref state) - 61)
          state,
      arg13_ref ↦
        read_call_memory
          ((read_register call_frame_pointer_ref state) - 62)
          state,
      arg14_ref ↦
        read_call_memory
          ((read_register call_frame_pointer_ref state) - 63)
          state,
      arg15_ref ↦
        read_call_memory
          ((read_register call_frame_pointer_ref state) - 64)
          state,
      static_data_frame_pointer_ref ↦
        read_call_memory
          ((read_register call_frame_pointer_ref state) - 65)
          state,
      static_data_stack_pointer_ref ↦
        read_call_memory
          ((read_register call_frame_pointer_ref state) - 66)
          state,
      instruction_pointer_ref ↦
        read_call_memory
          ((read_register call_frame_pointer_ref state) - 67)
          state
    )
⦇⦈⇩
```

end

2.3 Defining the Ironbark Instructions

theory *instruction_implementations*

imports
"Ironbark_state_manipulation.state_manipulation_implementations"
"HOL-Library.Monad_Syntax"

begin

This file describes all the instructions in the Ironbark processor.

We begin by assigning each instruction that we will define an opcode. We also try to group them into groups of related functionality. Note that 0x00 and 0xFF are errors. This is done so that when there is no program memory attached, the processor will error (assuming that reading from something that isn't there gives all-0s or all-1s, which we expect to be the most likely behaviour of any hardware implementation).

```
abbreviation (input) <ERROR0_opcode              :: 8 word ≡ 0x00>
abbreviation (input) <NOP_opcode                 :: 8 word ≡ 0x01>
— data movement
abbreviation (input) <LOAD_IMMEDIATE_opcode      :: 8 word ≡ 0x02>
abbreviation (input) <LOAD_STATIC_DATA_opcode    :: 8 word ≡ 0x03>
abbreviation (input) <STORE_STATIC_DATA_opcode   :: 8 word ≡ 0x04>
abbreviation (input) <LOAD_DYNAMIC_DATA_opcode   :: 8 word ≡ 0x05>
abbreviation (input) <STORE_DYNAMIC_DATA_opcode  :: 8 word ≡ 0x06>
abbreviation (input) <LOAD_INPUT_DATA_opcode     :: 8 word ≡ 0x07>
abbreviation (input) <STORE_OUTPUT_DATA_opcode   :: 8 word ≡ 0x08>
abbreviation (input) <COPY_opcode                :: 8 word ≡ 0x09>
— arithmetic
abbreviation (input) <ADD_opcode                 :: 8 word ≡ 0x0A>
abbreviation (input) <SUBTRACT_opcode            :: 8 word ≡ 0x0B>
abbreviation (input) <SHIFT_LEFT_opcode          :: 8 word ≡ 0x0C>
abbreviation (input) <SHIFT_RIGHT_opcode         :: 8 word ≡ 0x0D>
abbreviation (input) <BITWISE_AND_opcode         :: 8 word ≡ 0x0E>
abbreviation (input) <BITWISE_OR_opcode          :: 8 word ≡ 0x0F>
abbreviation (input) <BITWISE_XOR_opcode         :: 8 word ≡ 0x10>
abbreviation (input) <BITWISE_NAND_opcode        :: 8 word ≡ 0x11>
abbreviation (input) <BITWISE_NOT_opcode         :: 8 word ≡ 0x12>
abbreviation (input) <LESS_THAN_opcode           :: 8 word ≡ 0x13>
abbreviation (input) <GREATER_THAN_opcode        :: 8 word ≡ 0x14>
abbreviation (input) <EQUALS_opcode              :: 8 word ≡ 0x15>
abbreviation (input) <NOT_EQUALS_opcode          :: 8 word ≡ 0x16>
abbreviation (input) <RANDOMISE_opcode           :: 8 word ≡ 0x17>
— control flow
abbreviation (input) <END_JUMP_opcode            :: 8 word ≡ 0x18>
abbreviation (input) <END_JUMP_STRICT_opcode     :: 8 word ≡ 0x19>
abbreviation (input) <JUMP_opcode                :: 8 word ≡ 0x1A>
abbreviation (input) <CONDITIONAL_JUMP_opcode    :: 8 word ≡ 0x1B>
abbreviation (input) <END_CALL_opcode            :: 8 word ≡ 0x1C>
abbreviation (input) <CALL_opcode                :: 8 word ≡ 0x1D>
abbreviation (input) <END_RETURN_opcode          :: 8 word ≡ 0x1E>
abbreviation (input) <RETURN_opcode              :: 8 word ≡ 0x1F>
— misc
abbreviation (input) <HALT_opcode                :: 8 word ≡ 0x20>
abbreviation (input) <ERROR1_opcode              :: 8 word ≡ 0xFF>
```

Finally, we define what the behaviour is for each instruction. They are defined below in the same order as the opcodes given above. Most instructions are of the form:

```
if (guards) then
  do instruction
else
  error
```

```
definition
  ERROR0 :: ‹sequential_state ⇒ sequential_state›
  where
    ‹ERROR0 state ≡ (standard_error state)›

definition (in Ironbark_world)
  NOP :: ‹sequential_state ⇒ sequential_state›
  where
    ‹NOP state ≡
      if (typical_flags state) then
        standard_post_instruction common_instruction_duration state
      else
        standard_error state›

definition (in Ironbark_world)
  LOAD_IMMEDIATE :: ‹8 word ⇒ 64 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹LOAD_IMMEDIATE reg1 immediate state ≡
      if
        check_write_permission reg1
        ∧ typical_flags state
      then
        let
          state1 = write_register reg1 immediate state
        in
          standard_post_instruction common_instruction_duration state1
      else
        standard_error state›

definition (in Ironbark_world)
  LOAD_STATIC_DATA :: ‹8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹LOAD_STATIC_DATA reg1 reg2 state ≡
      if
        check_write_permission reg1
        ∧ check_read_permission reg2
        ∧ typical_flags state
      then
        let
          state1 =
            write_register
```

```
            reg1
            (read_static_memory (read_register reg2 state) state)
            state
      in
        standard_post_instruction memory_instruction_duration state1
    else
      standard_error state›

definition (in Ironbark_world)
  STORE_STATIC_DATA :: ‹8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹STORE_STATIC_DATA reg1 reg2 state ≡
      if
        check_read_permission reg1
        ∧ check_read_permission reg2
        ∧ typical_flags state
      then
        let
          state1 =
            write_static_memory
              (read_register reg1 state)
              (read_register reg2 state)
              state
        in
          standard_post_instruction memory_instruction_duration state1
      else
        standard_error state›

definition (in Ironbark_world)
  LOAD_DYNAMIC_DATA :: ‹8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹LOAD_DYNAMIC_DATA reg1 reg2 state ≡
      if
        check_write_permission reg1
        ∧ check_read_permission reg2
        ∧ typical_flags state
      then
        let
          state1 =
            write_register
              reg1
              (read_dynamic_memory (read_register reg2 state) state)
              state
        in
          standard_post_instruction memory_instruction_duration state1
      else
        standard_error state›

definition (in Ironbark_world)
```

```
  STORE_DYNAMIC_DATA :: <8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state>
  where
    <STORE_DYNAMIC_DATA reg1 reg2 state ≡
      if
        check_read_permission reg1
        ∧ check_read_permission reg2
        ∧ typical_flags state
      then
        let
          state1 =
            write_dynamic_memory
              (read_register reg1 state)
              (read_register reg2 state)
              state
        in
          standard_post_instruction memory_instruction_duration state1
      else
        standard_error state>

definition (in Ironbark_world)
  LOAD_INPUT_DATA :: <8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state>
  where
    <LOAD_INPUT_DATA reg1 reg2 state ≡
      if
        check_write_permission reg1
        ∧ check_read_permission reg2
        ∧ typical_flags state
      then
        let
          state1 =
            write_register
              reg1
              (read_input_memory (read_register reg2 state) state)
              state
        in
          standard_post_instruction memory_instruction_duration state1
      else
        standard_error state>

definition (in Ironbark_world)
  STORE_OUTPUT_DATA :: <8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state>
  where
    <STORE_OUTPUT_DATA reg1 reg2 state ≡
      if
        check_read_permission reg1
        ∧ check_read_permission reg2
        ∧ typical_flags state
      then
        let
```

```
      state1 =
        write_output_memory
          (read_register reg1 state)
          (read_register reg2 state)
          state
    in
      standard_post_instruction memory_instruction_duration state1
  else
    standard_error state›

definition (in Ironbark_world)
  COPY :: ‹8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹COPY reg1 reg2 state ≡
      if
        check_write_permission reg1
        ∧ check_read_permission reg2
        ∧ typical_flags state
      then
        let
          state1 =
            write_register
              reg1
              (read_register reg2 state)
              state
        in
          standard_post_instruction common_instruction_duration state1
      else
        standard_error state›

definition (in Ironbark_world)
  ADD :: ‹8 word ⇒ 8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹ADD reg1 reg2 reg3 state ≡
      if
        check_write_permission reg1
        ∧ check_read_permission reg2
        ∧ check_read_permission reg3
        ∧ typical_flags state
      then
        let
          state1 =
            write_register
              reg1
              ((read_register reg2 state) + (read_register reg3 state))
              state
        in
          standard_post_instruction common_instruction_duration state1
      else
```

```
    standard_error state›

definition (in Ironbark_world)
  SUBTRACT :: ‹8 word ⇒ 8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹SUBTRACT reg1 reg2 reg3 state ≡
      if
        check_write_permission reg1
        ∧ check_read_permission reg2
        ∧ check_read_permission reg3
        ∧ typical_flags state
      then
        let
          state1 =
            write_register
              reg1
              ((read_register reg2 state) - (read_register reg3 state))
              state
        in
          standard_post_instruction common_instruction_duration state1
      else
        standard_error state›

definition (in Ironbark_world)
  SHIFT_LEFT :: ‹8 word ⇒ 8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹SHIFT_LEFT reg1 reg2 reg3 state ≡
      if
        check_write_permission reg1
        ∧ check_read_permission reg2
        ∧ check_read_permission reg3
        ∧ typical_flags state
      then
        let
          state1 =
            write_register
              reg1
              (push_bit (unat (read_register reg3 state)) (read_register reg2 state))
              state
        in
          standard_post_instruction common_instruction_duration state1
      else
        standard_error state›

definition (in Ironbark_world)
  SHIFT_RIGHT :: ‹8 word ⇒ 8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹SHIFT_RIGHT reg1 reg2 reg3 state ≡
      if
```

```
    check_write_permission reg1
    ∧ check_read_permission reg2
    ∧ check_read_permission reg3
    ∧ typical_flags state
  then
    let
      state1 =
        write_register
          reg1
          (drop_bit (unat (read_register reg3 state)) (read_register reg2 state))
          state
    in
      standard_post_instruction common_instruction_duration state1
  else
    standard_error state›
```

definition (**in** *Ironbark_world*)

```
  BITWISE_AND :: ‹8 word ⇒ 8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹BITWISE_AND reg1 reg2 reg3 state ≡
      if
        check_write_permission reg1
        ∧ check_read_permission reg2
        ∧ check_read_permission reg3
        ∧ typical_flags state
      then
        let
          state1 =
            write_register
              reg1
              (and (read_register reg2 state) (read_register reg3 state))
              state
        in
          standard_post_instruction common_instruction_duration state1
      else
        standard_error state›
```

definition (**in** *Ironbark_world*)

```
  BITWISE_OR :: ‹8 word ⇒ 8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹BITWISE_OR reg1 reg2 reg3 state ≡
      if
        check_write_permission reg1
        ∧ check_read_permission reg2
        ∧ check_read_permission reg3
        ∧ typical_flags state
      then
        let
          state1 =
```

```
        write_register
          reg1
          (or (read_register reg2 state) (read_register reg3 state))
          state
    in
      standard_post_instruction common_instruction_duration state1
  else
    standard_error state›

definition (in Ironbark_world)
  BITWISE_XOR :: ‹8 word ⇒ 8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹BITWISE_XOR reg1 reg2 reg3 state ≡
      if
        check_write_permission reg1
        ∧ check_read_permission reg2
        ∧ check_read_permission reg3
        ∧ typical_flags state
      then
        let
          state1 =
            write_register
              reg1
              (xor (read_register reg2 state) (read_register reg3 state))
              state
        in
          standard_post_instruction common_instruction_duration state1
      else
        standard_error state›

definition (in Ironbark_world)
  BITWISE_NAND :: ‹8 word ⇒ 8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹BITWISE_NAND reg1 reg2 reg3 state ≡
      if
        check_write_permission reg1
        ∧ check_read_permission reg2
        ∧ check_read_permission reg3
        ∧ typical_flags state
      then
        let
          state1 =
            write_register
              reg1
              (not (and (read_register reg2 state) (read_register reg3 state)))
              state
        in
          standard_post_instruction common_instruction_duration state1
      else
```

```
    standard_error state›

definition (in Ironbark_world)
  BITWISE_NOT :: ‹8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹BITWISE_NOT reg1 reg2 state ≡
      if
        check_write_permission reg1
        ∧ check_read_permission reg2
        ∧ typical_flags state
      then
        let
          state1 =
            write_register
              reg1
              (not (read_register reg2 state))
              state
        in
          standard_post_instruction common_instruction_duration state1
      else
        standard_error state›

definition (in Ironbark_world)
  LESS_THAN :: ‹8 word ⇒ 8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹LESS_THAN reg1 reg2 reg3 state ≡
      if
        check_write_permission reg1
        ∧ check_read_permission reg2
        ∧ check_read_permission reg3
        ∧ typical_flags state
      then
        if (read_register reg2 state < read_register reg3 state) then
          let
            state1 = write_register reg1 0x01 state
          in
            standard_post_instruction common_instruction_duration state1
        else
          let
            state1 = write_register reg1 0x00 state
          in
            standard_post_instruction common_instruction_duration state1
      else
        standard_error state›

definition (in Ironbark_world)
  GREATER_THAN :: ‹8 word ⇒ 8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹GREATER_THAN reg1 reg2 reg3 state ≡
```

```
    if
      check_write_permission reg1
      ∧ check_read_permission reg2
      ∧ check_read_permission reg3
      ∧ typical_flags state
    then
      if (read_register reg2 state > read_register reg3 state) then
        let
          state1 = write_register reg1 0x01 state
        in
          standard_post_instruction common_instruction_duration state1
      else
        let
          state1 = write_register reg1 0x00 state
        in
          standard_post_instruction common_instruction_duration state1
    else
      standard_error state›
```

definition (**in** *Ironbark_world*)

```
  EQUALS :: ‹8 word ⇒ 8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹EQUALS reg1 reg2 reg3 state ≡
      if
        check_write_permission reg1
        ∧ check_read_permission reg2
        ∧ check_read_permission reg3
        ∧ typical_flags state
      then
        if (read_register reg2 state = read_register reg3 state) then
          let
            state1 = write_register reg1 0x01 state
          in
            standard_post_instruction common_instruction_duration state1
        else
          let
            state1 = write_register reg1 0x00 state
          in
            standard_post_instruction common_instruction_duration state1
      else
        standard_error state›
```

definition (**in** *Ironbark_world*)

```
  NOT_EQUALS :: ‹8 word ⇒ 8 word ⇒ 8 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹NOT_EQUALS reg1 reg2 reg3 state ≡
      if
        check_write_permission reg1
        ∧ check_read_permission reg2
```

```
    ∧ check_read_permission reg3
    ∧ typical_flags state
  then
    if (read_register reg2 state = read_register reg3 state) then
      let
        state1 = write_register reg1 0x00 state
      in
        standard_post_instruction common_instruction_duration state1
    else
      let
        state1 = write_register reg1 0x01 state
      in
        standard_post_instruction common_instruction_duration state1
  else
    standard_error state›
```

definition (**in** *Ironbark_world*)

```
  RANDOMISE :: ‹8 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹RANDOMISE reg1 state ≡
      if
        check_write_permission reg1
        ∧ typical_flags state
      then
        let
          state1 =
            write_register
              reg1
              (random_stream time)
              state
        in
          standard_post_instruction common_instruction_duration state1
      else
        standard_error state›
```

definition (**in** *Ironbark_world*)

```
  END_JUMP :: ‹64 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹END_JUMP immediate state ≡
      if
        get_error         state = 0x0
        ∧ get_halt        state = 0x0
        ∧ get_end_call    state = 0x0
        ∧ get_end_return  state = 0x0
      then
        if
          get_end_jump state = 0
          ∨ (read_register last_instruction_pointer_ref state = immediate)
        then
```

```
      let
        state1 = clear_end_jump state
      in
        standard_post_instruction common_instruction_duration state1
    else
      standard_error state
  else
    standard_error state›
```

definition (**in** *Ironbark_world*)

```
  END_JUMP_STRICT :: ‹64 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹END_JUMP_STRICT immediate state ≡
      if
        end_jump_flags state
        ∧ (read_register last_instruction_pointer_ref state = immediate)
      then
        let
          state1 = clear_end_jump state
        in
          standard_post_instruction common_instruction_duration state1
      else
        standard_error state›
```

definition (**in** *Ironbark_world*)

```
  JUMP :: ‹64 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹JUMP immediate state ≡
      if (typical_flags state) then
        let
          state1 = set_end_jump state;
          state2 =
            write_register
              last_instruction_pointer_ref
              (read_register instruction_pointer_ref state1)
              state1;
          state3 =
            write_register
              instruction_pointer_ref
              immediate
              state2
        in
          write_register
            cycles_register_ref
            ((read_register cycles_register_ref state3) + common_instruction_duration)
            state3
      else
        standard_error state›
```

```
definition (in Ironbark_world)
  CONDITIONAL_JUMP :: ‹8 word ⇒ 64 word ⇒ sequential_state ⇒ sequential_state›
  where
    ‹CONDITIONAL_JUMP reg1 immediate state ≡
      if
        check_read_permission reg1
        ∧ typical_flags state
      then
        if (read_register reg1 state = 0x00) then
          standard_post_instruction common_instruction_duration state
        else
          let
            state1 = set_end_jump state;
            state2 =
              write_register
                last_instruction_pointer_ref
                (read_register instruction_pointer_ref state1)
                state1;
            state3 =
              write_register
                instruction_pointer_ref
                immediate
                state2
          in
            write_register
              cycles_register_ref
              ((read_register cycles_register_ref state3)
                + common_instruction_duration)
              state3
      else
        standard_error state›


definition (in Ironbark_world)
  END_CALL :: ‹sequential_state ⇒ sequential_state›
  where
    ‹END_CALL state ≡
      if end_call_flags state then
        let
          state1 = clear_end_call state
        in
          standard_post_instruction common_instruction_duration state1
      else
        standard_error state›

definition (in Ironbark_world)
  CALL :: ‹64 word ⇒ sequential_state ⇒ sequential_state›
  where
```

```
  ‹CALL immediate state ≡
    if (typical_flags state) then
      let
        state1 = set_end_call state;
        state2 = backup_registers_before_call state1;
        state3 =
          write_register
            call_frame_pointer_ref
            ((read_register call_frame_pointer_ref state2) + 67)
            state2;
        state4 =
          write_register
            last_instruction_pointer_ref
            (read_register instruction_pointer_ref state3)
            state3;
        state5 =
          write_register
            instruction_pointer_ref
            immediate
            state4
      in
        write_register
          cycles_register_ref
          ((read_register cycles_register_ref state5) + call_duration)
          state5
    else
      standard_error state›
```

definition (**in** *Ironbark_world*)

```
  END_RETURN :: ‹64 word ⇒ sequential_state ⇒ sequential_state›
```

where

```
    ‹END_RETURN immediate state ≡
      if
        end_return_flags state
        ∧ (read_register last_instruction_pointer_ref state = immediate)
      then
        let
          state1 = clear_end_return state
        in
          standard_post_instruction common_instruction_duration state1
      else
        standard_error state›
```

definition (**in** *Ironbark_world*)

```
  RETURN :: ‹sequential_state ⇒ sequential_state›
```

where

```
    ‹RETURN state ≡
      if (typical_flags state) then
        let
```

```
      state1 = set_end_return state;
      state2 =
        write_register
          last_instruction_pointer_ref
          (read_register instruction_pointer_ref state1)
          state1;
      state3 = restore_registers_after_return state2;
      state4 =
        write_register
          call_frame_pointer_ref
          ((read_register call_frame_pointer_ref state3) - 67)
          state3;
      state5 =
        write_register
          instruction_pointer_ref
          ((read_register instruction_pointer_ref state4) + 1)
          state4
    in
      write_register
        cycles_register_ref
        ((read_register cycles_register_ref state5) + call_duration)
        state5
  else
    standard_error state›
```

```
definition (in Ironbark_world)
  HALT :: ‹sequential_state ⇒ sequential_state›
  where
    ‹HALT state ≡
      if (typical_flags state) then
        set_halt state
      else
        standard_error state›
```

```
definition
  ERROR1 :: ‹sequential_state ⇒ sequential_state›
  where
    ‹ERROR1 state ≡ (standard_error state)›
```

The following definition is for an 'illegal' instruction. An illegal instruction is executed whenever the opcode is not one of the defined opcodes. The exact operation is made apparent when viewing the definition of execute_instruction

```
definition (in Ironbark_world)
  ILLEGAL :: ‹sequential_state ⇒ sequential_state›
  where
    ‹ILLEGAL state ≡ standard_error state›
```

end

2.4 Defining Sequential Execution

theory *execution_implementations*

imports
"Ironbark_instruction.instruction_implementations"

begin

Execution is done in a fetch-decode-execute loop. We start by defining each of the steps.

Fetch is done by simply accessing of the program memory at the current instruction pointer.

definition
```
fetch_instruction :: ‹sequential_state ⇒ 96 word›
where
  ‹fetch_instruction state ≡
    read_program_memory
      (read_register instruction_pointer_ref state)
      state›
```

Since all instructions have the same form, decode is just the application of get_opcode, get_reg1, get_reg2, get_reg3, and get_immediate functions to the value fetched from memory.

definition
```
decode_instruction :: ‹96 word ⇒ (8 word * 8 word * 8 word * 8 word * 64 word)›
where
  ‹decode_instruction instruction ≡
    let
      opcode = get_opcode instruction;
      reg1 = get_reg1 instruction;
      reg2 = get_reg2 instruction;
      reg3 = get_reg3 instruction;
      immediate = get_immediate instruction
    in
      (opcode, reg1, reg2, reg3, immediate)›
```

Execution is essentially a large switch on the opcode. Since a switch is not available in Isabelle, we are forced to use if, else-if, else-if, else-if, ..., else. However, this is logically equivalent to a switch statement, and we expect an implementation would use a switch statement.

fun (in *Ironbark_world*)
```
execute_instruction ::
  ‹(8 word * 8 word * 8 word * 8 word * 64 word)
  ⇒ sequential_state
  ⇒ sequential_state›
where
  ‹execute_instruction (opcode, reg1, reg2, reg3, immediate) state =
  (
    — error
    if (opcode = ERROR0_opcode) then
      ERROR0 state
```

```
else if (opcode = ERROR1_opcode) then
  ERROR1 state

— no-operation
else if (opcode = NOP_opcode) then
  NOP state

— load immediate
else if (opcode = LOAD_IMMEDIATE_opcode) then
  LOAD_IMMEDIATE reg1 immediate state

— static data load/store
else if (opcode = LOAD_STATIC_DATA_opcode) then
  LOAD_STATIC_DATA reg1 reg2 state
else if (opcode = STORE_STATIC_DATA_opcode) then
  STORE_STATIC_DATA reg1 reg2 state

— dynamic data load/store
else if (opcode = LOAD_DYNAMIC_DATA_opcode) then
  LOAD_DYNAMIC_DATA reg1 reg2 state
else if (opcode = STORE_DYNAMIC_DATA_opcode) then
  STORE_DYNAMIC_DATA reg1 reg2 state

— mapped memory load/store
else if (opcode = LOAD_INPUT_DATA_opcode) then
  LOAD_INPUT_DATA reg1 reg2 state
else if (opcode = STORE_OUTPUT_DATA_opcode) then
  STORE_OUTPUT_DATA reg1 reg2 state

— arithmetic operations
else if (opcode = COPY_opcode) then
  COPY reg1 reg2 state
else if (opcode = ADD_opcode) then
  ADD reg1 reg2 reg3 state
else if (opcode = SUBTRACT_opcode) then
  SUBTRACT reg1 reg2 reg3 state
else if (opcode = SHIFT_LEFT_opcode) then
  SHIFT_LEFT reg1 reg2 reg3 state
else if (opcode = SHIFT_RIGHT_opcode) then
  SHIFT_RIGHT reg1 reg2 reg3 state

— bitwise operations
else if (opcode = BITWISE_AND_opcode) then
  BITWISE_AND reg1 reg2 reg3 state
else if (opcode = BITWISE_OR_opcode) then
  BITWISE_OR reg1 reg2 reg3 state
else if (opcode = BITWISE_XOR_opcode) then
  BITWISE_XOR reg1 reg2 reg3 state
else if (opcode = BITWISE_NAND_opcode) then
```

```
    BITWISE_NAND reg1 reg2 reg3 state
  else if (opcode = BITWISE_NOT_opcode) then
    BITWISE_NOT reg1 reg2 state

  — comparison operations
  else if (opcode = LESS_THAN_opcode) then
    LESS_THAN reg1 reg2 reg3 state
  else if (opcode = GREATER_THAN_opcode) then
    GREATER_THAN reg1 reg2 reg3 state
  else if (opcode = EQUALS_opcode) then
    EQUALS reg1 reg2 reg3 state
  else if (opcode = NOT_EQUALS_opcode) then
    NOT_EQUALS reg1 reg2 reg3 state

  — non-deterministic operations
  else if (opcode = RANDOMISE_opcode) then
    RANDOMISE reg1 state

  — control flow
  else if (opcode = END_JUMP_opcode) then
    END_JUMP immediate state
  else if (opcode = END_JUMP_STRICT_opcode) then
    END_JUMP_STRICT immediate state
  else if (opcode = JUMP_opcode) then
    JUMP immediate state
  else if (opcode = CONDITIONAL_JUMP_opcode) then
    CONDITIONAL_JUMP reg1 immediate state
  else if (opcode = END_CALL_opcode) then
    END_CALL state
  else if (opcode = CALL_opcode) then
    CALL immediate state
  else if (opcode = END_RETURN_opcode) then
    END_RETURN immediate state
  else if (opcode = RETURN_opcode) then
    RETURN state

  — termination
  else if (opcode = HALT_opcode) then
    HALT state
  else
    ILLEGAL state
)›
```

Because we defined a function via the Isabelle fun command, its definition is automatically put in the simp set. We do not want this, so we remove it.

```
declare (in Ironbark_world) execute_instruction.simps [simp del]
```

We now bundle the components of execution into a single function. We also at this point provide a guard for termination if there is either an error or halt flag set, to enforce termination. It is

debateable whether the cycles register should continue to increment after termination. We do not let it increment here, as we are not interested in reasoning about behaviour after termination, as there is nothing of note, and any deviation between this and an implementation where these change after termination is of no security impact, and is not a property we rely on for any proofs.

```
definition (in Ironbark_world)
  execute_next_instruction :: ‹sequential_state ⇒ sequential_state›
  where
    ‹execute_next_instruction state ≡
      if
        get_halt state = 0x1
        ∨ get_error state = 0x1
      then
        state
      else
        execute_instruction
          (decode_instruction (fetch_instruction state))
          state›
```

The characteristic behaviour of the Ironbark processor is to continue executing the next instruction as long as it is supplied with power, and does not halt or error, which we interpret laxly as forever. Thus we expect to see the next instruction executed for an infinite number of instructions.

To provide easier reasoning when executing multiple instructions, we provide a definition for executing a specific number of instructions.

```
primrec (in Ironbark_world)
  execute_multiple_instructions :: ‹sequential_state ⇒ nat ⇒ sequential_state›
  where
    execute_multiple_instructions_0: ‹execute_multiple_instructions state 0 = state›
  | execute_multiple_instructions_Suc:
      ‹execute_multiple_instructions state (Suc num_instructions)
        = execute_next_instruction
            (execute_multiple_instructions state num_instructions)›
```

The following to definitions require some explanation.

The finish_function_trace function returns the program trace of a program until just before it would return from a function. This should only be used in 'leaf' functions which do not call subfunctions. The step parameter is the maximum number of steps that can be taken, and is only provided so that we can show that the function terminates. Proofs that use this function are intended to be generalised for an arbitrary value of step.

Finally, we note that the details of these functions are the subject of some active debate, and have not been tested, are subject to change, and thus at the time of writing, caution against their use when building on this work.

```
primrec (in Ironbark_world)
   finish_function_trace :: ‹sequential_state list ⇒ nat ⇒ sequential_state list›
   where
     ‹finish_function_trace program_trace 0 = program_trace›
```

```
| ‹finish_function_trace program_trace (Suc step) =
  (
    let
      current_state = last (finish_function_trace program_trace step);
      instruction = fetch_instruction current_state
    in
      if (get_opcode instruction = RETURN_opcode) then
        finish_function_trace program_trace step
      else
        (finish_function_trace program_trace step)
          @ [execute_next_instruction current_state]
  )›
```

end

3 Key Results

3.1 Security Properties of Ironbark

theory *security_properties*

imports
"execution_auxiliary"

begin

In this section, we provide some key results which we believe are relevant to demonstrating the security of applications running on the Ironbark processor.

3.1.1 Program Memory Immutable

The first property we show is that there is no sequence of instructions that can result in program memory being written to. We show this over two steps. First, we show that an individual instruction cannot modify program memory, and then use an inductive step to show it for an arbitrary sequence.

lemma (in *Ironbark_world) program_memory_immutable:*
fixes
state :: sequential_state
shows
‹*program_memory (execute_next_instruction state) = program_memory state*›
apply(*simp add: execute_next_instruction_def)*
apply(*simp add: fetch_instruction_def)*
apply(*simp add: decode_instruction_def)*
apply(*simp add: execute_instruction.simps)*
apply(*safe)*
apply(*simp_all add: instruction_impl_defs)*
apply(*simp_all add: state_manipulation_simps Let_def)*
done

```
lemma (in Ironbark_world) program_memory_immutable':
  fixes
    state :: sequential_state
  shows
    ‹program_memory (execute_multiple_instructions state n) = program_memory state›
proof (induct ‹n›)
  case 0
  then show ?case
    using program_memory_immutable
    by(simp)
next
  case (Suc n)
  then show ?case
    using program_memory_immutable
    by(simp)
qed
```

From this, it follows that reading program memory also never changes, which we also show over the following two lemmas (single instruction first, then arbitrary sequence).

```
lemma (in Ironbark_world) read_program_memory_immutable:
  fixes
    state :: sequential_state
  shows
    ‹read_program_memory  a (execute_next_instruction state)
    = read_program_memory a state›
  by (auto simp add: read_program_memory_def program_memory_immutable)

lemma (in Ironbark_world) read_program_memory_immutable':
  fixes
    state :: sequential_state
  shows
    ‹read_program_memory  a (execute_multiple_instructions state n)
    = read_program_memory a state›
  by (auto simp add: read_program_memory_def program_memory_immutable')
```

From this, we then show that the fetch decode and execute operations will all do the same thing when the instruction pointer is the same

```
lemma (in Ironbark_world) fetch_consistent:
  assumes
    ‹read_register instruction_pointer_ref state1
    = read_register instruction_pointer_ref state2›

    ‹state2 = execute_multiple_instructions state1 n›
  shows
    ‹fetch_instruction state1 = fetch_instruction state2›
  using assms
  by (simp add: fetch_instruction_def read_program_memory_immutable')
```

```
lemma (in Ironbark_world) decode_consistent:
  assumes
    ‹read_register instruction_pointer_ref state1
    = read_register instruction_pointer_ref state2›

    ‹state2 = execute_multiple_instructions state1 n›
  shows
    ‹decode_instruction (fetch_instruction state1)
    = decode_instruction (fetch_instruction state2)›
  using assms
  by (simp add: fetch_consistent)

lemma (in Ironbark_world) execute_consistent:
  assumes
    ‹read_register instruction_pointer_ref state1
    = read_register instruction_pointer_ref state2›

    ‹state2 = execute_multiple_instructions state1 n›
  shows
    ‹execute_instruction (decode_instruction (fetch_instruction state1))
    = execute_instruction (decode_instruction (fetch_instruction state2))›
  using assms
  by (simp add: decode_consistent)
```

Since program memory is static, we are also able to show that it will be the same at the end of any function trace. We show this both for our weak finish_function_trace, and the more complete full_function_trace.

```
lemma (in Ironbark_world) trace_program_memory:
  ‹program_memory (last (finish_function_trace [x] y)) = program_memory x›
proof (induct ‹y›)
  case 0
  then show ?case
    by(simp add: program_memory_immutable)
next
  case (Suc y)
  then show ?case
    by(simp add: program_memory_immutable Let_def)
qed
```

From these, it trivially follows that reading program memory at the end of a function trace is unaffected as well.

```
lemma (in Ironbark_world) trace_read_program_memory:
  ‹read_program_memory addr (last (finish_function_trace [x] y))
  = read_program_memory addr x›
  by(simp add: read_program_memory_def trace_program_memory)
```

3.1.2 Restricted Call Memory Modification

Next, we show that the call memory cannot be written to by anything that is not a call instruction. We do this over two lemmas, first for a single instruction, and then for arbitrary sequences of instructions.

```
lemma (in Ironbark_world) call_memory_mostly_immutable:
  fixes
    state :: sequential_state
  defines
    ‹instruction_pointer_val ≡ read_register instruction_pointer_ref state›
  defines
    ‹instruction ≡ read_program_memory instruction_pointer_val state›
  defines
    ‹opcode ≡ get_opcode instruction›
  assumes
    ‹opcode ≠ CALL_opcode›
  shows
    ‹call_memory (execute_next_instruction state) = call_memory state›
  using assms
  apply(simp)
  apply(simp add: execute_next_instruction_def)
  apply(simp add: fetch_instruction_def)
  apply(simp add: decode_instruction_def)
  apply(simp add: execute_instruction.simps)
  apply(safe)
  apply(simp_all add: instruction_impl_defs)
  apply(simp_all add: state_manipulation_simps Let_def)
  done

lemma (in Ironbark_world) call_memory_mostly_immutable':
  fixes state :: sequential_state
  assumes
    not_call:
      ‹∀ m ≤ n .
        get_opcode
          (read_program_memory
            (read_register
              instruction_pointer_ref
              (execute_multiple_instructions state m))
            (execute_multiple_instructions state m))
        ≠ CALL_opcode›
  shows
    ‹call_memory (execute_multiple_instructions state n) = call_memory state›
proof -
  have
    ‹m ≤ n
    ⟶ call_memory (execute_multiple_instructions state m) = call_memory state›
    (is "?P m")
    for m
```

```
  proof (induct ‹m›)
    show
      ‹?P 0›
      by (simp)
  next
    fix m
    assume ‹?P m›
    then show
      ‹?P (Suc m)›
      apply(intro impI)
      apply(simp)
      apply(subst call_memory_mostly_immutable)
      apply(rule not_call[rule_format])
      apply(simp)
      apply(simp)
      done
  qed
  then show ?thesis
    by(simp)
qed
```

We also show that the only two instructions which can modify the call frame pointer are the call and return instructions. This is done over two lemmas, first for a single instruction, and then over an arbitrary sequence.

```
lemma (in Ironbark_world) call_frame_pointer_mostly_immutable:
  fixes
    state :: sequential_state
  defines
    ‹instruction ≡
      read_program_memory
        (read_register instruction_pointer_ref state)
        state›
  defines
    ‹r1 ≡ get_reg1 instruction›
    and ‹r2 ≡ get_reg2 instruction›
    and ‹r3 ≡ get_reg3 instruction›
    and ‹imm ≡ get_immediate instruction›
  assumes
    not_RET: ‹get_opcode (fetch_instruction state) ≠ RETURN_opcode›
    and not_CALL: ‹get_opcode (fetch_instruction state) ≠ CALL_opcode›
  shows
    ‹read_register call_frame_pointer_ref (execute_next_instruction state)
    = read_register call_frame_pointer_ref state›
proof (cases ‹get_halt state = 1 ∨ get_error state = 1›)
  case True
  then show ?thesis
    unfolding execute_next_instruction_def
    by(simp_all)
next
```

```
  case False
  also have
    ‹get_opcode instruction ≤ 255›
    apply (uint_arith)
    apply (auto)
    done
  ultimately show ?thesis
    using not_CALL not_RET
    unfolding execute_next_instruction_def
    apply (simp add: flag_normalisation)
    unfolding fetch_instruction_def decode_instruction_def
    apply (fold instruction_def)
    apply (fold r1_def r2_def r3_def imm_def)
    apply (simp add: Let_def)
    apply (simp add: word_le_decr)
    apply (elim disjE)
    apply (simp_all add: execute_instruction.simps)
    apply (simp_all (no_asm) add: instruction_impl_defs)
    apply (simp_all add: state_manipulation_simps Let_def)
    done
qed

lemma (in Ironbark_world) pointer_carry:
  assumes
    not_RET:
      ‹∀ i ≤ n .
        get_opcode (fetch_instruction (execute_multiple_instructions state i))
          ≠ RETURN_opcode›
  and
    not_CALL:
      ‹∀ i ≤ n .
        get_opcode (fetch_instruction (execute_multiple_instructions state i))
          ≠ CALL_opcode›
  shows
    ‹read_register call_frame_pointer_ref (execute_multiple_instructions state n)
      = read_register call_frame_pointer_ref state›
proof -
  have
    ‹i ≤ n ⟶
    (
      read_register call_frame_pointer_ref (execute_multiple_instructions state i)
      = read_register call_frame_pointer_ref state
    )›
  (is "?P i")
  for i
  proof (induct ‹i›)
    show
      ‹?P 0›
      by (simp)
```

```
  next
    fix i
    assume ‹?P i›
    then show
      ‹?P (Suc i)›
      apply(intro impI)
      apply(simp)
      apply(subst call_frame_pointer_mostly_immutable)
      apply(simp_all add: not_CALL not_RET)
      done
  qed
  then show ?thesis
    by simp
qed
```

3.1.3 Register Read and Write Guards

The following two lemmas show that our check_read_permission and check_write_permission work as expected. These lemmas have been shown earlier, and reproduced here as we consider it a security property that can be relied upon.

```
lemma (in Ironbark_world) check_read_permission_guard:
  fixes
    a :: ‹8 word›
  assumes
    ‹check_read_permission a›
  shows
    ‹a ∈ readable_registers›
  using assms
  by(simp add: check_read_permission_def)

lemma (in Ironbark_world) check_write_permission_guard:
  assumes
    ‹check_write_permission a›
  shows
    ‹a ∈ writeable_registers›
  using assms
  by(simp add: check_write_permission_def)
```

3.1.4 ROP Prevention

We provide two proofs for the base case for ROP protection. The following lemmas only shows the 'base' case, rather than the general case, which comes after this. Specifically, they show that if a function has no calls or returns in it, then when it returns, it will return to the point where it was called from.

```
lemma (in Ironbark_world) base_ROP_avoidance:
  fixes
    state1 state2 :: ‹sequential_state›
    and addr :: ‹64 word›
```

```
    and n :: "nat"
  assumes
    two_after_one:
      ‹state2 = execute_next_instruction state1›
    and three_after_two:
      ‹state3 = execute_multiple_instructions state2 n›
    and four_after_three:
      ‹state4 = execute_next_instruction state3›

    and first_instruction:
      ‹execute_instruction (decode_instruction (fetch_instruction state1)) state1
      = CALL addr state1›
    and end_function:
      ‹execute_instruction (decode_instruction (fetch_instruction state3)) state3
      = RETURN state3›

    and non_RET:
      ‹∀ i ≤ n .
        get_opcode (fetch_instruction (execute_multiple_instructions state2 i))
        ≠ RETURN_opcode›
    and non_CALL:
      ‹∀ i ≤ n .
        get_opcode (fetch_instruction (execute_multiple_instructions state2 i))
        ≠ CALL_opcode›

    and flags1: ‹typical_flags state1›
    and flags3: ‹typical_flags state3›
  shows
    ‹read_register instruction_pointer_ref state4
    = (read_register instruction_pointer_ref state1) + 1›
proof -
  have
    ‹execute_next_instruction state3 = RETURN state3›
    unfolding execute_next_instruction_def
    using flags3
    by(simp add: end_function)
  moreover have
    ‹execute_next_instruction state1 = CALL addr state1›
    unfolding execute_next_instruction_def
    using flags1
    by(simp add: first_instruction)
  moreover have
    ‹read_call_memory
      (read_register call_frame_pointer_ref state1)
      (CALL addr state1)
    = read_register instruction_pointer_ref state1›
    apply(simp add: instruction_decomp_manipulation flags1)
    apply(simp add: state_manipulation_simps)
    apply(simp add: state_manipulation_decomp)
```

```
    done
  moreover have
    ‹call_memory state3 = call_memory state2›
    using three_after_two non_CALL
    apply(simp)
    apply(subst call_memory_mostly_immutable')
    apply(simp_all add: fetch_instruction_def)
    done
  moreover from this have
    ‹∀ addr2 . read_call_memory addr2 state3 = read_call_memory addr2 state2›
    unfolding read_call_memory_def
    by(simp)
  moreover have
    ‹read_register call_frame_pointer_ref state3
    = read_register call_frame_pointer_ref state2›
    using three_after_two non_CALL non_RET
    by(simp add: pointer_carry)
  ultimately show ?thesis
    using assms
    by(simp add: instruction_simps)
qed

lemma (in Ironbark_world) base_ROP_avoidance':
  fixes
    state1 state2 state3 state4 :: ‹sequential_state›
    and n :: "nat"
    and addr :: ‹64 word›
  assumes
    two_after_one:
      ‹state2 = execute_next_instruction state1›
    and three_after_two:
      ‹state3 = execute_multiple_instructions state2 n›
    and four_after_three:
      ‹state4 = execute_next_instruction state3›

    and first_instruction:
      ‹get_opcode (fetch_instruction state1) = CALL_opcode›
      ‹get_immediate (fetch_instruction state1) = addr›
    and end_function:
      ‹get_opcode (fetch_instruction state3) = RETURN_opcode›

    and non_RET:
      ‹∀ i ≤ n .
        get_opcode (fetch_instruction (execute_multiple_instructions state2 i))
        ≠ RETURN_opcode›
    and non_CALL:
      ‹∀ i ≤ n .
        get_opcode (fetch_instruction (execute_multiple_instructions state2 i))
        ≠ CALL_opcode›
```

```
    and flags1: ‹typical_flags state1›
    and flags3: ‹typical_flags state3›
  shows
    ‹read_register instruction_pointer_ref state4
    = (read_register instruction_pointer_ref state1) + 1›
proof-
  have Ra1:
    ‹state2 = CALL addr state1›
    apply(simp add: two_after_one)
    apply(simp add: execute_next_instruction_def)
    apply(simp add: flags1)
    apply(simp add: decode_instruction_def)
    apply(simp add: first_instruction)
    apply(simp add: execute_instruction_simps)
    done
  have
    ‹state4 = RETURN state3›
    apply(simp add: four_after_three)
    apply(simp add: execute_next_instruction_def)
    apply(simp add: flags3)
    apply(simp add: decode_instruction_def)
    apply(simp add: end_function)
    apply(simp add: execute_instruction_simps)
    done
  with flags3 have
    ‹read_register instruction_pointer_ref state4
      = read_call_memory (read_register call_frame_pointer_ref state3 - 67) state3 + 1›
    by (simp add: instruction_simps)
  also have
    ‹...
      = read_call_memory (read_register call_frame_pointer_ref state2 - 67) state3 + 1›
    using pointer_carry [OF non_RET non_CALL, folded three_after_two]
    by simp
  also have
    ‹...
      = read_call_memory (read_register call_frame_pointer_ref state2 - 67) state2 + 1›
    using same_memory_same_addr(2) [OF call_memory_mostly_immutable'
                [folded fetch_instruction_def, OF non_CALL, folded three_after_two]]
    by simp
  also have
    ‹...
      = read_call_memory (read_register call_frame_pointer_ref state1) state2 + 1›
    using flags1
    by (simp add: Ra1 instruction_simps)
  also have
    ‹...
      = read_register instruction_pointer_ref state1 + 1›
    unfolding Ra1
```

```
    apply(simp add: CALL_decomp_manipulation flags1)
    apply(simp add: state_manipulation_simps)
    apply(simp add: state_manipulation_decomp)
    done
  finally show ?thesis .
qed
```

In working towards the general case, we rely on some auxiliary results about the operation of the call_frame_pointer, which are derived below.

```
lemma (in Ironbark_world) call_memory_preserved_step:
  fixes
    state1 state2 :: sequential_state
    and addr :: ‹64 word›
  defines
    ‹instruction1 ≡ fetch_instruction state1›
  defines
    ‹opcode1 ≡ get_opcode instruction1›
  defines
    ‹call_memory1 ≡ read_call_memory addr state1›
    and ‹call_memory2 ≡ read_call_memory addr state2›
  assumes
    two_after_one:
      ‹state2 = execute_next_instruction state1›
    and address_below_call_frame:
      ‹addr < (read_register call_frame_pointer_ref state1)›
    and no_overflow:
      ‹(read_register call_frame_pointer_ref state1) < (2^64 - 66)›
  shows
    ‹read_call_memory addr state1 = read_call_memory addr state2›
  by (simp add: call_memory_evolution [OF two_after_one]
          read_call_memory_backup_registers_before_call [OF address_below_call_frame no_overflow])

lemma (in Ironbark_world) call_memory_preserved_multi_stepJ [THEN mp, OF _ _ _ order_refl]:
  assumes
    start_small:
      ‹addr ≤ (2^64 - 67)›
    and not_terminated:
      ‹∀ i ≤ n .
        addr + 66
        < read_register call_frame_pointer_ref (execute_multiple_instructions state i)›
    and no_overflow:
      ‹∀ i ≤ n .
        read_register call_frame_pointer_ref (execute_multiple_instructions state i)  ≤ (2^64
- 67)›
  shows
    ‹i ≤ n ⟶
      read_call_memory addr (execute_multiple_instructions state i)
      = read_call_memory addr state›
  apply (induct i, auto)
```

```
  apply (rule call_memory_preserved_step [symmetric, THEN trans])
  apply (rule refl, simp_all)
  apply (rule order_less_trans [OF _ not_terminated [rule_format]], simp_all)
  using start_small
  apply (simp add: unat_arith_simps)
  apply (rule order_le_less_trans [OF no_overflow [rule_format]], simp_all)
  done
```

Following from this, we show an argument for ROP protection. Note that the assumption "depth_1" follows from call_frame_pointer_backsubstitute, which shows that the call_frame_pointer can only change by exactly 67, and only increases by 67 in a call, and only decreases by 67 in a return. Thus, the assumption can be read in English as "assume we are at a matching return".

```
lemma (in Ironbark_world) ROP_avoidance:
  fixes
    state1 state2 state3 state4 :: ‹sequential_state›
    and n :: "nat"
    and addr :: ‹64 word›
  assumes
    two_after_one:
      ‹state2 = CALL addr state1›
    and three_after_two:
      ‹state3 = execute_multiple_instructions state2 n›
    and four_after_three:
      ‹state4 = RETURN state3›

    and first_instruction:
      ‹get_opcode (fetch_instruction state1) = CALL_opcode›
      ‹get_immediate (fetch_instruction state1) = addr›
    and end_function:
      ‹get_opcode (fetch_instruction state3) = RETURN_opcode›
    and depth_1:
      ‹read_register call_frame_pointer_ref state3 = read_register call_frame_pointer_ref state1
+ 67›
    and start_small:
      ‹read_register call_frame_pointer_ref state1 ≤ (2^64 - 67)›
    and not_terminated:
      ‹∀ i ≤ n .
        read_register call_frame_pointer_ref state1 + 66
        < read_register call_frame_pointer_ref (execute_multiple_instructions state2 i)›
    and no_overflow:
      ‹∀i ≤ n .
        read_register call_frame_pointer_ref (execute_multiple_instructions state2 i)  ≤ (2^64
- 67)›

    and flags1: ‹typical_flags state1›
    and flags3: ‹typical_flags state3›
  shows
    ‹read_register instruction_pointer_ref state4
    = (read_register instruction_pointer_ref state1) + 1›
```

```
proof-
  from four_after_three flags3 have
    ‹read_register instruction_pointer_ref state4
      = read_call_memory (read_register call_frame_pointer_ref state3 - 67) state3 + 1›
    by (simp add: instruction_simps)
  also have
    ‹...
      = read_call_memory (read_register call_frame_pointer_ref state1) state3 + 1›
    using depth_1
    by simp
  also have
    ‹...
      = read_call_memory (read_register call_frame_pointer_ref state1) state2 + 1›
    unfolding three_after_two
    by (subst call_memory_preserved_multi_stepJ [OF start_small not_terminated no_overflow],
        rule refl)
  also have
    ‹...
      = read_register instruction_pointer_ref state1 + 1›
    unfolding two_after_one
    apply (simp add: CALL_decomp_manipulation flags1)
    apply (simp add: state_manipulation_simps)
    apply (simp add: state_manipulation_decomp)
    done
  finally show ?thesis .
qed


end
```

3.2 Correctness of Fibonacci as a Function

```
theory fibonacci_function

imports
  "Ironbark_execution.execution_top"

begin
```

This file contains a proof of correctness of a function which calculates Fibonacci numbers. Specifically, this proof is done on a function which can be freely relocated, unlike the versions seen in the fibonacci_invariant theory or fibonacci_invariant_piecemeal theory.

We start by defining the Fibonacci function from a pure mathematical point of view, which we will later show is equivalent to the return value of our function (modulo 2^{64}).

```
function
  fib :: ‹nat ⇒ nat›
  where
  ‹fib 0 = 1›
```

```
| ‹fib 1 = 1›
| ‹fib (n+2) = fib n + fib (n+1)›
  apply atomize_elim
  apply arith
  apply auto
  done
termination by lexicographic_order
```

Next, we derive a variety of rewriting. This is largely done to remove Suc from the proof state, as we prefer to operate on +, instead of Suc.

```
lemma fib_stuff:
  ‹fib (n+3) = fib (n+1) + fib (n+2)›
  ‹fib (Suc (Suc n)) = fib n + fib (n+1)›
  ‹fib (Suc (Suc (Suc n))) = fib (n+1) + fib (n+2)›
  ‹fib (Suc 0) = 1›
  ‹fib (Suc (Suc n)) = fib (n+2)›
  ‹fib (Suc n) = fib (n+1)›
  ‹fib (Suc (n+1)) = fib (n+2)›
  ‹fib (Suc (n+2)) = fib (n+3)›
  ‹fib (Suc (Suc 0)) = 2›
  ‹fib (n+0) = fib n›
  ‹fib (0+n) = fib n›
  ‹fib (Suc (Suc (Suc n))) = fib (n+3)›
  using fib.simps
  by(simp_all add: numeral_Bit1(1))
```

We found it is sometimes useful to be able to lookup the answers for early Fibonacci numbers, so we collect a few of the early answers for convenience in the following lemma.

```
lemma fib_answers:
  ‹fib 0 = 1›
  ‹fib 1 = 1›
  ‹fib (Suc 0) = 1›
  ‹fib 2 = 2›
  ‹fib (Suc 1) = 2›
  ‹fib (Suc (Suc 0)) = 2›
  ‹fib 3 = 3›
  ‹fib 4 = 5›
  using fib.simps
  by(simp_all add: numeral_3_eq_3 numeral_Bit0)
```

In order to support relocation of the function, we create a locale where we assume the branches and end branches have been appropriately setup for the location. The proof is done in the context of this locale.

```
locale fib_routine =
  fixes
    n :: ‹64 word›
    and hack_v1 hack_v2 hack_v3 hack_v4 :: ‹96 word›
  assumes
```

```
    decode_hacks:
      ‹decode_instruction(hack_v1) = (0x1B, 0x04, 0x00, 0x00, n + 0xe)›
      ‹decode_instruction(hack_v2) = (0x1A, 0x00, 0x00, 0x00, n + 0x6)›
      ‹decode_instruction(hack_v3) = (0x18, 0x00, 0x00, 0x00, n + 13)›
      ‹decode_instruction(hack_v4) = (0x18, 0x00, 0x00, 0x00, n + 8)›
begin
end

sublocale fib_routine ⊆ Ironbark_world
  done
```

We now define the program, which we will show correctly implements the Fibonacci function.

```
definition (in fib_routine)
  program :: ‹(64 word, 96 word) map›
  where
    ‹program ≡
      Map.empty
      (
        n    ↦ 0x1C00000000000000000000000,  — end_call
        n+1  ↦ 0x02210000000000000000000001,  — c01 = 0x1
        n+2  ↦ 0x02030000000000000000000000,  — r03 = 0x0
        n+3  ↦ 0x02000000000000000000000001,  — r00 = 0x1
        n+4  ↦ 0x02010000000000000000000001,  — r01 = 0x1
        n+5  ↦ 0x02020000000000000000000002,  — r02 = 0x2
        n+6  ↦ hack_v3,                       — end_jump (n + 13)
        n+7  ↦ 0x15040330000000000000000000,  — r04 = r03 == arg00
        n+8  ↦ hack_v1,                       — if (r04) jump (n + 14)
        n+9  ↦ 0x09000100000000000000000000,  — r00 = r01
        n+10 ↦ 0x09010200000000000000000000,  — r01 = r02
        n+11 ↦ 0x0A02000100000000000000000,   — r02 = r00 + r01
        n+12 ↦ 0x0A03032100000000000000000,   — r03 = r03 + c01
        n+13 ↦ hack_v2,                       — jump (n + 6)
        n+14 ↦ hack_v4,                       — end_jump (n + 8)
        n+15 ↦ 0x09400000000000000000000000,  — ret00 = r00
        n+16 ↦ 0x1F00000000000000000000000    — return
      )›
```

We also define what it means for our program to be 'correct'. Because we use an inductive process with this as our invariant, the approach we have taken is to state what we expect to be true at every step of the program. We've annotated this with the assembly being executed at each step, and what we really mean by correctness is what we show in the return step (at the end of the definition), and the rest of these are intermediate results useful for the inductive process.

```
definition (in fib_routine)
  check_trace :: ‹(sequential_state list) ⇒ bool›
  where
    ‹check_trace trace ≡
      trace ≠ [] ∧
      ((read_register instruction_pointer_ref (hd trace) = n) ⟶
      (list_all
```

```
(λ step .
  (read_register instruction_pointer_ref step ∈ dom program)

  — end_call
  ∧ (read_register instruction_pointer_ref step = n
      ⟶ end_call_flags step)

  — c01 = 0x1
  ∧ (read_register instruction_pointer_ref step = n+1
      ⟶ typical_flags step)

  — r03 = 0 //counter
  ∧ (read_register instruction_pointer_ref step = n+2 ⟶
  (
    typical_flags step
    ∧ read_register c01_ref step = 1
  ))

  — r00 = 1 //fib(counter)
  ∧ (read_register instruction_pointer_ref step = n+3 ⟶
  (
    typical_flags step
    ∧ read_register c01_ref step = 1
    ∧ read_register r03_ref step = 0
    ∧ read_register r03_ref step ≤ read_register arg00_ref step
  ))

  — r01 = 1 //fib(counter+1)
  ∧ (read_register instruction_pointer_ref step = n+4 ⟶
  (
    typical_flags step
    ∧ read_register c01_ref step = 1
    ∧ read_register r03_ref step = 0
    ∧ read_register r03_ref step ≤ read_register arg00_ref step
    ∧ read_register r00_ref step
      = of_nat (fib (unat (read_register r03_ref step)))
  ))

  — r02 = 2 //fib(counter+2))
  ∧ (read_register instruction_pointer_ref step = n+5 ⟶
  (
    typical_flags step
    ∧ read_register c01_ref step = 1
    ∧ read_register r03_ref step = 0
    ∧ read_register r03_ref step ≤ read_register arg00_ref step
    ∧ read_register r00_ref step
      = of_nat (fib (unat (read_register r03_ref step)))
    ∧ read_register r01_ref step
      = of_nat (fib (unat (read_register r03_ref step) + 1))
```

```
))

— end_jump
∧ (read_register instruction_pointer_ref step = n+6 ⟶
(
  (typical_flags step ∨ end_jump_flags step)
∧ (typical_flags step
  ⟶ (read_register last_instruction_pointer_ref step = n+5))
∧ (end_jump_flags step
  ⟶ (read_register last_instruction_pointer_ref step = n+13))
∧ read_register c01_ref step = 1
∧ read_register r03_ref step ≤ read_register arg00_ref step
∧ read_register r00_ref step
  = of_nat (fib (unat (read_register r03_ref step)))
∧ read_register r01_ref step
  = of_nat (fib (unat (read_register r03_ref step) + 1))
∧ read_register r02_ref step
  = of_nat (fib (unat (read_register r03_ref step) + 2))
))

— r04 = r03 == arg00
∧ (read_register instruction_pointer_ref step = n+7 ⟶
(
  typical_flags step
  ∧ read_register c01_ref step = 1
  ∧ read_register r03_ref step ≤ read_register arg00_ref step
  ∧ read_register r00_ref step
    = of_nat (fib (unat (read_register r03_ref step)))
  ∧ read_register r01_ref step
    = of_nat (fib (unat (read_register r03_ref step) + 1))
  ∧ read_register r02_ref step
    = of_nat (fib (unat (read_register r03_ref step) + 2))
))

— if (r04) jump finish //if (counter == n) finish
∧ (read_register instruction_pointer_ref step = n+8 ⟶
(
  typical_flags step
  ∧ read_register c01_ref step = 1
  ∧ read_register r03_ref step ≤ read_register arg00_ref step
  ∧ read_register r00_ref step
    = of_nat (fib (unat (read_register r03_ref step)))
  ∧ read_register r01_ref step
    = of_nat (fib (unat (read_register r03_ref step) + 1))
  ∧ read_register r02_ref step
    = of_nat (fib (unat (read_register r03_ref step) + 2))
  ∧ (read_register r03_ref step = read_register arg00_ref step
    ⟶ (read_register r04_ref step = 1))
  ∧ (read_register r03_ref step ≠ read_register arg00_ref step
```

```
    ⟶ (read_register r04_ref step = 0))
  ∧ (read_register r04_ref step = 1 ∨ read_register r04_ref step = 0)
))
```

— r00 = r01 //fib(counter) <- fib(counter+1)

```
∧ (read_register instruction_pointer_ref step = n+9 ⟶
(
  typical_flags step
  ∧ read_register c01_ref step = 1
  ∧ read_register r03_ref step < read_register arg00_ref step
  ∧ read_register r00_ref step
    = of_nat (fib (unat (read_register r03_ref step)))
  ∧ read_register r01_ref step
    = of_nat (fib (unat (read_register r03_ref step) + 1))
  ∧ read_register r02_ref step
    = of_nat (fib (unat (read_register r03_ref step) + 2))
))
```

— r01 = r02 //fib(counter+1) <- fib(counter+2)

```
∧ (read_register instruction_pointer_ref step = n+10 ⟶
(
  typical_flags step
  ∧ read_register c01_ref step = 1
  ∧ read_register r03_ref step < read_register arg00_ref step
  ∧ read_register r00_ref step
    = of_nat (fib (unat (read_register r03_ref step) + 1))
  ∧ read_register r01_ref step
    = of_nat (fib (unat (read_register r03_ref step) + 1))
  ∧ read_register r02_ref step
    = of_nat (fib (unat (read_register r03_ref step) + 2))
))
```

— r02 = r00 + r01 //fib(counter+2) <- fib(counter+3)

```
∧ (read_register instruction_pointer_ref step = n+11 ⟶
(
  typical_flags step
  ∧ read_register c01_ref step = 1
  ∧ read_register r03_ref step < read_register arg00_ref step
  ∧ read_register r00_ref step
    = of_nat (fib (unat (read_register r03_ref step) + 1))
  ∧ read_register r01_ref step
    = of_nat (fib (unat (read_register r03_ref step) + 2))
  ∧ read_register r02_ref step
    = of_nat (fib (unat (read_register r03_ref step) + 2))
))
```

— r03 = r03 + c01 //counter++

```
∧ (read_register instruction_pointer_ref step = n+12 ⟶
(
```

```
  typical_flags step
  ∧ read_register c01_ref step = 1
  ∧ read_register r03_ref step < read_register arg00_ref step
  ∧ read_register r00_ref step
    = of_nat (fib (unat (read_register r03_ref step) + 1))
  ∧ read_register r01_ref step
    = of_nat (fib (unat (read_register r03_ref step) + 2))
  ∧ read_register r02_ref step
    = of_nat (fib (unat (read_register r03_ref step) + 3))
))
```

— jump main_loop

```
∧ (read_register instruction_pointer_ref step = n+13 ⟶
(
  typical_flags step
  ∧ read_register c01_ref step = 1
  ∧ read_register r03_ref step ≤ read_register arg00_ref step
  ∧ read_register r00_ref step
    = of_nat (fib (unat (read_register r03_ref step)))
  ∧ read_register r01_ref step
    = of_nat (fib (unat (read_register r03_ref step) + 1))
  ∧ read_register r02_ref step
    = of_nat (fib (unat (read_register r03_ref step) + 2))
))
```

— end_jump

```
∧ (read_register instruction_pointer_ref step = n+14 ⟶
(
  end_jump_flags step
  ∧ read_register last_instruction_pointer_ref step = n+8
  ∧ read_register r00_ref step
    = of_nat (fib (unat (read_register arg00_ref step)))
))
```

— ret00 = r00

```
∧ (read_register instruction_pointer_ref step = n+15 ⟶
(
  typical_flags step
  ∧ read_register r00_ref step
    = of_nat (fib (unat (read_register arg00_ref step)))
))
```

— return

```
∧ (read_register instruction_pointer_ref step = n+16 ⟶
(
  typical_flags step
  ∧ read_register ret00_ref step
    = of_nat (fib (unat (read_register arg00_ref step)))
))
```

```
    ) trace
  ))›
```

Finally, we present a proof that our program is correct. We note that the body of the proof is very large, with many subproofs. The structure of this proof is to start with some preliminary transformation, then decompose the check_trace into the goals for each instruction (see have goals named check_trace_decomposition_n*), then establish what instruction will be executed in each step (see have goal named current_instruction), before showing that the properties claimed in check_trace will be held after executing the next instruction (see have goals named n*_h5). After showing all those subgoals, we combine them to show the original goal.

```
lemma (in fib_routine) check_fib:
  assumes
    function_start: ‹read_register instruction_pointer_ref starting_state = n›
    and starting_flags: ‹end_call_flags starting_state›
    and program_loaded: ‹program_memory starting_state = program›
  shows
    ‹check_trace (finish_function_trace [starting_state] x)›
proof (induct ‹x›)
  case 0
  then show ?case
    apply(simp)
    apply(simp add: check_trace_def)
    apply(simp add: program_def starting_flags)
    done
next
  case (Suc x)
  then show ?case
    apply(simp add: Let_def)
    apply(intro impI)

  proof -
      fix x :: nat
      define current_state
        where
          ‹current_state = last (finish_function_trace [starting_state] x)›
      define next_state
        where
          ‹next_state = execute_next_instruction current_state›
    assume a1: ‹check_trace (finish_function_trace [starting_state] x)›
    assume a2: ‹get_opcode (fetch_instruction current_state) ≠ 31›

    have trace_program_memory3:
      ‹∀ addr .
        read_program_memory addr current_state
        = read_program_memory addr starting_state›
      by(simp add: current_state_def trace_read_program_memory)

    have trace_program_memory4:
```

```
  ‹program_memory current_state = program_memory starting_state›
  by(simp add: current_state_def trace_program_memory)

have state_in_trace:
  ‹current_state ∈ set (finish_function_trace [starting_state] x)›
  by(simp add: current_state_def trace_rules)

have check_trace_decomposition_n0:
  ‹read_register instruction_pointer_ref current_state = n
  ⟹ end_call_flags current_state›
  using a1 trace_rules function_start
  by(simp_all add: check_trace_def list_all_def Ball_def state_in_trace)

have check_trace_decomposition_n1:
  ‹read_register instruction_pointer_ref current_state = n + 1
  ⟹ typical_flags current_state›
  using a1 trace_rules function_start
  by(simp_all add: check_trace_def list_all_def Ball_def state_in_trace)

have check_trace_decomposition_n2:
  ‹read_register instruction_pointer_ref current_state = n + 2
  ⟹ typical_flags current_state›

  ‹read_register instruction_pointer_ref current_state = n + 2
  ⟹ read_register c01_ref current_state = 1›
  using a1 trace_rules function_start
  by(simp_all add: check_trace_def list_all_def Ball_def state_in_trace)

have check_trace_decomposition_n3:
  ‹read_register instruction_pointer_ref current_state = n + 3
  ⟹ typical_flags current_state›

  ‹read_register instruction_pointer_ref current_state = n + 3
  ⟹ read_register c01_ref current_state = 1›

  ‹read_register instruction_pointer_ref current_state = n + 3
  ⟹ read_register r03_ref current_state = 0›

  ‹read_register instruction_pointer_ref current_state = n + 3
  ⟹ read_register r03_ref current_state
    ≤ read_register arg00_ref current_state›
  using a1 trace_rules function_start
  by(simp_all add: check_trace_def list_all_def Ball_def state_in_trace)

have check_trace_decomposition_n4:
  ‹read_register instruction_pointer_ref current_state = n + 4
  ⟹ typical_flags current_state›

  ‹read_register instruction_pointer_ref current_state = n + 4
```

```
  ⟹ read_register c01_ref current_state = 1›

  ‹read_register instruction_pointer_ref current_state = n + 4
  ⟹ read_register r03_ref current_state = 0›

  ‹read_register instruction_pointer_ref current_state = n + 4
  ⟹ read_register r03_ref current_state
    ≤ read_register arg00_ref current_state›

  ‹read_register instruction_pointer_ref current_state = n + 4
  ⟹ read_register r00_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state)))›
  using a1 trace_rules function_start
  by(simp_all add: check_trace_def list_all_def Ball_def state_in_trace)

have check_trace_decomposition_n5:
  ‹read_register instruction_pointer_ref current_state = n + 5
  ⟹ typical_flags current_state›

  ‹read_register instruction_pointer_ref current_state = n + 5
  ⟹ read_register c01_ref current_state = 1›

  ‹read_register instruction_pointer_ref current_state = n + 5
  ⟹ read_register r03_ref current_state = 0›

  ‹read_register instruction_pointer_ref current_state = n + 5
  ⟹ read_register r03_ref current_state ≤ read_register arg00_ref current_state›

  ‹read_register instruction_pointer_ref current_state = n + 5
  ⟹ read_register r00_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state)))›

  ‹read_register instruction_pointer_ref current_state = n + 5
  ⟹ read_register r01_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 1))›
  using a1 trace_rules function_start
  by(simp_all add: check_trace_def list_all_def Ball_def state_in_trace)

have check_trace_decomposition_n6:
  ‹read_register instruction_pointer_ref current_state = n + 6
  ⟹ (typical_flags current_state ∨ end_jump_flags current_state)›

  ‹read_register instruction_pointer_ref current_state = n + 6
  ⟹ read_register c01_ref current_state = 1›

  ‹read_register instruction_pointer_ref current_state = n + 6
  ⟹ (typical_flags current_state
    ⟶ read_register last_instruction_pointer_ref current_state = n + 5)›
```

```
  ‹read_register instruction_pointer_ref current_state = n + 6
  ⟹ (end_jump_flags current_state
    ⟶ read_register last_instruction_pointer_ref current_state = n + 13)›

  ‹read_register instruction_pointer_ref current_state = n + 6
  ⟹ read_register r03_ref current_state
    ≤ read_register arg00_ref current_state›

  ‹read_register instruction_pointer_ref current_state = n + 6
  ⟹ read_register r00_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state)))›

  ‹read_register instruction_pointer_ref current_state = n + 6
  ⟹ read_register r01_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 1))›

  ‹read_register instruction_pointer_ref current_state = n + 6
  ⟹ read_register r02_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 2))›
  using a1 trace_rules function_start
  by(simp_all add: check_trace_def list_all_def Ball_def state_in_trace)

have check_trace_decomposition_n7:
  ‹read_register instruction_pointer_ref current_state = n + 7
  ⟹ typical_flags current_state›

  ‹read_register instruction_pointer_ref current_state = n + 7
  ⟹ read_register c01_ref current_state = 1›

  ‹read_register instruction_pointer_ref current_state = n + 7
  ⟹ read_register r03_ref current_state
    ≤ read_register arg00_ref current_state›

  ‹read_register instruction_pointer_ref current_state = n + 7
  ⟹ read_register r00_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state)))›

  ‹read_register instruction_pointer_ref current_state = n + 7
  ⟹ read_register r01_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 1))›

  ‹read_register instruction_pointer_ref current_state = n + 7
  ⟹ read_register r02_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 2))›
  using a1 trace_rules function_start
  by(simp_all add: check_trace_def list_all_def Ball_def state_in_trace)

have check_trace_decomposition_n8:
  ‹read_register instruction_pointer_ref current_state = n + 8
```

```
  ⟹ typical_flags current_state›

  ‹read_register instruction_pointer_ref current_state = n + 8
  ⟹ read_register c01_ref current_state = 1›

  ‹read_register instruction_pointer_ref current_state = n + 8
  ⟹ read_register r03_ref current_state
    ≤ read_register arg00_ref current_state›

  ‹read_register instruction_pointer_ref current_state = n + 8
  ⟹ read_register r00_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state)))›

  ‹read_register instruction_pointer_ref current_state = n + 8
  ⟹ read_register r01_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 1))›

  ‹read_register instruction_pointer_ref current_state = n + 8
  ⟹ read_register r02_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 2))›

  ‹read_register instruction_pointer_ref current_state = n + 8
  ⟹ (read_register r03_ref current_state = read_register arg00_ref current_state)
    ⟶ (read_register r04_ref current_state = 1)›

  ‹read_register instruction_pointer_ref current_state = n + 8
  ⟹ (read_register r03_ref current_state ≠ read_register arg00_ref current_state)
    ⟶ (read_register r04_ref current_state = 0)›

  ‹read_register instruction_pointer_ref current_state = n + 8
  ⟹ (read_register r04_ref current_state = 1
    ∨ read_register r04_ref current_state = 0)›
  using a1 trace_rules function_start
  by(simp_all add: check_trace_def list_all_def Ball_def state_in_trace)

have check_trace_decomposition_n9:
  ‹read_register instruction_pointer_ref current_state = n + 9
  ⟹ typical_flags current_state›

  ‹read_register instruction_pointer_ref current_state = n + 9
  ⟹ read_register c01_ref current_state = 1›

  ‹read_register instruction_pointer_ref current_state = n + 9
  ⟹ read_register r03_ref current_state
    < read_register arg00_ref current_state›

  ‹read_register instruction_pointer_ref current_state = n + 9
  ⟹ read_register r00_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state)))›
```

```
  ‹read_register instruction_pointer_ref current_state = n + 9
  ⟹ read_register r01_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 1))›

  ‹read_register instruction_pointer_ref current_state = n + 9
  ⟹ read_register r02_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 2))›
  using a1 trace_rules function_start
  by(simp_all add: check_trace_def list_all_def Ball_def state_in_trace)

have check_trace_decomposition_n10:
  ‹read_register instruction_pointer_ref current_state = n + 10
  ⟹ typical_flags current_state›

  ‹read_register instruction_pointer_ref current_state = n + 10
  ⟹ read_register c01_ref current_state = 1›

  ‹read_register instruction_pointer_ref current_state = n + 10
  ⟹ read_register r03_ref current_state
    < read_register arg00_ref current_state›

  ‹read_register instruction_pointer_ref current_state = n + 10
  ⟹ read_register r00_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 1))›

  ‹read_register instruction_pointer_ref current_state = n + 10
  ⟹ read_register r01_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 1))›

  ‹read_register instruction_pointer_ref current_state = n + 10
  ⟹ read_register r02_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 2))›
  using a1 trace_rules function_start
  by(simp_all add: check_trace_def list_all_def Ball_def state_in_trace)

have check_trace_decomposition_n11:
  ‹read_register instruction_pointer_ref current_state = n + 11
  ⟹ typical_flags current_state›

  ‹read_register instruction_pointer_ref current_state = n + 11
  ⟹ read_register c01_ref current_state = 1›

  ‹read_register instruction_pointer_ref current_state = n + 11
  ⟹ read_register r03_ref current_state
    < read_register arg00_ref current_state›

  ‹read_register instruction_pointer_ref current_state = n + 11
  ⟹ read_register r00_ref current_state
```

```
    = of_nat (fib (unat (read_register r03_ref current_state) + 1))⟩

  ⟨read_register instruction_pointer_ref current_state = n + 11
  ⟹ read_register r01_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 2))⟩

  ⟨read_register instruction_pointer_ref current_state = n + 11
  ⟹ read_register r02_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 2))⟩
  using a1 trace_rules function_start
  by(simp_all add: check_trace_def list_all_def Ball_def state_in_trace)

have check_trace_decomposition_n12:
  ⟨read_register instruction_pointer_ref current_state = n + 12
  ⟹ typical_flags current_state⟩

  ⟨read_register instruction_pointer_ref current_state = n + 12
  ⟹ read_register c01_ref current_state = 1⟩

  ⟨read_register instruction_pointer_ref current_state = n + 12
  ⟹ read_register r03_ref current_state
    < read_register arg00_ref current_state⟩

  ⟨read_register instruction_pointer_ref current_state = n + 12
  ⟹ read_register r00_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 1))⟩

  ⟨read_register instruction_pointer_ref current_state = n + 12
  ⟹ read_register r01_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 2))⟩

  ⟨read_register instruction_pointer_ref current_state = n + 12
  ⟹ read_register r02_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 3))⟩
  using a1 trace_rules function_start
  by(simp_all add: check_trace_def list_all_def Ball_def state_in_trace)

have check_trace_decomposition_n13:
  ⟨read_register instruction_pointer_ref current_state = n + 13
  ⟹ typical_flags current_state⟩

  ⟨read_register instruction_pointer_ref current_state = n + 13
  ⟹ read_register c01_ref current_state = 1⟩

  ⟨read_register instruction_pointer_ref current_state = n + 13
  ⟹ read_register r03_ref current_state
    ≤ read_register arg00_ref current_state⟩

  ⟨read_register instruction_pointer_ref current_state = n + 13
```

```
  ⟹ read_register r00_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state)))›

  ‹read_register instruction_pointer_ref current_state = n + 13
  ⟹ read_register r01_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 1))›

  ‹read_register instruction_pointer_ref current_state = n + 13
  ⟹ read_register r02_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 2))›
  using a1 trace_rules function_start
  by(simp_all add: check_trace_def list_all_def Ball_def state_in_trace)

have check_trace_decomposition_n14:
  ‹read_register instruction_pointer_ref current_state = n + 14
  ⟹ end_jump_flags current_state›

  ‹read_register instruction_pointer_ref current_state = n + 14
  ⟹ read_register last_instruction_pointer_ref current_state = n+8›

  ‹read_register instruction_pointer_ref current_state = n + 14
  ⟹ read_register r00_ref current_state
    = of_nat (fib (unat (read_register arg00_ref current_state)))›
  using a1 trace_rules function_start
  by(simp_all add: check_trace_def list_all_def Ball_def state_in_trace)

have check_trace_decomposition_n15:
  ‹read_register instruction_pointer_ref current_state = n + 15
  ⟹ typical_flags current_state›

  ‹read_register instruction_pointer_ref current_state = n + 15
  ⟹ read_register r00_ref current_state
    = of_nat (fib (unat (read_register arg00_ref current_state)))›
  using a1 trace_rules function_start
  by(simp_all add: check_trace_def list_all_def Ball_def state_in_trace)

have check_trace_decomposition_n16:
  ‹read_register instruction_pointer_ref current_state = n + 16
  ⟹ typical_flags current_state›

  ‹read_register instruction_pointer_ref current_state = n + 16
  ⟹ read_register ret00_ref current_state
    = of_nat (fib (unat (read_register arg00_ref current_state)))›
  using a1 trace_rules function_start
  by(simp_all add: check_trace_def list_all_def Ball_def state_in_trace)

have current_instruction:
  ‹read_register instruction_pointer_ref current_state = n
  ⟹ next_state = END_CALL current_state›
```

```
⟨read_register instruction_pointer_ref current_state = n + 1
⟹ next_state = LOAD_IMMEDIATE c01_ref 1 current_state⟩

⟨read_register instruction_pointer_ref current_state = n + 2
⟹ next_state = LOAD_IMMEDIATE r03_ref 0 current_state⟩

⟨read_register instruction_pointer_ref current_state = n + 3
⟹ next_state = LOAD_IMMEDIATE r00_ref 1 current_state⟩

⟨read_register instruction_pointer_ref current_state = n + 4
⟹ next_state = LOAD_IMMEDIATE r01_ref 1 current_state⟩

⟨read_register instruction_pointer_ref current_state = n + 5
⟹ next_state = LOAD_IMMEDIATE r02_ref 2 current_state⟩

⟨read_register instruction_pointer_ref current_state = n + 6
⟹ next_state = END_JUMP (n+13) current_state⟩

⟨read_register instruction_pointer_ref current_state = n + 7
⟹ next_state = EQUALS r04_ref r03_ref arg00_ref current_state⟩

⟨read_register instruction_pointer_ref current_state = n + 8
⟹ next_state = CONDITIONAL_JUMP r04_ref (n+14) current_state⟩

⟨read_register instruction_pointer_ref current_state = n + 9
⟹ next_state = COPY r00_ref r01_ref current_state⟩

⟨read_register instruction_pointer_ref current_state = n + 10
⟹ next_state = COPY r01_ref r02_ref current_state⟩

⟨read_register instruction_pointer_ref current_state = n + 11
⟹ next_state = ADD r02_ref r00_ref r01_ref current_state⟩

⟨read_register instruction_pointer_ref current_state = n + 12
⟹ next_state = ADD r03_ref r03_ref c01_ref current_state⟩

⟨read_register instruction_pointer_ref current_state = n + 13
⟹ next_state = JUMP (n+6) current_state⟩

⟨read_register instruction_pointer_ref current_state = n + 14
⟹ next_state = END_JUMP (n+8) current_state⟩

⟨read_register instruction_pointer_ref current_state = n + 15
⟹ next_state = COPY ret00_ref r00_ref current_state⟩

⟨read_register instruction_pointer_ref current_state = n + 16
⟹ next_state = RETURN current_state⟩
```
using

```
    check_trace_decomposition_n0(1)
    check_trace_decomposition_n1(1)
    check_trace_decomposition_n2(1)
    check_trace_decomposition_n3(1)
    check_trace_decomposition_n4(1)
    check_trace_decomposition_n5(1)
    check_trace_decomposition_n6(1)
    check_trace_decomposition_n7(1)
    check_trace_decomposition_n8(1)
    check_trace_decomposition_n9(1)
    check_trace_decomposition_n10(1)
    check_trace_decomposition_n11(1)
    check_trace_decomposition_n12(1)
    check_trace_decomposition_n13(1)
    check_trace_decomposition_n14(1)
    check_trace_decomposition_n15(1)
    check_trace_decomposition_n16(1)
  apply(simp_all add: next_state_def)
  unfolding execute_next_instruction_def
  apply(simp_all)
  unfolding fetch_instruction_def
  apply(simp_all add: trace_program_memory3)
  apply(simp_all add: read_program_memory_def program_loaded program_def)
  apply(simp_all add: decode_hacks)
  apply(simp_all add: decode_decomp)
  apply(simp_all add: execute_instruction_simps)
  apply(elim disjE)
  apply(simp_all)
  done

have n0:
  ‹read_register instruction_pointer_ref next_state = n + 1›
  ‹typical_flags next_state›
  if a0: ‹read_register instruction_pointer_ref current_state = n›
proof -
  have current_instruction:
    ‹read_register instruction_pointer_ref current_state = n
    ⟹ next_state = END_CALL current_state›
    using check_trace_decomposition_n0(1)
    apply(simp add: next_state_def)
    apply(simp add: execute_next_instruction_def)
    apply(simp add: fetch_instruction_def)
    apply(simp_all add: trace_program_memory3)
    apply(simp_all add: read_program_memory_def program_loaded program_def)
    apply(simp_all add: decode_decomp)
    apply(simp_all add: execute_instruction_simps)
    done
  show
    ‹read_register instruction_pointer_ref next_state = n + 1›
```

```
    ‹typical_flags next_state›
    using a0 current_instruction check_trace_decomposition_n0
    by(simp_all add: END_CALL_simps)
qed

have n1:
  ‹read_register instruction_pointer_ref next_state = n + 2›
  ‹typical_flags next_state›
  ‹read_register c01_ref next_state = 1›
  if a0: ‹read_register instruction_pointer_ref current_state = n + 1›
  using
    a0
    current_instruction(2)
    check_trace_decomposition_n1
    check_write_permission_set
  by(simp_all add: LOAD_IMMEDIATE_simps)

have n2:
  ‹read_register instruction_pointer_ref next_state = n + 3›
  ‹typical_flags next_state›
  ‹read_register c01_ref next_state = 1›
  ‹read_register r03_ref next_state = 0›
  ‹read_register r03_ref next_state ≤ read_register arg00_ref next_state›
  if a0: ‹read_register instruction_pointer_ref current_state = n + 2›
  using
    a0
    current_instruction(3)
    check_trace_decomposition_n2
    check_write_permission_set
  by(simp_all add: LOAD_IMMEDIATE_simps)

have n3:
  ‹read_register instruction_pointer_ref next_state = n + 4›
  ‹typical_flags next_state›
  ‹read_register c01_ref next_state = 1›
  ‹read_register r03_ref next_state = 0›
  ‹read_register r03_ref next_state ≤ read_register arg00_ref next_state›

  ‹read_register r00_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state)))›
  if a0: ‹read_register instruction_pointer_ref current_state = n + 3›
  using
    a0
    current_instruction(4)
    check_trace_decomposition_n3
    check_write_permission_set
  by(simp_all add: LOAD_IMMEDIATE_simps)

have n4:
```

```
  ‹read_register instruction_pointer_ref next_state = n + 5›
  ‹typical_flags next_state›
  ‹read_register c01_ref next_state = 1›
  ‹read_register r03_ref next_state = 0›
  ‹read_register r03_ref next_state ≤ read_register arg00_ref next_state›

  ‹read_register r00_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state)))›

  ‹read_register r01_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 1))›
  if a0: ‹read_register instruction_pointer_ref current_state = n + 4›
  using
    a0
    current_instruction(5)
    check_trace_decomposition_n4
    check_write_permission_set
  by(simp_all add: LOAD_IMMEDIATE_simps fib_answers)

have n5:
  ‹read_register instruction_pointer_ref next_state = n + 6›
  ‹typical_flags next_state›
  ‹get_end_jump next_state = 0›
  ‹read_register last_instruction_pointer_ref next_state = n + 5›
  ‹read_register c01_ref next_state = 1›
  ‹read_register r03_ref next_state = 0›
  ‹read_register r03_ref next_state ≤ read_register arg00_ref next_state›

  ‹read_register r00_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state)))›

  ‹read_register r01_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

  ‹read_register r02_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
  if a0: ‹read_register instruction_pointer_ref current_state = n + 5›
  using a0 current_instruction(6) check_trace_decomposition_n5 check_write_permission_set
  by(simp_all add: LOAD_IMMEDIATE_simps fib_answers)

have n6:
  ‹read_register instruction_pointer_ref next_state = n + 7›
  ‹typical_flags next_state›
  ‹read_register c01_ref next_state = 1›
  ‹read_register r03_ref next_state ≤ read_register arg00_ref next_state›

  ‹read_register r00_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state)))›
```

```
  ‹read_register r01_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

  ‹read_register r02_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
  if a0: ‹read_register instruction_pointer_ref current_state = n + 6›
proof (all ‹cases ‹typical_flags current_state››)
  assume ‹typical_flags current_state›
  then show
    ‹read_register instruction_pointer_ref next_state = n + 7›
    ‹typical_flags next_state›
    ‹read_register c01_ref next_state = 1›
    ‹read_register r03_ref next_state ≤ read_register arg00_ref next_state›

    ‹read_register r00_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state)))›

    ‹read_register r01_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

    ‹read_register r02_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
    using a0 current_instruction(7) check_trace_decomposition_n6
    by(simp_all add: END_JUMP_simps)
next
  assume ‹¬ typical_flags current_state›
  then have ‹end_jump_flags current_state›
    using a0 check_trace_decomposition_n6(1)
    by(linarith)
  then show
    ‹read_register instruction_pointer_ref next_state = n + 7›
    ‹typical_flags next_state›
    ‹read_register c01_ref next_state = 1›
    ‹read_register r03_ref next_state ≤ read_register arg00_ref next_state›

    ‹read_register r00_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state)))›

    ‹read_register r01_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

    ‹read_register r02_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
    using a0 current_instruction(7) check_trace_decomposition_n6
    by(simp_all add: END_JUMP_simps)
qed

have n7:
  ‹read_register instruction_pointer_ref next_state = n + 8›
```

```
⟨typical_flags next_state⟩
⟨read_register c01_ref next_state = 1⟩
⟨read_register r03_ref next_state ≤ read_register arg00_ref next_state⟩

⟨read_register r00_ref next_state
= of_nat (fib (unat (read_register r03_ref next_state)))⟩

⟨read_register r01_ref next_state
= of_nat (fib (unat (read_register r03_ref next_state) + 1))⟩

⟨read_register r02_ref next_state
= of_nat (fib (unat (read_register r03_ref next_state) + 2))⟩

⟨(read_register r03_ref next_state = read_register arg00_ref next_state)
⟶ (read_register r04_ref next_state = 1)⟩

⟨(read_register r03_ref next_state ≠ read_register arg00_ref next_state)
⟶ (read_register r04_ref next_state = 0)⟩

⟨read_register r04_ref next_state = 1 ∨ read_register r04_ref next_state = 0⟩
if a0: ⟨read_register instruction_pointer_ref current_state = n + 7⟩
using
  a0
  current_instruction(8)
  check_trace_decomposition_n7
  check_write_permission_set
  check_read_permission_set
by(simp_all add: EQUALS_simps)

have n8:
  ⟨read_register r04_ref current_state = 0
  ⟶ (read_register instruction_pointer_ref next_state = n + 9)⟩

  ⟨read_register r04_ref current_state = 1
  ⟶ (read_register instruction_pointer_ref next_state = n + 14)⟩

  ⟨read_register r04_ref current_state = 0
  ⟶ typical_flags next_state⟩

  ⟨read_register r04_ref current_state = 1
  ⟶ end_jump_flags next_state⟩

  ⟨read_register r04_ref current_state = 1
  ⟶ (read_register last_instruction_pointer_ref next_state = n + 8)⟩

  ⟨read_register c01_ref next_state = 1⟩

  ⟨read_register r04_ref current_state = 0
  ⟶ (read_register r03_ref next_state < read_register arg00_ref next_state)⟩
```

```
  ‹read_register r04_ref current_state = 1
  ⟶ (read_register r03_ref next_state = read_register arg00_ref next_state)›

  ‹read_register r00_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state)))›

  ‹read_register r01_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

  ‹read_register r02_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
if a0: ‹read_register instruction_pointer_ref current_state = n + 8›
proof (all ‹cases ‹read_register r04_ref current_state = 0››)
  have h1:
    ‹check_read_permission r04_ref›
    by(simp add: check_read_permission_set)
  {
    assume ‹read_register 4 current_state = 0›
    then show
      ‹(read_register r04_ref current_state = 0)
      ⟶ (read_register instruction_pointer_ref next_state = n + 9)›

      ‹(read_register r04_ref current_state = 1)
      ⟶ (read_register instruction_pointer_ref next_state = n + 14)›

      ‹(read_register r04_ref current_state = 0)
      ⟶ typical_flags next_state›

      ‹(read_register r04_ref current_state = 1)
      ⟶ end_jump_flags next_state›

      ‹(read_register r04_ref current_state = 1)
      ⟶ (read_register last_instruction_pointer_ref next_state = n + 8)›

      ‹read_register c01_ref next_state = 1›

      ‹(read_register r04_ref current_state = 0)
      ⟶ (read_register r03_ref next_state < read_register arg00_ref next_state)›

      ‹read_register r04_ref current_state = 1
      ⟶ (read_register r03_ref next_state = read_register arg00_ref next_state)›

      ‹read_register r00_ref next_state
      = of_nat (fib (unat (read_register r03_ref next_state)))›

      ‹read_register r01_ref next_state
      = of_nat (fib (unat (read_register r03_ref next_state) + 1))›
```

```
      ‹read_register r02_ref next_state
      = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
      using a0 h1 current_instruction(9) check_trace_decomposition_n8
      by(simp_all add: CONDITIONAL_JUMP_simps)
    next
      assume ‹read_register 4 current_state ≠ 0›
      then show
      ‹read_register r04_ref current_state = 0
      ⟶ (read_register instruction_pointer_ref next_state = n + 9)›

      ‹read_register r04_ref current_state = 1
      ⟶ (read_register instruction_pointer_ref next_state = n + 14)›

      ‹read_register r04_ref current_state = 0
      ⟶ typical_flags next_state›

      ‹read_register r04_ref current_state = 1
      ⟶ end_jump_flags next_state›

      ‹read_register r04_ref current_state = 1
      ⟶ (read_register last_instruction_pointer_ref next_state = n + 8)›

      ‹read_register c01_ref next_state = 1›

      ‹read_register r04_ref current_state = 0
      ⟶ (read_register r03_ref next_state < read_register arg00_ref next_state)›

      ‹read_register r04_ref current_state = 1
      ⟶ (read_register r03_ref next_state = read_register arg00_ref next_state)›

      ‹read_register r00_ref next_state
      = of_nat (fib (unat (read_register r03_ref next_state)))›

      ‹read_register r01_ref next_state
      = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

      ‹read_register r02_ref next_state
      = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
      using a0 h1 current_instruction(9) check_trace_decomposition_n8
      by(simp_all add: CONDITIONAL_JUMP_simps)
  }
qed

have n9:
  ‹read_register instruction_pointer_ref next_state = n + 10›
  ‹typical_flags next_state›
  ‹read_register c01_ref next_state = 1›
  ‹read_register r03_ref next_state < read_register arg00_ref next_state›
```

```
  ‹read_register r00_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

  ‹read_register r01_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

  ‹read_register r02_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
  if a0: ‹read_register instruction_pointer_ref current_state = n + 9›
proof -
  have
    ‹check_write_permission r00_ref›
    ‹check_read_permission r01_ref›
    by(simp_all add: permission_simps)
  then show
    ‹read_register instruction_pointer_ref next_state = n + 10›
    ‹typical_flags next_state›
    ‹read_register c01_ref next_state = 1›
    ‹read_register r03_ref next_state < read_register arg00_ref next_state›

    ‹read_register r00_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

    ‹read_register r01_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

    ‹read_register r02_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
    using a0 current_instruction(10) check_trace_decomposition_n9
    by(simp_all add: COPY_simps)
qed

have n10:
  ‹read_register instruction_pointer_ref next_state = n + 11›
  ‹typical_flags next_state›
  ‹read_register c01_ref next_state = 1›
  ‹read_register r03_ref next_state < read_register arg00_ref next_state›

  ‹read_register r00_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

  ‹read_register r01_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 2))›

  ‹read_register r02_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
  if a0: ‹read_register instruction_pointer_ref current_state = n + 10›
 proof -
  have
```

```
    ‹check_write_permission r01_ref›
    ‹check_read_permission r02_ref›
    by(simp_all add: permission_simps)
  then show
    ‹read_register instruction_pointer_ref next_state = n + 11›
    ‹typical_flags next_state›
    ‹read_register c01_ref next_state = 1›
    ‹read_register r03_ref next_state < read_register arg00_ref next_state›

    ‹read_register r00_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

    ‹read_register r01_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 2))›

    ‹read_register r02_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
    using a0 current_instruction(11) check_trace_decomposition_n10
    by(simp_all add: COPY_simps)
qed

have n11:
  ‹read_register instruction_pointer_ref next_state = n + 12›
  ‹typical_flags next_state›
  ‹read_register c01_ref next_state = 1›
  ‹read_register r03_ref next_state < read_register arg00_ref next_state›

  ‹read_register r00_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

  ‹read_register r01_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 2))›

  ‹read_register r02_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 3))›
  if a0: ‹read_register instruction_pointer_ref current_state = n + 11›
proof -
  have
    ‹check_write_permission r02_ref›
    ‹check_read_permission r00_ref›
    ‹check_read_permission r01_ref›
    by(simp_all add: permission_simps)
  then show
    ‹read_register instruction_pointer_ref next_state = n + 12›
    ‹typical_flags next_state›
    ‹read_register c01_ref next_state = 1›
    ‹read_register r03_ref next_state < read_register arg00_ref next_state›

    ‹read_register r00_ref next_state
```

```
    = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

    ‹read_register r01_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 2))›

    ‹read_register r02_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 3))›
    using a0 current_instruction(12) check_trace_decomposition_n11
    by(simp_all add: ADD_simps fib_stuff(1))
qed

have n12:
  ‹read_register instruction_pointer_ref next_state = n + 13›
  ‹typical_flags next_state›
  ‹read_register c01_ref next_state = 1›
  ‹read_register r03_ref next_state ≤ read_register arg00_ref next_state›

  ‹read_register r00_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state)))›

  ‹read_register r01_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

  ‹read_register r02_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
  if a0: ‹read_register instruction_pointer_ref current_state = n + 12›
proof -
  have
    ‹check_write_permission r03_ref›
    ‹check_read_permission r03_ref›
    ‹check_read_permission c01_ref›
    by(simp_all add: permission_simps)
  then show
    ‹read_register instruction_pointer_ref next_state = n + 13›
    ‹typical_flags next_state›
    ‹read_register c01_ref next_state = 1›
    ‹read_register r03_ref next_state ≤ read_register arg00_ref next_state›

    ‹read_register r00_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state)))›

    ‹read_register r01_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

    ‹read_register r02_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
    using a0 current_instruction(13) check_trace_decomposition_n12
    by(simp_all add: ADD_simps fib_stuff(12) add_with_less_than inc_le)
qed
```

```
have n13:
  ‹read_register instruction_pointer_ref next_state = n + 6›
  ‹end_jump_flags next_state›
  ‹get_end_jump next_state = 1›
  ‹read_register c01_ref next_state = 1›
  ‹read_register last_instruction_pointer_ref next_state = n + 13›
  ‹read_register r03_ref next_state ≤ read_register arg00_ref next_state›

  ‹read_register r00_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state)))›

  ‹read_register r01_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

  ‹read_register r02_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
  if a0: ‹read_register instruction_pointer_ref current_state = n + 13›
  using a0 current_instruction(14) check_trace_decomposition_n13
  by(simp_all add: JUMP_simps)

have n14:
  ‹read_register instruction_pointer_ref next_state = n + 15›
  ‹typical_flags next_state›
  ‹read_register r00_ref next_state
  = of_nat (fib (unat (read_register arg00_ref next_state)))›
  if a0: ‹read_register instruction_pointer_ref current_state = n + 14›
  using a0 current_instruction(15) check_trace_decomposition_n14
  by(simp_all add: END_JUMP_simps)

have n15:
  ‹read_register instruction_pointer_ref next_state = n + 16›
  ‹typical_flags next_state›

  ‹read_register ret00_ref next_state
  = of_nat (fib (unat(read_register arg00_ref next_state)))›
  if a0: ‹read_register instruction_pointer_ref current_state = n + 15›
proof -
  have
    ‹check_write_permission ret00_ref›
    ‹check_read_permission r00_ref›
    by(simp_all add: permission_simps)
  then show
    ‹read_register instruction_pointer_ref next_state = n + 16›
    ‹typical_flags next_state›

    ‹read_register ret00_ref next_state
    = of_nat (fib (unat (read_register arg00_ref next_state)))›
    using a0 current_instruction(16) check_trace_decomposition_n15
```

```
    by(simp_all add: COPY_simps)
qed

have n16:
  ‹read_register instruction_pointer_ref current_state = n + 16
  ⟹ get_opcode (fetch_instruction current_state) = 31›
  apply(simp add: fetch_instruction_def)
  apply(simp add: trace_program_memory3)
  apply(simp add: read_program_memory_def)
  apply(simp add: program_loaded program_def)
  apply(simp add: get_opcode_def slice_def slice1_def)
  done

show
  ‹check_trace (finish_function_trace [starting_state] x
    @ [next_state])›
  using a1 a2
  apply(subst check_trace_def)
  apply(simp)
  apply(subst (asm) check_trace_def)
  using function_start
  apply(simp)
  using trace_rules
  apply(simp)

  apply(simp add: list_all_def Ball_def)
  apply(elim allE [where x=‹current_state›])
  apply(simp add: state_in_trace)
  apply(erule conjE)
  apply(simp add: program_def)
  apply(elim disjE)
  apply(simp_all)

  apply(simp add: n16)
  apply(simp add: n15 program_def)
  apply(simp add: n14)
  apply(simp add: n13)
  apply(simp add: n12)
  apply(simp add: n11)
  apply(simp add: n10)
  apply(simp add: n9)

  apply(cases ‹read_register r04_ref current_state = 0›)
  apply(simp add: n8)
  apply(simp add: n8)

  apply(simp add: n7)
  apply(simp add: n6)
  apply(simp add: n5)
```

```
      apply (simp add: n4)
      apply (simp add: n3)
      apply (simp add: n2)
      apply (simp add: n1)
      apply (simp add: n0)
      done
  qed
qed

end
```

A Preliminaries

```
theory preliminaries

imports
  Main
  "HOL-Library.Word"
  "HOL-Eisbach.Eisbach"

begin
```

This file collects various technical lemmas that extend the HOL/Word base but aren't specifically about Ironbark. This is also the entry point into Ironbark, so these lemmas are available to all subsequent theories. Also, it's useful to collect them to see if they can be rationalised, generalised, or removed.

The following lemmas is used to exhaust the possible values of a word when it has been bounded by a less than or equal to operation.

```
lemma word_le_decr:
  fixes
    z :: ‹('a :: len) word›
  assumes
    ‹w > 0›
  shows
    ‹z ≤ w ⟷ z ≤ w - 1 ∨ z = w›
  using assms
  by (uint_arith)
```

We found that the following lemma is useful when trying to show equivalence between a term of word type, and a term nat type. The particular case shown here is one that we encounter in our Fibonacci proofs, but we expect that other applications will require other variations.

```
lemma add_with_less_than:
  fixes
    a b :: ‹64 word›
```

```
  assumes
    ‹a < b›
  shows
    ‹unat(a + 1) = unat(a) + 1›
proof -
  have
    ‹a < a+1›
    using assms
    by(uint_arith)
  then show ?thesis
    apply(subst unat_1[symmetric])
    apply(subst unat_plus_simple[symmetric])
    apply(simp)
    done
qed
```

The following lemma is used to standardise the way we represent single bit words ("flags"). We do this mostly to save us writing lemmas for each of the various ways that flags can end up being represented.

```
lemma flag_normalisation:
  fixes
    a :: ‹1 word›
  shows
    ‹a ∈ {0, 1}›
    ‹¬(bit (a :: 1 word) 0) ⟷ a = 0›
    ‹even (a :: 1 word) ⟷ a = 0›
    ‹(bit (a :: 1 word) 0) ⟷ a = 1›
    ‹odd (a :: 1 word) ⟷ a = 1›
    ‹a ≠ 1 ⟷ a = 0›
    ‹a ≠ 0 ⟷ a = 1›
  by(cases ‹a›, auto simp add: less_2_cases_iff)+

end
```

B State Level Lemmas

B.1 Properties of the Processor State Definition

```
theory state_auxiliary

imports
  state_implementations

begin
```

We provide a trivial proof here that shows that two states can be shown to be equivalent if all of the components that make up the state are the same. This is used in future proofs to attack larger problems in a 'piecemeal' fashion.

```
lemma sequential_state_equality:
  fixes
    state1 state2 :: ‹sequential_state›
  assumes
    ‹flag_state     state1 = flag_state     state2›
    ‹register_state state1 = register_state state2›
    ‹program_memory state1 = program_memory state2›
    ‹call_memory    state1 = call_memory    state2›
    ‹static_memory  state1 = static_memory  state2›
    ‹dynamic_memory state1 = dynamic_memory state2›
    ‹input_memory   state1 = input_memory   state2›
    ‹output_memory  state1 = output_memory  state2›
  shows
    ‹state1 = state2›
  using assms
  by(simp)

end
```

B.2 Properties of Register Set Definitions

In this section we will show that the way we defined different sets of registers are somewhat sensible, and various properties hold that may be expected. These properties are heavily used in future proofs that show that it is possible to write both secure programs and secure functions that always behave correctly if the assumptions made in the definitions hold.

```
theory register_sets

imports
  state_auxiliary

begin
```

The following lemma shows that we didn't assign a register ID to multiple registers in different sets, by showing that all register sets are disjoint (i.e. unique sets with no overlap).

```
lemma generic_special_disjnt:
  ‹disjnt generic_registers         special_purpose_registers›
  ‹disjnt generic_registers         special_address_registers›
  ‹disjnt special_purpose_registers special_address_registers›
  unfolding
    generic_registers_def
    special_purpose_registers_def
    special_address_registers_def
  by (simp_all)
```

This is a rewrite of the above lemma using the pairwise disjoint operator.

```
lemma generic_special_disjnt':
  ‹pairwise disjnt {
    generic_registers,
```

```
    special_purpose_registers,
    special_address_registers
  }›
  unfolding
    generic_registers_def
    special_purpose_registers_def
    special_address_registers_def
  by (simp add: pairwise_def)
```

The following lemma shows that we didn't miss any registers in the generic registers, showing that the numbers from 0x00 to 0x4f (inclusive) are the same as the generic register set.

```
lemma le_ret15_ref_generic:
  ‹r ≤ 0x4f ⟷ r ∈ generic_registers›
proof -
  have
    ‹r ∈ generic_registers ⟹ r ≤ 0x4f›
    unfolding generic_registers_def
    by(auto)
  moreover have
    ‹r ≤ 0x4f ⟹ r ∈ generic_registers›
    unfolding generic_registers_def
    by(simp add: word_le_decr)
  ultimately show ?thesis
    by(auto)
qed
```

The following collection of lemmas shows that all of our subsets of registers are part of the full set of registers. It also shows that all generic registers are readable and writeable.

```
lemma register_subsets:
  ‹regID ∈ readable_registers         ⟹ regID ∈ registers›
  ‹regID ∈ writeable_registers        ⟹ regID ∈ registers›
  ‹regID ∈ special_address_registers  ⟹ regID ∈ registers›
  ‹regID ∈ special_purpose_registers  ⟹ regID ∈ registers›
  ‹regID ∈ generic_registers          ⟹ regID ∈ registers›
  ‹regID ∈ generic_registers ⟹ regID ∈ readable_registers›
  ‹regID ∈ generic_registers ⟹ regID ∈ writeable_registers›
  unfolding
    registers_def
    writeable_registers_def
    readable_registers_def
    generic_registers_def
    special_purpose_registers_def
    special_address_registers_def
  by(auto)
```

Here we write out the full set of the readable registers, which is convenient for future proofs.

```
lemma readable_set:
  assumes
```

```
  ‹regID ∈ {
    r00_ref,   r01_ref,   r02_ref,   r03_ref,
    r04_ref,   r05_ref,   r06_ref,   r07_ref,
    r08_ref,   r09_ref,   r10_ref,   r11_ref,
    r12_ref,   r13_ref,   r14_ref,   r15_ref,
    p00_ref,   p01_ref,   p02_ref,   p03_ref,
    p04_ref,   p05_ref,   p06_ref,   p07_ref,
    p08_ref,   p09_ref,   p10_ref,   p11_ref,
    p12_ref,   p13_ref,   p14_ref,   p15_ref,
    c00_ref,   c01_ref,   c02_ref,   c03_ref,
    c04_ref,   c05_ref,   c06_ref,   c07_ref,
    c08_ref,   c09_ref,   c10_ref,   c11_ref,
    c12_ref,   c13_ref,   c14_ref,   c15_ref,
    arg00_ref, arg01_ref, arg02_ref, arg03_ref,
    arg04_ref, arg05_ref, arg06_ref, arg07_ref,
    arg08_ref, arg09_ref, arg10_ref, arg11_ref,
    arg12_ref, arg13_ref, arg14_ref, arg15_ref,
    ret00_ref, ret01_ref, ret02_ref, ret03_ref,
    ret04_ref, ret05_ref, ret06_ref, ret07_ref,
    ret08_ref, ret09_ref, ret10_ref, ret11_ref,
    ret12_ref, ret13_ref, ret14_ref, ret15_ref,

    arg_frame_pointer_ref,
    arg_stack_pointer_ref,
    dynamic_data_frame_pointer_ref,
    dynamic_data_stack_pointer_ref,
    static_data_frame_pointer_ref,
    static_data_stack_pointer_ref,

    cycles_register_ref
  }›
shows
  ‹regID ∈ readable_registers›
using assms
unfolding readable_registers_def
unfolding generic_registers_def
unfolding special_address_registers_def
by(simp add: insert_commute)
```

Occasionally, we need to show that a register is not readable. We provide the following lemma for this purpose.

```
lemma readable_exclude_set:
  assumes
    ‹regID ∈ readable_registers›
  shows
    ‹regID ≠ instruction_pointer_ref›
    ‹regID ≠ last_instruction_pointer_ref›
    ‹regID ≠ call_frame_pointer_ref›
  using assms
```

```
  unfolding readable_registers_def
  unfolding generic_registers_def
  unfolding special_address_registers_def
  by(auto)
```

As with readable registers, we also write out the full set of writeable registers for the convenience of future proofs.

```
lemma writeable_set:
  assumes
    ‹regID ∈ {
      r00_ref,   r01_ref,   r02_ref,   r03_ref,
      r04_ref,   r05_ref,   r06_ref,   r07_ref,
      r08_ref,   r09_ref,   r10_ref,   r11_ref,
      r12_ref,   r13_ref,   r14_ref,   r15_ref,
      p00_ref,   p01_ref,   p02_ref,   p03_ref,
      p04_ref,   p05_ref,   p06_ref,   p07_ref,
      p08_ref,   p09_ref,   p10_ref,   p11_ref,
      p12_ref,   p13_ref,   p14_ref,   p15_ref,
      c00_ref,   c01_ref,   c02_ref,   c03_ref,
      c04_ref,   c05_ref,   c06_ref,   c07_ref,
      c08_ref,   c09_ref,   c10_ref,   c11_ref,
      c12_ref,   c13_ref,   c14_ref,   c15_ref,
      arg00_ref, arg01_ref, arg02_ref, arg03_ref,
      arg04_ref, arg05_ref, arg06_ref, arg07_ref,
      arg08_ref, arg09_ref, arg10_ref, arg11_ref,
      arg12_ref, arg13_ref, arg14_ref, arg15_ref,
      ret00_ref, ret01_ref, ret02_ref, ret03_ref,
      ret04_ref, ret05_ref, ret06_ref, ret07_ref,
      ret08_ref, ret09_ref, ret10_ref, ret11_ref,
      ret12_ref, ret13_ref, ret14_ref, ret15_ref,

      arg_frame_pointer_ref,
      arg_stack_pointer_ref,
      dynamic_data_frame_pointer_ref,
      dynamic_data_stack_pointer_ref,
      static_data_frame_pointer_ref,
      static_data_stack_pointer_ref
    }›
  shows
    ‹regID ∈ writeable_registers›
  using assms
  unfolding writeable_registers_def
  unfolding generic_registers_def
  unfolding special_address_registers_def
  by(simp add: insert_commute)
```

As with readable registers, we occasionally need to show that a register is not writeable. We provide the following lemma for this purpose.

```
lemma writeable_exclude_set:
```

```
  assumes
    ‹regID ∈ writeable_registers›
  shows
    ‹regID ≠ cycles_register_ref›
    ‹regID ≠ instruction_pointer_ref›
    ‹regID ≠ last_instruction_pointer_ref›
    ‹regID ≠ call_frame_pointer_ref›
  using assms
  unfolding writeable_registers_def
  unfolding generic_registers_def
  unfolding special_address_registers_def
  by(auto)
```

Finally, we also write out the full set of all registers for the convenience of future proofs.

```
lemma register_set:
  assumes
    ‹regID ∈ {
      r00_ref,   r01_ref,   r02_ref,   r03_ref,
      r04_ref,   r05_ref,   r06_ref,   r07_ref,
      r08_ref,   r09_ref,   r10_ref,   r11_ref,
      r12_ref,   r13_ref,   r14_ref,   r15_ref,
      p00_ref,   p01_ref,   p02_ref,   p03_ref,
      p04_ref,   p05_ref,   p06_ref,   p07_ref,
      p08_ref,   p09_ref,   p10_ref,   p11_ref,
      p12_ref,   p13_ref,   p14_ref,   p15_ref,
      c00_ref,   c01_ref,   c02_ref,   c03_ref,
      c04_ref,   c05_ref,   c06_ref,   c07_ref,
      c08_ref,   c09_ref,   c10_ref,   c11_ref,
      c12_ref,   c13_ref,   c14_ref,   c15_ref,
      arg00_ref, arg01_ref, arg02_ref, arg03_ref,
      arg04_ref, arg05_ref, arg06_ref, arg07_ref,
      arg08_ref, arg09_ref, arg10_ref, arg11_ref,
      arg12_ref, arg13_ref, arg14_ref, arg15_ref,
      ret00_ref, ret01_ref, ret02_ref, ret03_ref,
      ret04_ref, ret05_ref, ret06_ref, ret07_ref,
      ret08_ref, ret09_ref, ret10_ref, ret11_ref,
      ret12_ref, ret13_ref, ret14_ref, ret15_ref,

      cycles_register_ref,
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      call_frame_pointer_ref,

      arg_frame_pointer_ref,
      arg_stack_pointer_ref,
      dynamic_data_frame_pointer_ref,
      dynamic_data_stack_pointer_ref,
      static_data_frame_pointer_ref,
      static_data_stack_pointer_ref
```

```
    }›
  shows
    ‹regID ∈ registers›
  using assms
  unfolding registers_def
  unfolding
    generic_registers_def
    special_purpose_registers_def
    special_address_registers_def
  by(simp add: insert_commute)

end
```

B.3 State Interface Layer

```
theory state_top

imports
  register_sets

begin
```

Here we collect up all the results that are useful for future proofs, and place them under the one name for convenient reference later.

```
lemmas state_simps =
  register_subsets
  readable_set
  readable_exclude_set
  writeable_set
  writeable_exclude_set
  register_set

end
```

C State Manipulation Level Lemmas

C.1 Properties of State Manipulation Definitions

```
theory state_manipulation_auxiliary

imports
  "Ironbark_state.state_top"
  state_manipulation_implementations

begin
```

We provide a trivial proof here that the 8-bit opcode for any instruction will be less than or equal to 255. This is obvious from the fact that it is the maximum value that can be stored in an 8-bit integer.

```
lemma opcode_bound:
  ‹get_opcode instruction ≤ 255›
  by (unat_arith, auto)
```

We also provide a trivial proof here that if the underlying memory is the same, then a read from that memory space will give the same result. This is not included in the simplification rules as it tends to cause the simplifier to get stuck in an infinite loop.

```
lemma (in Ironbark_world) same_memory_same_addr:
  shows
    ‹program_memory state1 = program_memory state2
      ⟹ read_program_memory addr state1 = read_program_memory addr state2›

    ‹call_memory state1 = call_memory state2
      ⟹ read_call_memory addr state1 = read_call_memory addr state2›

    ‹static_memory state1 = static_memory state2
      ⟹ read_static_memory addr state1 = read_static_memory addr state2›

    ‹dynamic_memory state1 = dynamic_memory state2
      ⟹ read_dynamic_memory addr state1 = read_dynamic_memory addr state2›

    ‹input_memory state1 = input_memory state2
      ⟹ read_input_memory addr state1 = read_input_memory addr state2›
  by (simp_all add: read_program_memory_def read_call_memory_def read_static_memory_def
      read_dynamic_memory_def read_input_memory_def)
```

end

C.2 State Manipulation Decomposition Rules

C.2.1 Decomposition Rules for Flags

theory *flag_decomposition*

imports
  state_manipulation_auxiliary

begin

In this section we collect up the various definitions relating to reading and writing flags for easy reference later. These are used to decompose state manipulation operations into operations on the state components.

We first collect all the definitions for reading flags.

lemmas *read_flag_decomp =*

```
  get_error_def
  get_halt_def
  get_end_jump_def
  get_end_call_def
  get_end_return_def
```

We then collect all the definitions for writing (set and clear).

```
lemmas set_clear_flag_decomp =
  set_end_return_def
  set_end_call_def
  set_end_jump_def
  set_halt_def
  set_error_def
  clear_end_return_def
  clear_end_call_def
  clear_end_jump_def
  clear_halt_def
  clear_error_def
```

We then bundle the reading and writing versions together for easier reference when needed.

```
lemmas flag_decomp =
  read_flag_decomp
  set_clear_flag_decomp
```

end

C.2.2 Decomposition Rules for Registers

theory *register_decomposition*

imports
state_manipulation_auxiliary

begin

In this section we collect the definition for reading a register, and show how writing registers can be decomposed to state level operations. These are used to decompose state manipulation operations into operations on the state components.

We first collect the definition for reading registers - essentially just renaming it to match the naming pattern of other decomposition rules.

```
lemmas read_register_decomp =
  read_register_def
```

We then provide two ways of decomposing writing registers. The second is applicable to a subset of the first, but is convenient as we often have check_write_permission in the assumptions of future proofs, so we provide it here for convenience.

lemma *write_register_set_decomp:*

```
  assumes
    ‹regID ∈ registers›
  shows
    ‹write_register regID value state
    = state(|register_state := (register_state state)(regID ↦ value)|)›
  using assms
  unfolding write_register_def
  by(simp add: state_simps)

lemma write_check_permission_decomp:
  assumes
    ‹check_write_permission regID›
  shows
    ‹write_register regID value state
    = state(|register_state := (register_state state)(regID ↦ value)|)›
  using assms
  unfolding write_register_def check_write_permission_def
  by(simp add: state_simps)
```

We then bundle the reading and writing versions together for easier reference when needed.

```
lemmas register_decomp =
  read_register_decomp
  write_register_set_decomp
  write_check_permission_decomp

end
```

C.2.3 Decomposition Rules for Memory

```
theory memory_decomposition

imports
  state_manipulation_auxiliary

begin
```

In this section we collect up the various definitions relating to reading and writing memory for easy reference later. These are used to decompose state manipulation operations into operations on the state.

We first collect all the definitions for reading memory.

```
lemmas (in Ironbark_world) read_memory_decomp =
  read_program_memory_def
  read_call_memory_def
  read_static_memory_def
  read_dynamic_memory_def
  read_input_memory_def
```

We then collect all the definitions for writing memory.

```
lemmas write_memory_decomp =
  write_program_memory_def
  write_call_memory_def
  write_static_memory_def
  write_dynamic_memory_def
  write_output_memory_def
```

We then bundle the reading and writing versions together for easier reference when needed.

```
lemmas (in Ironbark_world) memory_decomp =
  read_memory_decomp
  write_memory_decomp
```

end

C.2.4 Decomposition Rules for standard_post_instruction

theory *post_instruction_decomp*

imports
state_manipulation_auxiliary

begin

In this section we show how the standard_post_instruction definition can be decomposed either into several simpler state manipulation operations, or into operations on the state.

```
lemma post_instruction_decomp_manipulation:
  shows
    ‹standard_post_instruction instruction_duration state =
      write_register
        instruction_pointer_ref
        ((read_register instruction_pointer_ref state) + 0x1)
        (write_register
          last_instruction_pointer_ref
          (read_register instruction_pointer_ref state)
            (write_register
              cycles_register_ref
              ((read_register cycles_register_ref state) + instruction_duration)
              state))›
  apply(simp add: standard_post_instruction_def)
  apply(simp add: Let_def)
  apply(simp add: read_register_def write_register_def)
  done

lemma post_instruction_decomp_state:
  shows
    ‹standard_post_instruction instruction_duration state =
      state
      (|
        register_state :=
```

```
          (register_state state)
          (
            instruction_pointer_ref
              ↦ ((read_register instruction_pointer_ref state) + 0x1),
            last_instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state),
            cycles_register_ref
              ↦ ((read_register cycles_register_ref state) + instruction_duration)
          )
      ⦆›
  apply(simp add: post_instruction_decomp_manipulation)
  apply(simp add: read_register_def write_register_def)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

end
```

C.2.5 Interface for State Manipulation Decomposition

theory *state_manipulation_decomposition*

imports
post_instruction_decomp
register_decomposition
flag_decomposition
memory_decomposition

begin

In this section we collect up all the decomposition rules for decomposing state manipulation operations into state level operations.

Note that we also include the register_set lemmas as a matter of convenience because we found that it is often needed in order to apply the decomposition rules. We also include slice_def and slice1_def, primarily to help the get_opcode, get_reg1, get_reg2, reg_reg3, and get_immediate operations decompose into a more understandable form.

lemmas (in *Ironbark_world*) *state_manipulation_decomp* =
register_set
register_decomp
flag_decomp
memory_decomp

check_read_permission_def
check_write_permission_def

halted_def
returned_def
called_def

```
  get_opcode_def
  get_immediate_def
  get_reg1_def
  get_reg2_def
  get_reg3_def
  slice_def
  slice1_def

  standard_error_def
  post_instruction_decomp_state
  backup_registers_before_call_def
  restore_registers_after_return_def
```

end

C.3 State Manipulation Simplification Rules

C.3.1 Simplification Rules Over Flags

theory *flag_simps*

imports
state_manipulation_decomposition

begin

This file contains simplification rules that apply get, set, or clear flag operations to a state.

We order the rules according to the secondary operator that is applied last in the normalised form.

At the state layer, we bundle all the simplification rules into the one lemma, which essentially shows that set/clear flag only affects the flag state (and therefore the register_state, program_memory, call_memory, static_memory, dynamic_memory, input_memory, and output_memory are all unaffected, regardless of which set or clear operation is performed.

```
lemma (in Ironbark_world) register_memory_state_set_clear_flag:
  assumes
    ‹f ∈ {set_end_jump,   set_end_call,   set_end_return,   set_halt,   set_error,
          clear_end_jump, clear_end_call, clear_end_return, clear_halt, clear_error}›
  shows
    ‹register_state (f state) = register_state  state›
    ‹program_memory (f state) = program_memory  state›
    ‹call_memory    (f state) = call_memory     state›
    ‹static_memory  (f state) = static_memory   state›
    ‹dynamic_memory (f state) = dynamic_memory  state›
    ‹input_memory   (f state) = input_memory    state›
    ‹output_memory  (f state) = output_memory   state›
  using assms
  by(auto simp add: state_manipulation_decomp)
```

At the state manipulation layer, we show the same non-interference properties for the respective

read functions.

```
lemma (in Ironbark_world) read_register_memory_set_clear_flag:
  assumes
    ‹f ∈ {set_end_jump,   set_end_call,   set_end_return,   set_halt,   set_error,
          clear_end_jump, clear_end_call, clear_end_return, clear_halt, clear_error}›
  shows
    ‹read_register        regID (f state) = read_register       regID state›
    ‹read_program_memory  addr1 (f state) = read_program_memory addr1 state›
    ‹read_call_memory     addr2 (f state) = read_call_memory    addr2 state›
    ‹read_static_memory   addr2 (f state) = read_static_memory  addr2 state›
    ‹read_dynamic_memory  addr2 (f state) = read_dynamic_memory addr2 state›
    ‹read_input_memory    addr2 (f state) = read_input_memory   addr2 state›
  using assms
  by(auto simp add: state_manipulation_decomp)
```

The following lemmas show that the set/clear flags only set their flag, and the other flags are not affected. We also include rules that show if you read the flag you write, you get what was written.

```
lemma (in Ironbark_world) get_end_jump_set_clear_other:
  assumes
    ‹f ∈ {set_end_call,   set_end_return,   set_halt,   set_error,
          clear_end_call, clear_end_return, clear_halt, clear_error}›
  shows
    ‹get_end_jump (f state) = get_end_jump state›
  using assms
  by(auto simp add: state_manipulation_decomp)

lemma (in Ironbark_world) get_end_jump_set_clear_same:
  ‹get_end_jump (set_end_jump   state) = 1›
  ‹get_end_jump (clear_end_jump state) = 0›
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) get_end_call_set_clear_other:
  assumes
    ‹f ∈ {set_end_jump,   set_end_return,   set_halt,   set_error,
          clear_end_jump, clear_end_return, clear_halt, clear_error}›
  shows
    ‹get_end_call (f state) = get_end_call state›
  using assms
  by(auto simp add: state_manipulation_decomp)

lemma (in Ironbark_world) get_end_call_set_clear_same:
  ‹get_end_call (set_end_call   state) = 1›
  ‹get_end_call (clear_end_call state) = 0›
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) get_end_return_set_clear_other:
  assumes
    ‹f ∈ {set_end_jump,   set_end_call,   set_halt,   set_error,
          clear_end_jump, clear_end_call, clear_halt, clear_error}›
```

```
  shows
    ‹get_end_return (f state) = get_end_return state›
  using assms
  by(auto simp add: state_manipulation_decomp)

lemma (in Ironbark_world) get_end_return_set_clear_same:
  ‹get_end_return (set_end_return   state) = 1›
  ‹get_end_return (clear_end_return state) = 0›
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) get_halt_set_clear_other:
  assumes
    ‹f ∈ {set_end_jump,   set_end_call,   set_end_return,   set_error,
          clear_end_jump, clear_end_call, clear_end_return, clear_error}›
  shows
    ‹get_halt (f state) = get_halt state›
  using assms
  by(auto simp add: state_manipulation_decomp)

lemma (in Ironbark_world) get_halt_set_clear_same:
  ‹get_halt (set_halt   state) = 1›
  ‹get_halt (clear_halt state) = 0›
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) get_error_set_clear_other:
  assumes
    ‹f ∈ {set_end_jump,   set_end_call,   set_end_return,   set_halt,
          clear_end_jump, clear_end_call, clear_end_return, clear_halt}›
  shows
    ‹get_error (f state) = get_error state›
  using assms
  by(auto simp add: state_manipulation_decomp)

lemma (in Ironbark_world) get_error_set_clear_same:
  ‹get_error (set_error   state) = 1›
  ‹get_error (clear_error state) = 0›
  by(simp_all add: state_manipulation_decomp)

end
```

C.3.2 Simplification Rules Over Register

```
theory register_simps

imports
  state_manipulation_decomposition

begin
```

This file contains simplification rules that apply register read and write operations to a state.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma flag_state_write_register:
  assumes
    ‹regID ∈ registers›
  shows
    ‹flag_state (write_register regID value state) = flag_state state›
  using assms by(simp add: write_register_def)

lemma memory_write_register:
  ‹program_memory (write_register regID value state) = program_memory state›
  ‹call_memory    (write_register regID value state) = call_memory    state›
  ‹static_memory  (write_register rgeID value state) = static_memory  state›
  ‹dynamic_memory (write_register regID value state) = dynamic_memory state›
  ‹input_memory   (write_register regID value state) = input_memory   state›
  ‹output_memory  (write_register regID value state) = output_memory  state›
  by(simp_all add: write_register_def)

lemma (in Ironbark_world) read_flag_write_register:
  assumes
    ‹regID ∈ registers›
  shows
    ‹get_end_jump   (write_register regID value state) = get_end_jump   state›
    ‹get_end_call   (write_register regID value state) = get_end_call   state›
    ‹get_end_return (write_register regID value state) = get_end_return state›
    ‹get_halt       (write_register regID value state) = get_halt       state›
    ‹get_error      (write_register regID value state) = get_error      state›
  using assms by (simp_all add: flag_state_write_register state_manipulation_decomp)

lemma (in Ironbark_world) common_flags_write_register:
  assumes
    ‹regID ∈ registers›
  shows
    ‹typical_flags    (write_register regID value state) = typical_flags    state›
    ‹end_call_flags   (write_register regID value state) = end_call_flags   state›
    ‹end_return_flags (write_register regID value state) = end_return_flags state›
    ‹end_jump_flags   (write_register regID value state) = end_jump_flags   state›
    ‹halt_flags       (write_register regID value state) = halt_flags       state›
    ‹error_flags      (write_register regID value state) = error_flags      state›
  using assms
  by(simp_all add: read_flag_write_register)

lemma same_registers_read_write:
  assumes
    ‹regID ∈ registers›
  shows
    ‹read_register regID (write_register regID value state) = value›
  using assms by(simp add: read_register_def write_register_def)
```

```
lemma diff_registers_read_write:
  assumes
    ‹regID1 ≠ regID2›
  shows
    ‹read_register  regID1 (write_register regID2 value state)
    = read_register regID1 state›
  using assms by(simp add: write_register_def read_register_def)

lemma (in Ironbark_world) read_memory_write_register:
  ‹read_program_memory  address (write_register regID value state)
  = read_program_memory address state›

  ‹read_call_memory     address (write_register regID value state)
  = read_call_memory    address state›

  ‹read_static_memory   address (write_register regID value state)
  = read_static_memory  address state›

  ‹read_dynamic_memory  address (write_register regID value state)
  = read_dynamic_memory address state›

  ‹read_input_memory    address (write_register regID value state)
  = read_input_memory   address state›
  by (simp_all add: memory_write_register
      read_program_memory_def read_call_memory_def
      read_static_memory_def read_dynamic_memory_def
      read_input_memory_def)

lemma same_registers_write_read:
  assumes
    ‹regID ∈ registers›
    ‹register_state state regID = Some value›
  shows
    ‹write_register regID (read_register regID state) state = state›
  using assms by(simp add: read_register_def write_register_def map_upd_triv)

lemma same_registers_write_write:
  shows
    ‹write_register   regID value1 (write_register regID value2 state)
    = write_register  regID value1 state›
  by(simp add: write_register_def)

lemma register_state_write_backup:
  shows
    ‹register_state   (write_register regID value1 (backup_registers_before_call state))
    = register_state  (write_register regID value1 state)›
  by(simp add: backup_registers_before_call_def write_register_def)

end
```

C.3.3 Simplification Rules Over Memory

theory *memory_simps*

imports
state_manipulation_decomposition

begin

This file contains simplification rules that apply state manipulation memory operations to a state.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

At the state layer, we show non-interference of writing memory on flags, registers and any memory address space that was not written to.

lemma (in *Ironbark_world) state_write_memory:*
— non-interference with flags
```
‹flag_state (write_program_memory address value1 state) = flag_state state›
‹flag_state (write_call_memory    address value2 state) = flag_state state›
‹flag_state (write_static_memory  address value2 state) = flag_state state›
‹flag_state (write_dynamic_memory address value2 state) = flag_state state›
‹flag_state (write_output_memory  address value2 state) = flag_state state›
```

— non-interference with registers
```
‹register_state (write_program_memory address value1 state) = register_state state›
‹register_state (write_call_memory    address value2 state) = register_state state›
‹register_state (write_static_memory  address value2 state) = register_state state›
‹register_state (write_dynamic_memory address value2 state) = register_state state›
‹register_state (write_output_memory  address value2 state) = register_state state›
```

— non-interference with other memory address spaces
```
‹program_memory (write_call_memory    address value2 state) = program_memory  state›
‹program_memory (write_static_memory  address value2 state) = program_memory  state›
‹program_memory (write_dynamic_memory address value2 state) = program_memory  state›
‹program_memory (write_output_memory  address value2 state) = program_memory  state›

‹call_memory    (write_program_memory address value1 state) = call_memory     state›
‹call_memory    (write_static_memory  address value2 state) = call_memory     state›
‹call_memory    (write_dynamic_memory address value2 state) = call_memory     state›
‹call_memory    (write_output_memory  address value2 state) = call_memory     state›

‹static_memory  (write_program_memory address value1 state) = static_memory   state›
‹static_memory  (write_call_memory    address value2 state) = static_memory   state›
‹static_memory  (write_dynamic_memory address value2 state) = static_memory   state›
‹static_memory  (write_output_memory  address value2 state) = static_memory   state›

‹dynamic_memory (write_program_memory address value1 state) = dynamic_memory  state›
‹dynamic_memory (write_call_memory    address value2 state) = dynamic_memory  state›
‹dynamic_memory (write_static_memory  address value2 state) = dynamic_memory  state›
‹dynamic_memory (write_output_memory  address value2 state) = dynamic_memory  state›
```

```
  ‹input_memory   (write_program_memory address value1 state) = input_memory    state›
  ‹input_memory   (write_call_memory    address value2 state) = input_memory    state›
  ‹input_memory   (write_static_memory  address value2 state) = input_memory    state›
  ‹input_memory   (write_dynamic_memory address value2 state) = input_memory    state›
  ‹input_memory   (write_output_memory  address value2 state) = input_memory    state›

  ‹output_memory  (write_program_memory address value1 state) = output_memory   state›
  ‹output_memory  (write_call_memory    address value2 state) = output_memory   state›
  ‹output_memory  (write_static_memory  address value2 state) = output_memory   state›
  ‹output_memory  (write_dynamic_memory address value2 state) = output_memory   state›
  by(simp_all add: state_manipulation_decomp)
```

The following lemma shows that if you write the value you read, then nothing changes.

```
lemma (in Ironbark_world) same_addr_write_read:
  ‹(program_memory state) address ≠ None
  ⟹ write_program_memory address (read_program_memory address state) state = state›

  ‹(call_memory state) address ≠ None
  ⟹ write_call_memory address (read_call_memory address state) state = state›

  ‹(static_memory state) address ≠ None
  ⟹ write_static_memory address (read_static_memory address state) state = state›

  ‹(dynamic_memory state) address ≠ None
  ⟹ write_dynamic_memory address (read_dynamic_memory address state) state = state›
  by(auto simp add: state_manipulation_decomp)
```

At the state manipulation layer, we show many of the same non-interference properties, but using the appropriate read functions.

The following lemma shows that individual flags are not affected by writing memory.

```
lemma (in Ironbark_world) read_flag_write_memory:
  assumes
    ‹f ∈ {
      get_end_jump,
      get_end_call,
      get_end_return,
      get_halt,
      get_error
    }›
  shows
    ‹f (write_program_memory  address value1 state) = f state›
    ‹f (write_call_memory     address value2 state) = f state›
    ‹f (write_static_memory   address value2 state) = f state›
    ‹f (write_dynamic_memory  address value2 state) = f state›
    ‹f (write_output_memory   address value2 state) = f state›
  using assms
  by(auto simp add: state_manipulation_decomp)
```

From this, we then show that our defined flag states will also be unaffected by writing memory.

```
lemma (in Ironbark_world) common_flags_write_memory:
  assumes
    ‹f ∈ {
      typical_flags,
      end_call_flags,
      end_return_flags,
      end_jump_flags,
      halt_flags,
      error_flags
    }›
  shows
    ‹f (write_program_memory  address value1 state) = f state›
    ‹f (write_call_memory     address value2 state) = f state›
    ‹f (write_static_memory   address value2 state) = f state›
    ‹f (write_dynamic_memory  address value2 state) = f state›
    ‹f (write_output_memory   address value2 state) = f state›
  using assms
  by(auto simp add: read_flag_write_memory)
```

The following lemma shows that reading a register is unaffected by writing memory.

```
lemma (in Ironbark_world) read_register_write_memory:
  ‹read_register  regID (write_program_memory  address value1 state)
  = read_register regID state›

  ‹read_register  regID (write_call_memory     address value2 state)
  = read_register regID state›

  ‹read_register  regID (write_static_memory   address value2 state)
  = read_register regID state›

  ‹read_register  regID (write_dynamic_memory  address value2 state)
  = read_register regID state›

  ‹read_register  regID (write_output_memory   address value2 state)
  = read_register regID state›
  by(simp_all add: state_manipulation_decomp)
```

Next, we show that reading from one memory region is unaffected by writes to another (different) memory region.

```
lemma (in Ironbark_world) diff_memory_read_write:
  fixes
    value2 :: ‹64 word›
    and value1 :: ‹96 word›
  shows
    ‹read_program_memory  address1 (write_call_memory     address2 value2 state)
    = read_program_memory address1 state›
    ‹read_program_memory  address1 (write_static_memory   address2 value2 state)
```

```
  = read_program_memory address1 state⟩
  ⟨read_program_memory  address1 (write_dynamic_memory  address2 value2 state)
  = read_program_memory address1 state⟩
  ⟨read_program_memory  address1 (write_output_memory   address2 value2 state)
  = read_program_memory address1 state⟩

  ⟨read_call_memory     address1 (write_program_memory  address2 value1 state)
  = read_call_memory    address1 state⟩
  ⟨read_call_memory     address1 (write_static_memory   address2 value2 state)
  = read_call_memory    address1 state⟩
  ⟨read_call_memory     address1 (write_dynamic_memory  address2 value2 state)
  = read_call_memory    address1 state⟩
  ⟨read_call_memory     address1 (write_output_memory   address2 value2 state)
  = read_call_memory    address1 state⟩

  ⟨read_static_memory   address1 (write_program_memory  address2 value1 state)
  = read_static_memory  address1 state⟩
  ⟨read_static_memory   address1 (write_call_memory     address2 value2 state)
  = read_static_memory  address1 state⟩
  ⟨read_static_memory   address1 (write_dynamic_memory  address2 value2 state)
  = read_static_memory  address1 state⟩
  ⟨read_static_memory   address1 (write_output_memory   address2 value2 state)
  = read_static_memory  address1 state⟩

  ⟨read_dynamic_memory  address1 (write_program_memory  address2 value1 state)
  = read_dynamic_memory address1 state⟩
  ⟨read_dynamic_memory  address1 (write_call_memory     address2 value2 state)
  = read_dynamic_memory address1 state⟩
  ⟨read_dynamic_memory  address1 (write_static_memory   address2 value2 state)
  = read_dynamic_memory address1 state⟩
  ⟨read_dynamic_memory  address1 (write_output_memory   address2 value2 state)
  = read_dynamic_memory address1 state⟩

  ⟨read_input_memory    address1 (write_program_memory  address2 value1 state)
  = read_input_memory   address1 state⟩
  ⟨read_input_memory    address1 (write_call_memory     address2 value2 state)
  = read_input_memory   address1 state⟩
  ⟨read_input_memory    address1 (write_static_memory   address2 value2 state)
  = read_input_memory   address1 state⟩
  ⟨read_input_memory    address1 (write_dynamic_memory  address2 value2 state)
  = read_input_memory   address1 state⟩
  ⟨read_input_memory    address1 (write_output_memory   address2 value2 state)
  = read_input_memory   address1 state⟩
by (simp_all add: state_manipulation_decomp)
```

Here we show that if you read from the address that you write a value to, you will get the value you wrote.

```
lemma (in Ironbark_world) same_addr_read_write:
  fixes
```

```
    value1 :: ‹96 word›
    and value2 :: ‹64 word›
  shows
    ‹read_program_memory  address (write_program_memory address value1 state) = value1›
    ‹read_call_memory     address (write_call_memory    address value2 state) = value2›
    ‹read_static_memory   address (write_static_memory  address value2 state) = value2›
    ‹read_dynamic_memory  address (write_dynamic_memory address value2 state) = value2›
  by (simp_all add: state_manipulation_decomp)
```

We then show that reading from one address is not affected by writing to another (different) address.

```
lemma (in Ironbark_world) diff_addr_read_write:
  fixes
    value1 :: ‹96 word›
    and value2 :: ‹64 word›
  assumes
    ‹address1 ≠ address2›
  shows
    ‹read_program_memory  address1 (write_program_memory  address2 value1 state)
    = read_program_memory address1 state›

    ‹read_call_memory     address1 (write_call_memory     address2 value2 state)
    = read_call_memory    address1 state›

    ‹read_static_memory   address1 (write_static_memory   address2 value2 state)
    = read_static_memory  address1 state›

    ‹read_dynamic_memory  address1 (write_dynamic_memory  address2 value2 state)
    = read_dynamic_memory address1 state›
  using assms by (simp_all add: state_manipulation_decomp)
```

Finally, we show that if you overwrite something, only the last value is retained.

```
lemma (in Ironbark_world) same_addr_write_write:
  ‹write_program_memory   address value1 (write_program_memory  address value2 state)
  = write_program_memory  address value1 state›

  ‹write_call_memory      address value3 (write_call_memory     address value4 state)
  = write_call_memory     address value3 state›

  ‹write_static_memory    address value3 (write_static_memory   address value4 state)
  = write_static_memory   address value3 state›

  ‹write_dynamic_memory   address value3 (write_dynamic_memory  address value4 state)
  = write_dynamic_memory  address value3 state›

  ‹write_output_memory    address value3 (write_output_memory   address value4 state)
  = write_output_memory   address value3 state›
  by (simp_all add: state_manipulation_decomp)
```

end

C.3.4 Simplification Rules Over the Initial State

theory *initial_state_simps*

imports
state_manipulation_decomposition

begin

This file contains simplification rules where the state is the initial state.

We order the rules according to the operator that is applied last in the normalised form.

At the state layer, we bundle all the simplification rules into the one lemma, which shows that memory is empty, except for input memory, which is set to a function with no definition.

lemma (**in** *Ironbark_world*) *memory_initial_state:*
```
  ‹program_memory initial_state = Map.empty›
  ‹call_memory    initial_state = Map.empty›
  ‹static_memory  initial_state = Map.empty›
  ‹dynamic_memory initial_state = Map.empty›
  ‹input_memory   initial_state = input_memory_stream›
  ‹output_memory  initial_state = Map.empty›
  by (simp_all add: initial_state_def)
```

At the state manipulation layer we show the same properties, but using the equivalent read functions. These functions also make it easier to express the state of registers and flags, so we also include those. We also found it is sometimes useful to include lemmas which include an arbitrary value in program memory.

lemma (**in** *Ironbark_world*) *flags_initial_state:*
```
  — flags on the initial state
  ‹get_end_jump   initial_state = 0›
  ‹get_end_call   initial_state = 0›
  ‹get_end_return initial_state = 0›
  ‹get_halt       initial_state = 0›
  ‹get_error      initial_state = 0›
  — with arbitrary program
  ‹get_end_jump   (initial_state(|program_memory := program|)) = 0›
  ‹get_end_call   (initial_state(|program_memory := program|)) = 0›
  ‹get_end_return (initial_state(|program_memory := program|)) = 0›
  ‹get_halt       (initial_state(|program_memory := program|)) = 0›
  ‹get_error      (initial_state(|program_memory := program|)) = 0›
  unfolding initial_state_def
  by(auto simp add: state_manipulation_decomp)
```

lemma (**in** *Ironbark_world*) *typical_flags_initial_state:*
```
  ‹typical_flags initial_state›
  ‹typical_flags (initial_state(|program_memory := program|))›
  by (simp_all add: initial_state_def state_manipulation_decomp)
```

lemma (**in** *Ironbark_world*) *read_register_initial_state:*

```
  ‹rID ∈ registers
  ⟹ read_register rID initial_state = 0›

  ‹rID ∈ registers
  ⟹ read_register rID (initial_state(|program_memory := program|)) = 0›
  unfolding registers_def
  unfolding
    generic_registers_def
    special_purpose_registers_def
    special_address_registers_def
  by(auto simp add: initial_state_def read_register_def)
```

end

C.3.5 Simplification Rules Over standard_error

theory *standard_error_simps*

imports
state_manipulation_decomposition

begin

This file contains simplification rules that apply standard_error to a state.
We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) memory_register_state_standard_error:
  ‹program_memory (standard_error state) = program_memory state›
  ‹call_memory    (standard_error state) = call_memory    state›
  ‹static_memory  (standard_error state) = static_memory  state›
  ‹dynamic_memory (standard_error state) = dynamic_memory state›
  ‹input_memory   (standard_error state) = input_memory   state›
  ‹output_memory  (standard_error state) = output_memory  state›
  ‹register_state (standard_error state) = register_state state›
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) read_flag_standard_error:
  ‹get_error      (standard_error state) = 1›
  ‹get_halt       (standard_error state) = 1›
  ‹get_end_jump   (standard_error state) = get_end_jump   state›
  ‹get_end_call   (standard_error state) = get_end_call   state›
  ‹get_end_return (standard_error state) = get_end_return state›
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) read_memory_register_state_standard_error:
  ‹read_program_memory  address (standard_error state)
  = read_program_memory address state›

  ‹read_call_memory     address (standard_error state)
```

```
  = read_call_memory    address state›

  ‹read_static_memory   address (standard_error state)
  = read_static_memory  address state›

  ‹read_dynamic_memory  address (standard_error state)
  = read_dynamic_memory address state›

  ‹read_register        regID   (standard_error state)
  = read_register       regID   state›
  by(simp_all add: state_manipulation_decomp)
```

end

C.3.6 Simplification Rules Over standard_post_instruction

theory *post_instruction_simps*

imports
state_manipulation_decomposition

begin

This file contains simplification rules that apply standard_post_instruction to a state.
We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) read_register_standard_post_instruction:
  assumes
    ‹regID ∉ {
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      cycles_register_ref
    }›
  shows
    ‹read_register  regID (standard_post_instruction instruction_duration state)
    = read_register regID state›
  using assms
  by(simp add: state_manipulation_decomp)

lemma (in Ironbark_world) read_last_instruction_pointer_standard_post_instruction:
  ‹read_register
    last_instruction_pointer_ref
    (standard_post_instruction instruction_duration state)
  = read_register instruction_pointer_ref state›
  by(simp add: state_manipulation_decomp)

lemma (in Ironbark_world) read_instruction_pointer_standard_post_instruction:
  ‹read_register
    instruction_pointer_ref
```

```
    (standard_post_instruction instruction_duration state)
  = read_register instruction_pointer_ref state + 1›
  by(simp add: state_manipulation_decomp)

lemma (in Ironbark_world) read_cycles_register_standard_post_instruction:
  ‹read_register
    cycles_register_ref
    (standard_post_instruction instruction_duration state)
  = read_register cycles_register_ref state + instruction_duration›
  by(simp add: state_manipulation_decomp)

lemma (in Ironbark_world) memory_standard_post_instruction:
  ‹program_memory   (standard_post_instruction instruction_duration state)
  = program_memory  state›

  ‹call_memory      (standard_post_instruction instruction_duration state)
  = call_memory     state›

  ‹static_memory    (standard_post_instruction instruction_duration state)
  = static_memory   state›

  ‹dynamic_memory   (standard_post_instruction instruction_duration state)
  = dynamic_memory  state›

  ‹input_memory     (standard_post_instruction instruction_duration state)
  = input_memory    state›

  ‹output_memory    (standard_post_instruction instruction_duration state)
  = output_memory   state›

  ‹flag_state       (standard_post_instruction instruction_duration state)
  = flag_state      state›
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) read_memory_standard_post_instruction:
  ‹read_program_memory  address (standard_post_instruction instruction_duration state)
  = read_program_memory address state›

  ‹read_call_memory     address (standard_post_instruction instruction_duration state)
  = read_call_memory    address state›

  ‹read_static_memory   address (standard_post_instruction instruction_duration state)
  = read_static_memory  address state›

  ‹read_dynamic_memory  address (standard_post_instruction instruction_duration state)
  = read_dynamic_memory address state›

  ‹get_end_jump     (standard_post_instruction instruction_duration state)
  = get_end_jump    state›
```

```
  ‹get_end_call     (standard_post_instruction instruction_duration state)
  = get_end_call    state›

  ‹get_end_return   (standard_post_instruction instruction_duration state)
  = get_end_return  state›

  ‹get_halt         (standard_post_instruction instruction_duration state)
  = get_halt        state›

  ‹get_error        (standard_post_instruction instruction_duration state)
  = get_error       state›
  by(simp_all add: state_manipulation_decomp memory_standard_post_instruction)

lemma (in Ironbark_world) check_status_post_instruction:
  ‹typical_flags  (standard_post_instruction instruction_duration state)
  = typical_flags state›
  by(auto simp add: read_memory_standard_post_instruction)

lemma (in Ironbark_world) register_state_passover:
  assumes
    ‹register_state state1 = register_state state2›
  shows
    ‹register_state   (standard_post_instruction instruction_duration state1)
    = register_state  (standard_post_instruction instruction_duration state2)›
  using assms
  by(simp add: state_manipulation_decomp)

end
```

C.3.7 Simplification Rules Over Check Permission Operations

```
theory permission_sets

imports
  state_manipulation_decomposition

begin
```

This file contains lemmas relating to the check_read_permission and check_write_permission operations and their underlying readable_registers and writeable_registers definitions.

Specifically, we show the sets that a regID can and can't belong to based on if it passes the check_write_permission and check_read_permission check.

```
lemma check_permission_sets:
  shows
    ‹check_write_permission regID ⟹ regID ∈ registers›
    ‹check_write_permission regID ⟹ regID ∈ writeable_registers›
    ‹check_write_permission regID ⟹ regID ∉ special_purpose_registers›
```

```
    ‹check_write_permission regID ⟹ regID ∉ {
      cycles_register_ref,
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      call_frame_pointer_ref
    }›
    ‹check_read_permission  regID ⟹ regID ∈ registers›
    ‹check_read_permission  regID ⟹ regID ∈ readable_registers›
    ‹check_read_permission  regID ⟹ regID ∉ {
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      call_frame_pointer_ref
    }›
proof -
  have ‹∀ x . (x ∈ special_purpose_registers ⟶ x ∉ writeable_registers)›
    unfolding special_purpose_registers_def writeable_registers_def
    unfolding generic_registers_def special_address_registers_def
    by (simp)
  then show
    ‹check_write_permission regID ⟹ regID ∉ special_purpose_registers›
    ‹check_write_permission regID ⟹ regID ∉ {
      cycles_register_ref,
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      call_frame_pointer_ref
    }›
    unfolding check_write_permission_def
    unfolding special_purpose_registers_def
    by (auto)
  have
    ‹instruction_pointer_ref      ∉ readable_registers›
    ‹last_instruction_pointer_ref ∉ readable_registers›
    ‹call_frame_pointer_ref       ∉ readable_registers›
    unfolding special_purpose_registers_def readable_registers_def
    unfolding generic_registers_def special_address_registers_def
    by (simp_all)
  then show
    ‹check_read_permission regID ⟹ regID ∉ {
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      call_frame_pointer_ref
    }›
    unfolding check_read_permission_def
    by (auto)
  show
    ‹check_read_permission  regID ⟹ regID ∈ registers›
    ‹check_read_permission  regID ⟹ regID ∈ readable_registers›
    ‹check_write_permission regID ⟹ regID ∈ registers›
    ‹check_write_permission regID ⟹ regID ∈ writeable_registers›
```

```
    using register_subsets
    by(simp_all add: check_write_permission_def check_read_permission_def)
qed

lemma check_write_permission_exclusion:
  assumes
    ‹check_write_permission regID›
  shows
    ‹regID ≠ cycles_register_ref›
    ‹regID ≠ instruction_pointer_ref›
    ‹regID ≠ last_instruction_pointer_ref›
    ‹regID ≠ call_frame_pointer_ref›
  using assms check_permission_sets
  by(simp_all)

lemma check_read_permission_exclusion:
  assumes
    ‹check_read_permission regID›
  shows
    ‹regID ≠ instruction_pointer_ref›
    ‹regID ≠ last_instruction_pointer_ref›
    ‹regID ≠ call_frame_pointer_ref›
  using assms check_permission_sets
  by(simp_all)

lemma check_read_permission_set:
  assumes
    ‹regID ∈ {
      r00_ref,   r01_ref,   r02_ref,   r03_ref,
      r04_ref,   r05_ref,   r06_ref,   r07_ref,
      r08_ref,   r09_ref,   r10_ref,   r11_ref,
      r12_ref,   r13_ref,   r14_ref,   r15_ref,

      p00_ref,   p01_ref,   p02_ref,   p03_ref,
      p04_ref,   p05_ref,   p06_ref,   p07_ref,
      p08_ref,   p09_ref,   p10_ref,   p11_ref,
      p12_ref,   p13_ref,   p14_ref,   p15_ref,

      c00_ref,   c01_ref,   c02_ref,   c03_ref,
      c04_ref,   c05_ref,   c06_ref,   c07_ref,
      c08_ref,   c09_ref,   c10_ref,   c11_ref,
      c12_ref,   c13_ref,   c14_ref,   c15_ref,

      arg00_ref, arg01_ref, arg02_ref, arg03_ref,
      arg04_ref, arg05_ref, arg06_ref, arg07_ref,
      arg08_ref, arg09_ref, arg10_ref, arg11_ref,
      arg12_ref, arg13_ref, arg14_ref, arg15_ref,

      ret00_ref, ret01_ref, ret02_ref, ret03_ref,
```

```
    ret04_ref, ret05_ref, ret06_ref, ret07_ref,
    ret08_ref, ret09_ref, ret10_ref, ret11_ref,
    ret12_ref, ret13_ref, ret14_ref, ret15_ref,

    arg_frame_pointer_ref,
    arg_stack_pointer_ref,
    dynamic_data_frame_pointer_ref,
    dynamic_data_stack_pointer_ref,
    static_data_frame_pointer_ref,
    static_data_stack_pointer_ref,

    cycles_register_ref
  }›
 shows
  ‹check_read_permission regID›
 using assms
 unfolding check_read_permission_def
 unfolding readable_registers_def
 unfolding generic_registers_def special_address_registers_def
 by(simp add: insert_commute)

lemma check_write_permission_set:
 assumes
  ‹regID ∈ {
    r00_ref,   r01_ref,   r02_ref,   r03_ref,
    r04_ref,   r05_ref,   r06_ref,   r07_ref,
    r08_ref,   r09_ref,   r10_ref,   r11_ref,
    r12_ref,   r13_ref,   r14_ref,   r15_ref,

    p00_ref,   p01_ref,   p02_ref,   p03_ref,
    p04_ref,   p05_ref,   p06_ref,   p07_ref,
    p08_ref,   p09_ref,   p10_ref,   p11_ref,
    p12_ref,   p13_ref,   p14_ref,   p15_ref,

    c00_ref,   c01_ref,   c02_ref,   c03_ref,
    c04_ref,   c05_ref,   c06_ref,   c07_ref,
    c08_ref,   c09_ref,   c10_ref,   c11_ref,
    c12_ref,   c13_ref,   c14_ref,   c15_ref,

    arg00_ref, arg01_ref, arg02_ref, arg03_ref,
    arg04_ref, arg05_ref, arg06_ref, arg07_ref,
    arg08_ref, arg09_ref, arg10_ref, arg11_ref,
    arg12_ref, arg13_ref, arg14_ref, arg15_ref,

    ret00_ref, ret01_ref, ret02_ref, ret03_ref,
    ret04_ref, ret05_ref, ret06_ref, ret07_ref,
    ret08_ref, ret09_ref, ret10_ref, ret11_ref,
    ret12_ref, ret13_ref, ret14_ref, ret15_ref,
```

```
    arg_frame_pointer_ref,
    arg_stack_pointer_ref,
    dynamic_data_frame_pointer_ref,
    dynamic_data_stack_pointer_ref,
    static_data_frame_pointer_ref,
    static_data_stack_pointer_ref
  }›
shows
  ‹check_write_permission regID›
using assms
unfolding check_write_permission_def
unfolding writeable_registers_def
unfolding generic_registers_def special_address_registers_def
by(simp add: insert_commute)

end
```

C.3.8 Simplification Rules Over Other State Manipulation Operations

theory *special_simps*

imports
state_manipulation_decomposition

begin

This file contains simplification rules that apply backup_registers_before_call or restore_registers_after_return to a state.
We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma flags_backup_registers_before_call:
  ‹flag_state (backup_registers_before_call state) = flag_state state›
  by (simp add: backup_registers_before_call_def)

lemma registers_backup_registers_before_call:
  ‹register_state (backup_registers_before_call state) = register_state state›
  by (simp add: backup_registers_before_call_def)

lemma memory_backup_registers_before_call:
  ‹program_memory (backup_registers_before_call state) = program_memory state›
  ‹static_memory  (backup_registers_before_call state) = static_memory  state›
  ‹dynamic_memory (backup_registers_before_call state) = dynamic_memory state›
  ‹input_memory   (backup_registers_before_call state) = input_memory   state›
  ‹output_memory  (backup_registers_before_call state) = output_memory  state›
  by (simp_all add: backup_registers_before_call_def)

lemma flags_restore_registers_after_return:
  ‹flag_state (restore_registers_after_return state) = flag_state state›
  by (simp add: restore_registers_after_return_def)
```

```
lemma memory_restore_registers_after_return:
  ‹program_memory (restore_registers_after_return state) = program_memory state›
  ‹call_memory    (restore_registers_after_return state) = call_memory    state›
  ‹static_memory  (restore_registers_after_return state) = static_memory  state›
  ‹dynamic_memory (restore_registers_after_return state) = dynamic_memory state›
  ‹input_memory   (restore_registers_after_return state) = input_memory   state›
  ‹output_memory  (restore_registers_after_return state) = output_memory  state›
  by (simp_all add: restore_registers_after_return_def)

lemma read_flags_backup_registers_before_call:
  ‹get_end_jump   (backup_registers_before_call state) = get_end_jump   state›
  ‹get_end_call   (backup_registers_before_call state) = get_end_call   state›
  ‹get_end_return (backup_registers_before_call state) = get_end_return state›
  ‹get_halt       (backup_registers_before_call state) = get_halt       state›
  ‹get_error      (backup_registers_before_call state) = get_error      state›
  by (simp_all add: flags_backup_registers_before_call
      get_end_jump_def get_end_call_def get_end_return_def get_halt_def get_error_def)

lemma read_registers_backup_registers_before_call:
  ‹read_register  regID (backup_registers_before_call state)
  = read_register regID state›
  by (simp add: registers_backup_registers_before_call read_register_def)

lemma (in Ironbark_world) read_memory_backup_registers_before_call:
  ‹read_program_memory  address (backup_registers_before_call state)
  = read_program_memory address state›

  ‹read_static_memory   address (backup_registers_before_call state)
  = read_static_memory  address state›

  ‹read_dynamic_memory  address (backup_registers_before_call state)
  = read_dynamic_memory address state›

  ‹read_input_memory    address (backup_registers_before_call state)
  = read_input_memory   address state›
  by (simp_all add: Let_def
        backup_registers_before_call_def
        read_program_memory_def read_call_memory_def read_static_memory_def
        read_dynamic_memory_def read_input_memory_def)

lemma read_call_memory_backup_registers_before_call:
  assumes
    ‹address < (read_register call_frame_pointer_ref state)›
    ‹(read_register call_frame_pointer_ref state) < (2^64 - 66)›
  shows
    ‹read_call_memory   address (backup_registers_before_call state)
    = read_call_memory  address state›
proof -
```

```
  have
    ‹address ≠ read_register call_frame_pointer_ref state›
    using assms
    by(auto)
  moreover have
    ‹∀n ∈ {0 .. 66} . address ≠ read_register call_frame_pointer_ref state + n›
    using assms
    by(uint_arith, auto)
  ultimately show ?thesis
    unfolding backup_registers_before_call_def
    by(simp add: read_call_memory_def)
qed

lemma read_call_memory_backup_registers_before_call':
  assumes
    ‹∀n ∈ {0 .. 66} . address ≠ read_register call_frame_pointer_ref state + n›
  shows
    ‹read_call_memory   address (backup_registers_before_call state)
    = read_call_memory  address state›
  using assms
  by(auto simp add: backup_registers_before_call_def read_call_memory_def)

lemma read_flag_restore_registers_after_return:
  ‹get_end_jump   (restore_registers_after_return state) = get_end_jump   state›
  ‹get_end_call   (restore_registers_after_return state) = get_end_call   state›
  ‹get_end_return (restore_registers_after_return state) = get_end_return state›
  ‹get_halt       (restore_registers_after_return state) = get_halt       state›
  ‹get_error      (restore_registers_after_return state) = get_error      state›
  by (simp_all add: flags_restore_registers_after_return
      get_end_jump_def get_end_call_def get_end_return_def get_halt_def get_error_def)

lemma read_register_after_return_same:
  assumes
    ‹regID ∉ {
      r00_ref,   r01_ref,   r02_ref,   r03_ref,
      r04_ref,  r05_ref,    r06_ref,   r07_ref,
      r08_ref,   r09_ref,   r10_ref,   r11_ref,
      r12_ref,  r13_ref,    r14_ref,   r15_ref,
      p00_ref,   p01_ref,   p02_ref,   p03_ref,
      p04_ref,  p05_ref,    p06_ref,   p07_ref,
      p08_ref,   p09_ref,   p10_ref,   p11_ref,
      p12_ref,  p13_ref,    p14_ref,   p15_ref,
      c00_ref,   c01_ref,   c02_ref,   c03_ref,
      c04_ref,   c05_ref,   c06_ref,   c07_ref,
      c08_ref,   c09_ref,   c10_ref,   c11_ref,
      c12_ref,   c13_ref,   c14_ref,   c15_ref,
      arg00_ref, arg01_ref, arg02_ref, arg03_ref,
      arg04_ref, arg05_ref, arg06_ref, arg07_ref,
      arg08_ref, arg09_ref, arg10_ref, arg11_ref,
```

```
    arg12_ref, arg13_ref, arg14_ref, arg15_ref,
    ret00_ref, ret01_ref, ret02_ref, ret03_ref,
    ret04_ref, ret05_ref, ret06_ref, ret07_ref,
    ret08_ref, ret09_ref, ret10_ref, ret11_ref,
    ret12_ref, ret13_ref, ret14_ref, ret15_ref,
    static_data_frame_pointer_ref,
    static_data_stack_pointer_ref,
    instruction_pointer_ref
  }›
 shows
  ‹read_register    regID (restore_registers_after_return state)
    = read_register regID state›
 using assms
 by(simp_all add: restore_registers_after_return_def read_register_def)

lemma read_register_restore_registers_after_return_diff:
 ‹read_register r00_ref  (restore_registers_after_return state)
 = read_call_memory      ((read_register call_frame_pointer_ref state) -  1) state›

 ‹read_register r01_ref  (restore_registers_after_return state)
 = read_call_memory      ((read_register call_frame_pointer_ref state) -  2) state›

 ‹read_register r02_ref  (restore_registers_after_return state)
 = read_call_memory      ((read_register call_frame_pointer_ref state) -  3) state›

 ‹read_register r03_ref  (restore_registers_after_return state)
 = read_call_memory      ((read_register call_frame_pointer_ref state) -  4) state›

 ‹read_register r04_ref  (restore_registers_after_return state)
 = read_call_memory      ((read_register call_frame_pointer_ref state) -  5) state›

 ‹read_register r05_ref  (restore_registers_after_return state)
 = read_call_memory      ((read_register call_frame_pointer_ref state) -  6) state›

 ‹read_register r06_ref  (restore_registers_after_return state)
 = read_call_memory      ((read_register call_frame_pointer_ref state) -  7) state›

 ‹read_register r07_ref  (restore_registers_after_return state)
 = read_call_memory      ((read_register call_frame_pointer_ref state) -  8) state›

 ‹read_register r08_ref  (restore_registers_after_return state)
 = read_call_memory      ((read_register call_frame_pointer_ref state) -  9) state›

 ‹read_register r09_ref  (restore_registers_after_return state)
 = read_call_memory      ((read_register call_frame_pointer_ref state) - 10) state›

 ‹read_register r10_ref  (restore_registers_after_return state)
 = read_call_memory      ((read_register call_frame_pointer_ref state) - 11) state›
```

```
<read_register r11_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 12) state>

<read_register r12_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 13) state>

<read_register r13_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 14) state>

<read_register r14_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 15) state>

<read_register r15_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 16) state>

<read_register p00_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 17) state>

<read_register p01_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 18) state>

<read_register p02_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 19) state>

<read_register p03_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 20) state>

<read_register p04_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 21) state>

<read_register p05_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 22) state>

<read_register p06_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 23) state>

<read_register p07_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 24) state>

<read_register p08_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 25) state>

<read_register p09_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 26) state>

<read_register p10_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 27) state>

<read_register p11_ref  (restore_registers_after_return state)
```

```
= read_call_memory      ((read_register call_frame_pointer_ref state) - 28) state>

<read_register p12_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 29) state>

<read_register p13_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 30) state>

<read_register p14_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 31) state>

<read_register p15_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 32) state>

<read_register c00_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 33) state>

<read_register c01_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 34) state>

<read_register c02_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 35) state>

<read_register c03_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 36) state>

<read_register c04_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 37) state>

<read_register c05_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 38) state>

<read_register c06_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 39) state>

<read_register c07_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 40) state>

<read_register c08_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 41) state>

<read_register c09_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 42) state>

<read_register c10_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 43) state>

<read_register c11_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 44) state>
```

```
<read_register c12_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 45) state>

<read_register c13_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 46) state>

<read_register c14_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 47) state>

<read_register c15_ref  (restore_registers_after_return state)
= read_call_memory      ((read_register call_frame_pointer_ref state) - 48) state>

<read_register arg00_ref  (restore_registers_after_return state)
= read_call_memory        ((read_register call_frame_pointer_ref state) - 49) state>

<read_register arg01_ref  (restore_registers_after_return state)
= read_call_memory        ((read_register call_frame_pointer_ref state) - 50) state>

<read_register arg02_ref  (restore_registers_after_return state)
= read_call_memory        ((read_register call_frame_pointer_ref state) - 51) state>

<read_register arg03_ref  (restore_registers_after_return state)
= read_call_memory        ((read_register call_frame_pointer_ref state) - 52) state>

<read_register arg04_ref  (restore_registers_after_return state)
= read_call_memory        ((read_register call_frame_pointer_ref state) - 53) state>

<read_register arg05_ref  (restore_registers_after_return state)
= read_call_memory        ((read_register call_frame_pointer_ref state) - 54) state>

<read_register arg06_ref  (restore_registers_after_return state)
= read_call_memory        ((read_register call_frame_pointer_ref state) - 55) state>

<read_register arg07_ref  (restore_registers_after_return state)
= read_call_memory        ((read_register call_frame_pointer_ref state) - 56) state>

<read_register arg08_ref  (restore_registers_after_return state)
= read_call_memory        ((read_register call_frame_pointer_ref state) - 57) state>

<read_register arg09_ref  (restore_registers_after_return state)
= read_call_memory        ((read_register call_frame_pointer_ref state) - 58) state>

<read_register arg10_ref  (restore_registers_after_return state)
= read_call_memory        ((read_register call_frame_pointer_ref state) - 59) state>

<read_register arg11_ref  (restore_registers_after_return state)
= read_call_memory        ((read_register call_frame_pointer_ref state) - 60) state>
```

```
  ‹read_register arg12_ref  (restore_registers_after_return state)
  = read_call_memory        ((read_register call_frame_pointer_ref state) - 61) state›

  ‹read_register arg13_ref  (restore_registers_after_return state)
  = read_call_memory        ((read_register call_frame_pointer_ref state) - 62) state›

  ‹read_register arg14_ref  (restore_registers_after_return state)
  = read_call_memory        ((read_register call_frame_pointer_ref state) - 63) state›

  ‹read_register arg15_ref  (restore_registers_after_return state)
  = read_call_memory        ((read_register call_frame_pointer_ref state) - 64) state›

  ‹read_register static_data_frame_pointer_ref (restore_registers_after_return state)
  = read_call_memory ((read_register call_frame_pointer_ref state) - 65) state›

  ‹read_register static_data_stack_pointer_ref (restore_registers_after_return state)
  = read_call_memory ((read_register call_frame_pointer_ref state) - 66) state›

  ‹read_register instruction_pointer_ref       (restore_registers_after_return state)
  = read_call_memory ((read_register call_frame_pointer_ref state) - 67) state›
  by(simp_all add: restore_registers_after_return_def read_register_def)

lemma (in Ironbark_world) read_memory_restore_registers_after_return:
  ‹read_program_memory    address (restore_registers_after_return state)
  = read_program_memory address state›

  ‹read_call_memory     address (restore_registers_after_return state)
  = read_call_memory    address state›

  ‹read_static_memory   address (restore_registers_after_return state)
  = read_static_memory  address state›

  ‹read_dynamic_memory  address (restore_registers_after_return state)
  = read_dynamic_memory address state›

  ‹read_input_memory    address (restore_registers_after_return state)
  = read_input_memory   address state›
  by (simp_all add: restore_registers_after_return_def read_memory_decomp)

end
```

C.3.9 Interface for State Manipulation Simplification

```
theory state_manipulation_simps

imports
  flag_simps
  register_simps
  memory_simps
```

```
  special_simps
  initial_state_simps
  standard_error_simps
  post_instruction_simps
  permission_sets
```

begin

This file provides a single point for importing all the simplification rules at the state manipulation layer.

We also collect groups of related lemmas and bundle them for easier reference later.

```
lemmas (in Ironbark_world) flag_simps =
  register_memory_state_set_clear_flag
  read_register_memory_set_clear_flag
  get_end_jump_set_clear_other
  get_end_jump_set_clear_same
  get_end_call_set_clear_other
  get_end_call_set_clear_same
  get_end_return_set_clear_other
  get_end_return_set_clear_same
  get_halt_set_clear_other
  get_halt_set_clear_same
  get_error_set_clear_other
  get_error_set_clear_same

lemmas (in Ironbark_world) register_simps =
  flag_state_write_register
  memory_write_register
  read_flag_write_register
  common_flags_write_register
  same_registers_read_write
  diff_registers_read_write
  read_memory_write_register
  same_registers_write_read
  same_registers_write_write
  register_state_write_backup

lemmas (in Ironbark_world) memory_simps =
  state_write_memory
  read_flag_write_memory
  common_flags_write_memory
  read_register_write_memory
  diff_memory_read_write
  same_addr_read_write
  diff_addr_read_write
  same_addr_write_read
  same_addr_write_write

lemmas (in Ironbark_world) backup_simps =
```

```
  flags_backup_registers_before_call
  registers_backup_registers_before_call
  memory_backup_registers_before_call
  read_flags_backup_registers_before_call
  read_registers_backup_registers_before_call
  read_memory_backup_registers_before_call
  read_call_memory_backup_registers_before_call
  read_call_memory_backup_registers_before_call'

lemmas (in Ironbark_world) restore_simps =
  flags_restore_registers_after_return
  memory_restore_registers_after_return
  read_flag_restore_registers_after_return
  read_register_after_return_same
  read_register_restore_registers_after_return_diff
  read_memory_restore_registers_after_return

lemmas (in Ironbark_world) initial_state_simps =
  memory_initial_state
  read_register_initial_state
  flags_initial_state

lemmas (in Ironbark_world) standard_error_simps =
  memory_register_state_standard_error
  read_flag_standard_error
  read_memory_register_state_standard_error

lemmas (in Ironbark_world) post_instruction_simps =
  read_register_standard_post_instruction
  read_last_instruction_pointer_standard_post_instruction
  read_instruction_pointer_standard_post_instruction
  read_cycles_register_standard_post_instruction
  memory_standard_post_instruction
  read_memory_standard_post_instruction
  check_status_post_instruction

lemmas check_permission_exclusion =
  check_write_permission_exclusion
  check_read_permission_exclusion

lemmas permission_simps =
  check_permission_sets
  check_write_permission_exclusion
  check_read_permission_exclusion
  check_read_permission_set
  check_write_permission_set

lemmas (in Ironbark_world) state_manipulation_simps =
  register_set
```

flag_simps
register_simps
memory_simps
backup_simps
restore_simps
initial_state_simps
standard_error_simps
post_instruction_simps
permission_simps

end

C.4 State Manipulation Reordering Rules

C.4.1 Reordering Rules for Flags

theory *flag_reorder*

imports
state_manipulation_simps

begin

This file contains reordering rules that apply set or get flag operations to a state.
We order the rules based on the second operator, according to which would be applied last in the normalised form.

lemma (in *Ironbark_world*) *flag_flag_reorder:*
shows
```
‹set_end_call   (set_end_return   state) = set_end_return   (set_end_call   state)›
‹set_end_jump   (set_end_return   state) = set_end_return   (set_end_jump   state)›
‹set_halt       (set_end_return   state) = set_end_return   (set_halt       state)›
‹set_error      (set_end_return   state) = set_end_return   (set_error      state)›
‹clear_end_call (set_end_return   state) = set_end_return   (clear_end_call state)›
‹clear_end_jump (set_end_return   state) = set_end_return   (clear_end_jump state)›
‹clear_halt     (set_end_return   state) = set_end_return   (clear_halt     state)›
‹clear_error    (set_end_return   state) = set_end_return   (clear_error    state)›
‹set_end_call   (clear_end_return state) = clear_end_return (set_end_call   state)›
‹set_end_jump   (clear_end_return state) = clear_end_return (set_end_jump   state)›
‹set_halt       (clear_end_return state) = clear_end_return (set_halt       state)›
‹set_error      (clear_end_return state) = clear_end_return (set_error      state)›
‹clear_end_call (clear_end_return state) = clear_end_return (clear_end_call state)›
‹clear_end_jump (clear_end_return state) = clear_end_return (clear_end_jump state)›
‹clear_halt     (clear_end_return state) = clear_end_return (clear_halt     state)›
‹clear_error    (clear_end_return state) = clear_end_return (clear_error    state)›

‹set_end_jump   (set_end_call     state) = set_end_call   (set_end_jump     state)›
‹set_halt       (set_end_call     state) = set_end_call   (set_halt         state)›
‹set_error      (set_end_call     state) = set_end_call   (set_error        state)›
‹clear_end_jump (set_end_call     state) = set_end_call   (clear_end_jump   state)›
```

```
  ‹clear_halt     (set_end_call     state) = set_end_call   (clear_halt       state)›
  ‹clear_error    (set_end_call     state) = set_end_call   (clear_error      state)›
  ‹set_end_jump   (clear_end_call   state) = clear_end_call (set_end_jump     state)›
  ‹set_halt       (clear_end_call   state) = clear_end_call (set_halt         state)›
  ‹set_error      (clear_end_call   state) = clear_end_call (set_error        state)›
  ‹clear_end_jump (clear_end_call   state) = clear_end_call (clear_end_jump   state)›
  ‹clear_halt     (clear_end_call   state) = clear_end_call (clear_halt       state)›
  ‹clear_error    (clear_end_call   state) = clear_end_call (clear_error      state)›

  ‹set_halt       (set_end_jump     state) = set_end_jump   (set_halt         state)›
  ‹set_error      (set_end_jump     state) = set_end_jump   (set_error        state)›
  ‹clear_halt     (set_end_jump     state) = set_end_jump   (clear_halt       state)›
  ‹clear_error    (set_end_jump     state) = set_end_jump   (clear_error      state)›
  ‹set_halt       (clear_end_jump   state) = clear_end_jump (set_halt         state)›
  ‹set_error      (clear_end_jump   state) = clear_end_jump (set_error        state)›
  ‹clear_halt     (clear_end_jump   state) = clear_end_jump (clear_halt       state)›
  ‹clear_error    (clear_end_jump   state) = clear_end_jump (clear_error      state)›

  ‹set_error      (set_halt         state) = set_halt       (set_error        state)›
  ‹clear_error    (set_halt         state) = set_halt       (clear_error      state)›
  ‹set_error      (clear_halt       state) = clear_halt     (set_error        state)›
  ‹clear_error    (clear_halt       state) = clear_halt     (clear_error      state)›
by (simp_all add: state_manipulation_decomp)
```

lemma (**in** *Ironbark_world*) *post_instruction_flag_reorder:*

```
  ‹standard_post_instruction instruction_duration (set_end_jump      state)
  = set_end_jump     (standard_post_instruction instruction_duration state)›

  ‹standard_post_instruction instruction_duration (set_end_call      state)
  = set_end_call     (standard_post_instruction instruction_duration state)›

  ‹standard_post_instruction instruction_duration (set_end_return    state)
  = set_end_return   (standard_post_instruction instruction_duration state)›

  ‹standard_post_instruction instruction_duration (set_halt          state)
  = set_halt         (standard_post_instruction instruction_duration state)›

  ‹standard_post_instruction instruction_duration (set_error         state)
  = set_error        (standard_post_instruction instruction_duration state)›

  ‹standard_post_instruction instruction_duration (clear_end_jump    state)
  = clear_end_jump   (standard_post_instruction instruction_duration state)›

  ‹standard_post_instruction instruction_duration (clear_end_call    state)
  = clear_end_call   (standard_post_instruction instruction_duration state)›

  ‹standard_post_instruction instruction_duration (clear_end_return  state)
  = clear_end_return (standard_post_instruction instruction_duration state)›
```

```
  ‹standard_post_instruction instruction_duration (clear_halt        state)
  = clear_halt       (standard_post_instruction instruction_duration state)›

  ‹standard_post_instruction instruction_duration (clear_error       state)
  = clear_error      (standard_post_instruction instruction_duration state)›
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) backup_registers_flag_reorder:
  ‹backup_registers_before_call (set_end_jump state1)
  = set_end_jump (backup_registers_before_call state1)›

  ‹backup_registers_before_call (set_end_call state1)
  = set_end_call (backup_registers_before_call state1)›

  ‹backup_registers_before_call (set_end_return state1)
  = set_end_return (backup_registers_before_call state1)›

  ‹backup_registers_before_call (set_halt state1)
  = set_halt (backup_registers_before_call state1)›

  ‹backup_registers_before_call (set_error state1)
  = set_error (backup_registers_before_call state1)›
  by (simp_all add: state_manipulation_decomp)

end
```

C.4.2 Reordering Rules for Registers

```
theory register_reorder

imports
  state_manipulation_simps

begin
```

This file contains reordering rules that apply write register to a state.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) write_register_write_memory:
  fixes
    value1 :: ‹96 word›
    and value2 :: ‹64 word›
  assumes
    ‹regID ∈ registers›
  shows
    ‹write_register regID value (write_program_memory  address value1 state)
    = write_program_memory address value1 (write_register regID value state)›

    ‹write_register regID value (write_call_memory     address value2 state)
```

```
    = write_call_memory    address value2 (write_register regID value state)›

    ‹write_register regID value (write_static_memory   address value2 state)
    = write_static_memory  address value2 (write_register regID value state)›

    ‹write_register regID value (write_dynamic_memory  address value2 state)
    = write_dynamic_memory address value2 (write_register regID value state)›

    ‹write_register regID value (write_output_memory   address value2 state)
    = write_output_memory  address value2 (write_register regID value state)›
  using assms
  by(simp_all add: state_manipulation_decomp)

lemma diff_registers_write_write:
  assumes
    ‹regID1 ∈ registers›
    ‹regID2 ∈ registers›
    ‹regID1 ≠ regID2›
  shows
    ‹write_register   regID1 value1 (write_register regID2 value2 state)
    = write_register  regID2 value2 (write_register regID1 value1 state)›
  using assms by(simp add: write_register_def fun_upd_twist)

lemma (in Ironbark_world) register_post_reorder:
  assumes
    ‹regID ∉ {
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      cycles_register_ref
    }›
  shows
    ‹standard_post_instruction instruction_duration
      (write_register regID value state)
    = write_register regID value
      (standard_post_instruction instruction_duration state)›
  using assms
  apply(simp add: state_manipulation_decomp)
  apply(simp add: write_register_def)
  apply(simp add: fun_upd_twist)
  done

end
```

C.4.3 Reorder Rules For Memory

```
theory memory_reorder

imports
  state_manipulation_simps
```

begin

This file contains reordering rules that apply write memory operations to a state.

In this instance, we show that writes to the same memory space at different addresses can be freely reordered.

```
lemma (in Ironbark_world) diff_addr_write_write:
  fixes
    value1 value2 :: ‹96 word›
    and value3 value4 :: ‹64 word›
  assumes
    ‹address1 ≠ address2›
  shows
    ‹write_program_memory   address1 value1
      (write_program_memory address2 value2 state)
    = write_program_memory  address2 value2
      (write_program_memory address1 value1 state)›

    ‹write_call_memory      address1 value3
      (write_call_memory    address2 value4 state)
    = write_call_memory     address2 value4
      (write_call_memory    address1 value3 state)›

    ‹write_static_memory    address1 value3
      (write_static_memory  address2 value4 state)
    = write_static_memory   address2 value4
      (write_static_memory  address1 value3 state)›

    ‹write_dynamic_memory   address1 value3
      (write_dynamic_memory address2 value4 state)
    = write_dynamic_memory  address2 value4
      (write_dynamic_memory address1 value3 state)›

    ‹write_output_memory    address1 value3
      (write_output_memory  address2 value4 state)
    = write_output_memory   address2 value4
      (write_output_memory  address1 value3 state)›
  using assms by (simp_all add: state_manipulation_decomp fun_upd_twist)
```

end

C.4.4 Interface for State Manipulation Reordering

theory *state_manipulation_reorder*

imports
  flag_reorder
  register_reorder

memory_reorder

begin

This file provides a single point for importing all the content related to reordering at the state manipulation layer.
We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *register_reorder* =
write_register_write_memory
diff_registers_write_write
register_post_reorder

lemmas (**in** *Ironbark_world*) *flag_reorder* =
flag_flag_reorder
post_instruction_flag_reorder
backup_registers_flag_reorder

lemmas (**in** *Ironbark_world*) *state_manipulation_reorder* =
permission_simps
register_reorder
flag_reorder

end

C.5 State Manipulation Interface Layer

theory *state_manipulation_top*

imports
state_manipulation_reorder

begin

This file provides a single point for importing all the content at the state manipulation layer.
Since related lemmas have been bundled earlier, we do not need to bundle them here.

end

D Instruction Level Lemmas

D.1 Properties of Ironbark Instructions

theory *instruction_auxiliary*

imports
"Ironbark_state_manipulation.state_manipulation_top"

instruction_implementations

begin

We use this file as an anchor point for the start of the instruction layer of reasoning.

For convenience, we gather the definitions of each instruction into *instruction_impl_defs*.

lemmas (in *Ironbark_world)* *instruction_impl_defs* =
ERROR0_def
ERROR1_def
NOP_def
LOAD_IMMEDIATE_def
LOAD_STATIC_DATA_def
STORE_STATIC_DATA_def
LOAD_DYNAMIC_DATA_def
STORE_DYNAMIC_DATA_def
LOAD_INPUT_DATA_def
STORE_OUTPUT_DATA_def
COPY_def
ADD_def
SUBTRACT_def
SHIFT_LEFT_def
SHIFT_RIGHT_def
BITWISE_AND_def
BITWISE_OR_def
BITWISE_XOR_def
BITWISE_NAND_def
BITWISE_NOT_def
LESS_THAN_def
GREATER_THAN_def
EQUALS_def
NOT_EQUALS_def
RANDOMISE_def
END_JUMP_def
END_JUMP_STRICT_def
JUMP_def
CONDITIONAL_JUMP_def
END_CALL_def
CALL_def
END_RETURN_def
RETURN_def
HALT_def
ILLEGAL_def

end

D.2 Instruction Decomposition Rules

D.2.1 Decomposition Rules Over ERROR0 and ERROR1

theory *ERROR_decomposition*

imports
instruction_auxiliary

begin

We provide three lemmas which are different ways of decomposing the *error0* and *error1* instructions. The two instructions are bundled because they have identical functional operation.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second and third are decompositions purely expressed in state manipulation operations. However, the second lemma uses the standard error operation, while the third just uses set flag operations. As *error* only operates on flags, the usual mixed decomposition is not relevant.

```
lemma (in Ironbark_world) ERROR_decomp_state:
  ‹ERROR0 state = state (|flag_state := (flag_state state)(|error := 1, halt := 1|)|)›
  ‹ERROR1 state = state (|flag_state := (flag_state state)(|error := 1, halt := 1|)|)›
  by(simp_all add: instruction_impl_defs state_manipulation_decomp)

lemma (in Ironbark_world) ERROR_decomp_manipulation:
  ‹ERROR0 state = standard_error state›
  ‹ERROR1 state = standard_error state›
  by(simp_all add: instruction_impl_defs)

lemma (in Ironbark_world) ERROR_decomp_manipulation2:
  ‹ERROR0 state = set_halt (set_error state)›
  ‹ERROR1 state = set_halt (set_error state)›
  by(simp_all add: instruction_impl_defs state_manipulation_decomp)
```

end

D.2.2 Decomposition Rules Over NOP

theory *NOP_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *NOP* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each com-

ponent expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) NOP_decomp_state:
  assumes
    ‹typical_flags state›
  shows
    ‹NOP state =
      state
      (|
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) NOP_decomp_mixed:
  assumes
    ‹typical_flags state›
  shows
    ‹NOP state =
      state
      (|
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
```

```
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) NOP_decomp_manipulation:
  assumes
    ‹typical_flags state›
  shows
    ‹NOP state = standard_post_instruction common_instruction_duration state›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.3 Decomposition Rules Over LOAD_IMMEDIATE

theory *LOAD_IMMEDIATE_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *LOAD_IMMEDIATE* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) LOAD_IMMEDIATE_decomp_state:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
  shows
    ‹LOAD_IMMEDIATE reg1 immediate state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ immediate,
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
```

```
            + common_instruction_duration
        )
    ⦈›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) LOAD_IMMEDIATE_decomp_mixed:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
  shows
    ‹LOAD_IMMEDIATE reg1 immediate state =
      state
      (⦇
        register_state :=
          (register_state state)
          (
            reg1
              ↦ immediate,
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + common_instruction_duration
          )
      ⦈)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) LOAD_IMMEDIATE_decomp_manipulation:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
  shows
    ‹LOAD_IMMEDIATE reg1 immediate state =
      standard_post_instruction
        common_instruction_duration
        (write_register reg1 immediate state)›
  using assms
```

by *(simp add: instruction_impl_defs)*

end

D.2.4 Decomposition Rules Over LOAD_STATIC_DATA

theory *LOAD_STATIC_DATA_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *LOAD_STATIC_DATA* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) LOAD_STATIC_DATA_decomp_state:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹LOAD_STATIC_DATA reg1 reg2 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ the (static_memory state (the (register_state state reg2))),
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + memory_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
```

```
  done

lemma (in Ironbark_world) LOAD_STATIC_DATA_decomp_mixed:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹LOAD_STATIC_DATA reg1 reg2 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ read_static_memory (read_register reg2 state) state,
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + memory_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) LOAD_STATIC_DATA_decomp_manipulation:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹LOAD_STATIC_DATA reg1 reg2 state =
      standard_post_instruction
        memory_instruction_duration
        (write_register
          reg1
          (read_static_memory (read_register reg2 state) state)
          state)›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.5 Decomposition Rules Over STORE_STATIC_DATA

theory *STORE_STATIC_DATA_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the store static data instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL ('state'). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) STORE_STATIC_DATA_decomp_state:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹STORE_STATIC_DATA reg1 reg2 state =
      state
      (|
        static_memory :=
          (static_memory state)
          (
            the (register_state state reg1) ↦ the (register_state state reg2)
          ),
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                + memory_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) STORE_STATIC_DATA_decomp_mixed:
```

```
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹STORE_STATIC_DATA reg1 reg2 state =
      state
      (|
        static_memory :=
          (static_memory state)
          (
            read_register reg1 state ↦ read_register reg2 state
          ),
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + memory_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) STORE_STATIC_DATA_decomp_manipulation:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹STORE_STATIC_DATA reg1 reg2 state =
      standard_post_instruction
        memory_instruction_duration
        (write_static_memory
          (read_register reg1 state)
          (read_register reg2 state)
          state)›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.6 Decomposition Rules Over LOAD_DYNAMIC_DATA

theory *LOAD_DYNAMIC_DATA_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *LOAD_DYNAMIC_DATA* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL ('state'). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

lemma (**in** *Ironbark_world*) *LOAD_DYNAMIC_DATA_decomp_state:*
assumes
‹typical_flags state›
‹check_write_permission reg1›
‹check_read_permission reg2›
shows
‹LOAD_DYNAMIC_DATA reg1 reg2 state =
state(|
register_state :=
(register_state state)
(
reg1
↦ the (dynamic_memory state (the (register_state state reg2))),
last_instruction_pointer_ref
↦ read_register instruction_pointer_ref state,
instruction_pointer_ref
↦ (read_register instruction_pointer_ref state) + 1,
cycles_register_ref
↦ (read_register cycles_register_ref state)
+ memory_instruction_duration
)
|)›
using *assms*
apply(*simp add: instruction_impl_defs*)
apply(*simp add: state_manipulation_decomp*)
apply(*simp add: state_simps*)
apply(*simp add: fun_upd_twist*)
done

lemma (**in** *Ironbark_world*) *LOAD_DYNAMIC_DATA_decomp_mixed:*
assumes
‹typical_flags state›
‹check_write_permission reg1›

```
    ‹check_read_permission reg2›
  shows
    ‹LOAD_DYNAMIC_DATA reg1 reg2 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ read_dynamic_memory (read_register reg2 state) state,
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + memory_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) LOAD_DYNAMIC_DATA_decomp_manipulation:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹LOAD_DYNAMIC_DATA reg1 reg2 state =
      standard_post_instruction
        memory_instruction_duration
        (write_register
          reg1
          (read_dynamic_memory (read_register reg2 state) state)
          state)›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.7 Decomposition Rules Over STORE_DYNAMIC_DATA

```
theory STORE_DYNAMIC_DATA_decomposition

imports
```

```
  instruction_auxiliary
```

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the store dynamic data instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL ('state'). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) STORE_DYNAMIC_DATA_decomp_state:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹STORE_DYNAMIC_DATA reg1 reg2 state =
      state
      (|
        dynamic_memory :=
          (dynamic_memory state)
          (
            the (register_state state reg1) ↦ the (register_state state reg2)
          ),
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                + memory_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) STORE_DYNAMIC_DATA_decomp_mixed:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
```

```
  shows
    ‹STORE_DYNAMIC_DATA reg1 reg2 state =
      state(|
        dynamic_memory :=
          (dynamic_memory state)
          (
            read_register reg1 state ↦ read_register reg2 state
          ),
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + memory_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) STORE_DYNAMIC_DATA_decomp_manipulation:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹STORE_DYNAMIC_DATA reg1 reg2 state =
      standard_post_instruction
        memory_instruction_duration
        (write_dynamic_memory
          (read_register reg1 state)
          (read_register reg2 state)
          state)›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.8 Decomposition Rules Over LOAD_INPUT_DATA

```
theory LOAD_INPUT_DATA_decomposition

imports
```

```
  instruction_auxiliary
```

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *LOAD_INPUT_DATA* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) LOAD_INPUT_DATA_decomp_state:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹LOAD_INPUT_DATA reg1 reg2 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ the (input_memory state time (the (register_state state reg2))),
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + memory_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) LOAD_INPUT_DATA_decomp_mixed:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹LOAD_INPUT_DATA reg1 reg2 state =
```

```
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ read_input_memory (read_register reg2 state) state,
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + memory_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) LOAD_INPUT_DATA_decomp_manipulation:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹LOAD_INPUT_DATA reg1 reg2 state =
      standard_post_instruction
        memory_instruction_duration
        (write_register
          reg1
          (read_input_memory (read_register reg2 state) state)
          state)›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.9 Decomposition Rules Over STORE_OUTPUT_DATA

```
theory STORE_OUTPUT_DATA_decomposition

imports
  instruction_auxiliary

begin
```

As with most instructions, we provide three lemmas which are different ways of decomposing the *STORE_OUTPUT_DATA* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) STORE_OUTPUT_DATA_decomp_state:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹STORE_OUTPUT_DATA reg1 reg2 state =
      state
      (|
        output_memory :=
          (output_memory state)
          (
            the (register_state state reg1) ↦ the (register_state state reg2)
          ),
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                + memory_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) STORE_OUTPUT_DATA_decomp_mixed:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹STORE_OUTPUT_DATA reg1 reg2 state =
      state
      (|
```

```
        output_memory :=
          (output_memory state)
          (
            read_register reg1 state ↦ read_register reg2 state
          ),
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + memory_instruction_duration
          )
      ⦆›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) STORE_OUTPUT_DATA_decomp_manipulation:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹STORE_OUTPUT_DATA reg1 reg2 state =
      standard_post_instruction
        memory_instruction_duration
        (write_output_memory
          (read_register reg1 state)
          (read_register reg2 state)
          state)›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.10 Decomposition Rules Over COPY

```
theory COPY_decomposition

imports
  instruction_auxiliary

begin
```

As with most instructions, we provide three lemmas which are different ways of decomposing the *COPY* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) COPY_decomp_state:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹COPY reg1 reg2 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ the (register_state state reg2),
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) COPY_decomp_mixed:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹COPY reg1 reg2 state =
      state
      (|
        register_state :=
          (register_state state)
```

```
        (
          reg1
            ↦ read_register reg2 state,
          last_instruction_pointer_ref
            ↦ read_register instruction_pointer_ref state,
          instruction_pointer_ref
            ↦ (read_register instruction_pointer_ref state) + 1,
          cycles_register_ref
            ↦ (read_register cycles_register_ref state)
              + common_instruction_duration
        )
    ⦆›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) COPY_decomp_manipulation:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹COPY reg1 reg2 state =
      standard_post_instruction
        common_instruction_duration
        (write_register
          reg1
          (read_register reg2 state)
          state)›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.11 Decomposition Rules Over ADD

theory *ADD_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *ADD* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) ADD_decomp_state:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹ADD reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ the (register_state state reg2) + the (register_state state reg3),
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) ADD_decomp_mixed:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹ADD reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
```

```
            ↦ (read_register reg2 state) + (read_register reg3 state),
          last_instruction_pointer_ref
            ↦ read_register instruction_pointer_ref state,
          instruction_pointer_ref
            ↦ (read_register instruction_pointer_ref state) + 1,
          cycles_register_ref
            ↦ (read_register cycles_register_ref state)
              + common_instruction_duration
        )
    ⦈›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) ADD_decomp_manipulation:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹ADD reg1 reg2 reg3 state =
      standard_post_instruction
        common_instruction_duration
        (write_register
          reg1
          ((read_register reg2 state) + (read_register reg3 state))
          state)›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.12 Decomposition Rules Over SUBTRACT

theory *SUBTRACT_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *SUBTRACT* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*).

The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) SUBTRACT_decomp_state:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹SUBTRACT reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ the (register_state state reg2) - the (register_state state reg3),
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
              + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) SUBTRACT_decomp_mixed:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹SUBTRACT reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ (read_register reg2 state) - (read_register reg3 state),
```

```
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + common_instruction_duration
          )
      ⦆›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) SUBTRACT_decomp_manipulation:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹SUBTRACT reg1 reg2 reg3 state =
      standard_post_instruction
        common_instruction_duration
        (write_register
          reg1
          ((read_register reg2 state) - (read_register reg3 state))
          state)›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.13 Decomposition Rules Over SHIFT_LEFT

theory *SHIFT_LEFT_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *SHIFT_LEFT* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (`state`). The second is a decomposition that is a mix of native Isabelle/HOL components, with each com-

ponent expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) SHIFT_LEFT_decomp_state:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹SHIFT_LEFT reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1 ↦
              push_bit
                (unat (the (register_state state reg3)))
                (the (register_state state reg2)),
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) SHIFT_LEFT_decomp_mixed:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹SHIFT_LEFT reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1 ↦
```

```
            push_bit
              (unat (read_register reg3 state))
              (read_register reg2 state),
          last_instruction_pointer_ref
            ↦ read_register instruction_pointer_ref state,
          instruction_pointer_ref
            ↦ (read_register instruction_pointer_ref state) + 1,
          cycles_register_ref
            ↦ (read_register cycles_register_ref state)
              + common_instruction_duration
        )
    ⟫›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) SHIFT_LEFT_decomp_manipulation:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹SHIFT_LEFT reg1 reg2 reg3 state =
        standard_post_instruction
          common_instruction_duration
          (write_register
            reg1
            (push_bit (unat (read_register reg3 state)) (read_register reg2 state))
            state)›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.14 Decomposition Rules Over SHIFT_RIGHT

theory *SHIFT_RIGHT_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *SHIFT_RIGHT* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) SHIFT_RIGHT_decomp_state:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹SHIFT_RIGHT reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1 ↦
              drop_bit
                (unat (the (register_state state reg3)))
                (the (register_state state reg2)),
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref) + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) SHIFT_RIGHT_decomp_mixed:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹SHIFT_RIGHT reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
```

```
            reg1 ↦
              drop_bit
                (unat (read_register reg3 state))
                (read_register reg2 state),
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state) + common_instruction_duration
          )
      ⟫›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) SHIFT_RIGHT_decomp_manipulation:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹SHIFT_RIGHT reg1 reg2 reg3 state =
        standard_post_instruction
          common_instruction_duration
          (write_register
            reg1
            (drop_bit (unat (read_register reg3 state)) (read_register reg2 state))
            state)›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.15 Decomposition Rules Over BITWISE_AND

theory *BITWISE_AND_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *BITWISE_AND* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) BITWISE_AND_decomp_state:
  includes
    bit_operations_syntax
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹BITWISE_AND reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ the (register_state state reg2) AND the (register_state state reg3),
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) BITWISE_AND_decomp_mixed:
  includes
    bit_operations_syntax
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹BITWISE_AND reg1 reg2 reg3 state =
      state
      (|
```

```
        register_state :=
          (register_state state)
          (
            reg1
              ↦ (read_register reg2 state) AND (read_register reg3 state),
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + common_instruction_duration
          )
      ⦇›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) BITWISE_AND_decomp_manipulation:
  includes
    bit_operations_syntax
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹BITWISE_AND reg1 reg2 reg3 state =
      standard_post_instruction common_instruction_duration
        (write_register
          reg1
          ((read_register reg2 state) AND (read_register reg3 state))
          state)›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.16 Decomposition Rules Over BITWISE_OR

```
theory BITWISE_OR_decomposition

imports
  instruction_auxiliary

begin
```

As with most instructions, we provide three lemmas which are different ways of decomposing the *BITWISE_OR* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) BITWISE_OR_decomp_state:
  includes
    bit_operations_syntax
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹BITWISE_OR reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ the (register_state state reg2) OR the (register_state state reg3),
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) BITWISE_OR_decomp_mixed:
  includes
    bit_operations_syntax
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
```

```
  shows
    ‹BITWISE_OR reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ (read_register reg2 state) OR (read_register reg3 state),
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state) + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) BITWISE_OR_decomp_manipulation:
  includes
    bit_operations_syntax
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹BITWISE_OR reg1 reg2 reg3 state =
      standard_post_instruction
        common_instruction_duration
        (write_register
          reg1
          ((read_register reg2 state) OR (read_register reg3 state))
          state)›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.17 Decomposition Rules Over BITWISE_XOR

```
theory BITWISE_XOR_decomposition
```

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *BITWISE_XOR* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) BITWISE_XOR_decomp_state:
  includes
    bit_operations_syntax
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹BITWISE_XOR reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ the (register_state state reg2) XOR the (register_state state reg3),
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) BITWISE_XOR_decomp_mixed:
  includes
    bit_operations_syntax
```

```
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹BITWISE_XOR reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ (read_register reg2 state) XOR (read_register reg3 state),
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) BITWISE_XOR_decomp_manipulation:
  includes
    bit_operations_syntax
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹BITWISE_XOR reg1 reg2 reg3 state =
      standard_post_instruction
        common_instruction_duration
        (write_register
          reg1
          ((read_register reg2 state) XOR (read_register reg3 state))
          state)›
  using assms
  by(simp add: instruction_impl_defs)
```

end

D.2.18 Decomposition Rules Over BITWISE_NAND

theory *BITWISE_NAND_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *BITWISE_NAND* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) BITWISE_NAND_decomp_state:
  includes
    bit_operations_syntax
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹BITWISE_NAND reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ NOT
                (the (register_state state reg2) AND the (register_state state reg3)),
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                  + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
```

```
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) BITWISE_NAND_decomp_mixed:
  includes
    bit_operations_syntax
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹BITWISE_NAND reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ NOT ((read_register reg2 state) AND (read_register reg3 state)),
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) BITWISE_NAND_decomp_manipulation:
  includes
    bit_operations_syntax
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹BITWISE_NAND reg1 reg2 reg3 state =
      standard_post_instruction
        common_instruction_duration
```

```
      (write_register
        reg1
        (NOT ((read_register reg2 state) AND (read_register reg3 state)))
        state)›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.19 Decomposition Rules Over BITWISE_NOT

theory *BITWISE_NOT_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *BITWISE_NOT* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) BITWISE_NOT_decomp_state:
  includes
    bit_operations_syntax
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹BITWISE_NOT reg1 reg2 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ NOT (the (register_state state reg2)),
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                + common_instruction_duration
```

```
        )
    ⦆›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) BITWISE_NOT_decomp_mixed:
includes
    bit_operations_syntax
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹BITWISE_NOT reg1 reg2 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ NOT (read_register reg2 state),
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state) + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) BITWISE_NOT_decomp_manipulation:
  includes
    bit_operations_syntax
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹BITWISE_NOT reg1 reg2 state =
```

```
      standard_post_instruction
        common_instruction_duration
        (write_register
          reg1
          (NOT (read_register reg2 state))
          state)›
  using assms
  by(simp add: instruction_impl_defs)
```

end

D.2.20 Decomposition Rules Over LESS_THAN

theory *LESS_THAN_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *LESS_THAN* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) LESS_THAN_decomp_state:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹LESS_THAN reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1 ↦
              if
                (the (register_state state reg2) < the (register_state state reg3))
              then
                0x1
              else
                0x0,
            last_instruction_pointer_ref
```

```
            ↦ the (register_state state instruction_pointer_ref),
          instruction_pointer_ref
            ↦ the (register_state state instruction_pointer_ref) + 1,
          cycles_register_ref
            ↦ the (register_state state cycles_register_ref)
              + common_instruction_duration
        )
    ⦆›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) LESS_THAN_decomp_mixed:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹LESS_THAN reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1 ↦
              if ((read_register reg2 state) < (read_register reg3 state)) then
                0x1
              else
                0x0,
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done
```

```
lemma (in Ironbark_world) LESS_THAN_decomp_manipulation:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹LESS_THAN reg1 reg2 reg3 state =
    (
      if ((read_register reg2 state) < (read_register reg3 state)) then
        standard_post_instruction
          common_instruction_duration
          (write_register reg1 0x01 state)
      else
        standard_post_instruction
          common_instruction_duration
          (write_register reg1 0x00 state)
    )›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.21 Decomposition Rules Over GREATER_THAN

theory *GREATER_THAN_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *GREATER_THAN* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) GREATER_THAN_decomp_state:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹GREATER_THAN reg1 reg2 reg3 state =
      state
```

```
    (|
      register_state :=
        (register_state state)
        (
          reg1 ↦
            if
              (the (register_state state reg2) > the (register_state state reg3))
            then
              0x1
            else
              0x0,
          last_instruction_pointer_ref
            ↦ the (register_state state instruction_pointer_ref),
          instruction_pointer_ref
            ↦ the (register_state state instruction_pointer_ref) + 1,
          cycles_register_ref
            ↦ the (register_state state cycles_register_ref)
              + common_instruction_duration
        )
    |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) GREATER_THAN_decomp_mixed:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹GREATER_THAN reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1 ↦
              if ((read_register reg2 state) > (read_register reg3 state)) then
                0x1
              else
                0x0,
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
```

```
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + common_instruction_duration
          )
      ⟫›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) GREATER_THAN_decomp_manipulation:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹GREATER_THAN reg1 reg2 reg3 state = (
      if ((read_register reg2 state) > (read_register reg3 state)) then
        standard_post_instruction
          common_instruction_duration
          (write_register reg1 0x01 state)
      else
        standard_post_instruction
          common_instruction_duration
          (write_register reg1 0x00 state)
      )›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.22 Decomposition Rules Over EQUALS

theory *EQUALS_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *EQUALS* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed

in state manipulation operations.

```
lemma (in Ironbark_world) EQUALS_decomp_state:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹EQUALS reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1 ↦
              if
                (the (register_state state reg2) = the (register_state state reg3))
              then
                0x1
              else
                0x0,
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) EQUALS_decomp_mixed:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹EQUALS reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
```

```
        (
          reg1 ↦
            if ((read_register reg2 state) = (read_register reg3 state)) then
              0x1
            else
              0x0,
          last_instruction_pointer_ref
            ↦ read_register instruction_pointer_ref state,
          instruction_pointer_ref
            ↦ (read_register instruction_pointer_ref state) + 1,
          cycles_register_ref
            ↦ (read_register cycles_register_ref state)
              + common_instruction_duration
        )
    ⟫
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) EQUALS_decomp_manipulation:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹EQUALS reg1 reg2 reg3 state =
    (
      if ((read_register reg2 state) = (read_register reg3 state)) then
        standard_post_instruction
          common_instruction_duration
          (write_register reg1 0x01 state)
      else
        standard_post_instruction
          common_instruction_duration
          (write_register reg1 0x00 state)
    )›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.23 Decomposition Rules Over NOT_EQUALS

theory *NOT_EQUALS_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *NOT_EQUALS* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.
The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) NOT_EQUALS_decomp_state:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹NOT_EQUALS reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1 ↦
              if (the (register_state state reg2) = the (register_state state reg3)) then
                0x0
              else
                0x1,
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) NOT_EQUALS_decomp_mixed:
  assumes
```

```
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹NOT_EQUALS reg1 reg2 reg3 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1 ↦
              if ((read_register reg2 state) = (read_register reg3 state)) then
                0x0
              else
                0x1,
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) NOT_EQUALS_decomp_manipulation:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹NOT_EQUALS reg1 reg2 reg3 state = (
      if ((read_register reg2 state) = (read_register reg3 state)) then
        standard_post_instruction
          common_instruction_duration
          (write_register reg1 0x00 state)
      else
        standard_post_instruction
          common_instruction_duration
          (write_register reg1 0x01 state)
      )›
```

```
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.24 Decomposition Rules Over RANDOMISE

```
theory RANDOMISE_decomposition

imports
  instruction_auxiliary

begin
```

As with most instructions, we provide three lemmas which are different ways of decomposing the *RANDOMISE* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) RANDOMISE_decomp_state:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
  shows
    ‹RANDOMISE reg1 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ random_stream time,
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
```

```
  done

lemma (in Ironbark_world) RANDOMISE_decomp_mixed:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
  shows
    ‹RANDOMISE reg1 state =
      state
      (|
        register_state :=
          (register_state state)
          (
            reg1
              ↦ random_stream time,
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: state_simps)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) RANDOMISE_decomp_manipulation:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
  shows
    ‹RANDOMISE reg1 state =
      standard_post_instruction
        common_instruction_duration
        (write_register
          reg1
          (random_stream time)
          state)›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.25 Decomposition Rules Over END_JUMP

theory *END_JUMP_decomposition*

imports
instruction_auxiliary

begin

Unlike most instructions, we provide six lemmas for decomposing the *END_JUMP* instruction. Three lemmas are for when the end jump follows a jump, and the other three are for when it is not after a jump (or 'falls through'). In all cases, they are the 'special' case where we assume the guards of the instruction will pass.

As with most instructions, we have lemmas in each case for how it decomposes to state level operations in native Isabelle/HOL (*state*), a mix of native Isabelle/HOL components (with each component expressed at the state manipulation level), and expressed purely in state manipulation operations.

```
lemma (in Ironbark_world) END_JUMP_decomp_state_jump:
  assumes
    ‹end_jump_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹END_JUMP immediate state =
      state
      (|
        flag_state := (flag_state state)(|end_jump := 0|),
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) END_JUMP_decomp_mixed_jump:
  assumes
    ‹end_jump_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
```

```
    ‹END_JUMP immediate state =
      state
      (|
        flag_state := flag_state (clear_end_jump state),
        register_state :=
          (register_state state) (
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) END_JUMP_decomp_manipulation_jump:
  assumes
    ‹end_jump_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹END_JUMP immediate state
    = standard_post_instruction common_instruction_duration (clear_end_jump state)›
  using assms
  by(simp add: instruction_impl_defs)

lemma (in Ironbark_world) END_JUMP_decomp_state_fall:
  assumes
    ‹typical_flags state›
  shows
    ‹END_JUMP immediate state =
      state
      (|
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                + common_instruction_duration
          )
```

```
    ⦈›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) END_JUMP_decomp_mixed_fall:
  assumes
    ‹typical_flags state›
  shows
    ‹END_JUMP immediate state =
      state
      (|
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + common_instruction_duration
          )
      ⦈›
  using assms
  apply(simp add: END_JUMP_decomp_state_fall)
  apply(simp add: state_manipulation_decomp)
  done

lemma (in Ironbark_world) END_JUMP_decomp_manipulation_fall:
  assumes
    ‹typical_flags state›
  shows
    ‹END_JUMP immediate state
    = standard_post_instruction common_instruction_duration state›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  done

end
```

D.2.26 Decomposition Rules Over END_JUMP_STRICT

theory *END_JUMP_STRICT_decomposition*

imports

```
  instruction_auxiliary
```

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the end jump strict instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL ('state'). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) END_JUMP_STRICT_decomp_state:
  assumes
    ‹end_jump_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹END_JUMP_STRICT immediate state =
      state
      (|
        flag_state := (flag_state state)(|end_jump := 0|),
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) END_JUMP_STRICT_decomp_mixed:
  assumes
    ‹end_jump_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹END_JUMP_STRICT immediate state =
      state
      (|
        flag_state := flag_state (clear_end_jump state),
        register_state :=
```

```
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + common_instruction_duration
          )
      ⦆›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) END_JUMP_STRICT_decomp_manipulation:
  assumes
    ‹end_jump_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹END_JUMP_STRICT immediate state
    = standard_post_instruction common_instruction_duration (clear_end_jump state)›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.27 Decomposition Rules Over JUMP

theory *JUMP_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *JUMP* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) JUMP_decomp_state:
  assumes
    ‹typical_flags state›
```

```
  shows
    ‹JUMP immediate state =
      state
      (|
        flag_state := (flag_state state)(|end_jump := 1|),
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ immediate,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_simps)
  apply(simp add: state_manipulation_decomp)
  done

lemma (in Ironbark_world) JUMP_decomp_mixed:
  assumes
    ‹typical_flags state›
  shows
    ‹JUMP immediate state =
      state
      (|
        flag_state := flag_state (set_end_jump state),
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ immediate,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_simps)
  apply(simp add: state_manipulation_decomp)
  done
```

```
lemma (in Ironbark_world) JUMP_decomp_manipulation:
  assumes
    ‹typical_flags state›
  shows
    ‹JUMP immediate state =
      write_register
        cycles_register_ref
        ((read_register cycles_register_ref state) + common_instruction_duration)
        (write_register
          instruction_pointer_ref
          immediate
          (write_register
            last_instruction_pointer_ref
            (read_register instruction_pointer_ref state)
            (set_end_jump state)))›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_simps)
  done

end
```

D.2.28 Decomposition Rules Over CONDITIONAL_JUMP

theory *CONDITIONAL_JUMP_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *CONDITIONAL_JUMP* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*CONDITIONAL_JUMP*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) CONDITIONAL_JUMP_decomp_state:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
  shows
    ‹CONDITIONAL_JUMP reg1 immediate state =
      state
      (|
        flag_state :=
          if the (register_state state reg1) = 0 then
```

```
        flag_state state
      else
        (flag_state state(|end_jump := 1|)),
    register_state :=
      (register_state state)
      (
        cycles_register_ref
          ↦ the (register_state state cycles_register_ref)
            + common_instruction_duration,
        last_instruction_pointer_ref
          ↦ the (register_state state instruction_pointer_ref),
        instruction_pointer_ref ↦
          if the (register_state state reg1) = 0 then
              the (register_state state instruction_pointer_ref) + 1
          else
            immediate
      )
  |)›
 using assms
 apply(simp add: instruction_impl_defs)
 apply(simp add: state_manipulation_decomp)
 apply(simp add: fun_upd_twist)
 done

lemma (in Ironbark_world) CONDITIONAL_JUMP_decomp_mixed:
 assumes
  ‹typical_flags state›
  ‹check_read_permission reg1›
 shows
  ‹CONDITIONAL_JUMP reg1 immediate state =
    state
    (|
      flag_state :=
        if (read_register reg1 state = 0) then
          flag_state state
        else
          flag_state (set_end_jump state),
      register_state :=
        (register_state state)
        (
          cycles_register_ref
            ↦ (read_register cycles_register_ref state)
              + common_instruction_duration,
          last_instruction_pointer_ref
            ↦ read_register instruction_pointer_ref state,
          instruction_pointer_ref ↦
            if (read_register reg1 state = 0) then
              (read_register instruction_pointer_ref state) + 1
            else
```

```
                immediate
            )
        ⦇⦈⟩
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) CONDITIONAL_JUMP_decomp_manipulation:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
  shows
    ‹CONDITIONAL_JUMP reg1 immediate state = (
      if (read_register reg1 state) = 0x00 then
        standard_post_instruction common_instruction_duration state
      else
        write_register
          cycles_register_ref
          ((read_register cycles_register_ref state) + common_instruction_duration)
          (write_register
            instruction_pointer_ref
            immediate
            (write_register
              last_instruction_pointer_ref
              (read_register instruction_pointer_ref state)
              (set_end_jump state))))›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_simps)
  done

end
```

D.2.29 Decomposition Rules Over END_CALL

theory *END_CALL_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *END_CALL* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each com-

ponent expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) END_CALL_decomp_state:
  assumes
    ‹end_call_flags state›
  shows
    ‹END_CALL state =
      state
      (|
        flag_state := (flag_state state)(|end_call := 0|),
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
            instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref) + 1,
            cycles_register_ref
              ↦ the (register_state state cycles_register_ref)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) END_CALL_decomp_mixed:
  assumes
    ‹end_call_flags state›
  shows
    ‹END_CALL state =
      state
      (|
        flag_state := flag_state (clear_end_call state),
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + common_instruction_duration
          )
      |)›
  using assms
```

```
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) END_CALL_decomp_manipulation:
  assumes
    ‹end_call_flags state›
  shows
    ‹END_CALL state =
      standard_post_instruction
        common_instruction_duration
        (clear_end_call state)›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.30 Decomposition Rules Over CALL

```
theory CALL_decomposition

imports
  instruction_auxiliary

begin
```

As with most instructions, we provide three lemmas which are different ways of decomposing the *CALL* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) CALL_decomp_state:
  assumes
    ‹typical_flags state›
  shows
    ‹CALL immediate state =
      state
      (|
        call_memory :=
          (call_memory state)
          (
            (read_register call_frame_pointer_ref state) + 66
              ↦ (read_register r00_ref state),
            (read_register call_frame_pointer_ref state) + 65
              ↦ (read_register r01_ref state),
```

```
(read_register call_frame_pointer_ref state) + 64
  ↦ (read_register r02_ref state),
(read_register call_frame_pointer_ref state) + 63
  ↦ (read_register r03_ref state),
(read_register call_frame_pointer_ref state) + 62
  ↦ (read_register r04_ref state),
(read_register call_frame_pointer_ref state) + 61
  ↦ (read_register r05_ref state),
(read_register call_frame_pointer_ref state) + 60
  ↦ (read_register r06_ref state),
(read_register call_frame_pointer_ref state) + 59
  ↦ (read_register r07_ref state),
(read_register call_frame_pointer_ref state) + 58
  ↦ (read_register r08_ref state),
(read_register call_frame_pointer_ref state) + 57
  ↦ (read_register r09_ref state),
(read_register call_frame_pointer_ref state) + 56
  ↦ (read_register r10_ref state),
(read_register call_frame_pointer_ref state) + 55
  ↦ (read_register r11_ref state),
(read_register call_frame_pointer_ref state) + 54
  ↦ (read_register r12_ref state),
(read_register call_frame_pointer_ref state) + 53
  ↦ (read_register r13_ref state),
(read_register call_frame_pointer_ref state) + 52
  ↦ (read_register r14_ref state),
(read_register call_frame_pointer_ref state) + 51
  ↦ (read_register r15_ref state),
(read_register call_frame_pointer_ref state) + 50
  ↦ (read_register p00_ref state),
(read_register call_frame_pointer_ref state) + 49
  ↦ (read_register p01_ref state),
(read_register call_frame_pointer_ref state) + 48
  ↦ (read_register p02_ref state),
(read_register call_frame_pointer_ref state) + 47
  ↦ (read_register p03_ref state),
(read_register call_frame_pointer_ref state) + 46
  ↦ (read_register p04_ref state),
(read_register call_frame_pointer_ref state) + 45
  ↦ (read_register p05_ref state),
(read_register call_frame_pointer_ref state) + 44
  ↦ (read_register p06_ref state),
(read_register call_frame_pointer_ref state) + 43
  ↦ (read_register p07_ref state),
(read_register call_frame_pointer_ref state) + 42
  ↦ (read_register p08_ref state),
(read_register call_frame_pointer_ref state) + 41
  ↦ (read_register p09_ref state),
(read_register call_frame_pointer_ref state) + 40
```

```
  ↦ (read_register p10_ref state),
(read_register call_frame_pointer_ref state) + 39
  ↦ (read_register p11_ref state),
(read_register call_frame_pointer_ref state) + 38
  ↦ (read_register p12_ref state),
(read_register call_frame_pointer_ref state) + 37
  ↦ (read_register p13_ref state),
(read_register call_frame_pointer_ref state) + 36
  ↦ (read_register p14_ref state),
(read_register call_frame_pointer_ref state) + 35
  ↦ (read_register p15_ref state),
(read_register call_frame_pointer_ref state) + 34
  ↦ (read_register c00_ref state),
(read_register call_frame_pointer_ref state) + 33
  ↦ (read_register c01_ref state),
(read_register call_frame_pointer_ref state) + 32
  ↦ (read_register c02_ref state),
(read_register call_frame_pointer_ref state) + 31
  ↦ (read_register c03_ref state),
(read_register call_frame_pointer_ref state) + 30
  ↦ (read_register c04_ref state),
(read_register call_frame_pointer_ref state) + 29
  ↦ (read_register c05_ref state),
(read_register call_frame_pointer_ref state) + 28
  ↦ (read_register c06_ref state),
(read_register call_frame_pointer_ref state) + 27
  ↦ (read_register c07_ref state),
(read_register call_frame_pointer_ref state) + 26
  ↦ (read_register c08_ref state),
(read_register call_frame_pointer_ref state) + 25
  ↦ (read_register c09_ref state),
(read_register call_frame_pointer_ref state) + 24
  ↦ (read_register c10_ref state),
(read_register call_frame_pointer_ref state) + 23
  ↦ (read_register c11_ref state),
(read_register call_frame_pointer_ref state) + 22
  ↦ (read_register c12_ref state),
(read_register call_frame_pointer_ref state) + 21
  ↦ (read_register c13_ref state),
(read_register call_frame_pointer_ref state) + 20
  ↦ (read_register c14_ref state),
(read_register call_frame_pointer_ref state) + 19
  ↦ (read_register c15_ref state),
(read_register call_frame_pointer_ref state) + 18
  ↦ (read_register arg00_ref state),
(read_register call_frame_pointer_ref state) + 17
  ↦ (read_register arg01_ref state),
(read_register call_frame_pointer_ref state) + 16
  ↦ (read_register arg02_ref state),
```

```
      (read_register call_frame_pointer_ref state) + 15
        ↦ (read_register arg03_ref state),
      (read_register call_frame_pointer_ref state) + 14
        ↦ (read_register arg04_ref state),
      (read_register call_frame_pointer_ref state) + 13
        ↦ (read_register arg05_ref state),
      (read_register call_frame_pointer_ref state) + 12
        ↦ (read_register arg06_ref state),
      (read_register call_frame_pointer_ref state) + 11
        ↦ (read_register arg07_ref state),
      (read_register call_frame_pointer_ref state) + 10
        ↦ (read_register arg08_ref state),
      (read_register call_frame_pointer_ref state) +  9
        ↦ (read_register arg09_ref state),
      (read_register call_frame_pointer_ref state) +  8
        ↦ (read_register arg10_ref state),
      (read_register call_frame_pointer_ref state) +  7
        ↦ (read_register arg11_ref state),
      (read_register call_frame_pointer_ref state) +  6
        ↦ (read_register arg12_ref state),
      (read_register call_frame_pointer_ref state) +  5
        ↦ (read_register arg13_ref state),
      (read_register call_frame_pointer_ref state) +  4
        ↦ (read_register arg14_ref state),
      (read_register call_frame_pointer_ref state) +  3
        ↦ (read_register arg15_ref state),
      (read_register call_frame_pointer_ref state) +  2
        ↦ (read_register static_data_frame_pointer_ref state),
      (read_register call_frame_pointer_ref state) +  1
        ↦ (read_register static_data_stack_pointer_ref state),
      (read_register call_frame_pointer_ref state) +  0
        ↦ (read_register instruction_pointer_ref state)
    ),
    flag_state := (flag_state state) (|end_call := 1|),
    register_state :=
      (register_state state)
      (
        last_instruction_pointer_ref
          ↦ the (register_state state instruction_pointer_ref),
        instruction_pointer_ref
          ↦ immediate,
        cycles_register_ref
          ↦ the (register_state state cycles_register_ref) + call_duration,
        call_frame_pointer_ref
          ↦ the (register_state state call_frame_pointer_ref) + 67
      )
  |⟩
using assms
apply(subst sequential_state_equality)
```

```
  apply(simp_all)
  apply(simp_all add: instruction_impl_defs)
  apply(simp_all add: state_manipulation_simps)
  apply(simp_all add: state_manipulation_decomp del: backup_registers_before_call_def)
  apply(simp_all add: state_manipulation_simps)
  apply(simp_all add: fun_upd_twist)
  apply(simp_all add: state_manipulation_decomp)
  apply(simp_all add: fun_upd_twist)
  done

lemma (in Ironbark_world) CALL_decomp_mixed:
  assumes
    ‹typical_flags state›
  shows
    ‹CALL immediate state =
      state
      (|
        call_memory := call_memory (backup_registers_before_call state),
        flag_state := flag_state (set_end_call state),
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ immediate,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state) + call_duration,
            call_frame_pointer_ref
              ↦ read_register call_frame_pointer_ref state + 67
          )
      |)›
  using assms
  apply(subst sequential_state_equality)
  apply(simp_all)
  apply(simp_all add: instruction_impl_defs)
  apply(simp_all add: state_manipulation_simps)
  apply(simp_all add: state_manipulation_decomp)
  apply(simp_all add: fun_upd_twist)
  done

lemma (in Ironbark_world) CALL_decomp_manipulation:
  assumes
    ‹typical_flags state›
  shows
    ‹CALL immediate state =
      write_register
        cycles_register_ref
        ((read_register cycles_register_ref state) + call_duration)
```

```
        (write_register
          instruction_pointer_ref
          immediate
          (write_register
            last_instruction_pointer_ref
            (read_register instruction_pointer_ref state)
              (write_register
                call_frame_pointer_ref
                ((read_register call_frame_pointer_ref state) + 67)
                (backup_registers_before_call
                  (set_end_call state)))))›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_simps Let_def)
  done

end
```

D.2.31 Decomposition Rules Over END_RETURN

```
theory END_RETURN_decomposition

imports
  instruction_auxiliary

begin
```

As with most instructions, we provide three lemmas which are different ways of decomposing the *END_RETURN* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) END_RETURN_decomp_state:
  assumes
    ‹end_return_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹END_RETURN immediate state =
      state
      (|
        flag_state := (flag_state state)(|end_return := 0|),
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),
```

```
          instruction_pointer_ref
            ↦ the (register_state state instruction_pointer_ref) + 1,
          cycles_register_ref
            ↦ the (register_state state cycles_register_ref)
              + common_instruction_duration
        )
    ⦈›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) END_RETURN_decomp_mixed:
  assumes
    ‹end_return_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹END_RETURN immediate state =
      state
      (|
        flag_state := flag_state (clear_end_return state),
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ (read_register instruction_pointer_ref state) + 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref state)
                + common_instruction_duration
          )
      |)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) END_RETURN_decomp_manipulation:
  assumes
    ‹end_return_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹END_RETURN immediate state
    = standard_post_instruction common_instruction_duration (clear_end_return state)›
  using assms
  by(simp add: instruction_impl_defs)
```

end

D.2.32 Decomposition Rules Over RETURN

theory *RETURN_decomposition*

imports
instruction_auxiliary

begin

As with most instructions, we provide three lemmas which are different ways of decomposing the *RETURN* instruction. The three lemmas are the 'special' case where we assume the guards of the instruction will pass.
The first lemma shows how it decomposes to state level operations in native Isabelle/HOL (*state*). The second is a decomposition that is a mix of native Isabelle/HOL components, with each component expressed at the state manipulation level. The third is a decomposition purely expressed in state manipulation operations.

```
lemma (in Ironbark_world) RETURN_decomp_state:
  assumes
    ‹typical_flags state›
  shows
    ‹RETURN state =
      state
      (|
        flag_state := (flag_state state)(|end_return := 1|),
        register_state :=
          (register_state state)
          (
            last_instruction_pointer_ref
              ↦ the (register_state state instruction_pointer_ref),

            cycles_register_ref
              ↦ the (register_state state cycles_register_ref) + call_duration,

            call_frame_pointer_ref
              ↦ the (register_state state call_frame_pointer_ref) - 67,

            r00_ref
              ↦ the
              (
                call_memory
                  state
                  (the (register_state state call_frame_pointer_ref) -  1)
              ),
            r01_ref
              ↦ the (
```

```
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) -  2)
  ),
r02_ref
  ↦ the
 (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) -  3)
  ),
r03_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) -  4)
  ),
r04_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) -  5)
  ),
r05_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) -  6)
  ),
r06_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) -  7)
  ),
r07_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) -  8)
  ),
r08_ref
  ↦ the
  (
```

```
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) -  9)
  ),
r09_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 10)
  ),
r10_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 11)
  ),
r11_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 12)
  ),
r12_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 13)
  ),
r13_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 14)
  ),
r14_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 15)
  ),
r15_ref
  ↦ the
  (
```

```
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 16)
  ),
p00_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 17)
  ),
p01_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 18)
  ),
p02_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 19)
  ),
p03_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 20)
  ),
p04_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 21)
  ),
p05_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 22)
  ),
p06_ref
  ↦ the
  (
```

```
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 23)
  ),
p07_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 24)
  ),
p08_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 25)
  ),
p09_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 26)
  ),
p10_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 27)
  ),
p11_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 28)
  ),
p12_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 29)
  ),
p13_ref
  ↦ the
  (
```

```
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 30)
  ),
p14_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 31)
  ),
p15_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 32)
  ),
c00_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 33)
  ),
c01_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 34)
  ),
c02_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 35)
  ),
c03_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 36)
  ),
c04_ref
  ↦ the
  (
```

```
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 37)
  ),
c05_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 38)
  ),
c06_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 39)
  ),
c07_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 40)
  ),
c08_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 41)
  ),
c09_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 42)
  ),
c10_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 43)
  ),
c11_ref
  ↦ the
  (
```

```
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 44)
  ),
c12_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 45)
  ),
c13_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 46)
  ),
c14_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 47)
  ),
c15_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 48)
  ),
arg00_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 49)
  ),
arg01_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 50)
  ),
arg02_ref
  ↦ the
  (
```

```
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 51)
  ),
arg03_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 52)
  ),
arg04_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 53)
  ),
arg05_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 54)
  ),
arg06_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 55)
  ),
arg07_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 56)
  ),
arg08_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 57)
  ),
arg09_ref
  ↦ the
  (
```

```
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 58)
  ),
arg10_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 59)
  ),
arg11_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 60)
  ),
arg12_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 61)
  ),
arg13_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 62)
  ),
arg14_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 63)
  ),
arg15_ref
  ↦ the
  (
    call_memory
      state
      (the (register_state state call_frame_pointer_ref) - 64)
  ),

static_data_frame_pointer_ref
  ↦ the
```

```
            (
              call_memory
                state
                (the (register_state state call_frame_pointer_ref) - 65)
            ),

          static_data_stack_pointer_ref
            ↦ the
            (
              call_memory
                state
                (the (register_state state call_frame_pointer_ref) - 66)
            ),

          instruction_pointer_ref
            ↦ the
            (
              call_memory
                state
                (the (register_state state call_frame_pointer_ref) - 67)
            ) + 1
        )
    ⦈›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_simps)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) RETURN_decomp_mixed:
  assumes
    ‹typical_flags state›
  shows
    ‹RETURN state =
      state
      (⦇
        flag_state := flag_state (set_end_return state),
        register_state :=
          (register_state (restore_registers_after_return state))
          (
            last_instruction_pointer_ref
              ↦ read_register instruction_pointer_ref state,
            instruction_pointer_ref
              ↦ read_call_memory
                ((read_register call_frame_pointer_ref state) - 67)
                state
              + 1,
            cycles_register_ref
```

```
            ↦ (read_register cycles_register_ref state) + call_duration,
          call_frame_pointer_ref
            ↦ (read_register call_frame_pointer_ref state) - 67
        )
    ⟫›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_simps)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: fun_upd_twist)
  done

lemma (in Ironbark_world) RETURN_decomp_manipulation:
  assumes
    ‹typical_flags state›
  shows
    ‹RETURN state =
      write_register
        cycles_register_ref
        ((read_register cycles_register_ref state) + call_duration)
        (write_register
          instruction_pointer_ref
          (read_call_memory
            ((read_register call_frame_pointer_ref state) - 67)
            state
          + 1)
          (write_register
            call_frame_pointer_ref
            ((read_register call_frame_pointer_ref state) - 67)
            (restore_registers_after_return (write_register
              last_instruction_pointer_ref
              (read_register instruction_pointer_ref state)
              (set_end_return state)))))›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_simps)
  done

end
```

D.2.33 Decomposition Rules Over HALT

theory *HALT_decomposition*

imports
instruction_auxiliary

begin

We provide two lemmas which are different ways of decomposing the *HALT* instruction. The two

lemmas are the 'special' case where we assume the guards of the instruction will pass.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL ('state'). The second is a decomposition purely expressed in state manipulation operations.

Because *HALT* only operates on flags, the typical mixed decomposition is not needed.

```
lemma (in Ironbark_world) HALT_decomp_state:
  assumes
    ‹typical_flags state›
  shows
    ‹HALT state = state (|flag_state := (flag_state state)(|halt := 1|)|)›
  using assms
  apply(simp add: instruction_impl_defs)
  apply(simp add: state_manipulation_decomp)
  done

lemma (in Ironbark_world) HALT_decomp_manipulation:
  assumes
    ‹typical_flags state›
  shows
    ‹HALT state = set_halt state›
  using assms
  by(simp add: instruction_impl_defs)

end
```

D.2.34 Decomposition Rules Over ILLEGAL

```
theory ILLEGAL_decomposition

imports
  instruction_auxiliary

begin
```

We provide three lemmas which are different ways of decomposing the *ILLEGAL* instruction. The two instructions are bundled because they have identical functional operation.

The first lemma shows how it decomposes to state level operations in native Isabelle/HOL ('state'). The second and third are decompositions purely expressed in state manipulation operations. However, the second lemma uses the standard error operation, while the third just uses set flag operations.

As halt only operates on flags, the usual mixed decomposition is not relevant.

```
lemma (in Ironbark_world) ILLEGAL_decomp_state:
  ‹ILLEGAL state = state(|flag_state := (flag_state state)(|error := 1, halt := 1|)|)›
  by(simp add: instruction_impl_defs state_manipulation_decomp)

lemma (in Ironbark_world) ILLEGAL_decomp_manipulation:
  ‹ILLEGAL state = standard_error state›
  by(simp add: instruction_impl_defs)
```

lemma *(in Ironbark_world) ILLEGAL_decomp_manipulation2:*
  ‹*ILLEGAL state = set_halt (set_error state)*›
  by *(simp add: instruction_impl_defs state_manipulation_decomp)*

end

D.3 Instruction Simplification Rules

D.3.1 Simplification Rules Over ERROR0 and ERROR1

theory *ERROR_simps*

imports
  ERROR_decomposition

begin

As with most instructions, we provide various simplification rules for the error0 and error1 instructions. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instructions.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

lemma *(in Ironbark_world) flag_ERROR:*
  ‹*flag_state (ERROR0 state) = flag_state (set_halt (set_error state))*›

  ‹*flag_state (ERROR1 state) = flag_state (set_halt (set_error state))*›
  by *(simp_all add: ERROR_decomp_manipulation standard_error_def)*

lemma *(in Ironbark_world) register_ERROR:*
  ‹*register_state (ERROR0 state) = register_state state*›

  ‹*register_state (ERROR1 state) = register_state state*›
  by *(simp_all add: ERROR_decomp_manipulation state_manipulation_simps)*

lemma *(in Ironbark_world) memory_ERROR:*
  ‹*program_memory (ERROR0 state) = program_memory state*›
  ‹*call_memory (ERROR0 state) = call_memory state*›
  ‹*static_memory (ERROR0 state) = static_memory state*›
  ‹*dynamic_memory (ERROR0 state) = dynamic_memory state*›
  ‹*input_memory (ERROR0 state) = input_memory state*›
  ‹*output_memory (ERROR0 state) = output_memory state*›

  ‹*program_memory (ERROR1 state) = program_memory state*›
  ‹*call_memory (ERROR1 state) = call_memory state*›
  ‹*static_memory (ERROR1 state) = static_memory state*›
  ‹*dynamic_memory (ERROR1 state) = dynamic_memory state*›
  ‹*input_memory (ERROR1 state) = input_memory state*›
  ‹*output_memory (ERROR1 state) = output_memory state*›

```
  by(simp_all add: ERROR_decomp_state)

lemma (in Ironbark_world) read_flag_ERROR:
  ‹get_end_jump   (ERROR0 state) = get_end_jump state›
  ‹get_end_call   (ERROR0 state) = get_end_call state›
  ‹get_end_return (ERROR0 state) = get_end_return state›
  ‹get_halt       (ERROR0 state) = 1›
  ‹get_error      (ERROR0 state) = 1›

  ‹get_end_jump   (ERROR1 state) = get_end_jump state›
  ‹get_end_call   (ERROR1 state) = get_end_call state›
  ‹get_end_return (ERROR1 state) = get_end_return state›
  ‹get_halt       (ERROR1 state) = 1›
  ‹get_error      (ERROR1 state) = 1›
  by(auto simp add: ERROR_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) common_flags_ERROR:
  ‹¬ typical_flags    (ERROR0 state)›
  ‹¬ end_jump_flags   (ERROR0 state)›
  ‹¬ end_call_flags   (ERROR0 state)›
  ‹¬ end_return_flags (ERROR0 state)›
  ‹¬ halt_flags       (ERROR0 state)›

  ‹¬ typical_flags    (ERROR1 state)›
  ‹¬ end_jump_flags   (ERROR1 state)›
  ‹¬ end_call_flags   (ERROR1 state)›
  ‹¬ end_return_flags (ERROR1 state)›
  ‹¬ halt_flags       (ERROR1 state)›
  by(simp_all add: read_flag_ERROR)

lemma (in Ironbark_world) read_register_ERROR:
  ‹read_register regID (ERROR0 state) = read_register regID state›

  ‹read_register regID (ERROR1 state) = read_register regID state›
  by(simp_all add: read_register_def register_ERROR)

lemma (in Ironbark_world) read_memory_ERROR:
  ‹read_program_memory  address (ERROR0 state) = read_program_memory  address state›
  ‹read_call_memory     address (ERROR0 state) = read_call_memory     address state›
  ‹read_static_memory   address (ERROR0 state) = read_static_memory   address state›
  ‹read_dynamic_memory  address (ERROR0 state) = read_dynamic_memory  address state›

  ‹read_program_memory  address (ERROR1 state) = read_program_memory  address state›
  ‹read_call_memory     address (ERROR1 state) = read_call_memory     address state›
  ‹read_static_memory   address (ERROR1 state) = read_static_memory   address state›
  ‹read_dynamic_memory  address (ERROR1 state) = read_dynamic_memory  address state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)
```

The following are lemmas of ERROR on the initial state.

```
lemma (in Ironbark_world) read_flag_ERROR_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (ERROR0 start_state) = 0›
    ‹get_end_call   (ERROR0 start_state) = 0›
    ‹get_end_return (ERROR0 start_state) = 0›
    ‹get_halt       (ERROR0 start_state) = 1›
    ‹get_error      (ERROR0 start_state) = 1›

    ‹get_end_jump   (ERROR1 start_state) = 0›
    ‹get_end_call   (ERROR1 start_state) = 0›
    ‹get_end_return (ERROR1 start_state) = 0›
    ‹get_halt       (ERROR1 start_state) = 1›
    ‹get_error      (ERROR1 start_state) = 1›
  using assms
  by(auto simp add: read_flag_ERROR initial_state_simps)

lemma (in Ironbark_world) common_flags_ERROR_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹error_flags (ERROR0 start_state)›

    ‹error_flags (ERROR1 start_state)›
  using assms
  by(simp_all add: read_flag_ERROR_initial_state)

lemma (in Ironbark_world) read_register_ERROR_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register regID (ERROR0 start_state) = read_register regID start_state›

    ‹read_register regID (ERROR1 start_state) = read_register regID start_state›
  using assms
  by(simp_all add: read_register_ERROR)

lemma (in Ironbark_world) read_memory_ERROR_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
```

shows
— error0
‹*read_program_memory address (ERROR0 start_state)*
= read_program_memory address start_state›

‹*read_call_memory address (ERROR0 start_state)*
= read_call_memory address start_state›

‹*read_static_memory address (ERROR0 start_state)*
= read_static_memory address start_state›

‹*read_dynamic_memory address (ERROR0 start_state)*
= read_dynamic_memory address start_state›

— error1
‹*read_program_memory address (ERROR1 start_state)*
= read_program_memory address start_state›

‹*read_call_memory address (ERROR1 start_state)*
= read_call_memory address start_state›

‹*read_static_memory address (ERROR1 start_state)*
= read_static_memory address start_state›

‹*read_dynamic_memory address (ERROR1 start_state)*
= read_dynamic_memory address start_state›
using *assms*
by *(simp_all add: read_memory_ERROR)*

end

D.3.2 Simplification Rules Over NOP

theory *NOP_simps*

imports
NOP_decomposition

begin

As with most instructions, we provide various simplification rules for the *NOP* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

lemma *(***in** *Ironbark_world) flag_NOP:*
assumes
‹*typical_flags state*›

```
  shows
    ‹flag_state (NOP state) = flag_state state›
  using assms
  by(simp add: NOP_decomp_mixed)

lemma (in Ironbark_world) register_NOP:
  assumes
    ‹typical_flags state›
  shows
    ‹register_state (NOP state)
    = register_state (standard_post_instruction common_instruction_duration state)›
  using assms
  by(simp add: NOP_decomp_manipulation)

lemma (in Ironbark_world) memory_NOP:
  ‹program_memory (NOP state) = program_memory  state›
  ‹call_memory    (NOP state) = call_memory     state›
  ‹static_memory  (NOP state) = static_memory   state›
  ‹dynamic_memory (NOP state) = dynamic_memory  state›
  ‹input_memory   (NOP state) = input_memory    state›
  ‹output_memory  (NOP state) = output_memory   state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_NOP:
  assumes
    ‹typical_flags state›
  shows
    ‹get_end_jump   (NOP state) = 0›
    ‹get_end_call   (NOP state) = 0›
    ‹get_end_return (NOP state) = 0›
    ‹get_halt       (NOP state) = 0›
    ‹get_error      (NOP state) = 0›
  using assms
  by(simp_all add: read_flag_decomp flag_NOP)

lemma (in Ironbark_world) common_flags_NOP:
  assumes
    ‹typical_flags state›
  shows
    ‹typical_flags (NOP state)›
  using assms
  by(simp_all add: read_flag_NOP)

lemma (in Ironbark_world) read_register_NOP:
  assumes
    ‹reg1 ∉ {
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      cycles_register_ref
```

```
    }›
  shows
    ‹read_register reg1 (NOP state) = read_register reg1 state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_ip_NOP:
  assumes
    ‹typical_flags state›
  shows
    ‹read_register instruction_pointer_ref (NOP state)
      = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: NOP_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_NOP:
  assumes
    ‹typical_flags state›
  shows
    ‹read_register cycles_register_ref (NOP state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: NOP_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_NOP:
  assumes
    ‹typical_flags state›
  shows
    ‹read_register last_instruction_pointer_ref (NOP state)
      = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: NOP_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_memory_NOP:
  ‹read_program_memory  address (NOP state) = read_program_memory address state›
  ‹read_call_memory     address (NOP state) = read_call_memory    address state›
  ‹read_static_memory   address (NOP state) = read_static_memory  address state›
  ‹read_dynamic_memory  address (NOP state) = read_dynamic_memory address state›
  using memory_NOP
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_NOP_same:
  assumes
    ‹typical_flags state›
  shows
    ‹write_register reg1 value (NOP state)
    = write_register
      reg1
      value
```

```
    (standard_post_instruction common_instruction_duration state)›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_reorder state_manipulation_simps)
```

The following are lemmas of NOP on the initial state.

```
lemma (in Ironbark_world) read_flag_NOP_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (NOP start_state) = 0›
    ‹get_end_call   (NOP start_state) = 0›
    ‹get_end_return (NOP start_state) = 0›
    ‹get_halt       (NOP start_state) = 0›
    ‹get_error      (NOP start_state) = 0›
  using assms
  by(simp_all add: read_flag_NOP initial_state_simps)

lemma (in Ironbark_world) common_flags_NOP_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (NOP start_state)›
  using assms
  by(simp add: read_flag_NOP_initial_state)

lemma (in Ironbark_world) read_memory_NOP_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (NOP start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (NOP start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (NOP start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (NOP start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_NOP)
```

```
lemma (in Ironbark_world) read_last_ip_NOP_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref (NOP start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_NOP initial_state_simps)

lemma (in Ironbark_world) read_ip_NOP_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (NOP start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_NOP initial_state_simps)

end
```

D.3.3 Simplification Rules Over LOAD_IMMEDIATE

```
theory LOAD_IMMEDIATE_simps

imports
  instruction_auxiliary
  LOAD_IMMEDIATE_decomposition

begin
```

As with most instructions, we provide various simplification rules for the *LOAD_IMMEDIATE* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_LOAD_IMMEDIATE:
  assumes
    ‹typical_flags state›
    ‹check_write_permission regID›
  shows
    ‹flag_state (LOAD_IMMEDIATE regID immediate state) = flag_state state›
  using assms
  by(simp add: LOAD_IMMEDIATE_decomp_mixed)
```

```
lemma (in Ironbark_world) memory_LOAD_IMMEDIATE:
  shows
    ‹program_memory (LOAD_IMMEDIATE regID immediate state) = program_memory state›
    ‹call_memory    (LOAD_IMMEDIATE regID immediate state) = call_memory    state›
    ‹static_memory  (LOAD_IMMEDIATE regID immediate state) = static_memory  state›
    ‹dynamic_memory (LOAD_IMMEDIATE regID immediate state) = dynamic_memory state›
    ‹input_memory   (LOAD_IMMEDIATE regID immediate state) = input_memory   state›
    ‹output_memory  (LOAD_IMMEDIATE regID immediate state) = output_memory  state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_LOAD_IMMEDIATE:
  assumes
    ‹typical_flags state›
    ‹check_write_permission regID›
  shows
    ‹get_end_jump   (LOAD_IMMEDIATE regID immediate state) = 0›
    ‹get_end_call   (LOAD_IMMEDIATE regID immediate state) = 0›
    ‹get_end_return (LOAD_IMMEDIATE regID immediate state) = 0›
    ‹get_halt       (LOAD_IMMEDIATE regID immediate state) = 0›
    ‹get_error      (LOAD_IMMEDIATE regID immediate state) = 0›
  using assms
  by(simp_all add: read_flag_decomp flag_LOAD_IMMEDIATE)

lemma (in Ironbark_world) common_flags_LOAD_IMMEDIATE:
  assumes
    ‹typical_flags state›
    ‹check_write_permission regID›
  shows
    ‹typical_flags (LOAD_IMMEDIATE regID immediate state)›
  using assms
  by(simp_all add: read_flag_LOAD_IMMEDIATE)

lemma (in Ironbark_world) read_register_LOAD_IMMEDIATE_diff:
  assumes
    ‹regID1 ≠ regID2›
    ‹regID1 ∉ {
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      cycles_register_ref
    }›
  shows
    ‹read_register  regID1 (LOAD_IMMEDIATE regID2 immediate state)
    = read_register regID1 state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_register_LOAD_IMMEDIATE_same:
  assumes
    ‹typical_flags state›
```

```
    ‹check_write_permission regID›
  shows
    ‹read_register regID (LOAD_IMMEDIATE regID immediate state) = immediate›
  using assms
  by(simp add: LOAD_IMMEDIATE_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_ip_LOAD_IMMEDIATE:
  assumes
    ‹typical_flags state›
    ‹check_write_permission regID›
  shows
    ‹read_register instruction_pointer_ref (LOAD_IMMEDIATE regID immediate state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: LOAD_IMMEDIATE_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_LOAD_IMMEDIATE:
  assumes
    ‹typical_flags state›
    ‹check_write_permission regID›
  shows
    ‹read_register cycles_register_ref (LOAD_IMMEDIATE regID immediate state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: LOAD_IMMEDIATE_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_LOAD_IMMEDIATE:
  assumes
    ‹typical_flags state›
    ‹check_write_permission regID›
  shows
    ‹read_register last_instruction_pointer_ref (LOAD_IMMEDIATE regID immediate state)
      = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: LOAD_IMMEDIATE_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_memory_LOAD_IMMEDIATE:
  shows
    ‹read_program_memory  address (LOAD_IMMEDIATE regID immediate state)
    = read_program_memory address state›

    ‹read_call_memory     address (LOAD_IMMEDIATE regID immediate state)
    = read_call_memory    address state›

    ‹read_static_memory   address (LOAD_IMMEDIATE regID immediate state)
    = read_static_memory  address state›

    ‹read_dynamic_memory  address (LOAD_IMMEDIATE regID immediate state)
    = read_dynamic_memory address state›
```

```
  using memory_LOAD_IMMEDIATE
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_LOAD_IMMEDIATE_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission regID›
  shows
    ‹write_register regID value (LOAD_IMMEDIATE regID immediate state) =
      write_register
        regID
        value
        (standard_post_instruction common_instruction_duration state)›
  using assms
  by(simp add:
      instruction_impl_defs
      state_manipulation_reorder
      state_manipulation_simps)
```

The following are lemmas of LOAD_IMMEDIATE on the initial state.

```
lemma (in Ironbark_world) read_flag_LOAD_IMMEDIATE_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission regID›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (LOAD_IMMEDIATE regID immediate start_state) = 0›
    ‹get_end_call   (LOAD_IMMEDIATE regID immediate start_state) = 0›
    ‹get_end_return (LOAD_IMMEDIATE regID immediate start_state) = 0›
    ‹get_halt       (LOAD_IMMEDIATE regID immediate start_state) = 0›
    ‹get_error      (LOAD_IMMEDIATE regID immediate start_state) = 0›
  using assms
  by(simp_all add: read_flag_LOAD_IMMEDIATE initial_state_simps)

lemma (in Ironbark_world) common_flags_LOAD_IMMEDIATE_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission regID›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (LOAD_IMMEDIATE regID immediate start_state)›
  using assms
  by(simp add: read_flag_LOAD_IMMEDIATE_initial_state)

lemma (in Ironbark_world) read_memory_LOAD_IMMEDIATE_initial_state:
```

```
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission regID›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (LOAD_IMMEDIATE regID immediate start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (LOAD_IMMEDIATE regID immediate start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (LOAD_IMMEDIATE regID immediate start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (LOAD_IMMEDIATE regID immediate start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_LOAD_IMMEDIATE initial_state_simps)

lemma (in Ironbark_world) read_last_ip_LOAD_IMMEDIATE_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission regID›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref
      (LOAD_IMMEDIATE regID immediate start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_LOAD_IMMEDIATE initial_state_simps)

lemma (in Ironbark_world) read_ip_LOAD_IMMEDIATE_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission regID›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (LOAD_IMMEDIATE regID immediate start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_LOAD_IMMEDIATE initial_state_simps)

end
```

D.3.4 Simplification Rules Over LOAD_STATIC_DATA

theory *LOAD_STATIC_DATA_simps*

imports
LOAD_STATIC_DATA_decomposition

begin

As with most instructions, we provide various simplification rules for the load static data instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.
We order the rules based on the second operator, according to which would be applied last in the normalised form.

lemma (**in** *Ironbark_world*) *flag_LOAD_STATIC_DATA:*
assumes
‹*typical_flags state*›
‹*check_write_permission reg1*›
‹*check_read_permission reg2*›
shows
‹*flag_state (LOAD_STATIC_DATA reg1 reg2 state) = flag_state state*›
using *assms*
by(*simp add: LOAD_STATIC_DATA_decomp_mixed*)

lemma (**in** *Ironbark_world*) *memory_LOAD_STATIC_DATA:*
‹*program_memory (LOAD_STATIC_DATA reg1 reg2 state) = program_memory state*›
‹*call_memory (LOAD_STATIC_DATA reg1 reg2 state) = call_memory state*›
‹*static_memory (LOAD_STATIC_DATA reg1 reg2 state) = static_memory state*›
‹*dynamic_memory (LOAD_STATIC_DATA reg1 reg2 state) = dynamic_memory state*›
‹*input_memory (LOAD_STATIC_DATA reg1 reg2 state) = input_memory state*›
‹*output_memory (LOAD_STATIC_DATA reg1 reg2 state) = output_memory state*›
by(*simp_all add: instruction_impl_defs state_manipulation_simps*)

lemma (**in** *Ironbark_world*) *read_flag_LOAD_STATIC_DATA:*
assumes
‹*typical_flags state*›
‹*check_write_permission reg1*›
‹*check_read_permission reg2*›
shows
‹*get_end_jump (LOAD_STATIC_DATA reg1 reg2 state) = 0*›
‹*get_end_call (LOAD_STATIC_DATA reg1 reg2 state) = 0*›
‹*get_end_return (LOAD_STATIC_DATA reg1 reg2 state) = 0*›
‹*get_halt (LOAD_STATIC_DATA reg1 reg2 state) = 0*›
‹*get_error (LOAD_STATIC_DATA reg1 reg2 state) = 0*›
using *assms*
by(*simp_all add: read_flag_decomp flag_LOAD_STATIC_DATA*)

lemma (**in** *Ironbark_world*) *common_flags_LOAD_STATIC_DATA:*

```
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹typical_flags (LOAD_STATIC_DATA reg1 reg2 state)›
  using assms
  by(simp_all add: read_flag_LOAD_STATIC_DATA)

lemma (in Ironbark_world) read_register_LOAD_STATIC_DATA_diff:
  assumes
    ‹reg1 ≠ reg2›
    ‹reg1 ∉ {
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      cycles_register_ref
    }›
  shows
    ‹read_register reg1 (LOAD_STATIC_DATA reg2 reg3 state) = read_register reg1 state›
  using assms
  by(simp add: instruction_impl_defs  state_manipulation_simps)

lemma (in Ironbark_world) read_register_LOAD_STATIC_DATA_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register reg1 (LOAD_STATIC_DATA reg1 reg2 state)
    = read_static_memory (read_register reg2 state) state›
  using assms
  by(simp add: LOAD_STATIC_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_ip_LOAD_STATIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register instruction_pointer_ref (LOAD_STATIC_DATA reg1 reg2 state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: LOAD_STATIC_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_LOAD_STATIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
```

```
  shows
    ‹read_register cycles_register_ref (LOAD_STATIC_DATA reg1 reg2 state)
    = (read_register cycles_register_ref state) + memory_instruction_duration›
  using assms
  by(simp add: LOAD_STATIC_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_LOAD_STATIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register last_instruction_pointer_ref (LOAD_STATIC_DATA reg1 reg2 state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: LOAD_STATIC_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_memory_LOAD_STATIC_DATA:
  ‹read_program_memory  address (LOAD_STATIC_DATA reg1 reg2 state)
  = read_program_memory address state›

  ‹read_call_memory     address (LOAD_STATIC_DATA reg1 reg2 state)
  = read_call_memory    address state›

  ‹read_static_memory   address (LOAD_STATIC_DATA reg1 reg2 state)
  = read_static_memory  address state›

  ‹read_dynamic_memory  address (LOAD_STATIC_DATA reg1 reg2 state)
  = read_dynamic_memory address state›
  using memory_LOAD_STATIC_DATA
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_LOAD_STATIC_DATA_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹write_register reg1 value (LOAD_STATIC_DATA reg1 reg2 state)
      = write_register
          reg1
          value
          (standard_post_instruction
            memory_instruction_duration
            state)›
  using assms
  by(simp add:
      instruction_impl_defs
      state_manipulation_reorder
```

```
    state_manipulation_simps)
```

The following are lemmas of LOAD_STATIC_DATA on the initial state.

```
lemma (in Ironbark_world) read_flag_LOAD_STATIC_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (LOAD_STATIC_DATA reg1 reg2 start_state) = 0›
    ‹get_end_call   (LOAD_STATIC_DATA reg1 reg2 start_state) = 0›
    ‹get_end_return (LOAD_STATIC_DATA reg1 reg2 start_state) = 0›
    ‹get_halt       (LOAD_STATIC_DATA reg1 reg2 start_state) = 0›
    ‹get_error      (LOAD_STATIC_DATA reg1 reg2 start_state) = 0›
  using assms
  by(simp_all add: read_flag_LOAD_STATIC_DATA initial_state_simps)

lemma (in Ironbark_world) common_flags_LOAD_STATIC_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (LOAD_STATIC_DATA reg1 reg2 start_state)›
  using assms
  by(simp add: read_flag_LOAD_STATIC_DATA_initial_state)

lemma (in Ironbark_world) read_memory_LOAD_STATIC_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (LOAD_STATIC_DATA reg1 reg2 start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (LOAD_STATIC_DATA reg1 reg2 start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (LOAD_STATIC_DATA reg1 reg2 start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (LOAD_STATIC_DATA reg1 reg2 start_state)
```

```
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_LOAD_STATIC_DATA)

lemma (in Ironbark_world) read_last_ip_LOAD_STATIC_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref
      (LOAD_STATIC_DATA reg1 reg2 start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_LOAD_STATIC_DATA initial_state_simps)

lemma (in Ironbark_world) read_ip_LOAD_STATIC_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (LOAD_STATIC_DATA reg1 reg2 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_LOAD_STATIC_DATA initial_state_simps)

end
```

D.3.5 Simplification Rules Over STORE_STATIC_DATA

```
theory STORE_STATIC_DATA_simps

imports
  STORE_STATIC_DATA_decomposition

begin
```

As with most instructions, we provide various simplification rules for the store static data instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the

normalised form.

```
lemma (in Ironbark_world) flag_STORE_STATIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹flag_state (STORE_STATIC_DATA reg1 reg2 state) = flag_state state›
  using assms
  by(simp add: STORE_STATIC_DATA_decomp_mixed)

lemma (in Ironbark_world) memory_STORE_STATIC_DATA_same:
  ‹program_memory (STORE_STATIC_DATA reg1 reg2 state) = program_memory  state›
  ‹call_memory    (STORE_STATIC_DATA reg1 reg2 state) = call_memory     state›
  ‹dynamic_memory (STORE_STATIC_DATA reg1 reg2 state) = dynamic_memory  state›
  ‹input_memory   (STORE_STATIC_DATA reg1 reg2 state) = input_memory    state›
  ‹output_memory  (STORE_STATIC_DATA reg1 reg2 state) = output_memory   state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) memory_STORE_STATIC_DATA_diff:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹static_memory (STORE_STATIC_DATA reg1 reg2 state) =
      static_memory
        (write_static_memory
          (read_register reg1 state)
          (read_register reg2 state)
          state
        )›
  using assms
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_STORE_STATIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹get_end_jump   (STORE_STATIC_DATA reg1 reg2 state) = 0›
    ‹get_end_call   (STORE_STATIC_DATA reg1 reg2 state) = 0›
    ‹get_end_return (STORE_STATIC_DATA reg1 reg2 state) = 0›
    ‹get_halt       (STORE_STATIC_DATA reg1 reg2 state) = 0›
    ‹get_error      (STORE_STATIC_DATA reg1 reg2 state) = 0›
  using assms
  by(simp_all add: read_flag_decomp flag_STORE_STATIC_DATA)
```

```
lemma (in Ironbark_world) common_flags_STORE_STATIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹typical_flags (STORE_STATIC_DATA reg1 reg2 state)›
  using assms
  by(simp_all add: read_flag_STORE_STATIC_DATA)

lemma (in Ironbark_world) read_register_STORE_STATIC_DATA:
  assumes
    ‹regID ∉ {
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      cycles_register_ref
    }›
  shows
    ‹read_register    regID (STORE_STATIC_DATA reg1 reg2 state)
    = read_register regID state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_ip_STORE_STATIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register instruction_pointer_ref (STORE_STATIC_DATA reg1 reg2 state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: STORE_STATIC_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_STORE_STATIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register cycles_register_ref (STORE_STATIC_DATA reg1 reg2 state)
    = (read_register cycles_register_ref state) + memory_instruction_duration›
  using assms
  by(simp add: STORE_STATIC_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_STORE_STATIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
```

```
    ‹check_read_permission reg2›
  shows
    ‹read_register last_instruction_pointer_ref (STORE_STATIC_DATA reg1 reg2 state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: STORE_STATIC_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_memory_STORE_STATIC_DATA_same:
  ‹read_program_memory  address (STORE_STATIC_DATA reg1 reg2 state)
  = read_program_memory address state›

  ‹read_call_memory     address (STORE_STATIC_DATA reg1 reg2 state)
  = read_call_memory    address state›

  ‹read_dynamic_memory  address (STORE_STATIC_DATA reg1 reg2 state)
  = read_dynamic_memory address state›
  using memory_STORE_STATIC_DATA_same
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) read_memory_STORE_STATIC_DATA_diff:
  assumes
    ‹address ≠ read_register reg1 state›
  shows
    ‹read_dynamic_memory  address (STORE_STATIC_DATA reg1 reg2 state)
    = read_dynamic_memory address state›
proof (cases
    ‹(check_read_permission reg1)
    ∧ (check_read_permission reg2)
    ∧ (typical_flags state)›)
  case True
  then show ?thesis
    using assms
    by(simp add: STORE_STATIC_DATA_decomp_manipulation state_manipulation_simps)
next
  case False
  then have
    ‹STORE_STATIC_DATA reg1 reg2 state = standard_error state›
    by(simp add: instruction_impl_defs state_manipulation_simps, linarith)
  then show ?thesis
    by(simp add: state_manipulation_simps)
qed
```

The following are lemmas of STORE_STATIC_DATA on the initial state.

```
lemma (in Ironbark_world) read_flag_STORE_STATIC_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
```

```
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (STORE_STATIC_DATA reg1 reg2 start_state) = 0›
    ‹get_end_call   (STORE_STATIC_DATA reg1 reg2 start_state) = 0›
    ‹get_end_return (STORE_STATIC_DATA reg1 reg2 start_state) = 0›
    ‹get_halt       (STORE_STATIC_DATA reg1 reg2 start_state) = 0›
    ‹get_error      (STORE_STATIC_DATA reg1 reg2 start_state) = 0›
  using assms
  by(simp_all add: read_flag_STORE_STATIC_DATA initial_state_simps)

lemma (in Ironbark_world) common_flags_STORE_STATIC_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (STORE_STATIC_DATA reg1 reg2 start_state)›
  using assms
  by(simp add: read_flag_STORE_STATIC_DATA_initial_state)

lemma (in Ironbark_world) read_memory_STORE_STATIC_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (STORE_STATIC_DATA reg1 reg2 start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (STORE_STATIC_DATA reg1 reg2 start_state)
    = read_call_memory    address start_state›

    ‹read_dynamic_memory  address (STORE_STATIC_DATA reg1 reg2 start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_STORE_STATIC_DATA_same)

lemma (in Ironbark_world) read_last_ip_STORE_STATIC_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
```

```
  shows
    ‹read_register last_instruction_pointer_ref
      (STORE_STATIC_DATA reg1 reg2 start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_STORE_STATIC_DATA initial_state_simps)

lemma (in Ironbark_world) read_ip_STORE_STATIC_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (STORE_STATIC_DATA reg1 reg2 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_STORE_STATIC_DATA initial_state_simps)

end
```

D.3.6 Simplification Rules Over LOAD_DYNAMIC_DATA

```
theory LOAD_DYNAMIC_DATA_simps

imports
  LOAD_DYNAMIC_DATA_decomposition

begin
```

As with most instructions, we provide various simplification rules for the *LOAD_DYNAMIC_DATA* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_LOAD_DYNAMIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹flag_state (LOAD_DYNAMIC_DATA reg1 reg2 state) = flag_state state›
  using assms
  by(simp add: LOAD_DYNAMIC_DATA_decomp_mixed)

lemma (in Ironbark_world) memory_LOAD_DYNAMIC_DATA:
```

```
  ‹program_memory (LOAD_DYNAMIC_DATA reg1 reg2 state) = program_memory  state›
  ‹call_memory    (LOAD_DYNAMIC_DATA reg1 reg2 state) = call_memory     state›
  ‹static_memory  (LOAD_DYNAMIC_DATA reg1 reg2 state) = static_memory   state›
  ‹dynamic_memory (LOAD_DYNAMIC_DATA reg1 reg2 state) = dynamic_memory  state›
  ‹input_memory   (LOAD_DYNAMIC_DATA reg1 reg2 state) = input_memory    state›
  ‹output_memory  (LOAD_DYNAMIC_DATA reg1 reg2 state) = output_memory   state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_LOAD_DYNAMIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹get_end_jump   (LOAD_DYNAMIC_DATA reg1 reg2 state) = 0›
    ‹get_end_call   (LOAD_DYNAMIC_DATA reg1 reg2 state) = 0›
    ‹get_end_return (LOAD_DYNAMIC_DATA reg1 reg2 state) = 0›
    ‹get_halt       (LOAD_DYNAMIC_DATA reg1 reg2 state) = 0›
    ‹get_error      (LOAD_DYNAMIC_DATA reg1 reg2 state) = 0›
  using assms
  by(simp_all add: read_flag_decomp flag_LOAD_DYNAMIC_DATA)

lemma (in Ironbark_world) common_flags_LOAD_DYNAMIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹typical_flags (LOAD_DYNAMIC_DATA reg1 reg2 state)›
  using assms
  by(simp_all add: read_flag_LOAD_DYNAMIC_DATA)

lemma (in Ironbark_world) read_register_LOAD_DYNAMIC_DATA_diff:
  assumes
    ‹reg1 ≠ reg2›
    ‹reg1 ∉ {
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      cycles_register_ref
    }›
  shows
    ‹read_register reg1 (LOAD_DYNAMIC_DATA reg2 reg3 state) = read_register reg1 state›
  using assms
  by(simp add: instruction_impl_defs  state_manipulation_simps)

lemma (in Ironbark_world) read_register_LOAD_DYNAMIC_DATA_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
```

```
    ‹check_read_permission reg2›
  shows
    ‹read_register reg1 (LOAD_DYNAMIC_DATA reg1 reg2 state)
    = read_dynamic_memory (read_register reg2 state) state›
  using assms
  by(simp add: LOAD_DYNAMIC_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_ip_LOAD_DYNAMIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register instruction_pointer_ref (LOAD_DYNAMIC_DATA reg1 reg2 state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: LOAD_DYNAMIC_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_LOAD_DYNAMIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register cycles_register_ref (LOAD_DYNAMIC_DATA reg1 reg2 state)
    = (read_register cycles_register_ref state) + memory_instruction_duration›
  using assms
  by(simp add: LOAD_DYNAMIC_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_LOAD_DYNAMIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register last_instruction_pointer_ref (LOAD_DYNAMIC_DATA reg1 reg2 state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: LOAD_DYNAMIC_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_memory_LOAD_DYNAMIC_DATA:
  ‹read_program_memory  address (LOAD_DYNAMIC_DATA reg1 reg2 state)
  = read_program_memory address state›

  ‹read_call_memory     address (LOAD_DYNAMIC_DATA reg1 reg2 state)
  = read_call_memory    address state›

  ‹read_static_memory   address (LOAD_DYNAMIC_DATA reg1 reg2 state)
  = read_static_memory  address state›
```

```
  ‹read_dynamic_memory  address (LOAD_DYNAMIC_DATA reg1 reg2 state)
  = read_dynamic_memory address state›
  using memory_LOAD_DYNAMIC_DATA
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_LOAD_DYNAMIC_DATA_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹write_register reg1 value (LOAD_DYNAMIC_DATA reg1 reg2 state)
    = write_register
        reg1
        value
        (standard_post_instruction
          memory_instruction_duration
          state)›
  using assms
  by(simp add:
      instruction_impl_defs
      state_manipulation_reorder
      state_manipulation_simps)
```

The following are lemmas of LOAD_DYNAMIC_DATA on the initial state.

```
lemma (in Ironbark_world) read_flag_LOAD_DYNAMIC_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (LOAD_DYNAMIC_DATA reg1 reg2 start_state) = 0›
    ‹get_end_call   (LOAD_DYNAMIC_DATA reg1 reg2 start_state) = 0›
    ‹get_end_return (LOAD_DYNAMIC_DATA reg1 reg2 start_state) = 0›
    ‹get_halt       (LOAD_DYNAMIC_DATA reg1 reg2 start_state) = 0›
    ‹get_error      (LOAD_DYNAMIC_DATA reg1 reg2 start_state) = 0›
  using assms
  by(simp_all add: read_flag_LOAD_DYNAMIC_DATA initial_state_simps)

lemma (in Ironbark_world) common_flags_LOAD_DYNAMIC_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
```

defines
‹start_state ≡ initial_state(|program_memory := program|)›
shows
‹typical_flags (LOAD_DYNAMIC_DATA reg1 reg2 start_state)›
using *assms*
by *(simp add: read_flag_LOAD_DYNAMIC_DATA_initial_state)*

lemma *(in Ironbark_world) read_memory_LOAD_DYNAMIC_DATA_initial_state:*
fixes
program :: ‹(64 word, 96 word) map›
defines
‹start_state ≡ initial_state(|program_memory := program|)›
shows
‹read_program_memory address (LOAD_DYNAMIC_DATA reg1 reg2 start_state)
= read_program_memory address start_state›

‹read_call_memory address (LOAD_DYNAMIC_DATA reg1 reg2 start_state)
= read_call_memory address start_state›

‹read_static_memory address (LOAD_DYNAMIC_DATA reg1 reg2 start_state)
= read_static_memory address start_state›

‹read_dynamic_memory address (LOAD_DYNAMIC_DATA reg1 reg2 start_state)
= read_dynamic_memory address start_state›
using *assms*
by *(simp_all add: read_memory_LOAD_DYNAMIC_DATA)*

lemma *(in Ironbark_world) read_last_ip_LOAD_DYNAMIC_DATA_initial_state:*
fixes
program :: ‹(64 word, 96 word) map›
assumes
‹check_write_permission reg1›
‹check_read_permission reg2›
defines
‹start_state ≡ initial_state(|program_memory := program|)›
shows
‹read_register last_instruction_pointer_ref
(LOAD_DYNAMIC_DATA reg1 reg2 start_state)
= (read_register instruction_pointer_ref start_state)›
using *assms*
by *(simp add: read_last_ip_LOAD_DYNAMIC_DATA initial_state_simps)*

lemma *(in Ironbark_world) read_ip_LOAD_DYNAMIC_DATA_initial_state:*
fixes
program :: ‹(64 word, 96 word) map›
assumes
‹check_write_permission reg1›
‹check_read_permission reg2›
defines

```
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (LOAD_DYNAMIC_DATA reg1 reg2 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_LOAD_DYNAMIC_DATA initial_state_simps)

end
```

D.3.7 Simplification Rules Over STORE_DYNAMIC_DATA

theory *STORE_DYNAMIC_DATA_simps*

imports
STORE_DYNAMIC_DATA_decomposition

begin

As with most instructions, we provide various simplification rules for the *STORE_DYNAMIC_DATA* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_STORE_DYNAMIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹flag_state (STORE_DYNAMIC_DATA reg1 reg2 state) = flag_state state›
  using assms
  by(simp add: STORE_DYNAMIC_DATA_decomp_mixed)

lemma (in Ironbark_world) memory_STORE_DYNAMIC_DATA_same:
  ‹program_memory (STORE_DYNAMIC_DATA reg1 reg2 state) = program_memory state›
  ‹call_memory    (STORE_DYNAMIC_DATA reg1 reg2 state) = call_memory    state›
  ‹static_memory  (STORE_DYNAMIC_DATA reg1 reg2 state) = static_memory  state›
  ‹input_memory   (STORE_DYNAMIC_DATA reg1 reg2 state) = input_memory   state›
  ‹output_memory  (STORE_DYNAMIC_DATA reg1 reg2 state) = output_memory  state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) memory_STORE_DYNAMIC_DATA_diff:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹dynamic_memory (STORE_DYNAMIC_DATA reg1 reg2 state) =
```

```
    dynamic_memory
      (write_dynamic_memory
        (read_register reg1 state)
        (read_register reg2 state)
        state)›
  using assms
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_STORE_DYNAMIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹get_end_jump   (STORE_DYNAMIC_DATA reg1 reg2 state) = 0›
    ‹get_end_call   (STORE_DYNAMIC_DATA reg1 reg2 state) = 0›
    ‹get_end_return (STORE_DYNAMIC_DATA reg1 reg2 state) = 0›
    ‹get_halt       (STORE_DYNAMIC_DATA reg1 reg2 state) = 0›
    ‹get_error      (STORE_DYNAMIC_DATA reg1 reg2 state) = 0›
  using assms
  by(simp_all add: read_flag_decomp flag_STORE_DYNAMIC_DATA)

lemma (in Ironbark_world) common_flags_STORE_DYNAMIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹typical_flags (STORE_DYNAMIC_DATA reg1 reg2 state)›
  using assms
  by(simp_all add: read_flag_STORE_DYNAMIC_DATA)

lemma (in Ironbark_world) read_register_STORE_DYNAMIC_DATA:
  assumes
    ‹regID ∉ {
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      cycles_register_ref
    }›
  shows
    ‹read_register    regID (STORE_DYNAMIC_DATA reg1 reg2 state)
      = read_register regID state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_ip_STORE_DYNAMIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
```

```
    ‹check_read_permission reg2›
  shows
    ‹read_register instruction_pointer_ref (STORE_DYNAMIC_DATA reg1 reg2 state)
      = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: STORE_DYNAMIC_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_STORE_DYNAMIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register cycles_register_ref (STORE_DYNAMIC_DATA reg1 reg2 state)
    = (read_register cycles_register_ref state) + memory_instruction_duration›
  using assms
  by(simp add: STORE_DYNAMIC_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_STORE_DYNAMIC_DATA:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register last_instruction_pointer_ref (STORE_DYNAMIC_DATA reg1 reg2 state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: STORE_DYNAMIC_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_memory_STORE_DYNAMIC_DATA_same:
  ‹read_program_memory  address (STORE_DYNAMIC_DATA reg1 reg2 state)
  = read_program_memory address state›

  ‹read_call_memory     address (STORE_DYNAMIC_DATA reg1 reg2 state)
  = read_call_memory    address state›

  ‹read_static_memory   address (STORE_DYNAMIC_DATA reg1 reg2 state)
  = read_static_memory  address state›
  using memory_STORE_DYNAMIC_DATA_same
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) read_memory_STORE_DYNAMIC_DATA_diff:
  assumes
    ‹address ≠ read_register reg1 state›
  shows
    ‹read_dynamic_memory address (STORE_DYNAMIC_DATA reg1 reg2 state)
    = read_dynamic_memory address state›
proof (cases
    ‹(check_read_permission reg1)
```

```
    ∧ (check_read_permission reg2)
    ∧ (typical_flags state)›)
  case True
  then show ?thesis
    using assms
    by(simp add: STORE_DYNAMIC_DATA_decomp_manipulation state_manipulation_simps)
next
  case False
  then have
    ‹STORE_DYNAMIC_DATA reg1 reg2 state = standard_error state›
    by(simp add: instruction_impl_defs state_manipulation_simps, linarith)
  then show ?thesis
    by(simp add: state_manipulation_simps)
qed
```

The following are lemmas of STORE_DYNAMIC_DATA on the initial state.

```
lemma (in Ironbark_world) read_flag_STORE_DYNAMIC_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (STORE_DYNAMIC_DATA reg1 reg2 start_state) = 0›
    ‹get_end_call   (STORE_DYNAMIC_DATA reg1 reg2 start_state) = 0›
    ‹get_end_return (STORE_DYNAMIC_DATA reg1 reg2 start_state) = 0›
    ‹get_halt       (STORE_DYNAMIC_DATA reg1 reg2 start_state) = 0›
    ‹get_error      (STORE_DYNAMIC_DATA reg1 reg2 start_state) = 0›
  using assms
  by(simp_all add: read_flag_STORE_DYNAMIC_DATA initial_state_simps)

lemma (in Ironbark_world) common_flags_STORE_DYNAMIC_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (STORE_DYNAMIC_DATA reg1 reg2 start_state)›
  using assms
  by(simp add: read_flag_STORE_DYNAMIC_DATA_initial_state)

lemma (in Ironbark_world) read_memory_STORE_DYNAMIC_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
```

```
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (STORE_DYNAMIC_DATA reg1 reg2 start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (STORE_DYNAMIC_DATA reg1 reg2 start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (STORE_DYNAMIC_DATA reg1 reg2 start_state)
    = read_static_memory  address start_state›
  using assms
  by(simp_all add: read_memory_STORE_DYNAMIC_DATA_same)

lemma (in Ironbark_world) read_last_ip_STORE_DYNAMIC_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref
      (STORE_DYNAMIC_DATA reg1 reg2 start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_STORE_DYNAMIC_DATA initial_state_simps)

lemma (in Ironbark_world) read_ip_STORE_DYNAMIC_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (STORE_DYNAMIC_DATA reg1 reg2 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_STORE_DYNAMIC_DATA initial_state_simps)

end
```

D.3.8 Simplification Rules Over LOAD_INPUT_DATA

```
theory LOAD_INPUT_DATA_simps
```

imports
LOAD_INPUT_DATA_decomposition

begin

As with most instructions, we provide various simplification rules for the *LOAD_INPUT_DATA* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

lemma (**in** *Ironbark_world*) *flag_LOAD_INPUT_DATA:*
assumes
‹*typical_flags state*›
‹*check_write_permission reg1*›
‹*check_read_permission reg2*›
shows
‹*flag_state (LOAD_INPUT_DATA reg1 reg2 state) = flag_state state*›
using *assms*
by(*simp add: LOAD_INPUT_DATA_decomp_mixed*)

lemma (**in** *Ironbark_world*) *memory_LOAD_INPUT_DATA:*
‹*program_memory (LOAD_INPUT_DATA reg1 reg2 state) = program_memory state*›
‹*call_memory (LOAD_INPUT_DATA reg1 reg2 state) = call_memory state*›
‹*static_memory (LOAD_INPUT_DATA reg1 reg2 state) = static_memory state*›
‹*dynamic_memory (LOAD_INPUT_DATA reg1 reg2 state) = dynamic_memory state*›
‹*input_memory (LOAD_INPUT_DATA reg1 reg2 state) = input_memory state*›
‹*output_memory (LOAD_INPUT_DATA reg1 reg2 state) = output_memory state*›
by(*simp_all add: instruction_impl_defs state_manipulation_simps Let_def*)

lemma (**in** *Ironbark_world*) *read_flag_LOAD_INPUT_DATA:*
assumes
‹*typical_flags state*›
‹*check_write_permission reg1*›
‹*check_read_permission reg2*›
shows
‹*get_end_jump (LOAD_INPUT_DATA reg1 reg2 state) = 0*›
‹*get_end_call (LOAD_INPUT_DATA reg1 reg2 state) = 0*›
‹*get_end_return (LOAD_INPUT_DATA reg1 reg2 state) = 0*›
‹*get_halt (LOAD_INPUT_DATA reg1 reg2 state) = 0*›
‹*get_error (LOAD_INPUT_DATA reg1 reg2 state) = 0*›
using *assms*
by(*simp_all add: read_flag_decomp flag_LOAD_INPUT_DATA*)

lemma (**in** *Ironbark_world*) *common_flags_LOAD_INPUT_DATA:*
assumes
‹*typical_flags state*›
‹*check_write_permission reg1*›

```
    ‹check_read_permission reg2›
  shows
    ‹typical_flags (LOAD_INPUT_DATA reg1 reg2 state)›
  using assms
  by(simp_all add: read_flag_LOAD_INPUT_DATA)

lemma (in Ironbark_world) read_register_LOAD_INPUT_DATA_diff:
  assumes
    ‹reg1 ≠ reg2›
    ‹reg1 ∉ {
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      cycles_register_ref
    }›
  shows
    ‹read_register reg1 (LOAD_INPUT_DATA reg2 reg3 state) = read_register reg1 state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps Let_def)

lemma (in Ironbark_world) read_register_LOAD_INPUT_DATA_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register reg1 (LOAD_INPUT_DATA reg1 reg2 state)
    = read_input_memory (read_register reg2 state) state›
  using assms
  by(simp add: LOAD_INPUT_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_ip_LOAD_INPUT_DATA:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register instruction_pointer_ref (LOAD_INPUT_DATA reg1 reg2 state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: LOAD_INPUT_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_LOAD_INPUT_DATA:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register cycles_register_ref (LOAD_INPUT_DATA reg1 reg2 state)
    = (read_register cycles_register_ref state) + memory_instruction_duration›
```

```
  using assms
  by(simp add: LOAD_INPUT_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_LOAD_INPUT_DATA:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register last_instruction_pointer_ref (LOAD_INPUT_DATA reg1 reg2 state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: LOAD_INPUT_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_memory_LOAD_INPUT_DATA:
  ‹read_program_memory  address (LOAD_INPUT_DATA reg1 reg2 state)
  = read_program_memory address state›

  ‹read_call_memory     address (LOAD_INPUT_DATA reg1 reg2 state)
  = read_call_memory    address state›

  ‹read_static_memory   address (LOAD_INPUT_DATA reg1 reg2 state)
  = read_static_memory  address state›

  ‹read_dynamic_memory  address (LOAD_INPUT_DATA reg1 reg2 state)
  = read_dynamic_memory address state›
  using memory_LOAD_INPUT_DATA
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_LOAD_INPUT_DATA_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹write_register reg1 value (LOAD_INPUT_DATA reg1 reg2 state) =
      write_register
        reg1
        value
        (standard_post_instruction
          memory_instruction_duration
          state)›
  using assms
  by(simp add:
      instruction_impl_defs
      state_manipulation_reorder
      state_manipulation_simps
      Let_def)
```

The following are lemmas of LOAD_INPUT_DATA on the initial state.

```
lemma (in Ironbark_world) read_flag_LOAD_INPUT_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (LOAD_INPUT_DATA reg1 reg2 start_state) = 0›
    ‹get_end_call   (LOAD_INPUT_DATA reg1 reg2 start_state) = 0›
    ‹get_end_return (LOAD_INPUT_DATA reg1 reg2 start_state) = 0›
    ‹get_halt       (LOAD_INPUT_DATA reg1 reg2 start_state) = 0›
    ‹get_error      (LOAD_INPUT_DATA reg1 reg2 start_state) = 0›
  using assms
  by(simp_all add: read_flag_LOAD_INPUT_DATA initial_state_simps)

lemma (in Ironbark_world) common_flags_LOAD_INPUT_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (LOAD_INPUT_DATA reg1 reg2 start_state)›
  using assms
  by(simp add: read_flag_LOAD_INPUT_DATA_initial_state)

lemma (in Ironbark_world) read_memory_LOAD_INPUT_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (LOAD_INPUT_DATA reg1 reg2 start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (LOAD_INPUT_DATA reg1 reg2 start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (LOAD_INPUT_DATA reg1 reg2 start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (LOAD_INPUT_DATA reg1 reg2 start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_LOAD_INPUT_DATA)
```

lemma *(in Ironbark_world) read_last_ip_LOAD_INPUT_DATA_initial_state:*
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref (LOAD_INPUT_DATA reg1 reg2 start_state)
    = (read_register instruction_pointer_ref start_state)›
  using *assms*
  by *(simp add: read_last_ip_LOAD_INPUT_DATA initial_state_simps)*

lemma *(in Ironbark_world) read_ip_LOAD_INPUT_DATA_initial_state:*
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (LOAD_INPUT_DATA reg1 reg2 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using *assms*
  by *(simp add: read_ip_LOAD_INPUT_DATA initial_state_simps)*

end

D.3.9 Simplification Rules Over STORE_OUTPUT_DATA

theory *STORE_OUTPUT_DATA_simps*

imports
  STORE_OUTPUT_DATA_decomposition

begin

As with most instructions, we provide various simplification rules for the *STORE_OUTPUT_DATA* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

lemma *(in Ironbark_world) flag_STORE_OUTPUT_DATA:*
  assumes
    ‹typical_flags state›

```
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹flag_state (STORE_OUTPUT_DATA reg1 reg2 state) = flag_state state›
  using assms
  by(simp add: STORE_OUTPUT_DATA_decomp_mixed)

lemma (in Ironbark_world) memory_STORE_OUTPUT_DATA_same:
  ‹program_memory (STORE_OUTPUT_DATA reg1 reg2 state) = program_memory  state›
  ‹call_memory    (STORE_OUTPUT_DATA reg1 reg2 state) = call_memory     state›
  ‹static_memory  (STORE_OUTPUT_DATA reg1 reg2 state) = static_memory   state›
  ‹dynamic_memory (STORE_OUTPUT_DATA reg1 reg2 state) = dynamic_memory  state›
  ‹input_memory   (STORE_OUTPUT_DATA reg1 reg2 state) = input_memory    state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) memory_STORE_OUTPUT_DATA_diff:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹output_memory (STORE_OUTPUT_DATA reg1 reg2 state) =
      output_memory
        (write_output_memory
          (read_register reg1 state)
          (read_register reg2 state)
          state)›
  using assms
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_STORE_OUTPUT_DATA:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹get_end_jump   (STORE_OUTPUT_DATA reg1 reg2 state) = 0›
    ‹get_end_call   (STORE_OUTPUT_DATA reg1 reg2 state) = 0›
    ‹get_end_return (STORE_OUTPUT_DATA reg1 reg2 state) = 0›
    ‹get_halt       (STORE_OUTPUT_DATA reg1 reg2 state) = 0›
    ‹get_error      (STORE_OUTPUT_DATA reg1 reg2 state) = 0›
  using assms
  by(simp_all add: read_flag_decomp flag_STORE_OUTPUT_DATA)

lemma (in Ironbark_world) common_flags_STORE_OUTPUT_DATA:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
```

```
  shows
    ‹typical_flags (STORE_OUTPUT_DATA reg1 reg2 state)›
  using assms
  by(simp_all add: read_flag_STORE_OUTPUT_DATA)

lemma (in Ironbark_world) read_register_STORE_OUTPUT_DATA:
  assumes
    ‹regID ∉ {
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      cycles_register_ref
    }›
  shows
    ‹read_register  regID (STORE_OUTPUT_DATA reg1 reg2 state)
    = read_register regID state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_ip_STORE_OUTPUT_DATA:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register instruction_pointer_ref (STORE_OUTPUT_DATA reg1 reg2 state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: STORE_OUTPUT_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_STORE_OUTPUT_DATA:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register cycles_register_ref (STORE_OUTPUT_DATA reg1 reg2 state)
    = (read_register cycles_register_ref state) + memory_instruction_duration›
  using assms
  by(simp add: STORE_OUTPUT_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_STORE_OUTPUT_DATA:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register last_instruction_pointer_ref (STORE_OUTPUT_DATA reg1 reg2 state)
    = (read_register instruction_pointer_ref state)›
  using assms
```

```
  by(simp add: STORE_OUTPUT_DATA_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_memory_STORE_OUTPUT_DATA_same:
  ‹read_program_memory  address (STORE_OUTPUT_DATA reg1 reg2 state)
  = read_program_memory address state›

  ‹read_call_memory     address (STORE_OUTPUT_DATA reg1 reg2 state)
  = read_call_memory    address state›

  ‹read_static_memory   address (STORE_OUTPUT_DATA reg1 reg2 state)
  = read_static_memory  address state›

  ‹read_dynamic_memory  address (STORE_OUTPUT_DATA reg1 reg2 state)
  = read_dynamic_memory address state›
  using memory_STORE_OUTPUT_DATA_same
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) read_memory_STORE_OUTPUT_DATA_diff:
  assumes
    ‹address ≠ read_register reg1 state›
  shows
    ‹read_dynamic_memory  address (STORE_OUTPUT_DATA reg1 reg2 state)
    = read_dynamic_memory address state›
proof (cases
    ‹(check_read_permission reg1)
    ∧ (check_read_permission reg2)
    ∧ (typical_flags state)›)
  case True
  then show ?thesis
    using assms
    by(simp add: STORE_OUTPUT_DATA_decomp_manipulation state_manipulation_simps)
next
  case False
  then have
    ‹STORE_OUTPUT_DATA reg1 reg2 state = standard_error state›
    by(simp add: instruction_impl_defs state_manipulation_simps, linarith)
  then show ?thesis
    by(simp add: state_manipulation_simps)
qed
```

The following are lemmas of STORE_OUTPUT_DATA on the initial state.

```
lemma (in Ironbark_world) read_flag_STORE_OUTPUT_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
```

```
  shows
    ‹get_end_jump   (STORE_OUTPUT_DATA reg1 reg2 start_state) = 0›
    ‹get_end_call   (STORE_OUTPUT_DATA reg1 reg2 start_state) = 0›
    ‹get_end_return (STORE_OUTPUT_DATA reg1 reg2 start_state) = 0›
    ‹get_halt       (STORE_OUTPUT_DATA reg1 reg2 start_state) = 0›
    ‹get_error      (STORE_OUTPUT_DATA reg1 reg2 start_state) = 0›
  using assms
  by(simp_all add: read_flag_STORE_OUTPUT_DATA initial_state_simps)

lemma (in Ironbark_world) common_flags_STORE_OUTPUT_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (STORE_OUTPUT_DATA reg1 reg2 start_state)›
  using assms
  by(simp add: read_flag_STORE_OUTPUT_DATA_initial_state)

lemma (in Ironbark_world) read_memory_STORE_OUTPUT_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (STORE_OUTPUT_DATA reg1 reg2 start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (STORE_OUTPUT_DATA reg1 reg2 start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (STORE_OUTPUT_DATA reg1 reg2 start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (STORE_OUTPUT_DATA reg1 reg2 start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_STORE_OUTPUT_DATA_same)

lemma (in Ironbark_world) read_last_ip_STORE_OUTPUT_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  defines
```

```
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref
      (STORE_OUTPUT_DATA reg1 reg2 start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_STORE_OUTPUT_DATA initial_state_simps)

lemma (in Ironbark_world) read_ip_STORE_OUTPUT_DATA_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_read_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (STORE_OUTPUT_DATA reg1 reg2 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_STORE_OUTPUT_DATA initial_state_simps)

end
```

D.3.10 Simplification Rules Over COPY

```
theory COPY_simps

imports
  COPY_decomposition

begin
```

As with most instructions, we provide various simplification rules for the *COPY* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_COPY:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹flag_state (COPY reg1 reg2 state) = flag_state state›
  using assms
  by(simp add: COPY_decomp_mixed)
```

```
lemma (in Ironbark_world) memory_COPY:
  ‹program_memory (COPY reg1 reg2 state) = program_memory state›
  ‹call_memory    (COPY reg1 reg2 state) = call_memory    state›
  ‹static_memory  (COPY reg1 reg2 state) = static_memory  state›
  ‹dynamic_memory (COPY reg1 reg2 state) = dynamic_memory state›
  ‹input_memory   (COPY reg1 reg2 state) = input_memory   state›
  ‹output_memory  (COPY reg1 reg2 state) = output_memory  state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_COPY:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹get_end_jump   (COPY reg1 reg2 state) = 0›
    ‹get_end_call   (COPY reg1 reg2 state) = 0›
    ‹get_end_return (COPY reg1 reg2 state) = 0›
    ‹get_halt       (COPY reg1 reg2 state) = 0›
    ‹get_error      (COPY reg1 reg2 state) = 0›
  using assms
  by(simp_all add: read_flag_decomp flag_COPY)

lemma (in Ironbark_world) common_flags_COPY:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹typical_flags (COPY reg1 reg2 state)›
  using assms
  by(simp_all add: read_flag_COPY)

lemma (in Ironbark_world) read_register_COPY_diff:
  assumes
    ‹reg1 ≠ reg2›
    ‹reg1 ∉ {
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      cycles_register_ref
    }›
  shows
    ‹read_register reg1 (COPY reg2 reg3 state) = read_register reg1 state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_register_COPY_same:
  assumes
    ‹typical_flags state›
```

```
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register reg1 (COPY reg1 reg2 state) = read_register reg2 state›
  using assms
  by(simp add: COPY_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_ip_COPY:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register instruction_pointer_ref (COPY reg1 reg2 state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: COPY_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_COPY:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register cycles_register_ref (COPY reg1 reg2 state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: COPY_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_COPY:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register last_instruction_pointer_ref (COPY reg1 reg2 state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: COPY_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_memory_COPY:
  ‹read_program_memory  address (COPY reg1 reg2 state)
  = read_program_memory address state›

  ‹read_call_memory     address (COPY reg1 reg2 state)
  = read_call_memory    address state›

  ‹read_static_memory   address (COPY reg1 reg2 state)
  = read_static_memory  address state›
```

```
  ‹read_dynamic_memory  address (COPY reg1 reg2 state)
  = read_dynamic_memory address state›
  using memory_COPY
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_COPY_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹write_register reg1 value (COPY reg1 reg2 state)
    = write_register
      reg1
      value
      (standard_post_instruction common_instruction_duration state)›
  using assms
  by(simp add: COPY_decomp_manipulation state_manipulation_reorder state_manipulation_simps)
```

The following are lemmas of COPY on the initial state.

```
lemma (in Ironbark_world) read_flag_COPY_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (COPY reg1 reg2 start_state) = 0›
    ‹get_end_call   (COPY reg1 reg2 start_state) = 0›
    ‹get_end_return (COPY reg1 reg2 start_state) = 0›
    ‹get_halt       (COPY reg1 reg2 start_state) = 0›
    ‹get_error      (COPY reg1 reg2 start_state) = 0›
  using assms
  by(simp_all add: read_flag_COPY initial_state_simps)

lemma (in Ironbark_world) common_flags_COPY_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (COPY reg1 reg2 start_state)›
  using assms
```

```
  by(simp add: read_flag_COPY_initial_state)

lemma (in Ironbark_world) read_memory_COPY_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (COPY reg1 reg2 start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (COPY reg1 reg2 start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (COPY reg1 reg2 start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (COPY reg1 reg2 start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_COPY)

lemma (in Ironbark_world) read_last_ip_COPY_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref (COPY reg1 reg2 start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_COPY initial_state_simps)

lemma (in Ironbark_world) read_ip_COPY_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (COPY reg1 reg2 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_COPY initial_state_simps)
```

end

D.3.11 Simplification Rules Over ADD

theory *ADD_simps*

imports
ADD_decomposition

begin

As with most instructions, we provide various simplification rules for the *ADD* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_ADD:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹flag_state (ADD reg1 reg2 reg3 state) = flag_state state›
  using assms
  by(simp add: ADD_decomp_mixed)

lemma (in Ironbark_world) memory_ADD:
  ‹program_memory (ADD reg1 reg2 reg3 state) = program_memory state›
  ‹call_memory    (ADD reg1 reg2 reg3 state) = call_memory    state›
  ‹static_memory  (ADD reg1 reg2 reg3 state) = static_memory  state›
  ‹dynamic_memory (ADD reg1 reg2 reg3 state) = dynamic_memory state›
  ‹input_memory   (ADD reg1 reg2 reg3 state) = input_memory   state›
  ‹output_memory  (ADD reg1 reg2 reg3 state) = output_memory  state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_ADD:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹get_end_jump   (ADD reg1 reg2 reg3 state) = 0›
    ‹get_end_call   (ADD reg1 reg2 reg3 state) = 0›
    ‹get_end_return (ADD reg1 reg2 reg3 state) = 0›
    ‹get_halt       (ADD reg1 reg2 reg3 state) = 0›
```

```
    ‹get_error      (ADD reg1 reg2 reg3 state) = 0›
  using assms
  by(simp_all add: read_flag_decomp flag_ADD)

lemma (in Ironbark_world) common_flags_ADD:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹typical_flags (ADD reg1 reg2 reg3 state)›
  using assms
  by(simp_all add: read_flag_ADD)

lemma (in Ironbark_world) read_register_ADD_diff:
  assumes
    ‹regID ≠ reg1›
    ‹regID ∉ {
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      cycles_register_ref
    }›
    shows
    ‹read_register regID (ADD reg1 reg2 reg3 state) = read_register regID state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_register_ADD_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register reg1 (ADD reg1 reg2 reg3 state)
    = read_register reg2 state + read_register reg3 state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_ip_ADD:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register    instruction_pointer_ref (ADD reg1 reg2 reg3 state)
    = (read_register  instruction_pointer_ref state) + 1›
```

```
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_ADD:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register cycles_register_ref (ADD reg1 reg2 reg3 state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_ADD:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register    last_instruction_pointer_ref  (ADD reg1 reg2 reg3 state)
    = (read_register  instruction_pointer_ref       state)›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_memory_ADD:
  ‹read_program_memory  address (ADD reg1 reg2 reg3 state)
  = read_program_memory address state›
  ‹read_call_memory     address (ADD reg1 reg2 reg3 state)
  = read_call_memory    address state›
  ‹read_static_memory   address (ADD reg1 reg2 reg3 state)
  = read_static_memory  address state›
  ‹read_dynamic_memory  address (ADD reg1 reg2 reg3 state)
  = read_dynamic_memory address state›
  using memory_ADD
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_ADD_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹write_register reg1 value (ADD reg1 reg2 reg3 state)
    = write_register reg1 value
        (standard_post_instruction common_instruction_duration state)›
```

```
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps state_manipulation_reorder)
```

The following are cases of ADD over the initial state.

```
lemma (in Ironbark_world) read_flag_ADD_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (ADD reg1 reg2 reg3 start_state) = 0›
    ‹get_end_call   (ADD reg1 reg2 reg3 start_state) = 0›
    ‹get_end_return (ADD reg1 reg2 reg3 start_state) = 0›
    ‹get_halt       (ADD reg1 reg2 reg3 start_state) = 0›
    ‹get_error      (ADD reg1 reg2 reg3 start_state) = 0›
  using assms
  by(simp_all add: read_flag_ADD initial_state_simps)

lemma (in Ironbark_world) common_flags_ADD_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (ADD reg1 reg2 reg3 start_state)›
  using assms
  by(simp_all add: read_flag_ADD_initial_state)

lemma (in Ironbark_world) read_memory_ADD_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (ADD reg1 reg2 reg3 start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (ADD reg1 reg2 reg3 start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (ADD reg1 reg2 reg3 start_state)
```

```
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (ADD reg1 reg2 reg3 start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_ADD)

lemma (in Ironbark_world) read_last_ip_ADD_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register  last_instruction_pointer_ref  (ADD reg1 reg2 reg3 start_state)
    = read_register instruction_pointer_ref       start_state›
  using assms
  by(simp add: read_last_ip_ADD initial_state_simps)

lemma (in Ironbark_world) read_ip_ADD_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register    instruction_pointer_ref (ADD reg1 reg2 reg3 start_state)
    = (read_register  instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_ADD initial_state_simps)

end
```

D.3.12 Simplification Rules Over SUBTRACT

```
theory SUBTRACT_simps

imports
  SUBTRACT_decomposition

begin
```

As with most instructions, we provide various simplification rules for the *SUBTRACT* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value

from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_SUBTRACT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹flag_state (SUBTRACT reg1 reg2 reg3 state) = flag_state state›
  using assms
  by(simp add: SUBTRACT_decomp_mixed)

lemma (in Ironbark_world) memory_SUBTRACT:
  ‹program_memory (SUBTRACT reg1 reg2 reg3 state) = program_memory  state›
  ‹call_memory    (SUBTRACT reg1 reg2 reg3 state) = call_memory     state›
  ‹static_memory  (SUBTRACT reg1 reg2 reg3 state) = static_memory   state›
  ‹dynamic_memory (SUBTRACT reg1 reg2 reg3 state) = dynamic_memory  state›
  ‹input_memory   (SUBTRACT reg1 reg2 reg3 state) = input_memory    state›
  ‹output_memory  (SUBTRACT reg1 reg2 reg3 state) = output_memory   state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_SUBTRACT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹get_end_jump   (SUBTRACT reg1 reg2 reg3 state) = 0›
    ‹get_end_call   (SUBTRACT reg1 reg2 reg3 state) = 0›
    ‹get_end_return (SUBTRACT reg1 reg2 reg3 state) = 0›
    ‹get_halt       (SUBTRACT reg1 reg2 reg3 state) = 0›
    ‹get_error      (SUBTRACT reg1 reg2 reg3 state) = 0›
  using assms
  by(simp_all add: read_flag_decomp flag_SUBTRACT)

lemma (in Ironbark_world) common_flags_SUBTRACT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹typical_flags (SUBTRACT reg1 reg2 reg3 state)›
  using assms
```

```
  by(simp_all add: read_flag_SUBTRACT)

lemma (in Ironbark_world) read_register_SUBTRACT_diff:
  assumes
    ‹regID ≠ reg1›
    ‹regID ∉ {
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      cycles_register_ref
    }›
  shows
    ‹read_register regID (SUBTRACT reg1 reg2 reg3 state) = read_register regID state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_register_SUBTRACT_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register reg1 (SUBTRACT reg1 reg2 reg3 state)
    = read_register reg2 state - read_register reg3 state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_ip_SUBTRACT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register instruction_pointer_ref (SUBTRACT reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_SUBTRACT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register cycles_register_ref (SUBTRACT reg1 reg2 reg3 state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
```

```
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_SUBTRACT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register last_instruction_pointer_ref (SUBTRACT reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_memory_SUBTRACT:
  ‹read_program_memory  address (SUBTRACT reg1 reg2 reg3 state)
  = read_program_memory address state›

  ‹read_call_memory     address (SUBTRACT reg1 reg2 reg3 state)
  = read_call_memory    address state›

  ‹read_static_memory   address (SUBTRACT reg1 reg2 reg3 state)
  = read_static_memory  address state›

  ‹read_dynamic_memory  address (SUBTRACT reg1 reg2 reg3 state)
  = read_dynamic_memory address state›
  using memory_SUBTRACT
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_SUBTRACT_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹write_register reg1 value (SUBTRACT reg1 reg2 reg3 state) =
      write_register
        reg1
        value
        (standard_post_instruction common_instruction_duration state)›
  using assms
  by(simp add:
      instruction_impl_defs
      state_manipulation_reorder
      state_manipulation_simps)
```

The following are cases of *SUBTRACT* over the initial state.

```
lemma (in Ironbark_world) read_flag_SUBTRACT_initial_state:
```

```
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (SUBTRACT reg1 reg2 reg3 start_state) = 0›
    ‹get_end_call   (SUBTRACT reg1 reg2 reg3 start_state) = 0›
    ‹get_end_return (SUBTRACT reg1 reg2 reg3 start_state) = 0›
    ‹get_halt       (SUBTRACT reg1 reg2 reg3 start_state) = 0›
    ‹get_error      (SUBTRACT reg1 reg2 reg3 start_state) = 0›
  using assms
  by(simp_all add: read_flag_SUBTRACT initial_state_simps)

lemma (in Ironbark_world) common_flags_SUBTRACT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (SUBTRACT reg1 reg2 reg3 start_state)›
  using assms
  by(simp add: read_flag_SUBTRACT_initial_state)

lemma (in Ironbark_world) read_memory_SUBTRACT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (SUBTRACT reg1 reg2 reg3 start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (SUBTRACT reg1 reg2 reg3 start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (SUBTRACT reg1 reg2 reg3 start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (SUBTRACT reg1 reg2 reg3 start_state)
    = read_dynamic_memory address start_state›
  using assms
```

```
  by(simp_all add: read_memory_SUBTRACT)

lemma (in Ironbark_world) read_last_ip_SUBTRACT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref (SUBTRACT reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_SUBTRACT initial_state_simps)

lemma (in Ironbark_world) read_ip_SUBTRACT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (SUBTRACT reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_SUBTRACT initial_state_simps)

end
```

D.3.13 Simplification Rules Over SHIFT_LEFT

```
theory SHIFT_LEFT_simps

imports
  SHIFT_LEFT_decomposition

begin
```

As with most instructions, we provide various simplification rules for the *SHIFT_LEFT* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_SHIFT_LEFT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹flag_state (SHIFT_LEFT reg1 reg2 reg3 state) = flag_state state›
  using assms
  by(simp add: SHIFT_LEFT_decomp_mixed)

lemma (in Ironbark_world) memory_SHIFT_LEFT:
  ‹program_memory (SHIFT_LEFT reg1 reg2 reg3 state) = program_memory  state›
  ‹call_memory    (SHIFT_LEFT reg1 reg2 reg3 state) = call_memory     state›
  ‹static_memory  (SHIFT_LEFT reg1 reg2 reg3 state) = static_memory   state›
  ‹dynamic_memory (SHIFT_LEFT reg1 reg2 reg3 state) = dynamic_memory  state›
  ‹input_memory   (SHIFT_LEFT reg1 reg2 reg3 state) = input_memory    state›
  ‹output_memory  (SHIFT_LEFT reg1 reg2 reg3 state) = output_memory   state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_SHIFT_LEFT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹get_end_jump   (SHIFT_LEFT reg1 reg2 reg3 state) = 0›
    ‹get_end_call   (SHIFT_LEFT reg1 reg2 reg3 state) = 0›
    ‹get_end_return (SHIFT_LEFT reg1 reg2 reg3 state) = 0›
    ‹get_halt       (SHIFT_LEFT reg1 reg2 reg3 state) = 0›
    ‹get_error      (SHIFT_LEFT reg1 reg2 reg3 state) = 0›
  using assms
  by(simp_all add: read_flag_decomp flag_SHIFT_LEFT)

lemma (in Ironbark_world) common_flags_SHIFT_LEFT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹typical_flags (SHIFT_LEFT reg1 reg2 reg3 state)›
  using assms
  by(simp_all add: read_flag_SHIFT_LEFT)

lemma (in Ironbark_world) read_register_SHIFT_LEFT_diff:
  assumes
    ‹regID ≠ reg1›
```

```
    ‹regID ∉ {
                instruction_pointer_ref,
                last_instruction_pointer_ref,
                cycles_register_ref
              }›
  shows
    ‹read_register regID (SHIFT_LEFT reg1 reg2 reg3 state) = read_register regID state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_register_SHIFT_LEFT_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register reg1 (SHIFT_LEFT reg1 reg2 reg3 state)
    = push_bit (unat (read_register reg3 state)) (read_register reg2 state)›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_ip_SHIFT_LEFT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register instruction_pointer_ref (SHIFT_LEFT reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_SHIFT_LEFT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register cycles_register_ref (SHIFT_LEFT reg1 reg2 reg3 state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_SHIFT_LEFT:
  assumes
    ‹typical_flags state›
```

```
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register last_instruction_pointer_ref (SHIFT_LEFT reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_memory_SHIFT_LEFT:
  ‹read_program_memory  address (SHIFT_LEFT reg1 reg2 reg3 state)
  = read_program_memory address state›

  ‹read_call_memory     address (SHIFT_LEFT reg1 reg2 reg3 state)
  = read_call_memory    address state›

  ‹read_static_memory   address (SHIFT_LEFT reg1 reg2 reg3 state)
  = read_static_memory  address state›

  ‹read_dynamic_memory  address (SHIFT_LEFT reg1 reg2 reg3 state)
  = read_dynamic_memory address state›
  using memory_SHIFT_LEFT
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_SHIFT_LEFT_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹write_register reg1 value (SHIFT_LEFT reg1 reg2 reg3 state) =
      write_register
        reg1
        value
        (standard_post_instruction common_instruction_duration state)›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_reorder state_manipulation_simps)
```

The following are cases of SHIFT_LEFT over the initial state.

```
lemma (in Ironbark_world) read_flag_SHIFT_LEFT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
```

```
  shows
    ‹get_end_jump   (SHIFT_LEFT reg1 reg2 reg3 start_state) = 0›
    ‹get_end_call   (SHIFT_LEFT reg1 reg2 reg3 start_state) = 0›
    ‹get_end_return (SHIFT_LEFT reg1 reg2 reg3 start_state) = 0›
    ‹get_halt       (SHIFT_LEFT reg1 reg2 reg3 start_state) = 0›
    ‹get_error      (SHIFT_LEFT reg1 reg2 reg3 start_state) = 0›
  using assms
  by(simp_all add: read_flag_SHIFT_LEFT initial_state_simps)

lemma (in Ironbark_world) common_flags_SHIFT_LEFT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (SHIFT_LEFT reg1 reg2 reg3 start_state)›
  using assms
  by(simp add: read_flag_SHIFT_LEFT_initial_state)

lemma (in Ironbark_world) read_memory_SHIFT_LEFT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (SHIFT_LEFT reg1 reg2 reg3 start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (SHIFT_LEFT reg1 reg2 reg3 start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (SHIFT_LEFT reg1 reg2 reg3 start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (SHIFT_LEFT reg1 reg2 reg3 start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_SHIFT_LEFT)

lemma (in Ironbark_world) read_last_ip_SHIFT_LEFT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
```

```
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref (SHIFT_LEFT reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_SHIFT_LEFT initial_state_simps)

lemma (in Ironbark_world) read_ip_SHIFT_LEFT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (SHIFT_LEFT reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_SHIFT_LEFT initial_state_simps)

end
```

D.3.14 Simplification Rules Over SHIFT_RIGHT

```
theory SHIFT_RIGHT_simps

imports
  SHIFT_RIGHT_decomposition

begin
```

As with most instructions, we provide various simplification rules for the *SHIFT_RIGHT* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_SHIFT_RIGHT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹flag_state (SHIFT_RIGHT reg1 reg2 reg3 state) = flag_state state›
```

```
  using assms
  by(simp add: SHIFT_RIGHT_decomp_mixed)

lemma (in Ironbark_world) memory_SHIFT_RIGHT:
  ‹program_memory (SHIFT_RIGHT reg1 reg2 reg3 state) = program_memory state›
  ‹call_memory    (SHIFT_RIGHT reg1 reg2 reg3 state) = call_memory    state›
  ‹static_memory  (SHIFT_RIGHT reg1 reg2 reg3 state) = static_memory  state›
  ‹dynamic_memory (SHIFT_RIGHT reg1 reg2 reg3 state) = dynamic_memory state›
  ‹input_memory   (SHIFT_RIGHT reg1 reg2 reg3 state) = input_memory   state›
  ‹output_memory  (SHIFT_RIGHT reg1 reg2 reg3 state) = output_memory  state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_SHIFT_RIGHT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹get_end_jump   (SHIFT_RIGHT reg1 reg2 reg3 state) = 0›
    ‹get_end_call   (SHIFT_RIGHT reg1 reg2 reg3 state) = 0›
    ‹get_end_return (SHIFT_RIGHT reg1 reg2 reg3 state) = 0›
    ‹get_halt       (SHIFT_RIGHT reg1 reg2 reg3 state) = 0›
    ‹get_error      (SHIFT_RIGHT reg1 reg2 reg3 state) = 0›
  using assms
  by(simp_all add: read_flag_decomp flag_SHIFT_RIGHT)

lemma (in Ironbark_world) common_flags_SHIFT_RIGHT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹typical_flags (SHIFT_RIGHT reg1 reg2 reg3 state)›
  using assms
  by(simp_all add: read_flag_SHIFT_RIGHT)

lemma (in Ironbark_world) read_register_SHIFT_RIGHT_diff:
  assumes
    ‹regID ≠ reg1›
    ‹regID ∉ {
                instruction_pointer_ref,
                last_instruction_pointer_ref,
                cycles_register_ref
              }›
  shows
    ‹read_register regID (SHIFT_RIGHT reg1 reg2 reg3 state)
    = read_register regID state›
```

```
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_register_SHIFT_RIGHT_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register reg1 (SHIFT_RIGHT reg1 reg2 reg3 state)
    = drop_bit (unat (read_register reg3 state)) (read_register reg2 state)›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_ip_SHIFT_RIGHT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register instruction_pointer_ref (SHIFT_RIGHT reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_SHIFT_RIGHT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register cycles_register_ref (SHIFT_RIGHT reg1 reg2 reg3 state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_SHIFT_RIGHT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register last_instruction_pointer_ref (SHIFT_RIGHT reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state)›
  using assms
```

```
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_memory_SHIFT_RIGHT:
  ‹read_program_memory  address (SHIFT_RIGHT reg1 reg2 reg3 state)
  = read_program_memory address state›

  ‹read_call_memory     address (SHIFT_RIGHT reg1 reg2 reg3 state)
  = read_call_memory    address state›

  ‹read_static_memory   address (SHIFT_RIGHT reg1 reg2 reg3 state)
  = read_static_memory  address state›

  ‹read_dynamic_memory  address (SHIFT_RIGHT reg1 reg2 reg3 state)
  = read_dynamic_memory address state›
  using memory_SHIFT_RIGHT
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_SHIFT_RIGHT_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹write_register reg1 value (SHIFT_RIGHT reg1 reg2 reg3 state)
    = write_register reg1 value
      (standard_post_instruction common_instruction_duration state)›
  using assms
  by(simp add:
      instruction_impl_defs
      state_manipulation_reorder
      state_manipulation_simps)
```

The following are cases of SHIFT_RIGHT over the initial state.

```
lemma (in Ironbark_world) read_flag_SHIFT_RIGHT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (SHIFT_RIGHT reg1 reg2 reg3 start_state) = 0›
    ‹get_end_call   (SHIFT_RIGHT reg1 reg2 reg3 start_state) = 0›
    ‹get_end_return (SHIFT_RIGHT reg1 reg2 reg3 start_state) = 0›
    ‹get_halt       (SHIFT_RIGHT reg1 reg2 reg3 start_state) = 0›
    ‹get_error      (SHIFT_RIGHT reg1 reg2 reg3 start_state) = 0›
```

```
  using assms
  by(simp_all add: read_flag_SHIFT_RIGHT initial_state_simps)

lemma (in Ironbark_world) common_flags_SHIFT_RIGHT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (SHIFT_RIGHT reg1 reg2 reg3 start_state)›
  using assms
  by(simp add: read_flag_SHIFT_RIGHT_initial_state)

lemma (in Ironbark_world) read_memory_SHIFT_RIGHT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (SHIFT_RIGHT reg1 reg2 reg3 start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (SHIFT_RIGHT reg1 reg2 reg3 start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (SHIFT_RIGHT reg1 reg2 reg3 start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (SHIFT_RIGHT reg1 reg2 reg3 start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_SHIFT_RIGHT)

lemma (in Ironbark_world) read_last_ip_SHIFT_RIGHT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref
      (SHIFT_RIGHT reg1 reg2 reg3 start_state)
```

```
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_SHIFT_RIGHT initial_state_simps)

lemma (in Ironbark_world) read_ip_SHIFT_RIGHT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (SHIFT_RIGHT reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_SHIFT_RIGHT initial_state_simps)

end
```

D.3.15 Simplification Rules Over BITWISE_AND

```
theory BITWISE_AND_simps

imports
  BITWISE_AND_decomposition

begin
```

As with most instructions, we provide various simplification rules for the *BITWISE_AND* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_BITWISE_AND:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹flag_state (BITWISE_AND reg1 reg2 reg3 state) = flag_state state›
  using assms
  by(simp_all add: BITWISE_AND_decomp_mixed)

lemma (in Ironbark_world) memory_BITWISE_AND:
  shows
```

```
    ‹program_memory (BITWISE_AND reg1 reg2 reg3 state) = program_memory state›
    ‹call_memory    (BITWISE_AND reg1 reg2 reg3 state) = call_memory    state›
    ‹static_memory  (BITWISE_AND reg1 reg2 reg3 state) = static_memory  state›
    ‹dynamic_memory (BITWISE_AND reg1 reg2 reg3 state) = dynamic_memory state›
    ‹input_memory   (BITWISE_AND reg1 reg2 reg3 state) = input_memory   state›
    ‹output_memory  (BITWISE_AND reg1 reg2 reg3 state) = output_memory  state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_BITWISE_AND:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹get_end_jump   (BITWISE_AND reg1 reg2 reg3 state) = 0›
    ‹get_end_call   (BITWISE_AND reg1 reg2 reg3 state) = 0›
    ‹get_end_return (BITWISE_AND reg1 reg2 reg3 state) = 0›
    ‹get_halt       (BITWISE_AND reg1 reg2 reg3 state) = 0›
    ‹get_error      (BITWISE_AND reg1 reg2 reg3 state) = 0›
  using assms
  by(simp_all add: read_flag_decomp flag_BITWISE_AND)

lemma (in Ironbark_world) common_flags_BITWISE_AND:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹typical_flags (BITWISE_AND reg1 reg2 reg3 state)›
  using assms
  by(simp_all add: read_flag_BITWISE_AND)

lemma (in Ironbark_world) read_register_BITWISE_AND_diff:
  assumes
    ‹regID ≠ reg1›
    ‹regID ∉ {
                instruction_pointer_ref,
                last_instruction_pointer_ref,
                cycles_register_ref
              }›
  shows
    ‹read_register regID (BITWISE_AND reg1 reg2 reg3 state)
      = read_register regID state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_register_BITWISE_AND_same:
```

```
  includes
    bit_operations_syntax
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register reg1 (BITWISE_AND reg1 reg2 reg3 state)
    = (read_register reg2 state) AND (read_register reg3 state)›
  using assms
  by(simp add: BITWISE_AND_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_ip_BITWISE_AND:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register instruction_pointer_ref (BITWISE_AND reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_BITWISE_AND:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register cycles_register_ref (BITWISE_AND reg1 reg2 reg3 state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_BITWISE_AND:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register last_instruction_pointer_ref (BITWISE_AND reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)
```

```
lemma (in Ironbark_world) read_memory_BITWISE_AND:
  shows
    ‹read_program_memory  address (BITWISE_AND reg1 reg2 reg3 state)
    = read_program_memory address state›

    ‹read_call_memory     address (BITWISE_AND reg1 reg2 reg3 state)
    = read_call_memory    address state›

    ‹read_static_memory   address (BITWISE_AND reg1 reg2 reg3 state)
    = read_static_memory  address state›

    ‹read_dynamic_memory  address (BITWISE_AND reg1 reg2 reg3 state)
    = read_dynamic_memory address state›
  using memory_BITWISE_AND
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_BITWISE_AND_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹write_register reg1 value (BITWISE_AND reg1 reg2 reg3 state) =
      write_register
        reg1
        value
        (standard_post_instruction common_instruction_duration state)›
  using assms
  by(simp add:
      instruction_impl_defs
      state_manipulation_reorder
      state_manipulation_simps)
```

The following are cases of BITWISE_AND over the initial state.

```
lemma (in Ironbark_world) read_flag_BITWISE_AND_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (BITWISE_AND reg1 reg2 reg3 start_state) = 0›
    ‹get_end_call   (BITWISE_AND reg1 reg2 reg3 start_state) = 0›
    ‹get_end_return (BITWISE_AND reg1 reg2 reg3 start_state) = 0›
    ‹get_halt       (BITWISE_AND reg1 reg2 reg3 start_state) = 0›
```

```
    ‹get_error      (BITWISE_AND reg1 reg2 reg3 start_state) = 0›
  using assms
  by(simp_all add: read_flag_BITWISE_AND initial_state_simps)

lemma (in Ironbark_world) common_flags_BITWISE_AND_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (BITWISE_AND reg1 reg2 reg3 start_state)›
  using assms
  by(simp add: read_flag_BITWISE_AND_initial_state)

lemma (in Ironbark_world) read_memory_BITWISE_AND_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (BITWISE_AND reg1 reg2 reg3 start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (BITWISE_AND reg1 reg2 reg3 start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (BITWISE_AND reg1 reg2 reg3 start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (BITWISE_AND reg1 reg2 reg3 start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_BITWISE_AND)

lemma (in Ironbark_world) read_last_ip_BITWISE_AND_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref
```

```
      (BITWISE_AND reg1 reg2 reg3 start_state)
    = read_register instruction_pointer_ref start_state›
  using assms
  by(simp add: read_last_ip_BITWISE_AND initial_state_simps)

lemma (in Ironbark_world) read_ip_BITWISE_AND_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (BITWISE_AND reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_BITWISE_AND initial_state_simps)

end
```

D.3.16 Simplification Rules Over BITWISE_OR

```
theory BITWISE_OR_simps

imports
  BITWISE_OR_decomposition

begin
```

As with most instructions, we provide various simplification rules for the bitwise or instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_BITWISE_OR:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹flag_state (BITWISE_OR reg1 reg2 reg3 state) = flag_state state›
  using assms
  by(simp add: BITWISE_OR_decomp_mixed)

lemma (in Ironbark_world) memory_BITWISE_OR:
```

```
  shows
    ‹program_memory (BITWISE_OR reg1 reg2 reg3 state) = program_memory  state›
    ‹call_memory    (BITWISE_OR reg1 reg2 reg3 state) = call_memory     state›
    ‹static_memory  (BITWISE_OR reg1 reg2 reg3 state) = static_memory   state›
    ‹dynamic_memory (BITWISE_OR reg1 reg2 reg3 state) = dynamic_memory  state›
    ‹input_memory   (BITWISE_OR reg1 reg2 reg3 state) = input_memory    state›
    ‹output_memory  (BITWISE_OR reg1 reg2 reg3 state) = output_memory   state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_BITWISE_OR:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹get_end_jump   (BITWISE_OR reg1 reg2 reg3 state) = 0›
    ‹get_end_call   (BITWISE_OR reg1 reg2 reg3 state) = 0›
    ‹get_end_return (BITWISE_OR reg1 reg2 reg3 state) = 0›
    ‹get_halt       (BITWISE_OR reg1 reg2 reg3 state) = 0›
    ‹get_error      (BITWISE_OR reg1 reg2 reg3 state) = 0›
  using assms
  by(simp_all add: read_flag_decomp flag_BITWISE_OR)

lemma (in Ironbark_world) common_flags_BITWISE_OR:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹typical_flags (BITWISE_OR reg1 reg2 reg3 state)›
  using assms
  by(simp_all add: read_flag_BITWISE_OR)

lemma (in Ironbark_world) read_register_BITWISE_OR_diff:
  assumes
    ‹regID ≠ reg1›
    ‹regID ∉ {
                instruction_pointer_ref,
                last_instruction_pointer_ref,
                cycles_register_ref
              }›
  shows
    ‹read_register regID (BITWISE_OR reg1 reg2 reg3 state) = read_register regID state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_register_BITWISE_OR_same:
```

```
  includes
    bit_operations_syntax
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register reg1 (BITWISE_OR reg1 reg2 reg3 state)
    = read_register reg2 state OR read_register reg3 state›
  using assms
  by(simp add: BITWISE_OR_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_ip_BITWISE_OR:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register instruction_pointer_ref (BITWISE_OR reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_BITWISE_OR:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register cycles_register_ref (BITWISE_OR reg1 reg2 reg3 state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_BITWISE_OR:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register last_instruction_pointer_ref (BITWISE_OR reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)
```

```
lemma (in Ironbark_world) read_memory_BITWISE_OR:
  shows
    ‹read_program_memory  address (BITWISE_OR reg1 reg2 reg3 state)
    = read_program_memory address state›

    ‹read_call_memory     address (BITWISE_OR reg1 reg2 reg3 state)
    = read_call_memory    address state›

    ‹read_static_memory   address (BITWISE_OR reg1 reg2 reg3 state)
    = read_static_memory  address state›

    ‹read_dynamic_memory  address (BITWISE_OR reg1 reg2 reg3 state)
    = read_dynamic_memory address state›
  using memory_BITWISE_OR
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_BITWISE_OR_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹write_register reg1 value (BITWISE_OR reg1 reg2 reg3 state)
    = write_register reg1 value
      (standard_post_instruction common_instruction_duration state)›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_reorder state_manipulation_simps)
```

The following are cases of BITWISE_OR over the initial state.

```
lemma (in Ironbark_world) read_flag_BITWISE_OR_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (BITWISE_OR reg1 reg2 reg3 start_state) = 0›
    ‹get_end_call   (BITWISE_OR reg1 reg2 reg3 start_state) = 0›
    ‹get_end_return (BITWISE_OR reg1 reg2 reg3 start_state) = 0›
    ‹get_halt       (BITWISE_OR reg1 reg2 reg3 start_state) = 0›
    ‹get_error      (BITWISE_OR reg1 reg2 reg3 start_state) = 0›
  using assms
  by(simp_all add: read_flag_BITWISE_OR initial_state_simps)

lemma (in Ironbark_world) common_flags_BITWISE_OR_initial_state:
```

```
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (BITWISE_OR reg1 reg2 reg3 start_state)›
  using assms
  by(simp add: read_flag_BITWISE_OR_initial_state)

lemma (in Ironbark_world) read_memory_BITWISE_OR_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory    address (BITWISE_OR reg1 reg2 reg3 start_state)
    = read_program_memory address start_state›

    ‹read_call_memory       address (BITWISE_OR reg1 reg2 reg3 start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory     address (BITWISE_OR reg1 reg2 reg3 start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory    address (BITWISE_OR reg1 reg2 reg3 start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_BITWISE_OR)

lemma (in Ironbark_world) read_last_ip_BITWISE_OR_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref (BITWISE_OR reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_BITWISE_OR initial_state_simps)

lemma (in Ironbark_world) read_ip_BITWISE_OR_initial_state:
```

```
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (BITWISE_OR reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_BITWISE_OR initial_state_simps)

end
```

D.3.17 Simplification Rules Over BITWISE_XOR

```
theory BITWISE_XOR_simps

imports
  BITWISE_XOR_decomposition

begin
```

As with most instructions, we provide various simplification rules for the bitwise xor instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_BITWISE_XOR:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹flag_state (BITWISE_XOR reg1 reg2 reg3 state) = flag_state state›
  using assms
  by(simp add: BITWISE_XOR_decomp_mixed)

lemma (in Ironbark_world) memory_BITWISE_XOR:
  shows
    ‹program_memory (BITWISE_XOR reg1 reg2 reg3 state) = program_memory state›
    ‹call_memory    (BITWISE_XOR reg1 reg2 reg3 state) = call_memory    state›
    ‹static_memory  (BITWISE_XOR reg1 reg2 reg3 state) = static_memory  state›
    ‹dynamic_memory (BITWISE_XOR reg1 reg2 reg3 state) = dynamic_memory state›
    ‹input_memory   (BITWISE_XOR reg1 reg2 reg3 state) = input_memory   state›
```

```
    ‹output_memory  (BITWISE_XOR reg1 reg2 reg3 state) = output_memory  state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_BITWISE_XOR:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹get_end_jump   (BITWISE_XOR reg1 reg2 reg3 state) = 0›
    ‹get_end_call   (BITWISE_XOR reg1 reg2 reg3 state) = 0›
    ‹get_end_return (BITWISE_XOR reg1 reg2 reg3 state) = 0›
    ‹get_halt       (BITWISE_XOR reg1 reg2 reg3 state) = 0›
    ‹get_error      (BITWISE_XOR reg1 reg2 reg3 state) = 0›
  using assms
  by(simp_all add: read_flag_decomp flag_BITWISE_XOR)

lemma (in Ironbark_world) common_flags_BITWISE_XOR:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹typical_flags (BITWISE_XOR reg1 reg2 reg3 state)›
  using assms
  by(simp_all add: read_flag_BITWISE_XOR)

lemma (in Ironbark_world) read_register_BITWISE_XOR_diff:
  assumes
    ‹regID ≠ reg1›
    ‹regID ∉ {
                instruction_pointer_ref,
                last_instruction_pointer_ref,
                cycles_register_ref
              }›
  shows
    ‹read_register    regID (BITWISE_XOR reg1 reg2 reg3 state)
    = read_register regID state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_register_BITWISE_XOR_same:
  includes
    bit_operations_syntax
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
```

```
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register reg1 (BITWISE_XOR reg1 reg2 reg3 state)
    = read_register reg2 state XOR read_register reg3 state›
  using assms
  by(simp add: BITWISE_XOR_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_ip_BITWISE_XOR:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register instruction_pointer_ref (BITWISE_XOR reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_BITWISE_XOR:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register cycles_register_ref (BITWISE_XOR reg1 reg2 reg3 state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_BITWISE_XOR:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register last_instruction_pointer_ref (BITWISE_XOR reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_memory_BITWISE_XOR:
  shows
    ‹read_program_memory  address (BITWISE_XOR reg1 reg2 reg3 state)
    = read_program_memory address state›
```

```
    ‹read_call_memory     address (BITWISE_XOR reg1 reg2 reg3 state)
    = read_call_memory    address state›

    ‹read_static_memory   address (BITWISE_XOR reg1 reg2 reg3 state)
    = read_static_memory  address state›

    ‹read_dynamic_memory  address (BITWISE_XOR reg1 reg2 reg3 state)
    = read_dynamic_memory address state›
  using memory_BITWISE_XOR
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_BITWISE_XOR_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹write_register reg1 value (BITWISE_XOR reg1 reg2 reg3 state)
    = write_register reg1 value
      (standard_post_instruction common_instruction_duration state)›
  using assms
  by(simp add:
      instruction_impl_defs
      state_manipulation_reorder
      state_manipulation_simps)
```

The following are cases of BITWISE_XOR over the initial state.

```
lemma (in Ironbark_world) read_flag_BITWISE_XOR_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (BITWISE_XOR reg1 reg2 reg3 start_state) = 0›
    ‹get_end_call   (BITWISE_XOR reg1 reg2 reg3 start_state) = 0›
    ‹get_end_return (BITWISE_XOR reg1 reg2 reg3 start_state) = 0›
    ‹get_halt       (BITWISE_XOR reg1 reg2 reg3 start_state) = 0›
    ‹get_error      (BITWISE_XOR reg1 reg2 reg3 start_state) = 0›
  using assms
  by(simp_all add: read_flag_BITWISE_XOR initial_state_simps)

lemma (in Ironbark_world) common_flags_BITWISE_XOR_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
```

```
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (BITWISE_XOR reg1 reg2 reg3 start_state)›
  using assms
  by(simp add: read_flag_BITWISE_XOR_initial_state)

lemma (in Ironbark_world) read_memory_BITWISE_XOR_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (BITWISE_XOR reg1 reg2 reg3 start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (BITWISE_XOR reg1 reg2 reg3 start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (BITWISE_XOR reg1 reg2 reg3 start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (BITWISE_XOR reg1 reg2 reg3 start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_BITWISE_XOR)

lemma (in Ironbark_world) read_last_ip_BITWISE_XOR_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref
      (BITWISE_XOR reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_BITWISE_XOR initial_state_simps)

lemma (in Ironbark_world) read_ip_BITWISE_XOR_initial_state:
  fixes
```

```
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (BITWISE_XOR reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_BITWISE_XOR initial_state_simps)

end
```

D.3.18 Simplification Rules Over BITWISE_NAND

theory *BITWISE_NAND_simps*

imports
BITWISE_NAND_decomposition

begin

As with most instructions, we provide various simplification rules for the *BITWISE_NAND* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_BITWISE_NAND:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹flag_state (BITWISE_NAND reg1 reg2 reg3 state) = flag_state state›
  using assms
  by(simp_all add: BITWISE_NAND_decomp_mixed)

lemma (in Ironbark_world) memory_BITWISE_NAND:
  shows
    ‹program_memory (BITWISE_NAND reg1 reg2 reg3 state) = program_memory  state›
    ‹call_memory    (BITWISE_NAND reg1 reg2 reg3 state) = call_memory     state›
    ‹static_memory  (BITWISE_NAND reg1 reg2 reg3 state) = static_memory   state›
    ‹dynamic_memory (BITWISE_NAND reg1 reg2 reg3 state) = dynamic_memory  state›
    ‹input_memory   (BITWISE_NAND reg1 reg2 reg3 state) = input_memory    state›
    ‹output_memory  (BITWISE_NAND reg1 reg2 reg3 state) = output_memory   state›
```

```
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_BITWISE_NAND:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹get_end_jump   (BITWISE_NAND reg1 reg2 reg3 state) = 0›
    ‹get_end_call   (BITWISE_NAND reg1 reg2 reg3 state) = 0›
    ‹get_end_return (BITWISE_NAND reg1 reg2 reg3 state) = 0›
    ‹get_halt       (BITWISE_NAND reg1 reg2 reg3 state) = 0›
    ‹get_error      (BITWISE_NAND reg1 reg2 reg3 state) = 0›
  using assms
  by(simp_all add: read_flag_decomp flag_BITWISE_NAND)

lemma (in Ironbark_world) common_flags_BITWISE_NAND:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹typical_flags (BITWISE_NAND reg1 reg2 reg3 state)›
  using assms
  by(simp_all add: read_flag_BITWISE_NAND)

lemma (in Ironbark_world) read_register_BITWISE_NAND_diff:
  assumes
    ‹regID ≠ reg1›
    ‹regID ∉ {
                instruction_pointer_ref,
                last_instruction_pointer_ref,
                cycles_register_ref
              }›
  shows
    ‹read_register  regID (BITWISE_NAND reg1 reg2 reg3 state)
    = read_register regID state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_register_BITWISE_NAND_same:
  includes
    bit_operations_syntax
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
```

```
    ‹check_read_permission reg3›
  shows
    ‹read_register reg1 (BITWISE_NAND reg1 reg2 reg3 state)
      = NOT (read_register reg2 state AND read_register reg3 state)›
  using assms
  by(simp add: BITWISE_NAND_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_ip_BITWISE_NAND:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register instruction_pointer_ref (BITWISE_NAND reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_BITWISE_NAND:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register cycles_register_ref (BITWISE_NAND reg1 reg2 reg3 state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_BITWISE_NAND:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register last_instruction_pointer_ref (BITWISE_NAND reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_memory_BITWISE_NAND:
  shows
    ‹read_program_memory  address (BITWISE_NAND reg1 reg2 reg3 state)
    = read_program_memory address state›

    ‹read_call_memory     address (BITWISE_NAND reg1 reg2 reg3 state)
```

```
    = read_call_memory    address state›

    ‹read_static_memory   address (BITWISE_NAND reg1 reg2 reg3 state)
    = read_static_memory  address state›

    ‹read_dynamic_memory  address (BITWISE_NAND reg1 reg2 reg3 state)
    = read_dynamic_memory address state›
  using memory_BITWISE_NAND
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_BITWISE_NAND_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹write_register reg1 value (BITWISE_NAND reg1 reg2 reg3 state)
    = write_register
      reg1
      value
      (standard_post_instruction common_instruction_duration state)›
  using assms
  by(simp add:
      instruction_impl_defs
      state_manipulation_reorder
      state_manipulation_simps)
```

The following are cases of BITWISE_NAND over the initial state.

```
lemma (in Ironbark_world) read_flag_BITWISE_NAND_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (BITWISE_NAND reg1 reg2 reg3 start_state) = 0›
    ‹get_end_call   (BITWISE_NAND reg1 reg2 reg3 start_state) = 0›
    ‹get_end_return (BITWISE_NAND reg1 reg2 reg3 start_state) = 0›
    ‹get_halt       (BITWISE_NAND reg1 reg2 reg3 start_state) = 0›
    ‹get_error      (BITWISE_NAND reg1 reg2 reg3 start_state) = 0›
  using assms
  by(simp_all add: read_flag_BITWISE_NAND initial_state_simps)

lemma (in Ironbark_world) common_flags_BITWISE_NAND_initial_state:
  fixes
```

```
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (BITWISE_NAND reg1 reg2 reg3 start_state)›
  using assms
  by(simp add: read_flag_BITWISE_NAND_initial_state)

lemma (in Ironbark_world) read_memory_BITWISE_NAND_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (BITWISE_NAND reg1 reg2 reg3 start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (BITWISE_NAND reg1 reg2 reg3 start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (BITWISE_NAND reg1 reg2 reg3 start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (BITWISE_NAND reg1 reg2 reg3 start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_BITWISE_NAND)

lemma (in Ironbark_world) read_last_ip_BITWISE_NAND_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref
      (BITWISE_NAND reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_BITWISE_NAND initial_state_simps)

lemma (in Ironbark_world) read_ip_BITWISE_NAND_initial_state:
```

```
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (BITWISE_NAND reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_BITWISE_NAND initial_state_simps)

end
```

D.3.19 Simplification Rules Over BITWISE_NOT

```
theory BITWISE_NOT_simps

imports
  BITWISE_NOT_decomposition

begin
```

As with most instructions, we provide various simplification rules for the *BITWISE_NOT* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_BITWISE_NOT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹flag_state (BITWISE_NOT reg1 reg2 state) = flag_state state›
  using assms
  by(simp add: BITWISE_NOT_decomp_mixed)

lemma (in Ironbark_world) memory_BITWISE_NOT:
  shows
    ‹program_memory (BITWISE_NOT reg1 reg2 state) = program_memory  state›
    ‹call_memory    (BITWISE_NOT reg1 reg2 state) = call_memory     state›
    ‹static_memory  (BITWISE_NOT reg1 reg2 state) = static_memory   state›
    ‹dynamic_memory (BITWISE_NOT reg1 reg2 state) = dynamic_memory  state›
    ‹input_memory   (BITWISE_NOT reg1 reg2 state) = input_memory    state›
    ‹output_memory  (BITWISE_NOT reg1 reg2 state) = output_memory   state›
```

```
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_BITWISE_NOT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹get_end_jump   (BITWISE_NOT reg1 reg2 state) = 0›
    ‹get_end_call   (BITWISE_NOT reg1 reg2 state) = 0›
    ‹get_end_return (BITWISE_NOT reg1 reg2 state) = 0›
    ‹get_halt       (BITWISE_NOT reg1 reg2 state) = 0›
    ‹get_error      (BITWISE_NOT reg1 reg2 state) = 0›
  using assms
  by(simp_all add: read_flag_decomp flag_BITWISE_NOT)

lemma (in Ironbark_world) common_flags_BITWISE_NOT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹typical_flags (BITWISE_NOT reg1 reg2 state)›
  using assms
  by(simp_all add: read_flag_BITWISE_NOT)

lemma (in Ironbark_world) read_register_BITWISE_NOT_diff:
  assumes
    ‹reg1 ≠ reg2›
    ‹reg1 ∉ {
              instruction_pointer_ref,
              last_instruction_pointer_ref,
              cycles_register_ref
            }›
  shows
    ‹read_register reg1 (BITWISE_NOT reg2 reg3 state) = read_register reg1 state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_register_BITWISE_NOT_same:
  includes
    bit_operations_syntax
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register reg1 (BITWISE_NOT reg1 reg2 state) = NOT (read_register reg2 state)›
  using assms
```

```
  by(simp add: BITWISE_NOT_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_ip_BITWISE_NOT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register instruction_pointer_ref (BITWISE_NOT reg1 reg2 state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: BITWISE_NOT_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_BITWISE_NOT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register cycles_register_ref (BITWISE_NOT reg1 reg2 state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: BITWISE_NOT_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_BITWISE_NOT:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹read_register last_instruction_pointer_ref (BITWISE_NOT reg1 reg2 state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: BITWISE_NOT_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_memory_BITWISE_NOT:
  shows
    ‹read_program_memory  address (BITWISE_NOT reg1 reg2 state)
    = read_program_memory address state›

    ‹read_call_memory     address (BITWISE_NOT reg1 reg2 state)
    = read_call_memory    address state›

    ‹read_static_memory   address (BITWISE_NOT reg1 reg2 state)
    = read_static_memory  address state›

    ‹read_dynamic_memory  address (BITWISE_NOT reg1 reg2 state)
    = read_dynamic_memory address state›
  using memory_BITWISE_NOT
```

```
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_BITWISE_NOT_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  shows
    ‹write_register reg1 value (BITWISE_NOT reg1 reg2 state)
    = write_register reg1 value
      (standard_post_instruction common_instruction_duration state)›
  using assms
  by(simp add:
      instruction_impl_defs
      state_manipulation_reorder
      state_manipulation_simps)
```

The following are lemmas of BITWISE_NOT on the initial state.

```
lemma (in Ironbark_world) read_flag_BITWISE_NOT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (BITWISE_NOT reg1 reg2 start_state) = 0›
    ‹get_end_call   (BITWISE_NOT reg1 reg2 start_state) = 0›
    ‹get_end_return (BITWISE_NOT reg1 reg2 start_state) = 0›
    ‹get_halt       (BITWISE_NOT reg1 reg2 start_state) = 0›
    ‹get_error      (BITWISE_NOT reg1 reg2 start_state) = 0›
  using assms
  by(simp_all add: read_flag_BITWISE_NOT initial_state_simps)

lemma (in Ironbark_world) common_flags_BITWISE_NOT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (BITWISE_NOT reg1 reg2 start_state)›
  using assms
  by(simp add: read_flag_BITWISE_NOT_initial_state)

lemma (in Ironbark_world) read_memory_BITWISE_NOT_initial_state:
```

```
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (BITWISE_NOT reg1 reg2 start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (BITWISE_NOT reg1 reg2 start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (BITWISE_NOT reg1 reg2 start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (BITWISE_NOT reg1 reg2 start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_BITWISE_NOT)

lemma (in Ironbark_world) read_last_ip_BITWISE_NOT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref (BITWISE_NOT reg1 reg2 start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_BITWISE_NOT initial_state_simps)

lemma (in Ironbark_world) read_ip_BITWISE_NOT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (BITWISE_NOT reg1 reg2 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_BITWISE_NOT initial_state_simps)

end
```

D.3.20 Simplification Rules Over LESS_THAN

theory *LESS_THAN_simps*

imports
LESS_THAN_decomposition

begin

As with most instructions, we provide various simplification rules for the *LESS_THAN* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

lemma (**in** *Ironbark_world*) *flag_LESS_THAN:*
assumes
‹*typical_flags state*›
‹*check_write_permission reg1*›
‹*check_read_permission reg2*›
‹*check_read_permission reg3*›
shows
‹*flag_state (LESS_THAN reg1 reg2 reg3 state) = flag_state state*›
using *assms*
by(*simp add: LESS_THAN_decomp_mixed*)

lemma (**in** *Ironbark_world*) *memory_LESS_THAN:*
‹*program_memory (LESS_THAN reg1 reg2 reg3 state) = program_memory state*›
‹*call_memory (LESS_THAN reg1 reg2 reg3 state) = call_memory state*›
‹*static_memory (LESS_THAN reg1 reg2 reg3 state) = static_memory state*›
‹*dynamic_memory (LESS_THAN reg1 reg2 reg3 state) = dynamic_memory state*›
‹*input_memory (LESS_THAN reg1 reg2 reg3 state) = input_memory state*›
‹*output_memory (LESS_THAN reg1 reg2 reg3 state) = output_memory state*›
by(*simp_all add: instruction_impl_defs state_manipulation_simps*)

lemma (**in** *Ironbark_world*) *read_flag_LESS_THAN:*
assumes
‹*typical_flags state*›
‹*check_write_permission reg1*›
‹*check_read_permission reg2*›
‹*check_read_permission reg3*›
shows
‹*get_end_jump (LESS_THAN reg1 reg2 reg3 state) = 0*›
‹*get_end_call (LESS_THAN reg1 reg2 reg3 state) = 0*›
‹*get_end_return (LESS_THAN reg1 reg2 reg3 state) = 0*›
‹*get_halt (LESS_THAN reg1 reg2 reg3 state) = 0*›
‹*get_error (LESS_THAN reg1 reg2 reg3 state) = 0*›
using *assms*
by(*simp_all add: read_flag_decomp flag_LESS_THAN*)

```
lemma (in Ironbark_world) common_flags_LESS_THAN:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹typical_flags (LESS_THAN reg1 reg2 reg3 state)›
  using assms
  by(simp_all add: read_flag_LESS_THAN)

lemma (in Ironbark_world) read_register_LESS_THAN_diff:
  assumes
    ‹regID ≠ reg1›
    ‹regID ∉ {
                instruction_pointer_ref,
                last_instruction_pointer_ref,
                cycles_register_ref
              }›
  shows
    ‹read_register regID (LESS_THAN reg1 reg2 reg3 state) = read_register regID state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_register_LESS_THAN_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register reg1 (LESS_THAN reg1 reg2 reg3 state) =
    (
      if ((read_register reg2 state < read_register reg3 state)) then
        0x1
      else
        0x0
    )›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_ip_LESS_THAN:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
```

```
    ‹read_register instruction_pointer_ref (LESS_THAN reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_LESS_THAN:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register cycles_register_ref (LESS_THAN reg1 reg2 reg3 state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_LESS_THAN:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register last_instruction_pointer_ref (LESS_THAN reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_memory_LESS_THAN:
    ‹read_program_memory  address (LESS_THAN reg1 reg2 reg3 state)
    = read_program_memory address state›

    ‹read_call_memory     address (LESS_THAN reg1 reg2 reg3 state)
    = read_call_memory    address state›

    ‹read_static_memory   address (LESS_THAN reg1 reg2 reg3 state)
    = read_static_memory  address state›

    ‹read_dynamic_memory  address (LESS_THAN reg1 reg2 reg3 state)
    = read_dynamic_memory address state›
  using memory_LESS_THAN
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_LESS_THAN_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
```

```
    ‹check_read_permission reg3›
  shows
    ‹write_register reg1 value (LESS_THAN reg1 reg2 reg3 state)
    = write_register
      reg1
      value
      (standard_post_instruction common_instruction_duration state)›
  using assms
  by(simp add:
      instruction_impl_defs
      state_manipulation_reorder
      state_manipulation_simps)
```

The following are cases of LESS_THAN over the initial state.

```
lemma (in Ironbark_world) read_flag_LESS_THAN_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (LESS_THAN reg1 reg2 reg3 start_state) = 0›
    ‹get_end_call   (LESS_THAN reg1 reg2 reg3 start_state) = 0›
    ‹get_end_return (LESS_THAN reg1 reg2 reg3 start_state) = 0›
    ‹get_halt       (LESS_THAN reg1 reg2 reg3 start_state) = 0›
    ‹get_error      (LESS_THAN reg1 reg2 reg3 start_state) = 0›
  using assms
  by(simp_all add: read_flag_LESS_THAN initial_state_simps)

lemma (in Ironbark_world) common_flags_LESS_THAN_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (LESS_THAN reg1 reg2 reg3 start_state)›
  using assms
  by(simp add: read_flag_LESS_THAN_initial_state)

lemma (in Ironbark_world) read_memory_LESS_THAN_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
```

```
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (LESS_THAN reg1 reg2 reg3 start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (LESS_THAN reg1 reg2 reg3 start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (LESS_THAN reg1 reg2 reg3 start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (LESS_THAN reg1 reg2 reg3 start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_LESS_THAN)

lemma (in Ironbark_world) read_last_ip_LESS_THAN_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref (LESS_THAN reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_LESS_THAN initial_state_simps)

lemma (in Ironbark_world) read_ip_LESS_THAN_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (LESS_THAN reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_LESS_THAN initial_state_simps)

end
```

D.3.21 Simplification Rules Over GREATER_THAN

theory *GREATER_THAN_simps*

imports
GREATER_THAN_decomposition

begin

As with most instructions, we provide various simplification rules for the *GREATER_THAN* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_GREATER_THAN:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹flag_state (GREATER_THAN reg1 reg2 reg3 state) = flag_state state›
  using assms
  by(simp add: GREATER_THAN_decomp_mixed)

lemma (in Ironbark_world) memory_GREATER_THAN:
  ‹program_memory (GREATER_THAN reg1 reg2 reg3 state) = program_memory  state›
  ‹call_memory    (GREATER_THAN reg1 reg2 reg3 state) = call_memory     state›
  ‹static_memory  (GREATER_THAN reg1 reg2 reg3 state) = static_memory   state›
  ‹dynamic_memory (GREATER_THAN reg1 reg2 reg3 state) = dynamic_memory  state›
  ‹input_memory   (GREATER_THAN reg1 reg2 reg3 state) = input_memory    state›
  ‹output_memory  (GREATER_THAN reg1 reg2 reg3 state) = output_memory   state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_GREATER_THAN:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹get_end_jump   (GREATER_THAN reg1 reg2 reg3 state) = 0›
    ‹get_end_call   (GREATER_THAN reg1 reg2 reg3 state) = 0›
    ‹get_end_return (GREATER_THAN reg1 reg2 reg3 state) = 0›
    ‹get_halt       (GREATER_THAN reg1 reg2 reg3 state) = 0›
    ‹get_error      (GREATER_THAN reg1 reg2 reg3 state) = 0›
  using assms
  by(simp_all add: read_flag_decomp flag_GREATER_THAN)
```

```
lemma (in Ironbark_world) common_flags_GREATER_THAN:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹typical_flags (GREATER_THAN reg1 reg2 reg3 state)›
  using assms
  by(simp_all add: read_flag_GREATER_THAN)

lemma (in Ironbark_world) read_register_GREATER_THAN_diff:
  assumes
    ‹regID ≠ reg1›
    ‹regID ∉ {
                instruction_pointer_ref,
                last_instruction_pointer_ref,
                cycles_register_ref
              }›
  shows
    ‹read_register  regID (GREATER_THAN reg1 reg2 reg3 state)
    = read_register regID state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_register_GREATER_THAN_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register reg1 (GREATER_THAN reg1 reg2 reg3 state) =
    (
      if ((read_register reg2 state > read_register reg3 state)) then
        0x1
      else
        0x0
    )›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_ip_GREATER_THAN:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
```

```
  shows
    ‹read_register instruction_pointer_ref (GREATER_THAN reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_GREATER_THAN:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register cycles_register_ref (GREATER_THAN reg1 reg2 reg3 state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_GREATER_THAN:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register last_instruction_pointer_ref (GREATER_THAN reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_memory_GREATER_THAN:
  shows
    ‹read_program_memory  address (GREATER_THAN reg1 reg2 reg3 state)
    = read_program_memory address state›

    ‹read_call_memory     address (GREATER_THAN reg1 reg2 reg3 state)
    = read_call_memory    address state›

    ‹read_static_memory   address (GREATER_THAN reg1 reg2 reg3 state)
    = read_static_memory  address state›

    ‹read_dynamic_memory  address (GREATER_THAN reg1 reg2 reg3 state)
    = read_dynamic_memory address state›
  using memory_GREATER_THAN
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_GREATER_THAN_same:
  assumes
    ‹typical_flags state›
```

```
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹write_register reg1 value (GREATER_THAN reg1 reg2 reg3 state)
    = write_register reg1 value
      (standard_post_instruction common_instruction_duration state)›
  using assms
  by(simp add:
      instruction_impl_defs
      state_manipulation_reorder
      state_manipulation_simps)
```

The following are cases of GREATER_THAN over the initial state.

```
lemma (in Ironbark_world) read_flag_GREATER_THAN_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (GREATER_THAN reg1 reg2 reg3 start_state) = 0›
    ‹get_end_call   (GREATER_THAN reg1 reg2 reg3 start_state) = 0›
    ‹get_end_return (GREATER_THAN reg1 reg2 reg3 start_state) = 0›
    ‹get_halt       (GREATER_THAN reg1 reg2 reg3 start_state) = 0›
    ‹get_error      (GREATER_THAN reg1 reg2 reg3 start_state) = 0›
  using assms
  by(simp_all add: read_flag_GREATER_THAN initial_state_simps)

lemma (in Ironbark_world) common_flags_GREATER_THAN_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (GREATER_THAN reg1 reg2 reg3 start_state)›
  using assms
  by(simp add: read_flag_GREATER_THAN_initial_state)

lemma (in Ironbark_world) read_memory_GREATER_THAN_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
```

```
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (GREATER_THAN reg1 reg2 reg3 start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (GREATER_THAN reg1 reg2 reg3 start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (GREATER_THAN reg1 reg2 reg3 start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (GREATER_THAN reg1 reg2 reg3 start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_GREATER_THAN)

lemma (in Ironbark_world) read_last_ip_GREATER_THAN_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref
      (GREATER_THAN reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_GREATER_THAN initial_state_simps)

lemma (in Ironbark_world) read_ip_GREATER_THAN_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (GREATER_THAN reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_GREATER_THAN initial_state_simps)

end
```

D.3.22 Simplification Rules Over EQUALS

theory *EQUALS_simps*

imports
EQUALS_decomposition

begin

As with most instructions, we provide various simplification rules for the ***EQUALS*** instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_EQUALS:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹flag_state (EQUALS reg1 reg2 reg3 state) = flag_state state›
  using assms
  by(simp add: EQUALS_decomp_mixed)

lemma (in Ironbark_world) memory_EQUALS:
  ‹program_memory (EQUALS reg1 reg2 reg3 state) = program_memory  state›
  ‹call_memory    (EQUALS reg1 reg2 reg3 state) = call_memory     state›
  ‹static_memory  (EQUALS reg1 reg2 reg3 state) = static_memory   state›
  ‹dynamic_memory (EQUALS reg1 reg2 reg3 state) = dynamic_memory  state›
  ‹input_memory   (EQUALS reg1 reg2 reg3 state) = input_memory    state›
  ‹output_memory  (EQUALS reg1 reg2 reg3 state) = output_memory   state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_EQUALS:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹get_end_jump   (EQUALS reg1 reg2 reg3 state) = 0›
    ‹get_end_call   (EQUALS reg1 reg2 reg3 state) = 0›
    ‹get_end_return (EQUALS reg1 reg2 reg3 state) = 0›
    ‹get_halt       (EQUALS reg1 reg2 reg3 state) = 0›
    ‹get_error      (EQUALS reg1 reg2 reg3 state) = 0›
  using assms
  by(simp_all add: read_flag_decomp flag_EQUALS)
```

```
lemma (in Ironbark_world) common_flags_EQUALS:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹typical_flags (EQUALS reg1 reg2 reg3 state)›
  using assms
  by(simp_all add: read_flag_EQUALS)

lemma (in Ironbark_world) read_register_EQUALS_diff:
  assumes
    ‹reg1 ≠ reg2›
    ‹reg1 ∉ {
              instruction_pointer_ref,
              last_instruction_pointer_ref,
              cycles_register_ref
            }›
  shows
    ‹read_register reg1 (EQUALS reg2 reg3 reg4 state) = read_register reg1 state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_register_EQUALS_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register reg1 (EQUALS reg1 reg2 reg3 state) =
    (
      if (read_register reg2 state = read_register reg3 state) then
        0x1
      else
        0x0
    )›
  using assms
  by(simp add: EQUALS_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_ip_EQUALS:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
```

```
    ‹read_register instruction_pointer_ref (EQUALS reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: EQUALS_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_EQUALS:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register cycles_register_ref (EQUALS reg1 reg2 reg3 state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: EQUALS_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_EQUALS:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register last_instruction_pointer_ref (EQUALS reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: EQUALS_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_memory_EQUALS:
  shows
    ‹read_program_memory  address (EQUALS reg1 reg2 reg3 state)
    = read_program_memory address state›
    ‹read_call_memory     address (EQUALS reg1 reg2 reg3 state)
    = read_call_memory    address state›
    ‹read_static_memory   address (EQUALS reg1 reg2 reg3 state)
    = read_static_memory  address state›
    ‹read_dynamic_memory  address (EQUALS reg1 reg2 reg3 state)
    = read_dynamic_memory address state›
  using memory_EQUALS
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_EQUALS_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
```

```
    ‹write_register reg1 value (EQUALS reg1 reg2 reg3 state)
    = write_register reg1 value
      (standard_post_instruction common_instruction_duration state)›
  using assms
  by(simp add:
      instruction_impl_defs
      state_manipulation_reorder
      state_manipulation_simps)
```

The following are lemmas of EQUALS on the initial state.

```
lemma (in Ironbark_world) read_flag_EQUALS_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (EQUALS reg1 reg2 reg3 start_state) = 0›
    ‹get_end_call   (EQUALS reg1 reg2 reg3 start_state) = 0›
    ‹get_end_return (EQUALS reg1 reg2 reg3 start_state) = 0›
    ‹get_halt       (EQUALS reg1 reg2 reg3 start_state) = 0›
    ‹get_error      (EQUALS reg1 reg2 reg3 start_state) = 0›
  using assms
  by(simp_all add: read_flag_EQUALS initial_state_simps)

lemma (in Ironbark_world) common_flags_EQUALS_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (EQUALS reg1 reg2 reg3 start_state)›
  using assms
  by(simp add: read_flag_EQUALS_initial_state)

lemma (in Ironbark_world) read_memory_EQUALS_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (EQUALS reg1 reg2 reg3 start_state)
```

```
    = read_program_memory address start_state›

    ‹read_call_memory     address (EQUALS reg1 reg2 reg3 start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (EQUALS reg1 reg2 reg3 start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (EQUALS reg1 reg2 reg3 start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_EQUALS)

lemma (in Ironbark_world) read_last_ip_EQUALS_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref (EQUALS reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_EQUALS initial_state_simps)

lemma (in Ironbark_world) read_ip_EQUALS_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (EQUALS reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_EQUALS initial_state_simps)

end
```

D.3.23 Simplification Rules Over NOT_EQUALS

theory *NOT_EQUALS_simps*

imports
NOT_EQUALS_decomposition

begin

As with most instructions, we provide various simplification rules for the *NOT_EQUALS* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

lemma (**in** *Ironbark_world*) *flag_NOT_EQUALS:*
assumes
‹*typical_flags state*›
‹*check_write_permission reg1*›
‹*check_read_permission reg2*›
‹*check_read_permission reg3*›
shows
‹*flag_state (NOT_EQUALS reg1 reg2 reg3 state) = flag_state state*›
using *assms*
by(*simp add: NOT_EQUALS_decomp_mixed*)

lemma (**in** *Ironbark_world*) *memory_NOT_EQUALS:*
‹*program_memory (NOT_EQUALS reg1 reg2 reg3 state) = program_memory state*›
‹*call_memory (NOT_EQUALS reg1 reg2 reg3 state) = call_memory state*›
‹*static_memory (NOT_EQUALS reg1 reg2 reg3 state) = static_memory state*›
‹*dynamic_memory (NOT_EQUALS reg1 reg2 reg3 state) = dynamic_memory state*›
‹*input_memory (NOT_EQUALS reg1 reg2 reg3 state) = input_memory state*›
‹*output_memory (NOT_EQUALS reg1 reg2 reg3 state) = output_memory state*›
by(*simp_all add: instruction_impl_defs state_manipulation_simps*)

lemma (**in** *Ironbark_world*) *read_flag_NOT_EQUALS:*
assumes
‹*typical_flags state*›
‹*check_write_permission reg1*›
‹*check_read_permission reg2*›
‹*check_read_permission reg3*›
shows
‹*get_end_jump (NOT_EQUALS reg1 reg2 reg3 state) = 0*›
‹*get_end_call (NOT_EQUALS reg1 reg2 reg3 state) = 0*›
‹*get_end_return (NOT_EQUALS reg1 reg2 reg3 state) = 0*›
‹*get_halt (NOT_EQUALS reg1 reg2 reg3 state) = 0*›
‹*get_error (NOT_EQUALS reg1 reg2 reg3 state) = 0*›
using *assms*
by(*simp_all add: read_flag_decomp flag_NOT_EQUALS*)

lemma (**in** *Ironbark_world*) *common_flags_NOT_EQUALS:*
assumes

```
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹typical_flags (NOT_EQUALS reg1 reg2 reg3 state)›
  using assms
  by(simp_all add: read_flag_NOT_EQUALS)

lemma (in Ironbark_world) read_register_NOT_EQUALS_diff:
  assumes
    ‹reg1 ≠ reg2›
    ‹reg1 ∉ {
              instruction_pointer_ref,
              last_instruction_pointer_ref,
              cycles_register_ref
            }›
  shows
    ‹read_register reg1 (NOT_EQUALS reg2 reg3 reg4 state) = read_register reg1 state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_register_NOT_EQUALS_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register reg1 (NOT_EQUALS reg1 reg2 reg3 state) =
    (
      if ((read_register reg2 state = read_register reg3 state)) then
        0x0
      else
        0x1
    )›
  using assms
  by(simp add: NOT_EQUALS_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_ip_NOT_EQUALS:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register instruction_pointer_ref (NOT_EQUALS reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
```

```
  by(simp add: NOT_EQUALS_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_NOT_EQUALS:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register cycles_register_ref (NOT_EQUALS reg1 reg2 reg3 state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: NOT_EQUALS_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_NOT_EQUALS:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
    ‹read_register last_instruction_pointer_ref (NOT_EQUALS reg1 reg2 reg3 state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: NOT_EQUALS_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_memory_NOT_EQUALS:
  shows
    ‹read_program_memory  address (NOT_EQUALS reg1 reg2 reg3 state)
    = read_program_memory address state›

    ‹read_call_memory     address (NOT_EQUALS reg1 reg2 reg3 state)
    = read_call_memory    address state›

    ‹read_static_memory   address (NOT_EQUALS reg1 reg2 reg3 state)
    = read_static_memory  address state›

    ‹read_dynamic_memory  address (NOT_EQUALS reg1 reg2 reg3 state)
    = read_dynamic_memory address state›
  using memory_NOT_EQUALS
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_NOT_EQUALS_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  shows
```

```
    ‹write_register reg1 value (NOT_EQUALS reg1 reg2 reg3 state)
    = write_register
      reg1
      value
      (standard_post_instruction common_instruction_duration state)›
  using assms
  by(simp add:
      instruction_impl_defs
      state_manipulation_reorder
      state_manipulation_simps)
```

The following are lemmas of NOT_EQUALS on the initial state.

```
lemma (in Ironbark_world) read_flag_NOT_EQUALS_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (NOT_EQUALS reg1 reg2 reg3 start_state) = 0›
    ‹get_end_call   (NOT_EQUALS reg1 reg2 reg3 start_state) = 0›
    ‹get_end_return (NOT_EQUALS reg1 reg2 reg3 start_state) = 0›
    ‹get_halt       (NOT_EQUALS reg1 reg2 reg3 start_state) = 0›
    ‹get_error      (NOT_EQUALS reg1 reg2 reg3 start_state) = 0›
  using assms
  by(simp_all add: read_flag_NOT_EQUALS initial_state_simps)

lemma (in Ironbark_world) common_flags_NOT_EQUALS_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (NOT_EQUALS reg1 reg2 reg3 start_state)›
  using assms
  by(simp add: read_flag_NOT_EQUALS_initial_state)

lemma (in Ironbark_world) read_memory_NOT_EQUALS_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
```

```
  shows
    ‹read_program_memory  address (NOT_EQUALS reg1 reg2 reg3 start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (NOT_EQUALS reg1 reg2 reg3 start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (NOT_EQUALS reg1 reg2 reg3 start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (NOT_EQUALS reg1 reg2 reg3 start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_NOT_EQUALS)

lemma (in Ironbark_world) read_last_ip_NOT_EQUALS_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref
      (NOT_EQUALS reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_NOT_EQUALS initial_state_simps)

lemma (in Ironbark_world) read_ip_NOT_EQUALS_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
    ‹check_read_permission reg3›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (NOT_EQUALS reg1 reg2 reg3 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_NOT_EQUALS initial_state_simps)

end
```

D.3.24 Simplification Rules Over RANDOMISE

theory *RANDOMISE_simps*

imports
RANDOMISE_decomposition

begin

As with most instructions, we provide various simplification rules for the *RANDOMISE* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.
We order the rules based on the second operator, according to which would be applied last in the normalised form.

lemma (**in** *Ironbark_world*) *flag_RANDOMISE:*
assumes
‹*typical_flags state*›
‹*check_write_permission reg1*›
shows
‹*flag_state (RANDOMISE reg1 state) = flag_state state*›
using *assms*
by (*simp add: RANDOMISE_decomp_mixed*)

lemma (**in** *Ironbark_world*) *memory_RANDOMISE:*
‹*program_memory (RANDOMISE reg1 state) = program_memory state*›
‹*call_memory (RANDOMISE reg1 state) = call_memory state*›
‹*static_memory (RANDOMISE reg1 state) = static_memory state*›
‹*dynamic_memory (RANDOMISE reg1 state) = dynamic_memory state*›
‹*input_memory (RANDOMISE reg1 state) = input_memory state*›
‹*output_memory (RANDOMISE reg1 state) = output_memory state*›
by (*simp_all add: instruction_impl_defs state_manipulation_simps*)

lemma (**in** *Ironbark_world*) *read_flag_RANDOMISE:*
assumes
‹*typical_flags state*›
‹*check_write_permission reg1*›
shows
‹*get_end_jump (RANDOMISE reg1 state) = 0*›
‹*get_end_call (RANDOMISE reg1 state) = 0*›
‹*get_end_return (RANDOMISE reg1 state) = 0*›
‹*get_halt (RANDOMISE reg1 state) = 0*›
‹*get_error (RANDOMISE reg1 state) = 0*›
using *assms*
by (*simp_all add: read_flag_decomp flag_RANDOMISE*)

lemma (**in** *Ironbark_world*) *common_flags_RANDOMISE:*
assumes
‹*typical_flags state*›

```
    ‹check_write_permission reg1›
  shows
    ‹typical_flags (RANDOMISE reg1 state)›
  using assms
  by(simp_all add: read_flag_RANDOMISE)

lemma (in Ironbark_world) read_register_RANDOMISE_diff:
  assumes
    ‹reg1 ≠ reg2›
    ‹reg1 ∉ {
              instruction_pointer_ref,
              last_instruction_pointer_ref,
              cycles_register_ref
            }›
  shows
    ‹read_register reg1 (RANDOMISE reg2 state) = read_register reg1 state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_register_RANDOMISE_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
  shows
    ‹read_register reg1 (RANDOMISE reg1 state) = random_stream time›
  using assms
  by(simp add: RANDOMISE_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_ip_RANDOMISE:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
  shows
    ‹read_register instruction_pointer_ref (RANDOMISE reg1 state)
      = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: RANDOMISE_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_RANDOMISE:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
  shows
    ‹read_register cycles_register_ref (RANDOMISE reg1 state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: RANDOMISE_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_RANDOMISE:
```

```
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
  shows
    ‹read_register last_instruction_pointer_ref (RANDOMISE reg1 state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: RANDOMISE_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_memory_RANDOMISE:
  ‹read_program_memory  address (RANDOMISE reg1 state)
  = read_program_memory address state›

  ‹read_call_memory     address (RANDOMISE reg1 state)
  = read_call_memory    address state›

  ‹read_static_memory   address (RANDOMISE reg1 state)
  = read_static_memory  address state›

  ‹read_dynamic_memory  address (RANDOMISE reg1 state)
  = read_dynamic_memory address state›
  using memory_RANDOMISE
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) write_register_RANDOMISE_same:
  assumes
    ‹typical_flags state›
    ‹check_write_permission reg1›
  shows
    ‹write_register reg1 value (RANDOMISE reg1 state) =
      write_register
        reg1
        value
        (standard_post_instruction common_instruction_duration state)›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_reorder state_manipulation_simps)
```

The following are lemmas of RANDOMISE on the initial state.

```
lemma (in Ironbark_world) read_flag_RANDOMISE_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (RANDOMISE reg1 start_state) = 0›
    ‹get_end_call   (RANDOMISE reg1 start_state) = 0›
    ‹get_end_return (RANDOMISE reg1 start_state) = 0›
```

```
    ‹get_halt       (RANDOMISE reg1 start_state) = 0›
    ‹get_error      (RANDOMISE reg1 start_state) = 0›
  using assms
  by(simp_all add: read_flag_RANDOMISE initial_state_simps)

lemma (in Ironbark_world) common_flags_RANDOMISE_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (RANDOMISE reg1 start_state)›
  using assms
  by(simp add: read_flag_RANDOMISE_initial_state)

lemma (in Ironbark_world) read_memory_RANDOMISE_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (RANDOMISE reg1 start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (RANDOMISE reg1 start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (RANDOMISE reg1 start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (RANDOMISE reg1 start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_RANDOMISE)

lemma (in Ironbark_world) read_last_ip_RANDOMISE_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref (RANDOMISE reg1 start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_RANDOMISE initial_state_simps)
```

```
lemma (in Ironbark_world) read_ip_RANDOMISE_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_write_permission reg1›
    ‹check_read_permission reg2›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (RANDOMISE reg1 start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_RANDOMISE initial_state_simps)

end
```

D.3.25 Simplification Rules Over END_JUMP

```
theory END_JUMP_simps

imports
  END_JUMP_decomposition

begin
```

As with most instructions, we provide various simplification rules for the *END_JUMP* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_END_JUMP_fall:
  assumes
    ‹typical_flags state›
  shows
    ‹flag_state (END_JUMP immediate state) = flag_state state›
  using assms
  by(simp add: END_JUMP_decomp_mixed_fall)

lemma (in Ironbark_world) memory_END_JUMP:
  ‹program_memory (END_JUMP immediate state) = program_memory state›
  ‹call_memory    (END_JUMP immediate state) = call_memory    state›
  ‹static_memory  (END_JUMP immediate state) = static_memory  state›
  ‹dynamic_memory (END_JUMP immediate state) = dynamic_memory state›
  ‹input_memory   (END_JUMP immediate state) = input_memory   state›
  ‹output_memory  (END_JUMP immediate state) = output_memory  state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)
```

```
lemma (in Ironbark_world) read_flag_END_JUMP_fall:
  assumes
    ‹typical_flags state›
  shows
    ‹get_end_jump   (END_JUMP immediate state) = 0›
    ‹get_end_call   (END_JUMP immediate state) = 0›
    ‹get_end_return (END_JUMP immediate state) = 0›
    ‹get_halt       (END_JUMP immediate state) = 0›
    ‹get_error      (END_JUMP immediate state) = 0›
  using assms
  by(simp_all add: read_flag_decomp flag_END_JUMP_fall)

lemma (in Ironbark_world) read_flag_END_JUMP_jump:
  assumes
    ‹end_jump_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹get_end_jump   (END_JUMP immediate state) = 0›
    ‹get_end_call   (END_JUMP immediate state) = 0›
    ‹get_end_return (END_JUMP immediate state) = 0›
    ‹get_halt       (END_JUMP immediate state) = 0›
    ‹get_error      (END_JUMP immediate state) = 0›
  using assms
  by(simp_all add: END_JUMP_decomp_manipulation_jump state_manipulation_simps)

lemma (in Ironbark_world) common_flags_END_JUMP_fall:
  assumes
    ‹typical_flags state›
  shows
    ‹typical_flags (END_JUMP immediate state)›
  using assms
  by(simp_all add: read_flag_END_JUMP_fall)

lemma (in Ironbark_world) common_flags_END_JUMP_jump:
  assumes
    ‹end_jump_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹typical_flags (END_JUMP immediate state)›
  using assms
  by(simp_all add: read_flag_END_JUMP_jump)

lemma (in Ironbark_world) read_register_END_JUMP:
  assumes
    ‹regID ∉ {
                instruction_pointer_ref,
                last_instruction_pointer_ref,
                cycles_register_ref
              }›
```

```
  shows
    ‹read_register regID (END_JUMP immediate state) = read_register regID state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_ip_END_JUMP_fall:
  assumes
    ‹typical_flags state›
  shows
    ‹read_register instruction_pointer_ref (END_JUMP immediate state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: END_JUMP_decomp_manipulation_fall state_manipulation_simps)

lemma (in Ironbark_world) read_ip_END_JUMP_jump:
  assumes
    ‹end_jump_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹read_register instruction_pointer_ref (END_JUMP immediate state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: END_JUMP_decomp_manipulation_jump state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_END_JUMP_fall:
  assumes
    ‹typical_flags state›
  shows
    ‹read_register cycles_register_ref (END_JUMP immediate state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: END_JUMP_decomp_manipulation_fall state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_END_JUMP_jump:
  assumes
    ‹end_jump_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹read_register cycles_register_ref (END_JUMP immediate state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: END_JUMP_decomp_manipulation_jump state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_END_JUMP_fall:
  assumes
    ‹typical_flags state›
  shows
    ‹read_register last_instruction_pointer_ref (END_JUMP immediate state)
    = (read_register instruction_pointer_ref state)›
```

```
  using assms
  by(simp add: END_JUMP_decomp_manipulation_fall state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_END_JUMP_jump:
  assumes
    ‹end_jump_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹read_register last_instruction_pointer_ref (END_JUMP immediate state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: END_JUMP_decomp_manipulation_jump state_manipulation_simps)

lemma (in Ironbark_world) read_memory_END_JUMP:
  ‹read_program_memory  address (END_JUMP immediate state)
  = read_program_memory address state›

  ‹read_call_memory     address (END_JUMP immediate state)
  = read_call_memory    address state›

  ‹read_static_memory   address (END_JUMP immediate state)
  = read_static_memory  address state›

  ‹read_dynamic_memory  address (END_JUMP immediate state)
  = read_dynamic_memory address state›
  using memory_END_JUMP
  by(simp_all add: state_manipulation_decomp)
```

The following are lemmas of END_JUMP on the initial state.

```
lemma (in Ironbark_world) read_flag_END_JUMP_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (END_JUMP immediate start_state) = 0›
    ‹get_end_call   (END_JUMP immediate start_state) = 0›
    ‹get_end_return (END_JUMP immediate start_state) = 0›
    ‹get_halt       (END_JUMP immediate start_state) = 0›
    ‹get_error      (END_JUMP immediate start_state) = 0›
  using assms
  by(simp_all add: read_flag_END_JUMP_fall initial_state_simps)

lemma (in Ironbark_world) common_flags_END_JUMP_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
```

```
    ‹typical_flags (END_JUMP immediate start_state)›
  using assms
  by(simp add: read_flag_END_JUMP_initial_state)

lemma (in Ironbark_world) read_memory_END_JUMP_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (END_JUMP immediate start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (END_JUMP immediate start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (END_JUMP immediate start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (END_JUMP immediate start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_END_JUMP initial_state_simps)

lemma (in Ironbark_world) read_last_ip_END_JUMP_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref (END_JUMP immediate start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_END_JUMP_fall initial_state_simps)

lemma (in Ironbark_world) read_ip_END_JUMP_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (END_JUMP immediate start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
  using assms
  by(simp add: read_ip_END_JUMP_fall initial_state_simps)

end
```

D.3.26 Simplification Rules Over END_JUMP_STRICT

theory *END_JUMP_STRICT_simps*

imports
END_JUMP_STRICT_decomposition

begin

As with most instructions, we provide various simplification rules for the end jump strict instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.
We order the rules based on the second operator, according to which would be applied last in the normalised form.

lemma (**in** *Ironbark_world*) *flag_END_JUMP_STRICT:*
assumes
‹end_jump_flags state›
‹read_register last_instruction_pointer_ref state = immediate›
shows
‹flag_state (END_JUMP_STRICT immediate state) = flag_state (clear_end_jump state)›
using *assms*
by *(simp add: END_JUMP_STRICT_decomp_mixed)*

lemma (**in** *Ironbark_world*) *memory_END_JUMP_STRICT:*
‹program_memory (END_JUMP_STRICT immediate state) = program_memory state›
‹call_memory (END_JUMP_STRICT immediate state) = call_memory state›
‹static_memory (END_JUMP_STRICT immediate state) = static_memory state›
‹dynamic_memory (END_JUMP_STRICT immediate state) = dynamic_memory state›
‹input_memory (END_JUMP_STRICT immediate state) = input_memory state›
‹output_memory (END_JUMP_STRICT immediate state) = output_memory state›
by *(simp_all add: instruction_impl_defs state_manipulation_simps)*

lemma (**in** *Ironbark_world*) *read_flag_END_JUMP_STRICT:*
assumes
‹end_jump_flags state›
‹read_register last_instruction_pointer_ref state = immediate›
shows
‹get_end_jump (END_JUMP_STRICT immediate state) = 0›
‹get_end_call (END_JUMP_STRICT immediate state) = 0›
‹get_end_return (END_JUMP_STRICT immediate state) = 0›
‹get_halt (END_JUMP_STRICT immediate state) = 0›
‹get_error (END_JUMP_STRICT immediate state) = 0›
using *assms*
by *(simp_all add: END_JUMP_STRICT_decomp_manipulation state_manipulation_simps)*

lemma (**in** *Ironbark_world*) *common_flags_END_JUMP_STRICT:*
assumes
‹end_jump_flags state›

```
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹typical_flags (END_JUMP_STRICT immediate state)›
  using assms
  by(simp_all add: read_flag_END_JUMP_STRICT)

lemma (in Ironbark_world) read_register_END_JUMP_STRICT:
  assumes
    ‹regID ∉ {
      instruction_pointer_ref,
      last_instruction_pointer_ref,
      cycles_register_ref
    }›
  shows
    ‹read_register regID (END_JUMP_STRICT immediate state) = read_register regID state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_ip_END_JUMP_STRICT:
  assumes
    ‹end_jump_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹read_register instruction_pointer_ref (END_JUMP_STRICT immediate state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: END_JUMP_STRICT_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_END_JUMP_STRICT:
  assumes
    ‹end_jump_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹read_register cycles_register_ref (END_JUMP_STRICT immediate state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: END_JUMP_STRICT_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_END_JUMP_STRICT:
  assumes
    ‹end_jump_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹read_register last_instruction_pointer_ref (END_JUMP_STRICT immediate state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: END_JUMP_STRICT_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_memory_END_JUMP_STRICT:
```

```
  shows
    ‹read_program_memory  address (END_JUMP_STRICT immediate state)
    = read_program_memory address state›

    ‹read_call_memory     address (END_JUMP_STRICT immediate state)
    = read_call_memory    address state›

    ‹read_static_memory   address (END_JUMP_STRICT immediate state)
    = read_static_memory  address state›

    ‹read_dynamic_memory  address (END_JUMP_STRICT immediate state)
    = read_dynamic_memory address state›
  using memory_END_JUMP_STRICT
  by(simp_all add: state_manipulation_decomp)
```

The following are lemmas of END_JUMP_STRICT on the initial state. Since the initial state is not after a jump, we get error. The simps for ERROR0 can be used in this instance.

```
lemma (in Ironbark_world) END_JUMP_STRICT_initial_state_errors:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹(END_JUMP_STRICT immediate start_state) = (ERROR0 start_state)›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

end
```

D.3.27 Simplification Rules Over JUMP

```
theory JUMP_simps

imports
  JUMP_decomposition

begin
```

As with most instructions, we provide various simplification rules for the *JUMP* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_JUMP:
  assumes
    ‹typical_flags state›
  shows
```

```
    ‹flag_state (JUMP immediate state) = flag_state (set_end_jump state)›
  using assms
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) memory_JUMP:
  ‹program_memory (JUMP immediate state) = program_memory state›
  ‹call_memory    (JUMP immediate state) = call_memory    state›
  ‹static_memory  (JUMP immediate state) = static_memory  state›
  ‹dynamic_memory (JUMP immediate state) = dynamic_memory state›
  ‹input_memory   (JUMP immediate state) = input_memory   state›
  ‹output_memory  (JUMP immediate state) = output_memory  state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_JUMP:
  assumes
    ‹typical_flags state›
  shows
    ‹get_end_jump   (JUMP immediate state) = 1›
    ‹get_end_call   (JUMP immediate state) = 0›
    ‹get_end_return (JUMP immediate state) = 0›
    ‹get_halt       (JUMP immediate state) = 0›
    ‹get_error      (JUMP immediate state) = 0›
  using assms
  by(simp_all add: JUMP_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) common_flags_JUMP:
  assumes
    ‹typical_flags state›
  shows
    ‹end_jump_flags (JUMP immediate state)›
  using assms
  by(simp_all add: read_flag_JUMP)

lemma (in Ironbark_world) read_register_JUMP:
  assumes
    ‹regID ∉ {
                instruction_pointer_ref,
                last_instruction_pointer_ref,
                cycles_register_ref
              }›
  shows
    ‹read_register regID (JUMP immediate state) = read_register regID state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_ip_JUMP:
  assumes
    ‹typical_flags state›
  shows
```

```
    ‹read_register instruction_pointer_ref (JUMP immediate state) = immediate›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_JUMP:
  assumes
    ‹typical_flags state›
  shows
    ‹read_register cycles_register_ref (JUMP immediate state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_JUMP:
  assumes
    ‹typical_flags state›
  shows
    ‹read_register last_instruction_pointer_ref (JUMP immediate state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: instruction_impl_defs  state_manipulation_simps)

lemma (in Ironbark_world) read_memory_JUMP:
  ‹read_program_memory  address (JUMP immediate state)
  = read_program_memory address state›

  ‹read_call_memory     address (JUMP immediate state)
  = read_call_memory    address state›

  ‹read_static_memory   address (JUMP immediate state)
  = read_static_memory  address state›

  ‹read_dynamic_memory  address (JUMP immediate state)
  = read_dynamic_memory address state›
  using memory_JUMP
  by(simp_all add: state_manipulation_decomp)
```

The following are cases of JUMP over the initial state.

```
lemma (in Ironbark_world) read_flag_JUMP_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (JUMP immediate start_state) = 1›
    ‹get_end_call   (JUMP immediate start_state) = 0›
    ‹get_end_return (JUMP immediate start_state) = 0›
    ‹get_halt       (JUMP immediate start_state) = 0›
    ‹get_error      (JUMP immediate start_state) = 0›
```

```
  using assms
  by(simp_all add: read_flag_JUMP initial_state_simps)

lemma (in Ironbark_world) common_flags_JUMP_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹end_jump_flags (JUMP immediate start_state)›
  using assms
  by(simp add: read_flag_JUMP_initial_state)

lemma (in Ironbark_world) read_memory_JUMP_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (JUMP immediate start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (JUMP immediate start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (JUMP immediate start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (JUMP immediate start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_JUMP)

lemma (in Ironbark_world) read_last_ip_JUMP_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_read_permission reg1›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref (JUMP immediate start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_JUMP initial_state_simps)

lemma (in Ironbark_world) read_ip_JUMP_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
```

```
  assumes
    ‹check_read_permission reg1›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (JUMP immediate start_state) = immediate›
proof -
  have
    ‹(read_register reg1 start_state) = 0x00›
    using assms
    by(simp_all add: state_manipulation_simps)
  then show ?thesis
    using assms
    by(simp_all add: read_ip_JUMP state_manipulation_simps)
qed

end
```

D.3.28 Simplification Rules Over CONDITIONAL_JUMP

```
theory CONDITIONAL_JUMP_simps

imports
  CONDITIONAL_JUMP_decomposition

begin
```

As with most instructions, we provide various simplification rules for the *CONDITIONAL_JUMP* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_CONDITIONAL_JUMP:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
  shows
    ‹(read_register reg1 state) = 0x00
    ⟹ flag_state (CONDITIONAL_JUMP reg1 immediate state) = flag_state state›

    ‹(read_register reg1 state) ≠ 0x00
    ⟹ flag_state (CONDITIONAL_JUMP reg1 immediate state)
      = flag_state (set_end_jump state)›
  using assms
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) memory_CONDITIONAL_JUMP:
  ‹program_memory (CONDITIONAL_JUMP reg1 immediate state) = program_memory  state›
```

```
  ‹call_memory    (CONDITIONAL_JUMP reg1 immediate state) = call_memory     state›
  ‹static_memory  (CONDITIONAL_JUMP reg1 immediate state) = static_memory   state›
  ‹dynamic_memory (CONDITIONAL_JUMP reg1 immediate state) = dynamic_memory  state›
  ‹input_memory   (CONDITIONAL_JUMP reg1 immediate state) = input_memory    state›
  ‹output_memory  (CONDITIONAL_JUMP reg1 immediate state) = output_memory   state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_CONDITIONAL_JUMP:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
  shows
    ‹get_end_call   (CONDITIONAL_JUMP reg1 immediate state) = 0›
    ‹get_end_return (CONDITIONAL_JUMP reg1 immediate state) = 0›
    ‹get_halt       (CONDITIONAL_JUMP reg1 immediate state) = 0›
    ‹get_error      (CONDITIONAL_JUMP reg1 immediate state) = 0›

    ‹read_register reg1 state = 0x00
    ⟹ get_end_jump (CONDITIONAL_JUMP reg1 immediate state) = 0›

    ‹read_register reg1 state ≠ 0x00
    ⟹ get_end_jump (CONDITIONAL_JUMP reg1 immediate state) = 1›
  using assms
  by(simp_all add: CONDITIONAL_JUMP_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) common_flags_CONDITIONAL_JUMP_fall:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹(read_register reg1 state) = 0x00›
  shows
    ‹typical_flags (CONDITIONAL_JUMP reg1 immediate state)›
  using assms
  by(simp_all add: read_flag_CONDITIONAL_JUMP)

lemma (in Ironbark_world) common_flags_CONDITIONAL_JUMP_jump:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹(read_register reg1 state) ≠ 0x00›
  shows
    ‹end_jump_flags (CONDITIONAL_JUMP reg1 immediate state)›
  using assms
  by(simp_all add: read_flag_CONDITIONAL_JUMP)

lemma (in Ironbark_world) read_register_CONDITIONAL_JUMP:
  assumes
    ‹regID ∉ {
                instruction_pointer_ref,
```

```
              last_instruction_pointer_ref,
              cycles_register_ref
            }›
  shows
    ‹read_register  regID (CONDITIONAL_JUMP reg1 immediate state)
    = read_register regID state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_ip_CONDITIONAL_JUMP_jump:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹(read_register reg1 state) ≠ 0x00›
  shows
    ‹read_register instruction_pointer_ref (CONDITIONAL_JUMP reg1 immediate state)
    = immediate›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_ip_CONDITIONAL_JUMP_fall:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
    ‹(read_register reg1 state) = 0x00›
  shows
    ‹read_register instruction_pointer_ref (CONDITIONAL_JUMP reg1 immediate state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_CONDITIONAL_JUMP:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
  shows
    ‹read_register cycles_register_ref (CONDITIONAL_JUMP reg1 immediate state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_CONDITIONAL_JUMP:
  assumes
    ‹typical_flags state›
    ‹check_read_permission reg1›
  shows
    ‹read_register last_instruction_pointer_ref (CONDITIONAL_JUMP reg1 immediate state)
    = (read_register instruction_pointer_ref state)›
  using assms
```

```
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_memory_CONDITIONAL_JUMP:
  ‹read_program_memory  address (CONDITIONAL_JUMP reg1 immediate state)
  = read_program_memory address state›

  ‹read_call_memory     address (CONDITIONAL_JUMP reg1 immediate state)
  = read_call_memory    address state›

  ‹read_static_memory   address (CONDITIONAL_JUMP reg1 immediate state)
  = read_static_memory  address state›

  ‹read_dynamic_memory  address (CONDITIONAL_JUMP reg1 immediate state)
  = read_dynamic_memory address state›
  using memory_CONDITIONAL_JUMP
  by(simp_all add: state_manipulation_decomp)
```

The following are cases of CONDITIONAL_JUMP over the initial state.

```
lemma (in Ironbark_world) read_flag_CONDITIONAL_JUMP_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_read_permission reg1›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (CONDITIONAL_JUMP reg1 immediate start_state) = 0›
    ‹get_end_call   (CONDITIONAL_JUMP reg1 immediate start_state) = 0›
    ‹get_end_return (CONDITIONAL_JUMP reg1 immediate start_state) = 0›
    ‹get_halt       (CONDITIONAL_JUMP reg1 immediate start_state) = 0›
    ‹get_error      (CONDITIONAL_JUMP reg1 immediate start_state) = 0›
  using assms
proof -
  have
    ‹typical_flags start_state›
    ‹(read_register reg1 start_state) = 0x00›
    using assms
    by(simp_all add: state_manipulation_simps)
  then show
    ‹get_end_jump   (CONDITIONAL_JUMP reg1 immediate start_state) = 0›
    ‹get_end_call   (CONDITIONAL_JUMP reg1 immediate start_state) = 0›
    ‹get_end_return (CONDITIONAL_JUMP reg1 immediate start_state) = 0›
    ‹get_halt       (CONDITIONAL_JUMP reg1 immediate start_state) = 0›
    ‹get_error      (CONDITIONAL_JUMP reg1 immediate start_state) = 0›
    using assms
    by(simp_all add: CONDITIONAL_JUMP_decomp_manipulation state_manipulation_simps)
qed

lemma (in Ironbark_world) common_flags_CONDITIONAL_JUMP_initial_state:
```

```
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_read_permission reg1›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹typical_flags (CONDITIONAL_JUMP reg1 immediate start_state)›
  using assms
  by(simp add: read_flag_CONDITIONAL_JUMP_initial_state)

lemma (in Ironbark_world) read_memory_CONDITIONAL_JUMP_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (CONDITIONAL_JUMP reg1 immediate start_state)
    = read_program_memory address start_state›

    ‹read_call_memory     address (CONDITIONAL_JUMP reg1 immediate start_state)
    = read_call_memory    address start_state›

    ‹read_static_memory   address (CONDITIONAL_JUMP reg1 immediate start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (CONDITIONAL_JUMP reg1 immediate start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_CONDITIONAL_JUMP)

lemma (in Ironbark_world) read_last_ip_CONDITIONAL_JUMP_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
    ‹check_read_permission reg1›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref
      (CONDITIONAL_JUMP reg1 immediate start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_CONDITIONAL_JUMP initial_state_simps)

lemma (in Ironbark_world) read_ip_CONDITIONAL_JUMP_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  assumes
```

```
    ‹check_read_permission reg1›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (CONDITIONAL_JUMP reg1 immediate start_state)
    = (read_register instruction_pointer_ref start_state) + 1›
proof -
  have
    ‹(read_register reg1 start_state) = 0x00›
    using assms
    by(simp_all add: state_manipulation_simps)
  then show ?thesis
    using assms
    by(simp_all add: read_ip_CONDITIONAL_JUMP_fall state_manipulation_simps)
qed

end
```

D.3.29 Simplification Rules Over END_CALL

```
theory END_CALL_simps

imports
  END_CALL_decomposition

begin
```

As with most instructions, we provide various simplification rules for the *END_CALL* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_END_CALL:
  assumes
    ‹end_call_flags state›
  shows
    ‹flag_state (END_CALL state) = flag_state (clear_end_call state)›
  using assms
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) register_END_CALL:
  assumes
    ‹end_call_flags state›
  shows
    ‹register_state (END_CALL state)
    = register_state (standard_post_instruction common_instruction_duration state)›
  using assms
  apply(simp_all add: instruction_impl_defs)
```

```
  apply(intro register_state_passover)
  apply(simp_all add: state_manipulation_simps)
  done

lemma (in Ironbark_world) memory_END_CALL:
  shows
    ‹program_memory (END_CALL state) = program_memory state›
    ‹call_memory    (END_CALL state) = call_memory    state›
    ‹static_memory  (END_CALL state) = static_memory  state›
    ‹dynamic_memory (END_CALL state) = dynamic_memory state›
    ‹input_memory   (END_CALL state) = input_memory   state›
    ‹output_memory  (END_CALL state) = output_memory  state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_END_CALL:
  assumes
    ‹end_call_flags state›
  shows
    ‹get_end_jump   (END_CALL state) = 0›
    ‹get_end_call   (END_CALL state) = 0›
    ‹get_end_return (END_CALL state) = 0›
    ‹get_halt       (END_CALL state) = 0›
    ‹get_error      (END_CALL state) = 0›
  using assms
  by(simp_all add: state_manipulation_decomp flag_END_CALL)

lemma (in Ironbark_world) common_flags_END_CALL:
  assumes
    ‹end_call_flags state›
  shows
    ‹typical_flags (END_CALL state)›
  using assms
  by(simp_all add: read_flag_END_CALL)

lemma (in Ironbark_world) read_register_END_CALL:
  assumes
    ‹regID ∉ {
                instruction_pointer_ref,
                last_instruction_pointer_ref,
                cycles_register_ref
              }›
  shows
    ‹read_register regID (END_CALL state) = read_register regID state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_ip_END_CALL:
  assumes
    ‹end_call_flags state›
```

```
  shows
    ‹read_register instruction_pointer_ref (END_CALL state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_END_CALL:
  assumes
    ‹end_call_flags state›
  shows
    ‹read_register cycles_register_ref (END_CALL state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_END_CALL:
  assumes
    ‹end_call_flags state›
  shows
    ‹read_register last_instruction_pointer_ref (END_CALL state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_memory_END_CALL:
  shows
    ‹read_program_memory  address (END_CALL state)
    = read_program_memory address state›

    ‹read_call_memory     address (END_CALL state)
    = read_call_memory    address state›

    ‹read_static_memory   address (END_CALL state)
    = read_static_memory  address state›

    ‹read_dynamic_memory  address (END_CALL state)
    = read_dynamic_memory address state›
  using memory_END_CALL
  by(simp_all add: instruction_impl_defs state_manipulation_simps)
```

The following are lemmas of END_CALL on the initial state. Since there is no call, we just get error, and we can use the ERROR0 lemmas for reasoning beyond this point.

```
lemma (in Ironbark_world) END_CALL_initial_state_errors:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹(END_CALL start_state) = (ERROR0 start_state)›
```

using *assms*
by(*simp add: instruction_impl_defs state_manipulation_simps*)

end

D.3.30 Simplification Rules Over CALL

theory *CALL_simps*

imports
CALL_decomposition

begin

As with most instructions, we provide various simplification rules for the *CALL* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

lemma (**in** *Ironbark_world*) *memory_CALL:*
‹program_memory (CALL immediate state) = program_memory state›
‹static_memory (CALL immediate state) = static_memory state›
‹dynamic_memory (CALL immediate state) = dynamic_memory state›
‹input_memory (CALL immediate state) = input_memory state›
‹output_memory (CALL immediate state) = output_memory state›
by(*simp_all add: instruction_impl_defs state_manipulation_simps*)

lemma (**in** *Ironbark_world*) *read_flag_CALL:*
assumes
‹typical_flags state›
shows
‹get_end_jump (CALL immediate state) = 0›
‹get_end_call (CALL immediate state) = 1›
‹get_end_return (CALL immediate state) = 0›
‹get_halt (CALL immediate state) = 0›
‹get_error (CALL immediate state) = 0›
using *assms*
by(*auto simp add: CALL_decomp_manipulation state_manipulation_simps*)

lemma (**in** *Ironbark_world*) *common_flags_CALL:*
assumes
‹typical_flags state›
shows
‹end_call_flags (CALL immediate state)›
using *assms*
by(*simp_all add: read_flag_CALL*)

lemma (**in** *Ironbark_world*) *read_register_CALL:*

```
  assumes
    ‹regID ∉ {
                instruction_pointer_ref,
                last_instruction_pointer_ref,
                cycles_register_ref,
                call_frame_pointer_ref
              }›
  shows
    ‹read_register regID (CALL immediate state) = read_register regID state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_ip_CALL:
  assumes
    ‹typical_flags state›
  shows
    ‹read_register instruction_pointer_ref (CALL immediate state) = immediate›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_CALL:
  assumes
    ‹typical_flags state›
  shows
    ‹read_register cycles_register_ref (CALL immediate state)
    = (read_register cycles_register_ref state) + call_duration›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_CALL:
  assumes
    ‹typical_flags state›
  shows
    ‹read_register last_instruction_pointer_ref (CALL immediate state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_call_pointer_CALL:
  assumes
    ‹typical_flags state›
  shows
    ‹read_register call_frame_pointer_ref (CALL immediate state)
    = (read_register call_frame_pointer_ref state) + 67›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_memory_CALL:
  ‹read_program_memory  address (CALL immediate state)
```

```
  = read_program_memory address state›
  ‹read_static_memory   address (CALL immediate state)
  = read_static_memory  address state›
  ‹read_dynamic_memory  address (CALL immediate state)
  = read_dynamic_memory address state›
  using memory_CALL
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) read_call_memory_CALL:
  assumes
    ‹typical_flags state›
    ‹∀n ∈ {0 .. 66} . address ≠ read_register call_frame_pointer_ref state + n›
    ‹address < (read_register call_frame_pointer_ref state)›
  shows
    ‹read_call_memory address (CALL immediate state) = read_call_memory address state›
  using assms
  by(simp add:
      instruction_impl_defs
      state_manipulation_reorder
      state_manipulation_simps)
```

The following are cases of CALL over the initial state

```
lemma (in Ironbark_world) read_flag_CALL_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (CALL immediate start_state) = 0›
    ‹get_end_call   (CALL immediate start_state) = 1›
    ‹get_end_return (CALL immediate start_state) = 0›
    ‹get_halt       (CALL immediate start_state) = 0›
    ‹get_error      (CALL immediate start_state) = 0›
  using assms
  by(simp_all add: read_flag_CALL initial_state_simps)

lemma (in Ironbark_world) common_flags_CALL_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹end_call_flags (CALL immediate start_state)›
  using assms
  by(simp add: read_flag_CALL_initial_state)

lemma (in Ironbark_world) read_memory_CALL_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
```

```
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (CALL immediate start_state)
    = read_program_memory address start_state›

    ‹read_static_memory   address (CALL immediate start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (CALL immediate start_state)
    = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_CALL)

lemma (in Ironbark_world) read_last_ip_CALL_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref (CALL immediate start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_CALL initial_state_simps)

lemma (in Ironbark_world) read_ip_CALL_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (CALL immediate start_state) = immediate›
  using assms
  by(simp add: read_ip_CALL initial_state_simps)

end
```

D.3.31 Simplification Rules Over END_RETURN

```
theory END_RETURN_simps

imports
  END_RETURN_decomposition

begin
```

As with most instructions, we provide various simplification rules for the *END_RETURN* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the

instruction will pass.
We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_END_RETURN:
  assumes
    ‹end_return_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹flag_state (END_RETURN immediate state) = flag_state (clear_end_return state)›
  using assms
  by(simp add: END_RETURN_decomp_mixed)

lemma (in Ironbark_world) memory_END_RETURN:
  ‹program_memory (END_RETURN immediate state) = program_memory state›
  ‹call_memory    (END_RETURN immediate state) = call_memory    state›
  ‹static_memory  (END_RETURN immediate state) = static_memory  state›
  ‹dynamic_memory (END_RETURN immediate state) = dynamic_memory state›
  ‹input_memory   (END_RETURN immediate state) = input_memory   state›
  ‹output_memory  (END_RETURN immediate state) = output_memory  state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_END_RETURN:
  assumes
    ‹end_return_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹get_end_jump   (END_RETURN immediate state) = 0›
    ‹get_end_call   (END_RETURN immediate state) = 0›
    ‹get_end_return (END_RETURN immediate state) = 0›
    ‹get_halt       (END_RETURN immediate state) = 0›
    ‹get_error      (END_RETURN immediate state) = 0›
  using assms
  by(simp_all add: END_RETURN_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) common_flags_END_RETURN:
  assumes
    ‹end_return_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹typical_flags (END_RETURN immediate state)›
  using assms
  by(simp_all add: read_flag_END_RETURN)

lemma (in Ironbark_world) read_register_END_RETURN:
  assumes
    ‹regID ∉ {
                instruction_pointer_ref,
                last_instruction_pointer_ref,
```

```
              cycles_register_ref
            }›
  shows
    ‹read_register regID (END_RETURN immediate state) = read_register regID state›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_ip_END_RETURN:
  assumes
    ‹end_return_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹read_register instruction_pointer_ref (END_RETURN immediate state)
    = (read_register instruction_pointer_ref state) + 1›
  using assms
  by(simp add: END_RETURN_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_cycles_END_RETURN:
  assumes
    ‹end_return_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹read_register cycles_register_ref (END_RETURN immediate state)
    = (read_register cycles_register_ref state) + common_instruction_duration›
  using assms
  by(simp add: END_RETURN_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_END_RETURN:
  assumes
    ‹end_return_flags state›
    ‹read_register last_instruction_pointer_ref state = immediate›
  shows
    ‹read_register last_instruction_pointer_ref (END_RETURN immediate state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: END_RETURN_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) read_memory_END_RETURN:
  shows
    ‹read_program_memory  address (END_RETURN immediate state)
    = read_program_memory address state›
    ‹read_call_memory     address (END_RETURN immediate state)
    = read_call_memory    address state›
    ‹read_static_memory   address (END_RETURN immediate state)
    = read_static_memory  address state›
    ‹read_dynamic_memory  address (END_RETURN immediate state)
    = read_dynamic_memory address state›
  using memory_END_RETURN
  by(simp_all add: state_manipulation_decomp)
```

The following are lemmas of END_RETURN on the initial state. Since the initial state is not after a jump, we get error. The simps for ERROR0 can be used in this instance.

lemma *(in Ironbark_world) END_RETURN_initial_state_errors:*
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹END_RETURN immediate start_state = ERROR0 start_state›
  using *assms*
  by*(simp add: instruction_impl_defs state_manipulation_simps)*

end

D.3.32 Simplification Rules Over RETURN

theory *RETURN_simps*

imports
  RETURN_decomposition

begin

As with most instructions, we provide various simplification rules for the ***RETURN*** instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

lemma *(in Ironbark_world) memory_RETURN:*
  ‹program_memory (RETURN state) = program_memory state›
  ‹call_memory   (RETURN state) = call_memory   state›
  ‹static_memory (RETURN state) = static_memory state›
  ‹dynamic_memory (RETURN state) = dynamic_memory state›
  ‹input_memory   (RETURN state) = input_memory   state›
  ‹output_memory (RETURN state) = output_memory state›
  by*(simp_all add: instruction_impl_defs state_manipulation_simps Let_def)*

lemma *(in Ironbark_world) read_flag_RETURN:*
  assumes
    ‹typical_flags state›
  shows
    ‹get_end_jump   (RETURN state) = 0›
    ‹get_end_call   (RETURN state) = 0›
    ‹get_end_return (RETURN state) = 1›
    ‹get_halt   (RETURN state) = 0›
    ‹get_error   (RETURN state) = 0›
  using *assms*

```
  by(auto simp add: RETURN_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) common_flags_RETURN:
  assumes
    ‹typical_flags state›
  shows
    ‹end_return_flags (RETURN state)›
  using assms
  by(simp_all add: read_flag_RETURN)

lemma (in Ironbark_world) read_ip_RETURN:
  assumes
    ‹typical_flags state›
  shows
    ‹read_register instruction_pointer_ref (RETURN state)
    = read_call_memory ((read_register call_frame_pointer_ref state) - 67) state + 1›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps Let_def)

lemma (in Ironbark_world) read_cycles_RETURN:
  assumes
    ‹typical_flags state›
  shows
    ‹read_register cycles_register_ref (RETURN state)
    = (read_register cycles_register_ref state) + call_duration›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_last_ip_RETURN:
  assumes
    ‹typical_flags state›
  shows
    ‹read_register last_instruction_pointer_ref (RETURN state)
    = (read_register instruction_pointer_ref state)›
  using assms
  by(simp add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_memory_RETURN:
  shows
    ‹read_program_memory  address (RETURN state) = read_program_memory  address state›
    ‹read_static_memory   address (RETURN state) = read_static_memory   address state›
    ‹read_dynamic_memory  address (RETURN state) = read_dynamic_memory  address state›
  using memory_RETURN
  by(simp_all add: state_manipulation_decomp)

lemma (in Ironbark_world) read_call_memory_RETURN:
  assumes
    ‹typical_flags state›
    ‹∀n ∈ {0 .. 50} . address ≠ read_register call_frame_pointer_ref state + 51 + n›
```

```
    ‹address < (read_register call_frame_pointer_ref state)›
  shows
    ‹read_call_memory address (RETURN state) = read_call_memory address state›
  using assms
  by(simp add:
      instruction_impl_defs
      state_manipulation_reorder
      state_manipulation_simps)
```

The following are cases of RETURN over the initial state

```
lemma (in Ironbark_world) read_flag_RETURN_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (RETURN start_state) = 0›
    ‹get_end_call   (RETURN start_state) = 0›
    ‹get_end_return (RETURN start_state) = 1›
    ‹get_halt       (RETURN start_state) = 0›
    ‹get_error      (RETURN start_state) = 0›
  using assms
  by(simp_all add: read_flag_RETURN initial_state_simps)

lemma (in Ironbark_world) common_flags_RETURN_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹end_return_flags (RETURN start_state)›
  using assms
  by(simp add: read_flag_RETURN_initial_state)

lemma (in Ironbark_world) read_memory_RETURN_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory  address (RETURN start_state)
    = read_program_memory address start_state›

    ‹read_static_memory   address (RETURN start_state)
    = read_static_memory  address start_state›

    ‹read_dynamic_memory  address (RETURN start_state)
    = read_dynamic_memory address start_state›
  using assms
```

```
  by(simp_all add: read_memory_RETURN)

lemma (in Ironbark_world) read_last_ip_RETURN_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register last_instruction_pointer_ref (RETURN start_state)
    = (read_register instruction_pointer_ref start_state)›
  using assms
  by(simp add: read_last_ip_RETURN initial_state_simps)

lemma (in Ironbark_world) read_ip_RETURN_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register instruction_pointer_ref (RETURN start_state) = the None + 1›
  using assms
  by(simp add: read_ip_RETURN initial_state_simps read_call_memory_def)

end
```

D.3.33 Simplification Rules Over HALT

```
theory HALT_simps

imports
  HALT_decomposition

begin
```

As with most instructions, we provide various simplification rules for the *HALT* instruction. The various rules are primarily showing non-interference, or the specific changes to a specific value from the instruction. Where needed, we use the 'special' case where we assume the guards of the instruction will pass.

We order the rules based on the second operator, according to which would be applied last in the normalised form.

```
lemma (in Ironbark_world) flag_HALT:
  assumes
    ‹typical_flags state›
  shows
    ‹flag_state (HALT state) = flag_state (set_halt state)›
  using assms
  by(simp add: HALT_decomp_manipulation standard_error_def)

lemma (in Ironbark_world) register_HALT:
```

```
  ‹register_state (HALT state) = register_state state›
proof (cases ‹typical_flags state›)
  case True
  then show ?thesis
    by(simp_all add: HALT_decomp_manipulation state_manipulation_simps)
next
  case False
  then show ?thesis
    by(simp add: instruction_impl_defs state_manipulation_simps)
qed

lemma (in Ironbark_world) memory_HALT:
  ‹program_memory (HALT state) = program_memory state›
  ‹call_memory    (HALT state) = call_memory    state›
  ‹static_memory  (HALT state) = static_memory  state›
  ‹dynamic_memory (HALT state) = dynamic_memory state›
  ‹input_memory   (HALT state) = input_memory   state›
  ‹output_memory  (HALT state) = output_memory  state›
  by(simp_all add: instruction_impl_defs state_manipulation_simps)

lemma (in Ironbark_world) read_flag_HALT:
  assumes
    ‹typical_flags state›
  shows
    ‹get_end_jump   (HALT state) = 0›
    ‹get_end_call   (HALT state) = 0›
    ‹get_end_return (HALT state) = 0›
    ‹get_halt       (HALT state) = 1›
    ‹get_error      (HALT state) = 0›
  using assms
  by(auto simp add: HALT_decomp_manipulation state_manipulation_simps)

lemma (in Ironbark_world) common_flags_HALT:
  assumes
    ‹typical_flags state›
  shows
    ‹halt_flags (HALT state)›
  using assms
  by(simp add: read_flag_HALT)

lemma (in Ironbark_world) read_register_HALT:
  ‹read_register regID (HALT state) = read_register regID state›
  by(simp_all add: read_register_def register_HALT)

lemma (in Ironbark_world) read_memory_HALT:
  ‹read_program_memory  address (HALT state) = read_program_memory  address state›
  ‹read_call_memory     address (HALT state) = read_call_memory     address state›
  ‹read_static_memory   address (HALT state) = read_static_memory   address state›
  ‹read_dynamic_memory  address (HALT state) = read_dynamic_memory  address state›
```

```
  by(simp_all add: instruction_impl_defs state_manipulation_simps)
```

The following are lemmas of HALT on the initial state.

```
lemma (in Ironbark_world) read_flag_HALT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹get_end_jump   (HALT start_state) = 0›
    ‹get_end_call   (HALT start_state) = 0›
    ‹get_end_return (HALT start_state) = 0›
    ‹get_halt       (HALT start_state) = 1›
    ‹get_error      (HALT start_state) = 0›
  using assms
  by(auto simp add: read_flag_HALT initial_state_simps)

lemma (in Ironbark_world) common_flags_HALT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹halt_flags (HALT start_state)›
  using assms
  by(simp_all add: read_flag_HALT_initial_state)

lemma (in Ironbark_world) read_register_HALT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_register regID (HALT start_state) = read_register regID start_state›
  using assms
  by(simp_all add: read_register_HALT)

lemma (in Ironbark_world) read_memory_HALT_initial_state:
  fixes
    program :: ‹(64 word, 96 word) map›
  defines
    ‹start_state ≡ initial_state(|program_memory := program|)›
  shows
    ‹read_program_memory    address (HALT start_state)
      = read_program_memory address start_state›

    ‹read_call_memory       address (HALT start_state)
      = read_call_memory    address start_state›
```

```
    ‹read_static_memory     address (HALT start_state)
      = read_static_memory  address start_state›

    ‹read_dynamic_memory    address (HALT start_state)
      = read_dynamic_memory address start_state›
  using assms
  by(simp_all add: read_memory_HALT)
```

end

D.3.34 Simplification Rules Over ILLEGAL

theory *ILLEGAL_simps*

imports
ILLEGAL_decomposition

begin

For the *ILLEGAL* instruction, to avoid duplication, we just show equivalence to error. The simplification rules for error can then be applied.

```
lemma (in Ironbark_world) ILLEGAL_equiv_ERROR:
  ‹ILLEGAL state = ERROR0 state›
  by(simp_all add: instruction_impl_defs)
```

end

D.4 Individual Instruction Interfaces

D.4.1 ERROR Instruction Interface

theory *ERROR_top*

imports
ERROR_decomposition
ERROR_simps

begin

This file provides a single point for importing all the content at the instruction layer.

We also collect groups of related lemmas and bundle them for easier reference later.

```
lemmas (in Ironbark_world) ERROR0_decomp_all =
  ERROR_decomp_state(1)
  ERROR_decomp_manipulation(1)
  ERROR_decomp_manipulation2(1)

lemmas (in Ironbark_world) ERROR1_decomp_all =
  ERROR_decomp_state(2)
  ERROR_decomp_manipulation(2)
```

```
  ERROR_decomp_manipulation2(2)

lemmas (in Ironbark_world) ERROR_decomp_all =
  ERROR0_decomp_all
  ERROR1_decomp_all

lemmas (in Ironbark_world) ERROR_simps =
  flag_ERROR
  register_ERROR
  memory_ERROR
  read_flag_ERROR
  common_flags_ERROR
  read_register_ERROR
  read_memory_ERROR
  read_flag_ERROR_initial_state
  common_flags_ERROR_initial_state
  read_register_ERROR_initial_state
  read_memory_ERROR_initial_state

end
```

D.4.2 NOP Instruction Interface

```
theory NOP_top

imports
  NOP_decomposition
  NOP_simps

begin
```

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

```
lemmas (in Ironbark_world) NOP_decomp_all =
  NOP_decomp_state
  NOP_decomp_mixed
  NOP_decomp_manipulation

lemmas (in Ironbark_world) NOP_simps =
  flag_NOP
  register_NOP
  memory_NOP
  read_flag_NOP
  common_flags_NOP
  read_register_NOP
  read_ip_NOP
  read_cycles_NOP
  read_last_ip_NOP
  read_memory_NOP
```

```
  write_register_NOP_same
  read_flag_NOP_initial_state
  common_flags_NOP_initial_state
  read_memory_NOP_initial_state
  read_last_ip_NOP_initial_state
  read_ip_NOP_initial_state
```

end

D.4.3 LOAD_IMMEDIATE Instruction Interface

theory *LOAD_IMMEDIATE_top*

imports
```
  LOAD_IMMEDIATE_decomposition
  LOAD_IMMEDIATE_simps
```

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *LOAD_IMMEDIATE_decomp_all* =
```
  LOAD_IMMEDIATE_decomp_state
  LOAD_IMMEDIATE_decomp_mixed
  LOAD_IMMEDIATE_decomp_manipulation
```

lemmas (**in** *Ironbark_world*) *LOAD_IMMEDIATE_simps* =
```
  flag_LOAD_IMMEDIATE
  memory_LOAD_IMMEDIATE
  read_flag_LOAD_IMMEDIATE
  common_flags_LOAD_IMMEDIATE
  read_register_LOAD_IMMEDIATE_diff
  read_register_LOAD_IMMEDIATE_same
  read_ip_LOAD_IMMEDIATE
  read_cycles_LOAD_IMMEDIATE
  read_last_ip_LOAD_IMMEDIATE
  read_memory_LOAD_IMMEDIATE
  write_register_LOAD_IMMEDIATE_same
  read_flag_LOAD_IMMEDIATE_initial_state
  common_flags_LOAD_IMMEDIATE_initial_state
  read_memory_LOAD_IMMEDIATE_initial_state
  read_last_ip_LOAD_IMMEDIATE_initial_state
  read_ip_LOAD_IMMEDIATE_initial_state
```

end

D.4.4 LOAD_STATIC_DATA Instruction Interface

theory *LOAD_STATIC_DATA_top*

imports
LOAD_STATIC_DATA_decomposition
LOAD_STATIC_DATA_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (in *Ironbark_world)* *LOAD_STATIC_DATA_decomp_all* =
LOAD_STATIC_DATA_decomp_state
LOAD_STATIC_DATA_decomp_mixed
LOAD_STATIC_DATA_decomp_manipulation

lemmas (in *Ironbark_world)* *LOAD_STATIC_DATA_simps* =
flag_LOAD_STATIC_DATA
memory_LOAD_STATIC_DATA
read_flag_LOAD_STATIC_DATA
common_flags_LOAD_STATIC_DATA
read_register_LOAD_STATIC_DATA_diff
read_register_LOAD_STATIC_DATA_same
read_ip_LOAD_STATIC_DATA
read_cycles_LOAD_STATIC_DATA
read_last_ip_LOAD_STATIC_DATA
read_memory_LOAD_STATIC_DATA
write_register_LOAD_STATIC_DATA_same
read_flag_LOAD_STATIC_DATA_initial_state
common_flags_LOAD_STATIC_DATA_initial_state
read_memory_LOAD_STATIC_DATA_initial_state
read_last_ip_LOAD_STATIC_DATA_initial_state
read_ip_LOAD_STATIC_DATA_initial_state

end

D.4.5 STORE_STATIC_DATA Instruction Interface

theory *STORE_STATIC_DATA_top*

imports
STORE_STATIC_DATA_decomposition
STORE_STATIC_DATA_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (in *Ironbark_world)* *STORE_STATIC_DATA_decomp_all* =
STORE_STATIC_DATA_decomp_state

STORE_STATIC_DATA_decomp_mixed
STORE_STATIC_DATA_decomp_manipulation

lemmas (**in** *Ironbark_world*) *STORE_STATIC_DATA_simps* =
flag_STORE_STATIC_DATA
memory_STORE_STATIC_DATA_same
memory_STORE_STATIC_DATA_diff
read_flag_STORE_STATIC_DATA
common_flags_STORE_STATIC_DATA
read_register_STORE_STATIC_DATA
read_ip_STORE_STATIC_DATA
read_cycles_STORE_STATIC_DATA
read_last_ip_STORE_STATIC_DATA
read_memory_STORE_STATIC_DATA_same
read_memory_STORE_STATIC_DATA_diff
read_flag_STORE_STATIC_DATA_initial_state
common_flags_STORE_STATIC_DATA_initial_state
read_memory_STORE_STATIC_DATA_initial_state
read_last_ip_STORE_STATIC_DATA_initial_state
read_ip_STORE_STATIC_DATA_initial_state

end

D.4.6 LOAD_DYNAMIC_DATA Instruction Interface

theory *LOAD_DYNAMIC_DATA_top*

imports
LOAD_DYNAMIC_DATA_decomposition
LOAD_DYNAMIC_DATA_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *LOAD_DYNAMIC_DATA_decomp_all* =
LOAD_DYNAMIC_DATA_decomp_state
LOAD_DYNAMIC_DATA_decomp_mixed
LOAD_DYNAMIC_DATA_decomp_manipulation

lemmas (**in** *Ironbark_world*) *LOAD_DYNAMIC_DATA_simps* =
flag_LOAD_DYNAMIC_DATA
memory_LOAD_DYNAMIC_DATA
read_flag_LOAD_DYNAMIC_DATA
common_flags_LOAD_DYNAMIC_DATA
read_register_LOAD_DYNAMIC_DATA_diff
read_register_LOAD_DYNAMIC_DATA_same
read_ip_LOAD_DYNAMIC_DATA
read_cycles_LOAD_DYNAMIC_DATA

read_last_ip_LOAD_DYNAMIC_DATA
read_memory_LOAD_DYNAMIC_DATA
write_register_LOAD_DYNAMIC_DATA_same
read_flag_LOAD_DYNAMIC_DATA_initial_state
common_flags_LOAD_DYNAMIC_DATA_initial_state
read_memory_LOAD_DYNAMIC_DATA_initial_state
read_last_ip_LOAD_DYNAMIC_DATA_initial_state
read_ip_LOAD_DYNAMIC_DATA_initial_state

end

D.4.7 STORE_DYNAMIC_DATA Instruction Interface

theory *STORE_DYNAMIC_DATA_top*

imports
STORE_DYNAMIC_DATA_decomposition
STORE_DYNAMIC_DATA_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *STORE_DYNAMIC_DATA_decomp_all* =
STORE_DYNAMIC_DATA_decomp_state
STORE_DYNAMIC_DATA_decomp_mixed
STORE_DYNAMIC_DATA_decomp_manipulation

lemmas (**in** *Ironbark_world*) *STORE_DYNAMIC_DATA_simps* =
flag_STORE_DYNAMIC_DATA
memory_STORE_DYNAMIC_DATA_same
memory_STORE_DYNAMIC_DATA_diff
read_flag_STORE_DYNAMIC_DATA
common_flags_STORE_DYNAMIC_DATA
read_register_STORE_DYNAMIC_DATA
read_ip_STORE_DYNAMIC_DATA
read_cycles_STORE_DYNAMIC_DATA
read_last_ip_STORE_DYNAMIC_DATA
read_memory_STORE_DYNAMIC_DATA_same
read_memory_STORE_DYNAMIC_DATA_diff
read_flag_STORE_DYNAMIC_DATA_initial_state
common_flags_STORE_DYNAMIC_DATA_initial_state
read_memory_STORE_DYNAMIC_DATA_initial_state
read_last_ip_STORE_DYNAMIC_DATA_initial_state
read_ip_STORE_DYNAMIC_DATA_initial_state

end

D.4.8 LOAD_INPUT_DATA Instruction Interface

theory *LOAD_INPUT_DATA_top*

imports
LOAD_INPUT_DATA_decomposition
LOAD_INPUT_DATA_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (in *Ironbark_world)* *LOAD_INPUT_DATA_decomp_all* =
LOAD_INPUT_DATA_decomp_state
LOAD_INPUT_DATA_decomp_mixed
LOAD_INPUT_DATA_decomp_manipulation

lemmas (in *Ironbark_world)* *LOAD_INPUT_DATA_simps* =
flag_LOAD_INPUT_DATA
memory_LOAD_INPUT_DATA
read_flag_LOAD_INPUT_DATA
common_flags_LOAD_INPUT_DATA
read_register_LOAD_INPUT_DATA_diff
read_register_LOAD_INPUT_DATA_same
read_ip_LOAD_INPUT_DATA
read_cycles_LOAD_INPUT_DATA
read_last_ip_LOAD_INPUT_DATA
read_memory_LOAD_INPUT_DATA
write_register_LOAD_INPUT_DATA_same
read_flag_LOAD_INPUT_DATA_initial_state
common_flags_LOAD_INPUT_DATA_initial_state
read_memory_LOAD_INPUT_DATA_initial_state
read_last_ip_LOAD_INPUT_DATA_initial_state
read_ip_LOAD_INPUT_DATA_initial_state

end

D.4.9 STORE_OUTPUT_DATA Instruction Interface

theory *STORE_OUTPUT_DATA_top*

imports
STORE_OUTPUT_DATA_decomposition
STORE_OUTPUT_DATA_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *STORE_OUTPUT_DATA_decomp_all* =
  STORE_OUTPUT_DATA_decomp_state
  STORE_OUTPUT_DATA_decomp_mixed
  STORE_OUTPUT_DATA_decomp_manipulation

lemmas (**in** *Ironbark_world*) *STORE_OUTPUT_DATA_simps* =
  flag_STORE_OUTPUT_DATA
  memory_STORE_OUTPUT_DATA_same
  memory_STORE_OUTPUT_DATA_diff
  read_flag_STORE_OUTPUT_DATA
  common_flags_STORE_OUTPUT_DATA
  read_register_STORE_OUTPUT_DATA
  read_ip_STORE_OUTPUT_DATA
  read_cycles_STORE_OUTPUT_DATA
  read_last_ip_STORE_OUTPUT_DATA
  read_memory_STORE_OUTPUT_DATA_same
  read_memory_STORE_OUTPUT_DATA_diff
  read_flag_STORE_OUTPUT_DATA_initial_state
  common_flags_STORE_OUTPUT_DATA_initial_state
  read_memory_STORE_OUTPUT_DATA_initial_state
  read_last_ip_STORE_OUTPUT_DATA_initial_state
  read_ip_STORE_OUTPUT_DATA_initial_state

end

D.4.10 COPY Instruction Interface

theory *COPY_top*

imports
  COPY_decomposition
  COPY_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *COPY_decomp_all* =
  COPY_decomp_state
  COPY_decomp_mixed
  COPY_decomp_manipulation

lemmas (**in** *Ironbark_world*) *COPY_simps* =
  flag_COPY
  memory_COPY
  read_flag_COPY
  common_flags_COPY
  read_register_COPY_diff
  read_register_COPY_same

```
  read_ip_COPY
  read_cycles_COPY
  read_last_ip_COPY
  read_memory_COPY
  write_register_COPY_same
  read_flag_COPY_initial_state
  common_flags_COPY_initial_state
  read_memory_COPY_initial_state
  read_last_ip_COPY_initial_state
  read_ip_COPY_initial_state
```

end

D.4.11 ADD Instruction Interface

theory *ADD_top*

imports
```
  ADD_decomposition
  ADD_simps
```

begin

This file provides a single point for importing all the content for the ADD instruction. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (in *Ironbark_world) ADD_decomp_all =*
```
  ADD_decomp_state
  ADD_decomp_mixed
  ADD_decomp_manipulation
```

lemmas (in *Ironbark_world) ADD_simps =*
```
  flag_ADD
  memory_ADD
  read_flag_ADD
  common_flags_ADD
  read_register_ADD_diff
  read_register_ADD_same
  read_ip_ADD
  read_cycles_ADD
  read_last_ip_ADD
  read_memory_ADD
  write_register_ADD_same
  read_flag_ADD_initial_state
  common_flags_ADD_initial_state
  read_memory_ADD_initial_state
  read_last_ip_ADD_initial_state
  read_ip_ADD_initial_state
```

end

D.4.12 SUBTRACT Instruction Interface

theory *SUBTRACT_top*

imports
SUBTRACT_decomposition
SUBTRACT_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (in *Ironbark_world)* *SUBTRACT_decomp_all* =
SUBTRACT_decomp_state
SUBTRACT_decomp_mixed
SUBTRACT_decomp_manipulation

lemmas (in *Ironbark_world)* *SUBTRACT_simps* =
flag_SUBTRACT
memory_SUBTRACT
read_flag_SUBTRACT
common_flags_SUBTRACT
read_register_SUBTRACT_diff
read_register_SUBTRACT_same
read_ip_SUBTRACT
read_cycles_SUBTRACT
read_last_ip_SUBTRACT
read_memory_SUBTRACT
write_register_SUBTRACT_same
read_flag_SUBTRACT_initial_state
common_flags_SUBTRACT_initial_state
read_memory_SUBTRACT_initial_state
read_last_ip_SUBTRACT_initial_state
read_ip_SUBTRACT_initial_state

end

D.4.13 SHIFT_LEFT Instruction Interface

theory *SHIFT_LEFT_top*

imports
SHIFT_LEFT_decomposition
SHIFT_LEFT_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *SHIFT_LEFT_decomp_all* =
  SHIFT_LEFT_decomp_state
  SHIFT_LEFT_decomp_mixed
  SHIFT_LEFT_decomp_manipulation

lemmas (**in** *Ironbark_world*) *SHIFT_LEFT_simps* =
  flag_SHIFT_LEFT
  memory_SHIFT_LEFT
  read_flag_SHIFT_LEFT
  common_flags_SHIFT_LEFT
  read_register_SHIFT_LEFT_diff
  read_register_SHIFT_LEFT_same
  read_ip_SHIFT_LEFT
  read_cycles_SHIFT_LEFT
  read_last_ip_SHIFT_LEFT
  read_memory_SHIFT_LEFT
  write_register_SHIFT_LEFT_same
  read_flag_SHIFT_LEFT_initial_state
  common_flags_SHIFT_LEFT_initial_state
  read_memory_SHIFT_LEFT_initial_state
  read_last_ip_SHIFT_LEFT_initial_state
  read_ip_SHIFT_LEFT_initial_state

end

D.4.14 SHIFT_RIGHT Instruction Interface

theory *SHIFT_RIGHT_top*

imports
  SHIFT_RIGHT_decomposition
  SHIFT_RIGHT_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *SHIFT_RIGHT_decomp_all* =
  SHIFT_RIGHT_decomp_state
  SHIFT_RIGHT_decomp_mixed
  SHIFT_RIGHT_decomp_manipulation

lemmas (**in** *Ironbark_world*) *SHIFT_RIGHT_simps* =
  flag_SHIFT_RIGHT
  memory_SHIFT_RIGHT
  read_flag_SHIFT_RIGHT
  common_flags_SHIFT_RIGHT
  read_register_SHIFT_RIGHT_diff
  read_register_SHIFT_RIGHT_same

```
  read_ip_SHIFT_RIGHT
  read_cycles_SHIFT_RIGHT
  read_last_ip_SHIFT_RIGHT
  read_memory_SHIFT_RIGHT
  write_register_SHIFT_RIGHT_same
  read_flag_SHIFT_RIGHT_initial_state
  common_flags_SHIFT_RIGHT_initial_state
  read_memory_SHIFT_RIGHT_initial_state
  read_last_ip_SHIFT_RIGHT_initial_state
  read_ip_SHIFT_RIGHT_initial_state
```

end

D.4.15 BITWISE_AND Instruction Interface

theory *BITWISE_AND_top*

imports

```
  BITWISE_AND_decomposition
  BITWISE_AND_simps
```

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (in *Ironbark_world)* *BITWISE_AND_decomp_all* **=**

```
  BITWISE_AND_decomp_state
  BITWISE_AND_decomp_mixed
  BITWISE_AND_decomp_manipulation
```

lemmas (in *Ironbark_world)* *BITWISE_AND_simps* **=**

```
  flag_BITWISE_AND
  memory_BITWISE_AND
  read_flag_BITWISE_AND
  common_flags_BITWISE_AND
  read_register_BITWISE_AND_diff
  read_register_BITWISE_AND_same
  read_ip_BITWISE_AND
  read_cycles_BITWISE_AND
  read_last_ip_BITWISE_AND
  read_memory_BITWISE_AND
  write_register_BITWISE_AND_same
  read_flag_BITWISE_AND_initial_state
  common_flags_BITWISE_AND_initial_state
  read_memory_BITWISE_AND_initial_state
  read_last_ip_BITWISE_AND_initial_state
  read_ip_BITWISE_AND_initial_state
```

end

D.4.16 BITWISE_OR Instruction Interface

theory *BITWISE_OR_top*

imports
BITWISE_OR_decomposition
BITWISE_OR_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (in *Ironbark_world)* *BITWISE_OR_decomp_all* =
BITWISE_OR_decomp_state
BITWISE_OR_decomp_mixed
BITWISE_OR_decomp_manipulation

lemmas (in *Ironbark_world)* *BITWISE_OR_simps* =
flag_BITWISE_OR
memory_BITWISE_OR
read_flag_BITWISE_OR
common_flags_BITWISE_OR
read_register_BITWISE_OR_diff
read_register_BITWISE_OR_same
read_ip_BITWISE_OR
read_cycles_BITWISE_OR
read_last_ip_BITWISE_OR
read_memory_BITWISE_OR
write_register_BITWISE_OR_same
read_flag_BITWISE_OR_initial_state
common_flags_BITWISE_OR_initial_state
read_memory_BITWISE_OR_initial_state
read_last_ip_BITWISE_OR_initial_state
read_ip_BITWISE_OR_initial_state

end

D.4.17 BITWISE_XOR Instruction Interface

theory *BITWISE_XOR_top*

imports
BITWISE_XOR_decomposition
BITWISE_XOR_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *BITWISE_XOR_decomp_all* =
  BITWISE_XOR_decomp_state
  BITWISE_XOR_decomp_mixed
  BITWISE_XOR_decomp_manipulation

lemmas (**in** *Ironbark_world*) *BITWISE_XOR_simps* =
  flag_BITWISE_XOR
  memory_BITWISE_XOR
  read_flag_BITWISE_XOR
  common_flags_BITWISE_XOR
  read_register_BITWISE_XOR_diff
  read_register_BITWISE_XOR_same
  read_ip_BITWISE_XOR
  read_cycles_BITWISE_XOR
  read_last_ip_BITWISE_XOR
  read_memory_BITWISE_XOR
  write_register_BITWISE_XOR_same
  read_flag_BITWISE_XOR_initial_state
  common_flags_BITWISE_XOR_initial_state
  read_memory_BITWISE_XOR_initial_state
  read_last_ip_BITWISE_XOR_initial_state
  read_ip_BITWISE_XOR_initial_state

end

D.4.18 BITWISE_NAND Instruction Interface

theory *BITWISE_NAND_top*

imports
  BITWISE_NAND_decomposition
  BITWISE_NAND_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *BITWISE_NAND_decomp_all* =
  BITWISE_NAND_decomp_state
  BITWISE_NAND_decomp_mixed
  BITWISE_NAND_decomp_manipulation

lemmas (**in** *Ironbark_world*) *BITWISE_NAND_simps* =
  flag_BITWISE_NAND
  memory_BITWISE_NAND
  read_flag_BITWISE_NAND
  common_flags_BITWISE_NAND
  read_register_BITWISE_NAND_diff
  read_register_BITWISE_NAND_same

```
  read_ip_BITWISE_NAND
  read_cycles_BITWISE_NAND
  read_last_ip_BITWISE_NAND
  read_memory_BITWISE_NAND
  write_register_BITWISE_NAND_same
  read_flag_BITWISE_NAND_initial_state
  common_flags_BITWISE_NAND_initial_state
  read_memory_BITWISE_NAND_initial_state
  read_last_ip_BITWISE_NAND_initial_state
  read_ip_BITWISE_NAND_initial_state
```

end

D.4.19 BITWISE_NOT Instruction Interface

theory *BITWISE_NOT_top*

imports

```
  BITWISE_NOT_decomposition
  BITWISE_NOT_simps
```

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (in *Ironbark_world) BITWISE_NOT_decomp_all =*

```
  BITWISE_NOT_decomp_state
  BITWISE_NOT_decomp_mixed
  BITWISE_NOT_decomp_manipulation
```

lemmas (in *Ironbark_world) BITWISE_NOT_simps =*

```
  flag_BITWISE_NOT
  memory_BITWISE_NOT
  read_flag_BITWISE_NOT
  common_flags_BITWISE_NOT
  read_register_BITWISE_NOT_diff
  read_register_BITWISE_NOT_same
  read_ip_BITWISE_NOT
  read_cycles_BITWISE_NOT
  read_last_ip_BITWISE_NOT
  read_memory_BITWISE_NOT
  write_register_BITWISE_NOT_same
  read_flag_BITWISE_NOT_initial_state
  common_flags_BITWISE_NOT_initial_state
  read_memory_BITWISE_NOT_initial_state
  read_last_ip_BITWISE_NOT_initial_state
  read_ip_BITWISE_NOT_initial_state
```

end

D.4.20 LESS_THAN Instruction Interface

theory *LESS_THAN_top*

imports
LESS_THAN_decomposition
LESS_THAN_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (in *Ironbark_world)* *LESS_THAN_decomp_all* =
LESS_THAN_decomp_state
LESS_THAN_decomp_mixed
LESS_THAN_decomp_manipulation

lemmas (in *Ironbark_world)* *LESS_THAN_simps* =
flag_LESS_THAN
memory_LESS_THAN
read_flag_LESS_THAN
common_flags_LESS_THAN
read_register_LESS_THAN_diff
read_register_LESS_THAN_same
read_ip_LESS_THAN
read_cycles_LESS_THAN
read_last_ip_LESS_THAN
read_memory_LESS_THAN
write_register_LESS_THAN_same
read_flag_LESS_THAN_initial_state
common_flags_LESS_THAN_initial_state
read_memory_LESS_THAN_initial_state
read_last_ip_LESS_THAN_initial_state
read_ip_LESS_THAN_initial_state

end

D.4.21 GREATER_THAN Instruction Interface

theory *GREATER_THAN_top*

imports
GREATER_THAN_decomposition
GREATER_THAN_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *GREATER_THAN_decomp_all* =
  GREATER_THAN_decomp_state
  GREATER_THAN_decomp_mixed
  GREATER_THAN_decomp_manipulation

lemmas (**in** *Ironbark_world*) *GREATER_THAN_simps* =
  flag_GREATER_THAN
  memory_GREATER_THAN
  read_flag_GREATER_THAN
  common_flags_GREATER_THAN
  read_register_GREATER_THAN_diff
  read_register_GREATER_THAN_same
  read_ip_GREATER_THAN
  read_cycles_GREATER_THAN
  read_last_ip_GREATER_THAN
  read_memory_GREATER_THAN
  write_register_GREATER_THAN_same
  read_flag_GREATER_THAN_initial_state
  common_flags_GREATER_THAN_initial_state
  read_memory_GREATER_THAN_initial_state
  read_last_ip_GREATER_THAN_initial_state
  read_ip_GREATER_THAN_initial_state

end

D.4.22 EQUALS Instruction Interface

theory *EQUALS_top*

imports
  EQUALS_decomposition
  EQUALS_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *EQUALS_decomp_all* =
  EQUALS_decomp_state
  EQUALS_decomp_mixed
  EQUALS_decomp_manipulation

lemmas (**in** *Ironbark_world*) *EQUALS_simps* =
  flag_EQUALS
  memory_EQUALS
  read_flag_EQUALS
  common_flags_EQUALS
  read_register_EQUALS_diff
  read_register_EQUALS_same

read_ip_EQUALS
read_cycles_EQUALS
read_last_ip_EQUALS
read_memory_EQUALS
write_register_EQUALS_same
read_flag_EQUALS_initial_state
common_flags_EQUALS_initial_state
read_memory_EQUALS_initial_state
read_last_ip_EQUALS_initial_state
read_ip_EQUALS_initial_state

end

D.4.23 NOT_EQUALS Instruction Interface

theory *NOT_EQUALS_top*

imports
NOT_EQUALS_decomposition
NOT_EQUALS_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (in *Ironbark_world)* *NOT_EQUALS_decomp_all* =
NOT_EQUALS_decomp_state
NOT_EQUALS_decomp_mixed
NOT_EQUALS_decomp_manipulation

lemmas (in *Ironbark_world)* *NOT_EQUALS_simps* =
flag_NOT_EQUALS
memory_NOT_EQUALS
read_flag_NOT_EQUALS
common_flags_NOT_EQUALS
read_register_NOT_EQUALS_diff
read_register_NOT_EQUALS_same
read_ip_NOT_EQUALS
read_cycles_NOT_EQUALS
read_last_ip_NOT_EQUALS
read_memory_NOT_EQUALS
write_register_NOT_EQUALS_same
read_flag_NOT_EQUALS_initial_state
common_flags_NOT_EQUALS_initial_state
read_memory_NOT_EQUALS_initial_state
read_last_ip_NOT_EQUALS_initial_state
read_ip_NOT_EQUALS_initial_state

end

D.4.24 RANDOMISE Instruction Interface

theory *RANDOMISE_top*

imports
RANDOMISE_decomposition
RANDOMISE_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (in *Ironbark_world)* *RANDOMISE_decomp_all* =
RANDOMISE_decomp_state
RANDOMISE_decomp_mixed
RANDOMISE_decomp_manipulation

lemmas (in *Ironbark_world)* *RANDOMISE_simps* =
flag_RANDOMISE
memory_RANDOMISE
read_flag_RANDOMISE
common_flags_RANDOMISE
read_register_RANDOMISE_diff
read_register_RANDOMISE_same
read_ip_RANDOMISE
read_cycles_RANDOMISE
read_last_ip_RANDOMISE
read_memory_RANDOMISE
write_register_RANDOMISE_same
read_flag_RANDOMISE_initial_state
common_flags_RANDOMISE_initial_state
read_memory_RANDOMISE_initial_state
read_last_ip_RANDOMISE_initial_state
read_ip_RANDOMISE_initial_state

end

D.4.25 END_JUMP Instruction Interface

theory *END_JUMP_top*

imports
END_JUMP_decomposition
END_JUMP_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *END_JUMP_decomp_all* =
  END_JUMP_decomp_state_jump
  END_JUMP_decomp_mixed_jump
  END_JUMP_decomp_manipulation_jump
  END_JUMP_decomp_state_fall
  END_JUMP_decomp_mixed_fall
  END_JUMP_decomp_manipulation_fall

lemmas (**in** *Ironbark_world*) *END_JUMP_simps* =
  flag_END_JUMP_fall
  memory_END_JUMP
  read_flag_END_JUMP_fall
  read_flag_END_JUMP_jump
  common_flags_END_JUMP_fall
  common_flags_END_JUMP_jump
  read_register_END_JUMP
  read_ip_END_JUMP_fall
  read_ip_END_JUMP_jump
  read_cycles_END_JUMP_fall
  read_cycles_END_JUMP_jump
  read_last_ip_END_JUMP_fall
  read_last_ip_END_JUMP_jump
  read_memory_END_JUMP
  read_flag_END_JUMP_initial_state
  common_flags_END_JUMP_initial_state
  read_memory_END_JUMP_initial_state
  read_last_ip_END_JUMP_initial_state
  read_ip_END_JUMP_initial_state

end

D.4.26 END_JUMP_STRICT Instruction Interface

theory *END_JUMP_STRICT_top*

imports
  END_JUMP_STRICT_decomposition
  END_JUMP_STRICT_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *END_JUMP_STRICT_decomp_all* =
  END_JUMP_STRICT_decomp_state
  END_JUMP_STRICT_decomp_mixed
  END_JUMP_STRICT_decomp_manipulation

lemmas (**in** *Ironbark_world*) *END_JUMP_STRICT_simps* =

flag_END_JUMP_STRICT
memory_END_JUMP_STRICT
read_flag_END_JUMP_STRICT
common_flags_END_JUMP_STRICT
read_register_END_JUMP_STRICT
read_ip_END_JUMP_STRICT
read_cycles_END_JUMP_STRICT
read_last_ip_END_JUMP_STRICT
read_memory_END_JUMP_STRICT
END_JUMP_STRICT_initial_state_errors

end

D.4.27 JUMP Instruction Interface

theory *JUMP_top*

imports
JUMP_decomposition
JUMP_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (in *Ironbark_world)* *JUMP_decomp_all* **=**
JUMP_decomp_state
JUMP_decomp_mixed
JUMP_decomp_manipulation

lemmas (in *Ironbark_world)* *JUMP_simps* **=**
flag_JUMP
memory_JUMP
read_flag_JUMP
common_flags_JUMP
read_register_JUMP
read_ip_JUMP
read_cycles_JUMP
read_last_ip_JUMP
read_memory_JUMP
read_flag_JUMP_initial_state
common_flags_JUMP_initial_state
read_memory_JUMP_initial_state
read_last_ip_JUMP_initial_state
read_ip_JUMP_initial_state

end

D.4.28 CONDITIONAL_JUMP Instruction Interface

theory *CONDITIONAL_JUMP_top*

imports
CONDITIONAL_JUMP_decomposition
CONDITIONAL_JUMP_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (in *Ironbark_world)* *CONDITIONAL_JUMP_decomp_all* =
CONDITIONAL_JUMP_decomp_state
CONDITIONAL_JUMP_decomp_mixed
CONDITIONAL_JUMP_decomp_manipulation

lemmas (in *Ironbark_world)* *CONDITIONAL_JUMP_simps* =
flag_CONDITIONAL_JUMP
memory_CONDITIONAL_JUMP
read_flag_CONDITIONAL_JUMP
common_flags_CONDITIONAL_JUMP_fall
common_flags_CONDITIONAL_JUMP_jump
read_register_CONDITIONAL_JUMP
read_ip_CONDITIONAL_JUMP_jump
read_ip_CONDITIONAL_JUMP_fall
read_cycles_CONDITIONAL_JUMP
read_last_ip_CONDITIONAL_JUMP
read_memory_CONDITIONAL_JUMP
read_flag_CONDITIONAL_JUMP_initial_state
common_flags_CONDITIONAL_JUMP_initial_state
read_memory_CONDITIONAL_JUMP_initial_state
read_last_ip_CONDITIONAL_JUMP_initial_state
read_ip_CONDITIONAL_JUMP_initial_state

end

D.4.29 END_CALL Instruction Interface

theory *END_CALL_top*

imports
END_CALL_decomposition
END_CALL_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *END_CALL_decomp_all* =
  END_CALL_decomp_state
  END_CALL_decomp_mixed
  END_CALL_decomp_manipulation

lemmas (**in** *Ironbark_world*) *END_CALL_simps* =
  flag_END_CALL
  memory_END_CALL
  read_flag_END_CALL
  common_flags_END_CALL
  read_register_END_CALL
  read_ip_END_CALL
  read_cycles_END_CALL
  read_last_ip_END_CALL
  read_memory_END_CALL
  END_CALL_initial_state_errors

end

D.4.30 CALL Instruction Interface

theory *CALL_top*

imports
  CALL_decomposition
  CALL_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *CALL_decomp_all* =
  CALL_decomp_state
  CALL_decomp_mixed
  CALL_decomp_manipulation

lemmas (**in** *Ironbark_world*) *CALL_simps* =
  memory_CALL
  read_flag_CALL
  common_flags_CALL
  read_register_CALL
  read_ip_CALL
  read_cycles_CALL
  read_last_ip_CALL
  read_call_pointer_CALL
  read_memory_CALL
  read_call_memory_CALL
  read_flag_CALL_initial_state
  common_flags_CALL_initial_state

read_memory_CALL_initial_state
read_last_ip_CALL_initial_state
read_ip_CALL_initial_state

end

D.4.31 END_RETURN Instruction Interface

theory *END_RETURN_top*

imports
END_RETURN_decomposition
END_RETURN_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *END_RETURN_decomp_all* =
END_RETURN_decomp_state
END_RETURN_decomp_mixed
END_RETURN_decomp_manipulation

lemmas (**in** *Ironbark_world*) *END_RETURN_simps* =
flag_END_RETURN
memory_END_RETURN
read_flag_END_RETURN
common_flags_END_RETURN
read_register_END_RETURN
read_ip_END_RETURN
read_cycles_END_RETURN
read_last_ip_END_RETURN
read_memory_END_RETURN
END_RETURN_initial_state_errors

end

D.4.32 RETURN Instruction Interface

theory *RETURN_top*

imports
RETURN_decomposition
RETURN_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *RETURN_decomp_all* =
  RETURN_decomp_state
  RETURN_decomp_mixed
  RETURN_decomp_manipulation

lemmas (**in** *Ironbark_world*) *RETURN_simps* =
  memory_RETURN
  read_flag_RETURN
  common_flags_RETURN
  read_ip_RETURN
  read_cycles_RETURN
  read_last_ip_RETURN
  read_memory_RETURN
  read_call_memory_RETURN
  read_flag_RETURN_initial_state
  common_flags_RETURN_initial_state
  read_memory_RETURN_initial_state
  read_last_ip_RETURN_initial_state
  read_ip_RETURN_initial_state

end

D.4.33 HALT Instruction Interface

theory *HALT_top*

imports
  HALT_decomposition
  HALT_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *HALT_decomp_all* =
  HALT_decomp_state
  HALT_decomp_manipulation

lemmas (**in** *Ironbark_world*) *HALT_simps* =
  flag_HALT
  register_HALT
  memory_HALT
  read_flag_HALT
  common_flags_HALT
  read_register_HALT
  read_memory_HALT
  read_flag_HALT_initial_state
  common_flags_HALT_initial_state
  read_register_HALT_initial_state

read_memory_HALT_initial_state

end

D.4.34 ILLEGAL Instruction Interface

theory *ILLEGAL_top*

imports
ILLEGAL_decomposition
ILLEGAL_simps

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (**in** *Ironbark_world*) *ILLEGAL_decomp_all* =
ILLEGAL_decomp_state
ILLEGAL_decomp_manipulation
ILLEGAL_decomp_manipulation2

lemmas (**in** *Ironbark_world*) *ILLEGAL_simps* =
ILLEGAL_equiv_ERROR

end

D.5 Instruction Interface Layer

theory *instruction_top*

imports
instruction_auxiliary
ERROR_top
NOP_top
LOAD_IMMEDIATE_top
LOAD_STATIC_DATA_top
STORE_STATIC_DATA_top
LOAD_DYNAMIC_DATA_top
STORE_DYNAMIC_DATA_top
LOAD_INPUT_DATA_top
STORE_OUTPUT_DATA_top
COPY_top
ADD_top
SUBTRACT_top
SHIFT_LEFT_top
SHIFT_RIGHT_top
BITWISE_AND_top

BITWISE_OR_top
BITWISE_XOR_top
BITWISE_NAND_top
BITWISE_NOT_top
LESS_THAN_top
GREATER_THAN_top
EQUALS_top
NOT_EQUALS_top
RANDOMISE_top
END_JUMP_top
END_JUMP_STRICT_top
JUMP_top
CONDITIONAL_JUMP_top
END_CALL_top
CALL_top
END_RETURN_top
RETURN_top
HALT_top
ILLEGAL_top

begin

This file provides a single point for importing all the content at the instruction layer. We also collect groups of related lemmas and bundle them for easier reference later.

lemmas (in *Ironbark_world)* *instruction_decomp_manipulation* =
ERROR_decomp_manipulation
NOP_decomp_manipulation
LOAD_IMMEDIATE_decomp_manipulation
LOAD_STATIC_DATA_decomp_manipulation
STORE_STATIC_DATA_decomp_manipulation
LOAD_DYNAMIC_DATA_decomp_manipulation
STORE_DYNAMIC_DATA_decomp_manipulation
LOAD_INPUT_DATA_decomp_manipulation
STORE_OUTPUT_DATA_decomp_manipulation
COPY_decomp_manipulation
ADD_decomp_manipulation
SUBTRACT_decomp_manipulation
SHIFT_LEFT_decomp_manipulation
SHIFT_RIGHT_decomp_manipulation
BITWISE_AND_decomp_manipulation
BITWISE_OR_decomp_manipulation
BITWISE_XOR_decomp_manipulation
BITWISE_NAND_decomp_manipulation
BITWISE_NOT_decomp_manipulation
LESS_THAN_decomp_manipulation
GREATER_THAN_decomp_manipulation
EQUALS_decomp_manipulation
NOT_EQUALS_decomp_manipulation
RANDOMISE_decomp_manipulation

END_JUMP_decomp_manipulation_jump
END_JUMP_decomp_manipulation_fall
END_JUMP_STRICT_decomp_manipulation
JUMP_decomp_manipulation
CONDITIONAL_JUMP_decomp_manipulation
END_CALL_decomp_manipulation
CALL_decomp_manipulation
END_RETURN_decomp_manipulation
RETURN_decomp_manipulation
HALT_decomp_manipulation
ILLEGAL_decomp_manipulation

lemmas (**in** *Ironbark_world*) *instruction_decomp_state* =
ERROR_decomp_state
NOP_decomp_state
LOAD_IMMEDIATE_decomp_state
LOAD_STATIC_DATA_decomp_state
STORE_STATIC_DATA_decomp_state
LOAD_DYNAMIC_DATA_decomp_state
STORE_DYNAMIC_DATA_decomp_state
LOAD_INPUT_DATA_decomp_state
STORE_OUTPUT_DATA_decomp_state
COPY_decomp_state
ADD_decomp_state
SUBTRACT_decomp_state
SHIFT_LEFT_decomp_state
SHIFT_RIGHT_decomp_state
BITWISE_AND_decomp_state
BITWISE_OR_decomp_state
BITWISE_XOR_decomp_state
BITWISE_NAND_decomp_state
BITWISE_NOT_decomp_state
LESS_THAN_decomp_state
GREATER_THAN_decomp_state
EQUALS_decomp_state
NOT_EQUALS_decomp_state
RANDOMISE_decomp_state
END_JUMP_decomp_state_jump
END_JUMP_decomp_state_fall
END_JUMP_STRICT_decomp_state
JUMP_decomp_state
CONDITIONAL_JUMP_decomp_state
END_CALL_decomp_state
CALL_decomp_state
END_RETURN_decomp_state
RETURN_decomp_state
HALT_decomp_state
ILLEGAL_decomp_state

lemmas (**in** *Ironbark_world*) *instruction_decomp_mixed* =
permission_simps

NOP_decomp_mixed
LOAD_IMMEDIATE_decomp_mixed
LOAD_STATIC_DATA_decomp_mixed
STORE_STATIC_DATA_decomp_mixed
LOAD_DYNAMIC_DATA_decomp_mixed
STORE_DYNAMIC_DATA_decomp_mixed
LOAD_INPUT_DATA_decomp_mixed
STORE_OUTPUT_DATA_decomp_mixed
COPY_decomp_mixed
ADD_decomp_mixed
SUBTRACT_decomp_mixed
SHIFT_LEFT_decomp_mixed
SHIFT_RIGHT_decomp_mixed
BITWISE_AND_decomp_mixed
BITWISE_OR_decomp_mixed
BITWISE_XOR_decomp_mixed
BITWISE_NAND_decomp_mixed
BITWISE_NOT_decomp_mixed
LESS_THAN_decomp_mixed
GREATER_THAN_decomp_mixed
EQUALS_decomp_mixed
NOT_EQUALS_decomp_mixed
RANDOMISE_decomp_mixed
END_JUMP_decomp_mixed_jump
END_JUMP_decomp_mixed_fall
END_JUMP_STRICT_decomp_mixed
JUMP_decomp_mixed
CONDITIONAL_JUMP_decomp_mixed
END_CALL_decomp_mixed
CALL_decomp_mixed
END_RETURN_decomp_mixed
RETURN_decomp_mixed

lemmas (**in** *Ironbark_world*) *instruction_simps* =
initial_state_simps
permission_simps

NOP_simps
LOAD_IMMEDIATE_simps
LOAD_STATIC_DATA_simps
STORE_STATIC_DATA_simps
LOAD_DYNAMIC_DATA_simps
STORE_DYNAMIC_DATA_simps
LOAD_INPUT_DATA_simps
STORE_OUTPUT_DATA_simps
COPY_simps

ADD_simps
SUBTRACT_simps
SHIFT_LEFT_simps
SHIFT_RIGHT_simps
BITWISE_AND_simps
BITWISE_OR_simps
BITWISE_XOR_simps
BITWISE_NAND_simps
BITWISE_NOT_simps
LESS_THAN_simps
GREATER_THAN_simps
EQUALS_simps
NOT_EQUALS_simps
RANDOMISE_simps
END_JUMP_simps
END_JUMP_STRICT_simps
JUMP_simps
CONDITIONAL_JUMP_simps
END_CALL_simps
CALL_simps
END_RETURN_simps
RETURN_simps
HALT_simps
ILLEGAL_simps
ERROR_simps

end

E Execution Level Lemmas

E.1 Properties of Sequential Execution Definitions

theory *execution_auxiliary*

imports
"Ironbark_instruction.instruction_top"
execution_implementations

begin

We collect up the various definitions relating to the decode operation for easy reference later. These are used to decompose a decode operation into the parameters of the execute instruction function.

lemmas *decode_decomp =*
decode_instruction_def
get_opcode_def
get_reg1_def
get_reg2_def
get_reg3_def

```
  get_immediate_def
  slice_def
  slice1_def
```

The execute multiple instructions is defined with execute next single instruction being applied to the state after the remaining multiple instructions. We show here that this is the same as having the next instruction single instruction done first instead.

```
lemma (in Ironbark_world) emi_Suc:
  ‹execute_multiple_instructions state (Suc num_instructions)
  = execute_multiple_instructions (execute_next_instruction state) num_instructions›
proof(induct ‹num_instructions› arbitrary: ‹state›)
  case 0
  then show ?case
    by(auto)
next
  case (Suc num_instructions)
  then show ?case
    by(auto)
qed
```

For convenience of future proofs, we provide the following lemma for converting the opcode to instruction to be executed.

```
lemma (in Ironbark_world) execute_instruction_simps:
  ‹execute_instruction (0x00, reg1, reg2, reg3, immediate) state
  = ERROR0 state›

  ‹execute_instruction (0x01, reg1, reg2, reg3, immediate) state
  = NOP state›

  ‹execute_instruction (0x02, reg1, reg2, reg3, immediate) state
  = LOAD_IMMEDIATE reg1 immediate state›

  ‹execute_instruction (0x03, reg1, reg2, reg3, immediate) state
  = LOAD_STATIC_DATA reg1 reg2 state›

  ‹execute_instruction (0x04, reg1, reg2, reg3, immediate) state
  = STORE_STATIC_DATA reg1 reg2 state›

  ‹execute_instruction (0x05, reg1, reg2, reg3, immediate) state
  = LOAD_DYNAMIC_DATA reg1 reg2 state›

  ‹execute_instruction (0x06, reg1, reg2, reg3, immediate) state
  = STORE_DYNAMIC_DATA reg1 reg2 state›

  ‹execute_instruction (0x07, reg1, reg2, reg3, immediate) state
  = LOAD_INPUT_DATA reg1 reg2 state›

  ‹execute_instruction (0x08, reg1, reg2, reg3, immediate) state
  = STORE_OUTPUT_DATA reg1 reg2 state›
```

```
‹execute_instruction (0x09, reg1, reg2, reg3, immediate) state
= COPY reg1 reg2 state›

‹execute_instruction (0x0A, reg1, reg2, reg3, immediate) state
= ADD reg1 reg2 reg3 state›

‹execute_instruction (0x0B, reg1, reg2, reg3, immediate) state
= SUBTRACT reg1 reg2 reg3 state›

‹execute_instruction (0x0C, reg1, reg2, reg3, immediate) state
= SHIFT_LEFT reg1 reg2 reg3 state›

‹execute_instruction (0x0D, reg1, reg2, reg3, immediate) state
= SHIFT_RIGHT reg1 reg2 reg3 state›

‹execute_instruction (0x0E, reg1, reg2, reg3, immediate) state
= BITWISE_AND reg1 reg2 reg3 state›

‹execute_instruction (0x0F, reg1, reg2, reg3, immediate) state
= BITWISE_OR reg1 reg2 reg3 state›

‹execute_instruction (0x10, reg1, reg2, reg3, immediate) state
= BITWISE_XOR reg1 reg2 reg3 state›

‹execute_instruction (0x11, reg1, reg2, reg3, immediate) state
= BITWISE_NAND reg1 reg2 reg3 state›

‹execute_instruction (0x12, reg1, reg2, reg3, immediate) state
= BITWISE_NOT reg1 reg2 state›

‹execute_instruction (0x13, reg1, reg2, reg3, immediate) state
= LESS_THAN reg1 reg2 reg3 state›

‹execute_instruction (0x14, reg1, reg2, reg3, immediate) state
= GREATER_THAN reg1 reg2 reg3 state›

‹execute_instruction (0x15, reg1, reg2, reg3, immediate) state
= EQUALS reg1 reg2 reg3 state›

‹execute_instruction (0x16, reg1, reg2, reg3, immediate) state
= NOT_EQUALS reg1 reg2 reg3 state›

‹execute_instruction (0x17, reg1, reg2, reg3, immediate) state
= RANDOMISE reg1 state›

‹execute_instruction (0x18, reg1, reg2, reg3, immediate) state
= END_JUMP immediate state›
```

```
  ‹execute_instruction (0x19, reg1, reg2, reg3, immediate) state
  = END_JUMP_STRICT immediate state›

  ‹execute_instruction (0x1A, reg1, reg2, reg3, immediate) state
  = JUMP immediate state›

  ‹execute_instruction (0x1B, reg1, reg2, reg3, immediate) state
  = CONDITIONAL_JUMP reg1 immediate state›

  ‹execute_instruction (0x1C, reg1, reg2, reg3, immediate) state
  = END_CALL state›

  ‹execute_instruction (0x1D, reg1, reg2, reg3, immediate) state
  = CALL immediate state›

  ‹execute_instruction (0x1E, reg1, reg2, reg3, immediate) state
  = END_RETURN immediate state›

  ‹execute_instruction (0x1F, reg1, reg2, reg3, immediate) state
  = RETURN state›

  ‹execute_instruction (0x20, reg1, reg2, reg3, immediate) state
  = HALT state›

  ‹execute_instruction (0xFF, reg1, reg2, reg3, immediate) state
  = ERROR1 state›
  by(simp_all add: execute_instruction.simps)
```

The following two lemmas use different methods of showing which opcodes are illegal.

```
lemma (in Ironbark_world) execute_instruction_ILLEGAL:
  fixes
    opcode :: ‹8 word›
  assumes
    ‹opcode > 0x20›
    ‹opcode < 0xFF›
  shows
    ‹execute_instruction (opcode, reg1, reg2, reg3, immediate) state = ILLEGAL state›
proof -
  have
    ‹opcode ∉ {
      0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  10, 11, 12, 13, 14, 15, 16,
      17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 0xFF
    }›
    using assms
    by(auto)
  then show ?thesis
    by(simp add: execute_instruction.simps)
qed
```

```
lemma (in Ironbark_world) execute_instruction_ILLEGAL':
  fixes
    opcode :: ‹8 word›
  assumes
    ‹opcode ∉ {
      0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  10, 11, 12, 13, 14, 15, 16,
      17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 0xFF
    }›
  shows
    ‹execute_instruction (opcode, reg1, reg2, reg3, immediate) state ≡ ILLEGAL state›
  using assms
  by(auto simp add: execute_instruction.simps)
```

The following lemmas are for understanding control flow in Ironbark, and the import of the flags for control flow instructions.

```
lemma decodeE:
  assumes
    ‹get_opcode (fetch_instruction state) = opcode›
  obtains reg1 reg2 reg3 immediate where
    ‹decode_instruction (fetch_instruction state) = (opcode, reg1, reg2, reg3, immediate)›
  using assms
  by(simp add: decode_instruction_def)

lemma (in Ironbark_world) typical_flags_backsubstitute:
  fixes
    state1 state2 :: ‹sequential_state›
  defines
    ‹instruction ≡ fetch_instruction state1›
  defines
    ‹opcode ≡ get_opcode instruction›
    and ‹addr ≡ get_immediate instruction›
  assumes
    two_after_one: ‹state2 = execute_next_instruction state1›
    and flags2: ‹typical_flags state2›
  shows
    ‹ if opcode = END_JUMP_opcode ∧ get_end_jump state1 = 1 then
        end_jump_flags state1 ∧
        read_register last_instruction_pointer_ref state1 = addr
      else if opcode = END_JUMP_STRICT_opcode then
        end_jump_flags state1 ∧
        read_register last_instruction_pointer_ref state1 = addr
      else if opcode = END_CALL_opcode then
        end_call_flags state1 ∧
        get_end_jump state1 = 0
      else if opcode = END_RETURN_opcode then
        end_return_flags state1 ∧
        read_register last_instruction_pointer_ref state1 = addr
      else
        typical_flags state1›
```

```
proof(cases ‹get_halt state1 = 1 ∨ get_error state1 = 1›)
  case True
  then show ?thesis
    using two_after_one flags2
    unfolding execute_next_instruction_def
    by(simp add: flag_normalisation)
next
  case False
  then show ?thesis
  proof(cases ‹opcode = 24›)
    case True
    moreover assume ‹¬ (get_halt state1 = 1 ∨ get_error state1 = 1)›
    ultimately show ?thesis
    using two_after_one flags2
    unfolding execute_next_instruction_def
    apply(simp add: flag_normalisation)
    unfolding opcode_def instruction_def addr_def
    apply(elim decodeE)
    apply(simp add: execute_instruction_simps)
    apply(simp add: instruction_impl_defs)
    apply(simp split: if_splits)
    apply(simp_all add: state_manipulation_simps)
    apply(intro impI)
    apply(simp)
    apply(simp add: decode_instruction_def)
    done
  next
    case False
    moreover assume ‹¬ (get_halt state1 = 1 ∨ get_error state1 = 1)›
    ultimately show ?thesis
    proof(cases ‹opcode = 25›)
      case True
      moreover assume ‹¬ (get_halt state1 = 1 ∨ get_error state1 = 1)›
      ultimately show ?thesis
        using two_after_one flags2
        unfolding execute_next_instruction_def
        apply(simp add: flag_normalisation)
        unfolding opcode_def instruction_def addr_def
        apply(elim decodeE)
        apply(simp add: execute_instruction_simps)
        apply(simp add: instruction_impl_defs)
        apply(simp split: if_splits)
        apply(simp_all add: state_manipulation_simps)
        apply(simp add: decode_instruction_def)
        done
    next
      case False
      moreover assume ‹opcode ≠ 24›
      moreover assume ‹¬ (get_halt state1 = 1 ∨ get_error state1 = 1)›
```

```
ultimately show ?thesis
proof(cases ‹opcode = 28›)
  case True
  moreover assume ‹¬ (get_halt state1 = 1 ∨ get_error state1 = 1)›
  ultimately show ?thesis
    using two_after_one flags2
    unfolding execute_next_instruction_def
    apply(simp add: flag_normalisation)
    unfolding opcode_def instruction_def addr_def
    apply(elim decodeE)
    apply(simp add: execute_instruction_simps)
    apply(simp add: instruction_impl_defs)
    apply(simp split: if_splits)
    apply(simp_all add: state_manipulation_simps)
    done
next
  case False
  moreover assume ‹opcode ≠ 24›
  moreover assume ‹opcode ≠ 25›
  moreover assume ‹¬ (get_halt state1 = 1 ∨ get_error state1 = 1)›
  ultimately show ?thesis
  proof(cases ‹opcode = 30›)
    case True
    moreover assume ‹¬ (get_halt state1 = 1 ∨ get_error state1 = 1)›
    ultimately show ?thesis
      using two_after_one flags2
      unfolding execute_next_instruction_def
      apply(simp add: flag_normalisation)
      unfolding opcode_def instruction_def addr_def
      apply(elim decodeE)
      apply(simp add: execute_instruction_simps)
      apply(simp add: instruction_impl_defs)
      apply(simp split: if_splits)
      apply(simp_all add: state_manipulation_simps)
      apply(simp add: decode_instruction_def)
      done
  next
    case False
    moreover assume ‹opcode ≠ 24›
    moreover assume ‹opcode ≠ 25›
    moreover assume ‹opcode ≠ 28›
    moreover assume ‹¬ (get_halt state1 = 1 ∨ get_error state1 = 1)›
    ultimately show ?thesis
      using two_after_one flags2
      apply(simp)
      unfolding execute_next_instruction_def
      apply(simp add: flag_normalisation)
      using opcode_bound [of ‹fetch_instruction state1›]
      unfolding opcode_def instruction_def
```

```
            apply (simp add: word_le_decr)
            apply (elim disjE)
            apply (simp_all add: decode_instruction_def)
            apply (simp_all add: execute_instruction_simps execute_instruction_ILLEGAL)
            apply (simp_all add: instruction_simps)
            apply (simp_all add: instruction_impl_defs)
            apply (simp_all split: if_splits)
            apply (simp_all add: state_manipulation_simps)
            done
        qed
      qed
    qed
  qed
qed

lemma (in Ironbark_world) endjump_set_backsubstitute:
  fixes
    state1 state2 :: ‹sequential_state›
  defines
    ‹instruction ≡ fetch_instruction state1›
  defines
    ‹opcode ≡ get_opcode instruction›
  defines
    ‹addr ≡ get_immediate instruction›
  assumes
    two_after_one: ‹state2 = execute_next_instruction state1›
  assumes
    flags2: ‹end_jump_flags state2›
  shows
    ‹ (opcode = JUMP_opcode ∨ opcode = CONDITIONAL_JUMP_opcode) ∧
      typical_flags state1›
proof (cases ‹get_halt state1 = 1 ∨ get_error state1 = 1›)
  case True
  then show ?thesis
    using two_after_one flags2
    unfolding execute_next_instruction_def
    by (simp)
next
  case False
  then show ?thesis
    using two_after_one flags2
    unfolding execute_next_instruction_def opcode_def instruction_def
    apply (simp add: flag_normalisation)
    using opcode_bound [of ‹fetch_instruction state1›]
    apply (simp add: word_le_decr)
    apply (elim disjE)
    apply (simp_all add: decode_instruction_def)
    apply (simp_all add: execute_instruction_simps execute_instruction_ILLEGAL)
    apply (simp_all add: instruction_simps)
```

```
    apply(simp_all add: instruction_impl_defs)
    apply(simp_all split: if_splits)
    apply(simp_all add: state_manipulation_simps)
    done
qed

lemma (in Ironbark_world) endcall_set_backsubstitute:
  fixes
    state1 state2 :: ‹sequential_state›
  defines
    ‹instruction ≡ fetch_instruction state1›
  defines
    ‹opcode ≡ get_opcode instruction›
  defines
    ‹addr ≡ get_immediate instruction›
  assumes
    two_after_one: ‹state2 = execute_next_instruction state1›
  assumes
      flags2: ‹end_call_flags state2›
  shows
    ‹ (opcode = CALL_opcode) ∧
      typical_flags state1›
proof(cases ‹get_halt state1 = 1 ∨ get_error state1 = 1›)
  case True
  then show ?thesis
    using two_after_one flags2
    unfolding execute_next_instruction_def
    by (simp)
next
  case False
  then show ?thesis
    using two_after_one flags2
    unfolding execute_next_instruction_def opcode_def instruction_def
    apply(simp add: flag_normalisation)
    using opcode_bound [of ‹fetch_instruction state1›]
    apply(simp add: word_le_decr)
    apply(elim disjE)
    apply(simp_all add: decode_instruction_def)
    apply(simp_all add: execute_instruction_simps execute_instruction_ILLEGAL)
    apply(simp_all add: instruction_simps)
    apply(simp_all add: instruction_impl_defs)
    apply(simp_all split: if_splits)
    apply(simp_all add: state_manipulation_simps)
    done
qed

lemma (in Ironbark_world) endreturn_set_backsubstitute:
  fixes
    state1 state2 :: ‹sequential_state›
```

```
  defines
    ‹instruction ≡ fetch_instruction state1›
  defines
    ‹opcode ≡ get_opcode instruction›
  defines
    ‹addr ≡ get_immediate instruction›
  assumes
    two_after_one: ‹state2 = execute_next_instruction state1›
  assumes
    flags2: ‹end_return_flags state2›
  shows
    ‹ (opcode = RETURN_opcode) ∧
      typical_flags state1›
proof(cases ‹get_halt state1 = 1 ∨ get_error state1 = 1›)
  case True
  then show ?thesis
    using two_after_one flags2
    unfolding execute_next_instruction_def
    by (simp)
next
  case False
  then show ?thesis
    using two_after_one flags2
    unfolding execute_next_instruction_def opcode_def instruction_def
    apply(simp add: flag_normalisation)
    using opcode_bound [of ‹fetch_instruction state1›]
    apply(simp add: word_le_decr)
    apply(elim disjE)
    apply(simp_all add: decode_instruction_def)
    apply(simp_all add: execute_instruction_simps execute_instruction_ILLEGAL)
    apply(simp_all add: instruction_simps)
    apply(simp_all add: instruction_impl_defs)
    apply(simp_all split: if_splits)
    apply(simp_all add: state_manipulation_simps)
    done
qed

lemma (in Ironbark_world) call_frame_pointer_evolution:
  fixes
    state :: sequential_state
  defines
    ‹instruction ≡ fetch_instruction state›
  defines
    ‹opcode ≡ get_opcode instruction›
  shows
    ‹read_register call_frame_pointer_ref (execute_next_instruction state)
    = (
        if typical_flags state ∧ opcode = CALL_opcode then
          read_register call_frame_pointer_ref state + 67
```

```
        else if typical_flags state ∧ opcode = RETURN_opcode then
          read_register call_frame_pointer_ref state - 67
        else
          read_register call_frame_pointer_ref state
      )›
proof(cases ‹typical_flags state›)
  case True
  then show ?thesis
    apply(simp add: execute_next_instruction_def opcode_def instruction_def)
    apply(intro conjI impI)
    apply(elim decodeE, simp add: execute_instruction.simps)
    apply(simp (no_asm) add: instruction_impl_defs)
    apply(simp add: state_manipulation_simps Let_def)
    apply(elim decodeE, simp add: execute_instruction.simps)
    apply(simp (no_asm) add: instruction_impl_defs)
    apply(simp add: state_manipulation_simps Let_def)
    using opcode_bound [of ‹instruction›]
    apply(fold instruction_def)
    unfolding fetch_instruction_def decode_instruction_def
    apply(simp add: word_le_decr)
    apply(elim disjE)
    apply(simp_all add: execute_instruction_simps execute_instruction_ILLEGAL)
    apply(simp_all add: instruction_simps)
    apply(simp_all add: instruction_impl_defs)
    apply(simp_all add: state_manipulation_simps Let_def)
    done
next
  case False
  moreover from this have
    ‹¬ (typical_flags state ∧ opcode = 29)›
    ‹¬ (typical_flags state ∧ opcode = 31)›
    by(safe)
  ultimately show ?thesis
    apply(simp only: if_splits)
    apply(simp add: flag_normalisation)
    apply(cases ‹get_halt state = 1 ∨ get_error state = 1›)
    apply(simp_all add: execute_next_instruction_def)
    apply(simp add: flag_normalisation)
    apply(fold instruction_def)
    unfolding fetch_instruction_def decode_instruction_def
    apply(simp add: word_le_decr)
    apply(elim disjE)
    apply(simp_all add: execute_instruction.simps)
    apply(simp_all add: instruction_simps)
    apply(safe)
    apply(simp_all add: instruction_simps)
    apply(simp_all (no_asm) add: instruction_impl_defs)
    apply(simp_all add: state_manipulation_simps)
    done
```

qed

lemma (**in** *Ironbark_world*) *call_frame_pointer_backsubstitute:*
fixes
state1 state2 :: sequential_state
defines
‹*instruction1* ≡ *fetch_instruction state1*›
defines
‹*opcode1* ≡ *get_opcode instruction1*›
defines
‹*call_frame_pointer1* ≡ *read_register call_frame_pointer_ref state1*›
and ‹*call_frame_pointer2* ≡ *read_register call_frame_pointer_ref state2*›
assumes
two_after_one:
‹*state2 = execute_next_instruction state1*›
shows
‹*read_register call_frame_pointer_ref state1*
= (
if typical_flags state1 ∧ opcode1 = CALL_opcode then
call_frame_pointer2 - 67
else if typical_flags state1 ∧ opcode1 = RETURN_opcode then
call_frame_pointer2 + 67
else
call_frame_pointer2
)›
unfolding *call_frame_pointer2_def two_after_one call_frame_pointer_evolution*
apply(*fold instruction1_def opcode1_def call_frame_pointer1_def*)
apply(*simp only: split: if_splits*)
apply(*intro impI conjI*)
apply(*simp_all*)
done

lemma (**in** *Ironbark_world*) *call_memory_evolution:*
fixes
state1 state2 :: sequential_state
and *addr* :: ‹*64 word*›
defines
‹*instruction1* ≡ *fetch_instruction state1*›
defines
‹*opcode1* ≡ *get_opcode instruction1*›
defines
‹*call_memory1* ≡ *read_call_memory addr state1*›
and ‹*call_memory2* ≡ *read_call_memory addr state2*›
assumes
two_after_one:
‹*state2 = execute_next_instruction state1*›
shows
‹*call_memory2*
= (

```
          if typical_flags state1 ∧ opcode1 = CALL_opcode then
            read_call_memory addr (backup_registers_before_call state1)
          else
            call_memory1
        )›
proof(cases ‹get_halt state1 = 1 ∨ get_error state1 = 1›)
  case True
  then show ?thesis
    unfolding call_memory2_def two_after_one
    apply(simp add: execute_next_instruction_def call_memory1_def)
    apply(auto)
    done
next
  case False
  then show ?thesis
    unfolding call_memory2_def two_after_one
    apply(simp add: execute_next_instruction_def call_memory1_def)
    using opcode_bound [of ‹instruction1›]
    apply(fold instruction1_def)
    unfolding fetch_instruction_def decode_instruction_def
    apply(fold opcode1_def)
    apply(simp add: word_le_decr)
    apply(elim disjE)
    apply(simp_all add: execute_instruction.simps)
    apply(simp_all (no_asm) add: instruction_impl_defs)
    apply(simp_all add: state_manipulation_simps)
    apply(simp_all add: state_manipulation_reorder)
    apply(simp_all add: state_manipulation_simps)
    done
qed
```

The last thing we show here is some trivial properties of our definition of finish_function_trace. Namely, we show that a trace is not empty, the start of the trace is the starting state, and the starting state is in the trace.

```
lemma (in Ironbark_world) trace_rules:
  ‹finish_function_trace [x] y ≠ []›
  ‹hd (finish_function_trace [x] y) = x›
  ‹x ∈ set (finish_function_trace [x] y)›
proof (induct ‹y›)
  case 0
  then show
   ‹finish_function_trace [x] 0 ≠ []›
   ‹hd (finish_function_trace [x] 0) = x›
   ‹x ∈ set (finish_function_trace [x] 0)›
   by(simp_all)
next
  case (Suc y)
  then show
    ‹finish_function_trace [x] (Suc y) ≠ []›
```

```
    ‹hd (finish_function_trace [x] (Suc y)) = x›
    ‹x ∈ set (finish_function_trace [x] (Suc y))›
    by(simp_all add: Let_def)
qed

end
```

E.2 Our Invariant 'Toolkit'

theory *invariant_toolkit*

imports
security_properties

begin

This file is for developing our invariant toolkit. The primary lemma is presented first, which shows that an invariant holds over a program if:

- the instruction pointer is in the program
- assuming the instruction pointer is in the program, the program hasn't changed, and the invariant was true in the previous state, then:
  - the instruction pointer stays in the program
  - the program won't change
  - the invariant is true in the next state

Note that the condition of the program not changing can be trivially shown from the security properties, but the other conditions require understanding the behaviour of the program and invariant being shown.

```
lemma (in Ironbark_world) inv_satI:
  fixes starting_state :: sequential_state
  assumes
      a0:
        ‹read_register
          instruction_pointer_ref
          (starting_state(|program_memory := program|))
        ∈ dom program›
    and
      a1:
        ‹predicates (starting_state (| program_memory := program |))›
    and
      a2:
        ‹⋀ state.
        ⟦
          (read_register instruction_pointer_ref state) ∈ dom program;
          program_memory state = program; predicates state
```

```
        ⟧
        ⟹ predicates (execute_next_instruction state)›
    and
      a3:
        ‹⋀ state.
        ⟦
          (read_register instruction_pointer_ref state) ∈ dom program;
          program_memory state = program;
          predicates state
        ⟧
        ⟹ (read_register instruction_pointer_ref (execute_next_instruction state))
          ∈ dom program›
  shows
    ‹predicates
      (execute_multiple_instructions
        (starting_state(|program_memory := program|))
        num_instructions)›

    ‹read_register
      instruction_pointer_ref
      (execute_multiple_instructions
        (starting_state(|program_memory := program|))
        num_instructions)
    ∈ dom program›
proof (induct ‹num_instructions›)
  case 0
  then show
    ‹predicates
      (execute_multiple_instructions (starting_state(|program_memory := program|)) 0)›

    ‹read_register
      instruction_pointer_ref
      (execute_multiple_instructions (starting_state(|program_memory := program|)) 0)
    ∈ dom program›
    using a0 a1
    by(simp_all)
next
  case (Suc num_instructions)
  then show
    ‹predicates
      (execute_multiple_instructions
        (starting_state(|program_memory := program|))
        (Suc num_instructions))›
    apply(simp)
    apply(subst a2)
    apply(simp_all add: program_memory_immutable')
    done
next
  case (Suc num_instructions)
```

```
  then show
    ‹read_register
      instruction_pointer_ref
      (execute_multiple_instructions
        (starting_state(|program_memory := program|))
        (Suc num_instructions))
    ∈ dom program›
    apply(simp)
    apply(subst a3)
    apply(simp_all add: program_memory_immutable')
    done
qed

end
```

E.3 Execution Interface Layer

theory *execution_top*

imports
execution_auxiliary
security_properties
invariant_toolkit

begin

This file provides a single point for importing all the content at the execution layer.
As this is also the final layer, this serves as the import point for the Ironbark processor.

end

F Example Programs

F.1 Correctness of basic program

theory *add_example*

imports
"Ironbark_execution.execution_top"

begin

This file just contains a proof of a specific program doing 1 + 1 add will get the correct result (i.e. 2), along with some other claims about the state.

The proof is structured to first have what instructions will be executed, before finally showing the proof goal.

```
lemma (in Ironbark_world) add_example:
  fixes
```

```
    state :: sequential_state
  defines
    ‹program ≡
      Map.empty
      (
        0  ↦ 0x020100000000000000000001,  — r01 = 0x1
        1  ↦ 0x0A020101000000000000000,  — r02 = r01 + r01
        2  ↦ 0x20000000000000000000000   — halt
      )›
  defines
    ‹check_state ≡ λ state .
        read_register instruction_pointer_ref state = 2
      ∧ halt_flags state
      ∧ read_register r02_ref state = 2
      ∧ read_register cycles_register_ref state = 2 * common_instruction_duration›
  shows
    ‹check_state
      (execute_multiple_instructions
        (initial_state(|program_memory := program|))
        3)›
proof -

  have
    ‹read_register instruction_pointer_ref (initial_state(|program_memory := program|))
    = 0›
    by(simp add: program_def initial_state_def read_register_decomp)

  have
    ‹execute_next_instruction (initial_state(|program_memory := program|))
    = LOAD_IMMEDIATE r01_ref 1 (initial_state(|program_memory := program|))›
    apply(simp add: execute_next_instruction_def)
    apply(simp add: state_manipulation_simps)
    apply(simp add: fetch_instruction_def)
    apply(simp add: state_manipulation_simps)
    apply(simp add: read_program_memory_def)
    apply(subst program_def)
    apply(simp add: decode_instruction_def)
    apply(simp add: state_manipulation_decomp)
    apply(simp add: execute_instruction_simps)
    done

  then have
    ‹execute_next_instruction
      (execute_next_instruction
        (initial_state(|program_memory := program|)))
    = ADD
        r02_ref
        r01_ref
        r01_ref
```

```
      (LOAD_IMMEDIATE r01_ref 1 (initial_state⦇program_memory := program⦈))⟩
  apply(simp add: execute_next_instruction_def)
  apply(simp add: instruction_simps)
  apply(simp add: fetch_instruction_def)
  apply(simp add: instruction_simps)
  apply(simp add: state_manipulation_simps)
  apply(simp add: read_program_memory_def)
  apply(subst program_def)
  apply(simp add: decode_instruction_def)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: execute_instruction_simps)
  done

then have
  ‹execute_next_instruction
    (execute_next_instruction
      (execute_next_instruction
        (initial_state⦇program_memory := program⦈)))
  = HALT
      (ADD
        r02_ref
        r01_ref
        r01_ref
        (LOAD_IMMEDIATE r01_ref 1 (initial_state⦇program_memory := program⦈)))⟩
  apply(simp)
  apply(simp (no_asm) add: execute_next_instruction_def)
  apply(simp add: instruction_simps)
  apply(simp add: fetch_instruction_def)
  apply(simp add: instruction_simps)
  apply(simp add: state_manipulation_simps)
  apply(simp add: read_program_memory_def)
  apply(subst program_def)
  apply(simp add: decode_instruction_def)
  apply(simp add: state_manipulation_decomp)
  apply(simp add: execute_instruction_simps)
  done

then show
  ?thesis
  apply(simp add: execute_multiple_instructions_def)
  apply(simp add: check_state_def)
  apply(intro conjI)
  apply(simp_all add: HALT_decomp_manipulation instruction_simps)
  apply(simp_all add: state_manipulation_simps)
  apply(simp_all add: instruction_simps)
  apply(simp_all add: state_manipulation_simps)
  done
qed
```

end

F.2 Pieacemeal Correctness of Fibonacci as a Program

theory *fibonacci_invariant_piecemeal*

imports
"Ironbark_execution.execution_top"

begin

This file contains a proof of correctness of a function which calculates Fibonacci numbers. Specifically, this proof is done as a series of individual smaller proofs, rather than a single large proof with many subcomponents.

We start by defining the Fibonacci function from a pure mathematical point of view, which we will later show is equivalent to the return value of our function (modulo 2^{64}).

```
function fib :: ‹nat ⇒ nat›
  where
  ‹fib 0 = 1›
| ‹fib 1 = 1›
| ‹fib (n+2) = fib n + fib (n+1)›
  apply(atomize_elim)
  apply(arith)
  apply(auto)
  done
termination by lexicographic_order
```

The following lemma tries to exhaust the various ways in which Suc can appear with the fib function and rewrite it in terms of +. This is done because we we prefer to work with the + operator.

```
lemma fib_stuff:
  ‹fib (n+3) = fib (n+1) + fib (n+2)›
  ‹fib (Suc (Suc n)) = fib n + fib (n+1)›
  ‹fib (Suc (Suc (Suc n))) = fib (n+1) + fib (n+2)›
  ‹fib (Suc 0) = 1›
  ‹fib (Suc (Suc n)) = fib (n+2)›
  ‹fib (Suc n) = fib (n+1)›
  ‹fib (Suc (n+1)) = fib (n+2)›
  ‹fib ((Suc n)+1) = fib (n+2)›
  ‹fib (Suc (n+2)) = fib (n+3)›
  ‹fib (n+1+2) = fib (n+3)›
  ‹fib (Suc (Suc 0)) = 2›
  ‹fib (n+0) = fib n›
  ‹fib (0+n) = fib n›
  ‹fib (Suc (Suc (Suc n))) = fib (n+3)›
  using fib.simps
  by(simp_all add: numeral_3_eq_3)
```

We found it is sometimes useful to be able to lookup the answers for early Fibonacci numbers, so

we collect a few of the early answers for convenience.

```
lemma fib_answers:
  ‹fib (Suc 0) = 1›
  ‹fib 2 = 2›
  ‹fib (Suc 1) = 2›
  ‹fib (Suc (Suc 0)) = 2›
  ‹fib 3 = 3›
  ‹fib 4 = 5›
  using fib.simps by(simp_all add: numeral_3_eq_3 numeral_Bit0)
```

We now define the program, which we will show correctly implements the Fibonacci function.

```
definition program :: ‹64 word ⇒ 96 word option›
  where
    ‹program ≡ Map.empty
    (
        0   ↦ 0x022100000000000000000001, — c01 = 0x1
        1   ↦ 0x020300000000000000000000, — r03 = 0x0
        2   ↦ 0x020000000000000000000001, — r00 = 0x1
        3   ↦ 0x020100000000000000000001, — r01 = 0x1
        4   ↦ 0x020200000000000000000002, — r02 = 0x2
        5   ↦ 0x18000000000000000000000C, — end_jump 0xC
        6   ↦ 0x150403300000000000000000, — r04 = r03 == arg00
        7   ↦ 0x1B040000000000000000000d, — if (r04) jump 0x000000000000000d
        8   ↦ 0x090001000000000000000000, — r00 = r01
        9   ↦ 0x090102000000000000000000, — r01 = r02
        10  ↦ 0x0A020001000000000000000, — r02 = r00 + r01
        11  ↦ 0x0A030321000000000000000, — r03 = r03 + c01
        12  ↦ 0x1A000000000000000000005, — jump 0x0000000000000005
        13  ↦ 0x18000000000000000000007, — end_jump 0x7
        14  ↦ 0x094000000000000000000000, — ret00 = r00
        15  ↦ 0x200000000000000000000000  — halt
    )›
```

We also define what it means for our program to be 'correct'. Because we use an inductive process with the following as our invariant, the approach we have taken is to state what we expect to be true at every step of the program. We've annotated this with the assembly being executed at each step, and what we really mean by correctness is what we show in the halt step (at the end of the definition), and the rest of these are intermediate results useful for the inductive process.

```
definition
  invariant :: ‹sequential_state ⇒ bool›
  where
    ‹invariant state ≡
      — c01 = 0x1
      (read_register instruction_pointer_ref state = 0
        ⟶ typical_flags state)

      — r03 = 0 //counter
      ∧ (read_register instruction_pointer_ref state = 1 ⟶
```

```
(
  typical_flags state
  ∧ read_register c01_ref state = 1
))

— r00 = 1 //fib(counter)
∧ (read_register instruction_pointer_ref state = 2 ⟶
(
  typical_flags state
  ∧ read_register c01_ref state = 1
  ∧ read_register r03_ref state = 0
  ∧ read_register r03_ref state ≤ read_register arg00_ref state
))

— r01 = 1 //fib(counter+1)
∧ (read_register instruction_pointer_ref state = 3 ⟶
(
  typical_flags state
  ∧ read_register c01_ref state = 1
  ∧ read_register r03_ref state = 0
  ∧ read_register r03_ref state ≤ read_register arg00_ref state
  ∧ read_register r00_ref state
    = of_nat (fib (unat (read_register r03_ref state)))
))

— r02 = 2 //fib(counter+2))
∧ (read_register instruction_pointer_ref state = 4 ⟶
(
  typical_flags state
  ∧ read_register c01_ref state = 1
  ∧ read_register r03_ref state = 0
  ∧ read_register r03_ref state ≤ read_register arg00_ref state
  ∧ read_register r00_ref state
    = of_nat (fib (unat (read_register r03_ref state)))
  ∧ read_register r01_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 1))
))

— end jump
∧ (read_register instruction_pointer_ref state = 5 ⟶
(
  (typical_flags state ∨ end_jump_flags state)
  ∧ (typical_flags state
    ⟶ (read_register last_instruction_pointer_ref state = 4))
  ∧ (end_jump_flags state
    ⟶ (read_register last_instruction_pointer_ref state = 12))
  ∧ read_register c01_ref state = 1
  ∧ read_register r03_ref state ≤ read_register arg00_ref state
  ∧ read_register r00_ref state
```

```
    = of_nat (fib (unat (read_register r03_ref state)))
  ∧ read_register r01_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 1))
  ∧ read_register r02_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 2))
))

— r04 = r03 == arg00
∧ (read_register instruction_pointer_ref state = 6 ⟶
(
  typical_flags state
  ∧ read_register c01_ref state = 1
  ∧ read_register r03_ref state ≤ read_register arg00_ref state
  ∧ read_register r00_ref state
    = of_nat (fib (unat (read_register r03_ref state)))
  ∧ read_register r01_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 1))
  ∧ read_register r02_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 2))
))

— if (r04) jump finish //if (counter == n) finish
∧ (read_register instruction_pointer_ref state = 7 ⟶
(
  typical_flags state
  ∧ read_register c01_ref state = 1
  ∧ read_register r03_ref state ≤ read_register arg00_ref state
  ∧ read_register r00_ref state
    = of_nat (fib (unat (read_register r03_ref state)))
  ∧ read_register r01_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 1))
  ∧ read_register r02_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 2))
  ∧ ((read_register r03_ref state = read_register arg00_ref state)
    ⟷ (read_register r04_ref state = 1))
  ∧ ((read_register r03_ref state ≠ read_register arg00_ref state)
    ⟷ (read_register r04_ref state = 0))
  ∧ (read_register r04_ref state = 1 ∨ read_register r04_ref state = 0)
))

— r00 = r01 //fib(counter) <- fib(counter+1)
∧ (read_register instruction_pointer_ref state = 8 ⟶
(
  typical_flags state
  ∧ read_register c01_ref state = 1
  ∧ read_register r03_ref state < read_register arg00_ref state
  ∧ read_register r00_ref state
    = of_nat (fib (unat (read_register r03_ref state)))
  ∧ read_register r01_ref state
```

```
    = of_nat (fib (unat (read_register r03_ref state) + 1))
  ∧ read_register r02_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 2))
))

— r01 = r02 //fib(counter+1) <- fib(counter+2)
∧ (read_register instruction_pointer_ref state = 9 ⟶
(
  typical_flags state
  ∧ read_register c01_ref state = 1
  ∧ read_register r03_ref state < read_register arg00_ref state
  ∧ read_register r00_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 1))
  ∧ read_register r01_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 1))
  ∧ read_register r02_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 2))
))

— r02 = r00 + r01 //fib(counter+2) <- fib(counter+3)
∧ (read_register instruction_pointer_ref state = 10 ⟶
(
  typical_flags state
  ∧ read_register c01_ref state = 1
  ∧ read_register r03_ref state < read_register arg00_ref state
  ∧ read_register r00_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 1))
  ∧ read_register r01_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 2))
  ∧ read_register r02_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 2))
))

— r03 = r03 + c01 //counter++
∧ (read_register instruction_pointer_ref state = 11 ⟶
(
  typical_flags state
  ∧ read_register c01_ref state = 1
  ∧ read_register r03_ref state < read_register arg00_ref state
  ∧ read_register r00_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 1))
  ∧ read_register r01_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 2))
  ∧ read_register r02_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 3))
))

— jump main loop
∧ (read_register instruction_pointer_ref state = 12 ⟶
```

```
(
  typical_flags state
  ∧ read_register c01_ref state = 1
  ∧ read_register r03_ref state ≤ read_register arg00_ref state
  ∧ read_register r00_ref state
    = of_nat (fib (unat (read_register r03_ref state)))
  ∧ read_register r01_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 1))
  ∧ read_register r02_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 2))
))

— end jump
∧ (read_register instruction_pointer_ref state = 13 ⟶
(
  end_jump_flags state
  ∧ read_register last_instruction_pointer_ref state = 7
  ∧ read_register c01_ref state = 1
  ∧ read_register r00_ref state
    = of_nat (fib (unat (read_register r03_ref state)))
  ∧ read_register r01_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 1))
  ∧ read_register r02_ref state
    = of_nat (fib (unat (read_register r03_ref state) + 2))
  ∧ read_register r03_ref state = read_register arg00_ref state
))

— ret0 = r0
∧ (read_register instruction_pointer_ref state = 14 ⟶
(
  typical_flags state
  ∧ read_register r00_ref state
    = of_nat (fib (unat (read_register r03_ref state)))
  ∧ read_register r03_ref state = read_register arg00_ref state
))

— halt
∧ (read_register instruction_pointer_ref state = 15 ⟶
(
  get_end_return state = 0
  ∧ get_end_call state = 0
  ∧ get_end_jump state = 0
  ∧ get_error state = 0
  ∧ read_register r03_ref state = read_register arg00_ref state
  ∧ read_register ret00_ref state
    = of_nat (fib (unat (read_register arg00_ref state)))
))›
```

The next series of lemmas show that the invariant holds after the next instruction has been executed,

assuming the invariant was true at the start. We do this for each step in the program. The structure of each of these proofs is to first show how the instruction will change the state, and then show that the state change will satisfy the invariant conditions.

Showing the invariant was true at the start will be done later.

```
lemma (in Ironbark_world) fib_equivalence_invariant_step0:
  fixes
    current_state :: sequential_state
  defines
    ‹last_step_invariant ≡ λ state .
      typical_flags state›
  defines
    ‹step_invariant ≡ λ next_state .
      typical_flags next_state ∧
      read_register c01_ref next_state = 1 ∧
      read_register instruction_pointer_ref next_state = 1›
  assumes
    previous_step_invariant:
      ‹last_step_invariant current_state›
      ‹read_register instruction_pointer_ref current_state = 0›
      ‹program_memory current_state = program›
  shows
    ‹step_invariant (execute_next_instruction current_state)›
proof -
  have
    ‹execute_next_instruction current_state =
      current_state
      (|
        register_state :=
          (register_state current_state)
          (
            c01_ref
              ↦ 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref current_state)
                + common_instruction_duration,
            last_instruction_pointer_ref
              ↦ 0,
            instruction_pointer_ref
              ↦ 1
          )
      |)›
    using previous_step_invariant
    apply(simp add: last_step_invariant_def)
    apply(unfold execute_next_instruction_def)
    apply(unfold fetch_instruction_def)
    apply(simp add: read_register_decomp read_program_memory_def)
    apply(simp (no_asm) add: program_def)
    apply(simp add: decode_decomp)
```

```
    apply (simp add: execute_instruction_simps)
    apply (simp add: instruction_decomp_mixed)
    apply (simp add: state_manipulation_decomp)
    apply (simp add: fun_upd_twist)
    done
  then have
    ‹read_register instruction_pointer_ref (execute_next_instruction current_state) = 1›
    ‹typical_flags (execute_next_instruction current_state)›
    ‹read_register c01_ref (execute_next_instruction current_state) = 1›
    using previous_step_invariant
    by (simp_all add: last_step_invariant_def state_manipulation_decomp)
  then show ?thesis
    by (simp add: step_invariant_def)
qed

lemma (in Ironbark_world) fib_equivalence_invariant_step1:
  fixes
    current_state :: sequential_state
  defines
    ‹last_step_invariant ≡ λ state .
      typical_flags state ∧
      read_register c01_ref state = 1›
  defines
    ‹step_invariant ≡ λ next_state .
      typical_flags next_state ∧
      read_register c01_ref next_state = 1 ∧
      read_register instruction_pointer_ref next_state = 2 ∧
      read_register r03_ref next_state = 0 ∧
      read_register r03_ref next_state ≤ read_register arg00_ref next_state›
  assumes
    previous_step_invariant:
      ‹last_step_invariant current_state›
      ‹read_register instruction_pointer_ref current_state = 1›
      ‹program_memory current_state = program›
  shows
    ‹step_invariant (execute_next_instruction current_state)›
proof -
  have
    ‹typical_flags current_state›
    ‹read_register c01_ref current_state = 1›
    using previous_step_invariant
    by (simp_all add: last_step_invariant_def)
  moreover have
    ‹execute_next_instruction current_state = LOAD_IMMEDIATE 3 0 current_state›
  proof -
    have
      ‹decode_instruction (fetch_instruction current_state) = (2, 3, 0, 0, 0)›
      using previous_step_invariant
      unfolding fetch_instruction_def
```

```
    apply(simp add: read_program_memory_def)
    apply(simp (no_asm) add: program_def)
    apply(simp add: decode_decomp)
    done
  then show
    ‹execute_next_instruction current_state = LOAD_IMMEDIATE 3 0 current_state›
    using previous_step_invariant
    unfolding last_step_invariant_def
    unfolding execute_next_instruction_def
    by(simp add: execute_instruction_simps)
qed
ultimately have
  ‹execute_next_instruction current_state =
    current_state
    (|
      register_state :=
        (register_state current_state)
        (
          r03_ref
            ↦ 0,
          last_instruction_pointer_ref
            ↦ 1,
          instruction_pointer_ref
            ↦ 2,
          cycles_register_ref
            ↦ (read_register cycles_register_ref current_state)
              + common_instruction_duration
        )
    |)›
  using previous_step_invariant
  by(simp add: instruction_decomp_mixed)
then have
  ‹read_register instruction_pointer_ref (execute_next_instruction current_state)
  = 2›

  ‹typical_flags (execute_next_instruction current_state)›
  ‹read_register c01_ref (execute_next_instruction current_state) = 1›
  ‹read_register r03_ref (execute_next_instruction current_state) = 0›

  ‹read_register r03_ref (execute_next_instruction current_state)
  ≤ read_register arg00_ref (execute_next_instruction current_state)›
  using previous_step_invariant
  by(simp_all add: last_step_invariant_def state_manipulation_decomp)
then show ?thesis
  by(simp add: step_invariant_def)
qed

lemma (in Ironbark_world) fib_equivalence_invariant_step2:
  fixes
```

```
    current_state :: sequential_state
  defines
    ‹last_step_invariant ≡ λ state .
      typical_flags state ∧
      read_register c01_ref state = 1 ∧
      read_register r03_ref state = 0 ∧
      read_register r03_ref state ≤ read_register arg00_ref state›
  defines
    ‹next_state ≡ execute_next_instruction current_state›
  defines
    ‹step_invariant ≡ λ state .
      typical_flags state ∧
      read_register c01_ref state = 1 ∧
      read_register instruction_pointer_ref state = 3 ∧
      read_register r03_ref state = 0 ∧
      read_register r03_ref state ≤ read_register arg00_ref state ∧
      read_register r00_ref state = of_nat (fib (unat (read_register r03_ref state)))›
  assumes
    previous_step_invariant: ‹last_step_invariant current_state›
    and previous_step:  ‹read_register instruction_pointer_ref current_state = 2›
    and program_state:  ‹program_memory current_state = program›
  shows
    ‹step_invariant next_state›
proof -
  have
    ‹next_state =
      current_state
      (|
        register_state :=
          (register_state current_state)
          (
            r00_ref
              ↦ 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref current_state)
                + common_instruction_duration,
            last_instruction_pointer_ref
              ↦ 2,
            instruction_pointer_ref
              ↦ 3
          )
      |)›
    unfolding next_state_def
    using previous_step_invariant previous_step
    apply(simp add: last_step_invariant_def)
    apply(elim conjE)
    apply(unfold execute_next_instruction_def)
    apply(simp)
    apply(unfold fetch_instruction_def)
```

```
    apply(simp add: read_program_memory_def)
    apply(simp add: program_state)
    apply(simp add: program_def)
    apply(simp add: decode_decomp)
    apply(simp add: execute_instruction_simps)
    apply(simp add: instruction_decomp_mixed)
    apply(simp add: fun_upd_twist)
    done
  then have
    ‹read_register instruction_pointer_ref next_state = 3›
    ‹typical_flags next_state›
    ‹read_register c01_ref next_state = 1›
    ‹read_register r03_ref next_state = 0›
    ‹read_register r03_ref next_state ≤ read_register arg00_ref next_state›
    ‹read_register r00_ref next_state = 1›

    ‹read_register r00_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state)))›
    using previous_step_invariant program_state
    by(simp_all add: last_step_invariant_def state_manipulation_decomp)
  then show ?thesis
    by(simp add: step_invariant_def)
qed

lemma (in Ironbark_world) fib_equivalence_invariant_step3:
  fixes
    current_state :: sequential_state
  defines
    ‹last_step_invariant ≡ λ state .
      typical_flags state
      ∧ read_register c01_ref state = 1
      ∧ read_register r03_ref state = 0
      ∧ read_register r03_ref state ≤ read_register arg00_ref state
      ∧ read_register r00_ref state
        = of_nat (fib (unat (read_register r03_ref state)))›
  defines
    ‹next_state ≡ execute_next_instruction current_state›
  defines
    ‹step_invariant ≡ λ state .
      typical_flags state
      ∧ read_register c01_ref state = 1
      ∧ read_register instruction_pointer_ref state = 4
      ∧ read_register r03_ref state = 0
      ∧ read_register r03_ref state ≤ read_register arg00_ref state
      ∧ read_register r00_ref state
        = of_nat (fib (unat (read_register r03_ref state)))
      ∧ read_register r01_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 1))›
  assumes
```

```
    previous_step_invariant: ‹last_step_invariant current_state›
    and previous_step:  ‹read_register instruction_pointer_ref current_state = 3›
    and program_state:  ‹program_memory current_state = program›
  shows
    ‹step_invariant next_state›
proof -
  have
    ‹next_state =
      current_state
      (|
        register_state :=
          (register_state current_state)
          (
            r01_ref
              ↦ 1,
            cycles_register_ref
              ↦ (read_register cycles_register_ref current_state)
                + common_instruction_duration,
            last_instruction_pointer_ref
              ↦ 3,
            instruction_pointer_ref
              ↦ 4
          )
      |)›
    unfolding next_state_def
    using previous_step_invariant previous_step
    apply(simp add: last_step_invariant_def)
    apply(elim conjE)
    apply(unfold execute_next_instruction_def)
    apply(simp)
    apply(unfold fetch_instruction_def)
    apply(simp add: read_program_memory_def)
    apply(simp add: program_state)
    apply(simp add: program_def)
    apply(simp add: decode_decomp)
    apply(simp add: execute_instruction_simps)
    apply(simp add: instruction_decomp_mixed)
    apply(simp add: fun_upd_twist)
    done
  then have
    ‹read_register instruction_pointer_ref next_state = 4›
    ‹typical_flags next_state›
    ‹read_register c01_ref next_state = 1›
    ‹read_register r03_ref next_state = 0›
    ‹read_register r03_ref next_state ≤ read_register arg00_ref next_state›
    ‹read_register r00_ref next_state = 1›
    ‹read_register r01_ref next_state = 1›

    ‹read_register r00_ref next_state
```

```
    = of_nat (fib (unat (read_register r03_ref next_state)))›

    ‹read_register r01_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 1))›
    using previous_step_invariant program_state
    by(simp_all add: last_step_invariant_def state_manipulation_decomp fib_answers)
  then show ?thesis
    by(simp add: step_invariant_def)
qed

lemma (in Ironbark_world) fib_equivalence_invariant_step4:
  fixes
    current_state :: sequential_state
  defines
    ‹last_step_invariant ≡ λ state .
      typical_flags state
      ∧ read_register c01_ref state = 1
      ∧ read_register r03_ref state = 0
      ∧ read_register r03_ref state ≤ read_register arg00_ref state
      ∧ read_register r00_ref state
        = of_nat (fib (unat (read_register r03_ref state)))
      ∧ read_register r01_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 1))›
  defines
    ‹next_state ≡ execute_next_instruction current_state›
  defines
    ‹step_invariant ≡ λ state .
      typical_flags state
      ∧ read_register c01_ref state = 1
      ∧ read_register instruction_pointer_ref state = 5
      ∧ read_register last_instruction_pointer_ref state = 4
      ∧ read_register r03_ref state = 0
      ∧ read_register r03_ref state ≤ read_register arg00_ref state
      ∧ read_register r00_ref state
        = of_nat (fib (unat (read_register r03_ref state)))
      ∧ read_register r01_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 1))
      ∧ read_register r02_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 2))›
  assumes
    previous_step_invariant: ‹last_step_invariant current_state›
    and previous_step: ‹read_register instruction_pointer_ref current_state = 4›
    and program_state: ‹program_memory current_state = program›
  shows
    ‹step_invariant next_state›
proof -
  have
    ‹next_state =
      current_state
```

```
    (|
      register_state :=
        (register_state current_state)
        (
          r02_ref
            ↦ 2,
          cycles_register_ref
            ↦ (read_register cycles_register_ref current_state)
              + common_instruction_duration,
          last_instruction_pointer_ref
            ↦ 4,
          instruction_pointer_ref
            ↦ 5
        )
    |)›
  unfolding next_state_def
  using previous_step_invariant previous_step
  apply(simp add: last_step_invariant_def)
  apply(elim conjE)
  apply(unfold execute_next_instruction_def)
  apply(simp)
  apply(unfold fetch_instruction_def)
  apply(simp add: read_program_memory_def)
  apply(simp add: program_state)
  apply(simp add: program_def)
  apply(simp add: decode_decomp)
  apply(simp add: execute_instruction_simps)
  apply(simp add: instruction_decomp_mixed)
  apply(simp add: fun_upd_twist)
  done
then have
  ‹read_register instruction_pointer_ref next_state = 5›
  ‹read_register last_instruction_pointer_ref next_state = 4›
  ‹typical_flags next_state›
  ‹read_register c01_ref next_state = 1›
  ‹read_register r03_ref next_state = 0›
  ‹read_register r03_ref next_state ≤ read_register arg00_ref next_state›
  ‹read_register r00_ref next_state = 1›
  ‹read_register r01_ref next_state = 1›
  ‹read_register r02_ref next_state = 2›

  ‹read_register r00_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state)))›

  ‹read_register r01_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

  ‹read_register r02_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
```

```
    using previous_step_invariant program_state
    by(simp_all add: last_step_invariant_def state_manipulation_decomp fib_answers
        del: One_nat_def add_2_eq_Suc')
  then show ?thesis
    by(simp add: step_invariant_def)
qed

lemma (in Ironbark_world) fib_equivalence_invariant_step5:
  fixes
    current_state :: sequential_state
  defines
    ‹next_state ≡ execute_next_instruction current_state›
  defines
    ‹step_invariant ≡ λ state .
      typical_flags state
      ∧ read_register c01_ref state = 1
      ∧ read_register instruction_pointer_ref state = 6
      ∧ read_register r03_ref state ≤ read_register arg00_ref state
      ∧ read_register r00_ref state
        = of_nat (fib (unat (read_register r03_ref state)))
      ∧ read_register r01_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 1))
      ∧ read_register r02_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 2))›
  assumes
    previous_step_invariant:
      ‹typical_flags current_state ∨ end_jump_flags current_state›

      ‹end_jump_flags current_state
      ⟶ (read_register last_instruction_pointer_ref current_state = 12)›

      ‹read_register c01_ref current_state = 1›
      ‹read_register r03_ref current_state ≤ read_register arg00_ref current_state›

      ‹read_register r00_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state)))›

      ‹read_register r01_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state) + 1))›

      ‹read_register r02_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state) + 2))›

    and previous_step:  ‹read_register instruction_pointer_ref current_state = 5›
    and program_state:  ‹program_memory current_state = program›
  shows
    ‹step_invariant next_state›
proof -
  have
```

```
  ‹next_state =
    current_state
    (|
      flag_state := (flag_state current_state)(|end_jump := 0|),
      register_state :=
        (register_state current_state)
        (
          cycles_register_ref
            ↦ (read_register cycles_register_ref current_state)
              + common_instruction_duration,
          last_instruction_pointer_ref
            ↦ 5,
          instruction_pointer_ref
            ↦ 6
        )
    |)›
  unfolding next_state_def
  using previous_step_invariant previous_step
  apply(simp add: step_invariant_def)
  apply(unfold execute_next_instruction_def)
  apply(simp)
  apply(unfold fetch_instruction_def)
  apply(simp add: read_program_memory_def)
  apply(simp add: program_state)
  apply(simp add: program_def)
  apply(simp add: decode_decomp)
  apply(simp add: execute_instruction_simps)
  apply(simp add: instruction_impl_defs)
  apply(cases ‹get_end_jump current_state = 0›)
  apply(simp_all add: flag_normalisation)
  apply(simp_all add: state_manipulation_decomp)
  apply(simp_all add: fun_upd_twist)
  done
then have
  ‹read_register instruction_pointer_ref next_state = 6›
  ‹typical_flags next_state›
  ‹read_register c01_ref next_state = 1›
  ‹read_register r03_ref next_state ≤ read_register arg00_ref next_state›
  ‹read_register r00_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state)))›

  ‹read_register r01_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

  ‹read_register r02_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
  using previous_step_invariant program_state
  by(auto simp add: read_register_decomp read_flag_decomp)
then show ?thesis
```

```
    by(simp add: step_invariant_def)
qed

lemma (in Ironbark_world) fib_equivalence_invariant_step6:
  fixes
    current_state :: sequential_state
  defines
    ‹next_state ≡ execute_next_instruction current_state›
  defines
    ‹step_invariant ≡ λ state .
      typical_flags state
      ∧ read_register c01_ref state = 1
      ∧ read_register instruction_pointer_ref state = 7
      ∧ read_register r03_ref state ≤ read_register arg00_ref state
      ∧ (read_register r04_ref state = 1 ∨ read_register r04_ref state = 0)
      ∧ read_register r00_ref state
        = of_nat (fib (unat (read_register r03_ref state)))
      ∧ read_register r01_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 1))
      ∧ read_register r02_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 2))

      ∧ ((read_register r03_ref state = read_register arg00_ref state)
        ⟷ (read_register r04_ref state = 1))

      ∧ ((read_register r03_ref state ≠ read_register arg00_ref state)
        ⟷ (read_register r04_ref state = 0))›
  assumes
    previous_step_invariant:
      ‹typical_flags current_state›
      ‹read_register c01_ref current_state = 1›
      ‹read_register r03_ref current_state ≤ read_register arg00_ref current_state›
      ‹read_register r00_ref current_state
        = of_nat (fib (unat (read_register r03_ref current_state)))›

      ‹read_register r01_ref current_state
        = of_nat (fib (unat (read_register r03_ref current_state) + 1))›

      ‹read_register r02_ref current_state
        = of_nat (fib (unat (read_register r03_ref current_state) + 2))›
    and previous_step: ‹read_register instruction_pointer_ref current_state = 6›
    and program_state: ‹program_memory current_state = program›
  shows
    ‹step_invariant next_state›
proof -
  have
    ‹next_state =
      current_state
      (|
```

```
    register_state := (register_state current_state)
    (
      r04_ref ↦
        if
          read_register arg00_ref current_state
          = read_register r03_ref current_state
        then
          1
        else
          0,
      cycles_register_ref
        ↦ (read_register cycles_register_ref current_state)
          + common_instruction_duration,
      last_instruction_pointer_ref
        ↦ 6,
      instruction_pointer_ref
        ↦ 7
    )
  ⟫›
  unfolding next_state_def
  using previous_step_invariant previous_step
  apply(simp)
  apply(unfold execute_next_instruction_def)
  apply(simp)
  apply(unfold fetch_instruction_def)
  apply(simp add: read_program_memory_def)
  apply(simp add: program_state)
  apply(simp add: program_def)
  apply(simp add: decode_decomp)
  apply(simp add: execute_instruction_simps)
  apply(simp add: instruction_decomp_mixed)
  apply(simp add: fun_upd_twist)
  done
then have
  ‹read_register instruction_pointer_ref next_state = 7›
  ‹typical_flags next_state›
  ‹read_register c01_ref next_state = 1›
  ‹read_register r03_ref next_state ≤ read_register arg00_ref next_state›
  ‹read_register r04_ref next_state = 1 ∨ read_register r04_ref next_state = 0›

  ‹read_register r00_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state)))›

  ‹read_register r01_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

  ‹read_register r02_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
```

```
    ‹(read_register r03_ref next_state = read_register arg00_ref next_state)
    ⟷ (read_register r04_ref next_state = 1)›

    ‹(read_register r03_ref next_state ≠ read_register arg00_ref next_state)
    ⟷ (read_register r04_ref next_state = 0)›

    ‹(read_register arg00_ref next_state ≠ read_register r03_ref next_state)
    ⟷ (read_register r03_ref next_state < read_register arg00_ref next_state)›
    using previous_step_invariant program_state
    apply(simp_all add: read_register_decomp read_flag_decomp
            del: One_nat_def add_2_eq_Suc')
    apply(auto)
    done
  then show ?thesis
    by(simp add: step_invariant_def)
qed

lemma (in Ironbark_world) fib_equivalence_invariant_step7:
  fixes
    current_state :: sequential_state
  defines
    ‹next_state ≡ execute_next_instruction current_state›
  defines
    ‹step_invariant ≡ λ state .
      (read_register r04_ref state = 1 ∨ read_register r04_ref state = 0)
      ∧ (read_register r04_ref state = 0 ⟶ typical_flags state)
      ∧ (read_register r04_ref state = 1 ⟶ end_jump_flags state)
      ∧ read_register c01_ref state = 1
      ∧ read_register r03_ref state ≤ read_register arg00_ref state

      ∧ read_register r00_ref state
        = of_nat (fib (unat (read_register r03_ref state)))

      ∧ read_register r01_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 1))

      ∧ read_register r02_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 2))

      ∧ read_register last_instruction_pointer_ref state = 7

      ∧ (read_register instruction_pointer_ref state = 8
          ∨ read_register instruction_pointer_ref state = 13)

      ∧ ((read_register r03_ref state = read_register arg00_ref state)
          ⟷ (read_register r04_ref state = 1))

      ∧ ((read_register r03_ref state ≠ read_register arg00_ref state)
          ⟷ (read_register r04_ref state = 0))
```

```
    ∧ ((read_register r04_ref state = 0)
        ⟷ (read_register instruction_pointer_ref state = 8))

    ∧ ((read_register r04_ref state = 1)
        ⟷ (read_register instruction_pointer_ref state = 13))›
  assumes
    previous_step_invariant:
      ‹typical_flags current_state›
      ‹read_register c01_ref current_state = 1›
      ‹read_register r03_ref current_state ≤ read_register arg00_ref current_state›
      ‹read_register r04_ref current_state = 1 ∨ read_register r04_ref current_state = 0›

      ‹read_register r00_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state)))›

      ‹read_register r01_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state) + 1))›

      ‹read_register r02_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state) + 2))›

      ‹(read_register r03_ref current_state = read_register arg00_ref current_state)
      ⟷ (read_register r04_ref current_state = 1)›

      ‹(read_register r03_ref current_state ≠ read_register arg00_ref current_state)
      ⟷ (read_register r04_ref current_state = 0)›
    and previous_step:  ‹read_register instruction_pointer_ref current_state = 7›
    and program_state:  ‹program_memory current_state = program›
  shows
    ‹step_invariant next_state›
proof -
  have
    ‹if read_register r04_ref current_state = 0 then
      next_state =
        current_state
        (|
          register_state :=
            (register_state current_state)
            (
              cycles_register_ref
                ↦ (read_register cycles_register_ref current_state)
                  + common_instruction_duration,
              last_instruction_pointer_ref
                ↦ 7,
              instruction_pointer_ref
                ↦ 8
            )
        |)
```

```
  else
    next_state =
      current_state
      (|
        flag_state := flag_state current_state(|
          end_jump := 1
        |),
        register_state :=
          (register_state current_state)
          (
            last_instruction_pointer_ref
              ↦ 7,
            instruction_pointer_ref
              ↦ 13,
            cycles_register_ref
              ↦ (read_register cycles_register_ref current_state)
                + common_instruction_duration
          )
      |)›
  unfolding next_state_def
  using previous_step_invariant previous_step
  apply(simp)
  apply(cases ‹read_register 4 current_state = 0›)
  apply(simp_all)
  apply(unfold execute_next_instruction_def)
  apply(simp_all)
  apply(unfold fetch_instruction_def)
  apply(simp_all add: read_program_memory_def)
  apply(simp_all add: program_state)
  apply(simp_all add: program_def)
  apply(simp_all add: decode_decomp)
  apply(simp_all add: execute_instruction_simps)
  apply(simp_all add: instruction_decomp_mixed)
  apply(simp_all add: fun_upd_twist)
  apply(simp_all add: state_manipulation_decomp)
  done
then have h1:
  ‹read_register last_instruction_pointer_ref next_state = 7›
  ‹read_register c01_ref next_state = 1›
  ‹read_register r03_ref next_state ≤ read_register arg00_ref next_state›
  ‹read_register r04_ref next_state = 0 ⟶ typical_flags next_state›

  ‹(read_register arg00_ref next_state = read_register r03_ref next_state)
  ⟷ (read_register r04_ref next_state = 1)›

  ‹(read_register r04_ref next_state = 0)
  ⟷ (read_register instruction_pointer_ref next_state = 8)›

  ‹(read_register r04_ref next_state = 1)
```

```
    ⟷ (read_register instruction_pointer_ref next_state = 13)›

    ‹read_register r04_ref next_state = 1 ⟶ end_jump_flags next_state›

    ‹(typical_flags next_state ∨ end_jump_flags next_state)›

    ‹read_register instruction_pointer_ref next_state = 8
    ∨ (read_register instruction_pointer_ref next_state = 13)›

    ‹read_register r00_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state)))›

    ‹read_register r01_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

    ‹read_register r02_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
    using previous_step_invariant program_state
    apply(simp_all add: read_register_decomp read_flag_decomp
          del: One_nat_def add_2_eq_Suc')
    apply(auto)
    done
  show ?thesis
    apply(simp add: step_invariant_def h1)
    apply(safe)
    using h1
    apply(simp_all)
    done
qed

lemma (in Ironbark_world) fib_equivalence_invariant_step8:
  fixes
    current_state :: sequential_state
  defines
    ‹next_state ≡ execute_next_instruction current_state›
  defines
    ‹step_invariant ≡ λ state .
      typical_flags state
      ∧ read_register c01_ref state = 1
      ∧ read_register instruction_pointer_ref state = 9
      ∧ read_register r03_ref state < read_register arg00_ref state
      ∧ read_register r00_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 1))
      ∧ read_register r01_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 1))
      ∧ read_register r02_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 2))›
  assumes
    previous_step_invariant:
```

```
    ‹typical_flags current_state›
    ‹read_register c01_ref current_state = 1›
    ‹read_register r03_ref current_state < read_register arg00_ref current_state›

    ‹read_register r00_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state)))›

    ‹read_register r01_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 1))›

    ‹read_register r02_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 2))›
  and previous_step:  ‹read_register instruction_pointer_ref current_state = 8›
  and program_state:  ‹program_memory current_state = program›
 shows
  ‹step_invariant next_state›
proof -
 have
  ‹next_state =
    current_state
    (|
      register_state :=
        (register_state current_state)
        (
          r00_ref
            ↦ read_register r01_ref current_state,
          cycles_register_ref
            ↦ (read_register cycles_register_ref current_state)
              + common_instruction_duration,
          last_instruction_pointer_ref
            ↦ 8,
          instruction_pointer_ref
            ↦ 9
        )
    |)›
  unfolding next_state_def
  using previous_step_invariant previous_step
  apply(simp)
  apply(unfold execute_next_instruction_def)
  apply(simp)
  apply(unfold fetch_instruction_def)
  apply(simp add: read_program_memory_def)
  apply(simp add: program_state)
  apply(simp add: program_def)
  apply(simp add: decode_decomp)
  apply(simp add: execute_instruction_simps)
  apply(simp add: instruction_decomp_mixed)
  apply(simp add: fun_upd_twist)
  done
```

```
  then have
    ‹read_register instruction_pointer_ref next_state = 9›
    ‹typical_flags next_state›
    ‹read_register c01_ref next_state = 1›
    ‹read_register r03_ref next_state < read_register arg00_ref next_state›

    ‹read_register r00_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

    ‹read_register r01_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

    ‹read_register r02_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
    using previous_step_invariant program_state
    by(simp_all add: read_register_decomp read_flag_decomp
            del: One_nat_def add_2_eq_Suc')
  then show ?thesis
    by(simp add: step_invariant_def)
qed

lemma (in Ironbark_world) fib_equivalence_invariant_step9:
  fixes
    current_state :: sequential_state
  defines
    ‹next_state ≡ execute_next_instruction current_state›
  defines
    ‹step_invariant ≡ λ state .
      typical_flags state
      ∧ read_register c01_ref state = 1
      ∧ read_register instruction_pointer_ref state = 10
      ∧ read_register r03_ref state < read_register arg00_ref state
      ∧ read_register r00_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 1))
      ∧ read_register r01_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 2))
      ∧ read_register r02_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 2))›
  assumes
    previous_step_invariant:
      ‹typical_flags current_state›
      ‹read_register c01_ref current_state = 1›
      ‹read_register r03_ref current_state < read_register arg00_ref current_state›

      ‹read_register r00_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state) + 1))›

      ‹read_register r01_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state) + 1))›
```

```
    ‹read_register r02_ref current_state
    = of_nat (fib (unat (read_register r03_ref current_state) + 2))›
  and previous_step:  ‹read_register instruction_pointer_ref current_state = 9›
  and program_state:  ‹program_memory current_state = program›
shows
  ‹step_invariant next_state›
proof -
  have
    ‹next_state =
      current_state
      (|
        register_state :=
          (register_state current_state)
          (
            r01_ref
              ↦ read_register r02_ref current_state,
            cycles_register_ref
              ↦ (read_register cycles_register_ref current_state)
                + common_instruction_duration,
            last_instruction_pointer_ref
              ↦ 9,
            instruction_pointer_ref
              ↦ 10
          )
      |)›
    unfolding next_state_def
    using previous_step_invariant previous_step
    apply(simp)
    apply(unfold execute_next_instruction_def)
    apply(simp)
    apply(unfold fetch_instruction_def)
    apply(simp add: read_program_memory_def)
    apply(simp add: program_state)
    apply(simp add: program_def)
    apply(simp add: decode_decomp)
    apply(simp add: execute_instruction_simps)
    apply(simp add: instruction_decomp_mixed)
    apply(simp add: fun_upd_twist)
    done
  then have
    ‹read_register instruction_pointer_ref next_state = 10›
    ‹typical_flags next_state›
    ‹read_register c01_ref next_state = 1›
    ‹read_register r03_ref next_state < read_register arg00_ref next_state›

    ‹read_register r00_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 1))›
```

```
    ‹read_register r01_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 2))›

    ‹read_register r02_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
    using previous_step_invariant program_state
    by(simp_all add: read_register_decomp read_flag_decomp
            del: One_nat_def add_2_eq_Suc')
  then show ?thesis
    by(simp add: step_invariant_def)
qed

lemma (in Ironbark_world) fib_equivalence_invariant_step10:
  fixes
    current_state :: sequential_state
  defines
    ‹next_state ≡ execute_next_instruction current_state›
  defines
    ‹step_invariant ≡ λ state .
      typical_flags state
      ∧ read_register c01_ref state = 1
      ∧ read_register instruction_pointer_ref state = 11
      ∧ read_register r03_ref state < read_register arg00_ref state
      ∧ read_register r00_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 1))
      ∧ read_register r01_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 2))
      ∧ read_register r02_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 3))›
  assumes
    previous_step_invariant:
      ‹typical_flags current_state›
      ‹read_register c01_ref current_state = 1›
      ‹read_register r03_ref current_state < read_register arg00_ref current_state›

      ‹read_register r00_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state) + 1))›

      ‹read_register r01_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state) + 2))›

      ‹read_register r02_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state) + 2))›
    and previous_step:  ‹read_register instruction_pointer_ref current_state = 10›
    and program_state:  ‹program_memory current_state = program›
  shows
    ‹step_invariant next_state›
proof -
  have
```

```
  ‹next_state =
    current_state
    (|
      register_state :=
        (register_state current_state)
        (
          r02_ref
            ↦ (read_register r00_ref current_state)
              + (read_register r01_ref current_state),
          cycles_register_ref
            ↦ (read_register cycles_register_ref current_state)
              + common_instruction_duration,
          last_instruction_pointer_ref
            ↦ 10,
          instruction_pointer_ref
            ↦ 11
        )
    |)›
  unfolding next_state_def
  using previous_step_invariant previous_step
  apply(simp)
  apply(unfold execute_next_instruction_def)
  apply(simp)
  apply(unfold fetch_instruction_def)
  apply(simp add: read_program_memory_def)
  apply(simp add: program_state)
  apply(simp add: program_def)
  apply(simp add: decode_decomp)
  apply(simp add: execute_instruction_simps)
  apply(simp add: instruction_decomp_mixed)
  apply(simp add: fun_upd_twist)
  done
then have
  ‹read_register instruction_pointer_ref next_state = 11›
  ‹typical_flags next_state›
  ‹read_register c01_ref next_state = 1›
  ‹read_register r03_ref next_state < read_register arg00_ref next_state›

  ‹read_register r00_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

  ‹read_register r01_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 2))›

  ‹read_register r02_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 3))›
  using previous_step_invariant program_state
  apply(simp_all add: read_register_decomp read_flag_decomp)
  using fib.simps
```

```
    apply (simp)
    apply (simp only: fib_stuff)
    apply (simp)
    done
  then show ?thesis
    by (simp add: step_invariant_def)
qed

lemma (in Ironbark_world) fib_equivalence_invariant_step11:
  fixes
    current_state :: sequential_state
  defines
    ‹next_state ≡ execute_next_instruction current_state›
  defines
    ‹step_invariant ≡ λ state .
      typical_flags state
      ∧ read_register c01_ref state = 1
      ∧ read_register instruction_pointer_ref state = 12
      ∧ read_register r03_ref state ≤ read_register arg00_ref state
      ∧ read_register r00_ref state
        = of_nat (fib (unat (read_register r03_ref state)))
      ∧ read_register r01_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 1))
      ∧ read_register r02_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 2))›
  assumes
    previous_step_invariant:
      ‹typical_flags current_state›
      ‹read_register c01_ref current_state = 1›
      ‹read_register r03_ref current_state < read_register arg00_ref current_state›

      ‹read_register r00_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state) + 1))›

      ‹read_register r01_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state) + 2))›

      ‹read_register r02_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state) + 3))›
    and previous_step:  ‹read_register instruction_pointer_ref current_state = 11›
    and program_state:  ‹program_memory current_state = program›
  shows
    ‹step_invariant next_state›
proof -
  have
    ‹next_state =
      current_state
      (|
        register_state :=
```

```
      (register_state current_state)
      (
        r03_ref
          ↦ (read_register r03_ref current_state) + 1,
        cycles_register_ref
          ↦ (read_register cycles_register_ref current_state)
            + common_instruction_duration,
        last_instruction_pointer_ref
          ↦ 11,
        instruction_pointer_ref
          ↦ 12
      )
    ⟩
  unfolding next_state_def
  using previous_step_invariant previous_step
  apply(simp)
  apply(unfold execute_next_instruction_def)
  apply(simp)
  apply(unfold fetch_instruction_def)
  apply(simp add: read_program_memory_def)
  apply(simp add: program_state)
  apply(simp add: program_def)
  apply(simp add: decode_decomp)
  apply(simp add: execute_instruction_simps)
  apply(simp add: instruction_decomp_mixed)
  apply(simp add: fun_upd_twist)
  done
then have
  ‹read_register instruction_pointer_ref next_state = 12›
  ‹typical_flags next_state›
  ‹read_register c01_ref next_state = 1›
  ‹read_register r03_ref next_state ≤ read_register arg00_ref next_state›
  ‹read_register r00_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state)))›

  ‹read_register r01_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

  ‹read_register r02_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
  apply(simp_all del: One_nat_def add_2_eq_Suc')
  using previous_step_invariant program_state
  apply(simp_all add: read_register_decomp read_flag_decomp
          del: One_nat_def add_2_eq_Suc')
  apply(simp_all add: add_with_less_than)
  apply(simp_all only: fib_stuff fib.simps(3))
  apply(simp_all)
  apply(simp add: inc_le)
  done
```

```
  then show ?thesis
    by(simp add: step_invariant_def)
qed

lemma (in Ironbark_world) fib_equivalence_invariant_step12:
  fixes
    current_state :: sequential_state
  defines
    ‹next_state ≡ execute_next_instruction current_state›
  defines
    ‹step_invariant ≡ λ state .
      end_jump_flags state
      ∧ read_register c01_ref state = 1
      ∧ read_register instruction_pointer_ref state = 5
      ∧ read_register last_instruction_pointer_ref state = 12
      ∧ read_register r03_ref state ≤ read_register arg00_ref state
      ∧ read_register r00_ref state
        = of_nat (fib (unat (read_register r03_ref state)))
      ∧ read_register r01_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 1))
      ∧ read_register r02_ref state
        = of_nat (fib (unat (read_register r03_ref state) + 2))›
  assumes
    previous_step_invariant:
      ‹typical_flags current_state›
      ‹read_register c01_ref current_state = 1›
      ‹read_register r03_ref current_state ≤ read_register arg00_ref current_state›

      ‹read_register r00_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state)))›

      ‹read_register r01_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state) + 1))›

      ‹read_register r02_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state) + 2))›
    and previous_step:  ‹read_register instruction_pointer_ref current_state = 12›
    and program_state:  ‹program_memory current_state = program›
  shows
    ‹step_invariant next_state›
proof -
  have
    ‹next_state =
      current_state
      (|
        flag_state := flag_state current_state(|end_jump := 1|),
        register_state :=
          (register_state current_state)
          (
```

```
          cycles_register_ref
            ↦ (read_register cycles_register_ref current_state)
              + common_instruction_duration,
          last_instruction_pointer_ref
            ↦ 12,
          instruction_pointer_ref
            ↦ 5
        )
    ⟫›
  unfolding next_state_def
  using previous_step_invariant previous_step
  apply(simp)
  apply(unfold execute_next_instruction_def)
  apply(simp)
  apply(unfold fetch_instruction_def)
  apply(simp add: read_program_memory_def)
  apply(simp add: program_state)
  apply(simp add: program_def)
  apply(simp add: decode_decomp)
  apply(simp add: execute_instruction_simps)
  apply(simp add: instruction_decomp_mixed)
  apply(simp add: fun_upd_twist)
  apply(simp add: state_manipulation_decomp)
  done
then have
  ‹read_register instruction_pointer_ref next_state = 5›
  ‹read_register last_instruction_pointer_ref next_state = 12›
  ‹end_jump_flags next_state›
  ‹read_register c01_ref next_state = 1›
  ‹read_register r03_ref next_state ≤ read_register arg00_ref next_state›

  ‹read_register r00_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state)))›

  ‹read_register r01_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 1))›

  ‹read_register r02_ref next_state
  = of_nat (fib (unat (read_register r03_ref next_state) + 2))›
  using previous_step_invariant program_state
  by(simp_all add: read_register_decomp read_flag_decomp
          del: One_nat_def add_2_eq_Suc')
then show ?thesis
  by(simp add: step_invariant_def)
qed

lemma (in Ironbark_world) fib_equivalence_invariant_step13:
  fixes
    current_state :: sequential_state
```

```
  defines
    ‹next_state ≡ execute_next_instruction current_state›
  defines
    ‹step_invariant ≡ λ state .
      typical_flags state
      ∧ read_register instruction_pointer_ref state = 14
      ∧ read_register r03_ref state = read_register arg00_ref state
      ∧ read_register r00_ref state
        = of_nat (fib (unat (read_register r03_ref state)))›
  assumes
    previous_step_invariant:
      ‹end_jump_flags current_state›
      ‹read_register last_instruction_pointer_ref current_state = 7›
      ‹read_register r03_ref current_state = read_register arg00_ref current_state›

      ‹read_register r00_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state)))›
    and previous_step: ‹read_register instruction_pointer_ref current_state = 13›
    and program_state: ‹program_memory current_state = program›
  shows
    ‹step_invariant next_state›
proof -
  have
    ‹next_state =
      current_state
      (|
        flag_state := (flag_state current_state)(|end_jump := 0|),
        register_state :=
          (register_state current_state)
          (
            cycles_register_ref
              ↦ (read_register cycles_register_ref current_state)
                + common_instruction_duration,
            last_instruction_pointer_ref
              ↦ 13,
            instruction_pointer_ref
              ↦ 14
          )
      |)›
    unfolding next_state_def
    using previous_step_invariant previous_step
    apply(simp)
    apply(unfold execute_next_instruction_def)
    apply(simp)
    apply(unfold fetch_instruction_def)
    apply(simp add: read_program_memory_def)
    apply(simp add: program_state)
    apply(simp add: program_def)
    apply(simp add: decode_decomp)
```

```
    apply(simp add: execute_instruction_simps)
    apply(simp add: instruction_decomp_mixed)
    apply(simp add: fun_upd_twist)
    apply(simp add: state_manipulation_decomp)
    done
  then have
    ‹read_register instruction_pointer_ref next_state = 14›
    ‹typical_flags next_state›
    ‹read_register r03_ref next_state = read_register arg00_ref next_state›

    ‹read_register r00_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state)))›
    using previous_step_invariant program_state
    by(simp_all add: read_register_decomp read_flag_decomp
            del: One_nat_def add_2_eq_Suc')
  then show ?thesis
    by(simp add: step_invariant_def)
qed

lemma (in Ironbark_world) fib_equivalence_invariant_step14:
  fixes
    current_state :: sequential_state
  defines
    ‹next_state ≡ execute_next_instruction current_state›
  defines
    ‹step_invariant ≡ λ state .
      typical_flags state
      ∧ get_end_return state = 0
      ∧ get_end_call state = 0
      ∧ get_end_jump state = 0
      ∧ get_error state = 0
      ∧ read_register instruction_pointer_ref state = 15
      ∧ read_register r03_ref state = read_register arg00_ref state
      ∧ read_register ret00_ref state
        = of_nat (fib (unat (read_register r03_ref state)))›
  assumes
    previous_step_invariant:
      ‹typical_flags current_state›
      ‹read_register r03_ref current_state = read_register arg00_ref current_state›

      ‹read_register r00_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state)))›
    and previous_step:  ‹read_register instruction_pointer_ref current_state = 14›
    and program_state:  ‹program_memory current_state = program›
  shows
    ‹step_invariant next_state›
proof -
  have
    ‹next_state =
```

```
      current_state
      (|
        register_state :=
          (register_state current_state)
          (
            ret00_ref
              ↦ read_register r00_ref current_state,
            cycles_register_ref
              ↦ (read_register cycles_register_ref current_state)
                + common_instruction_duration,
            last_instruction_pointer_ref
              ↦ 14,
            instruction_pointer_ref
              ↦ 15
          )
      |)›
    unfolding next_state_def
    using previous_step_invariant previous_step
    apply(simp)
    apply(unfold execute_next_instruction_def)
    apply(simp)
    apply(unfold fetch_instruction_def)
    apply(simp add: read_program_memory_def)
    apply(simp add: program_state)
    apply(simp add: program_def)
    apply(simp add: decode_decomp)
    apply(simp add: execute_instruction_simps)
    apply(simp add: instruction_decomp_mixed)
    apply(simp add: fun_upd_twist)
    done
  then have
    ‹read_register instruction_pointer_ref next_state = 15›
    ‹typical_flags next_state›
    ‹read_register r03_ref next_state = read_register arg00_ref next_state›

    ‹read_register r00_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state)))›

    ‹read_register ret00_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state)))›
    using previous_step_invariant program_state
    by(simp_all add: read_register_decomp read_flag_decomp
            del: One_nat_def add_2_eq_Suc')
  then show ?thesis
    by(simp add: step_invariant_def)
qed

lemma (in Ironbark_world) fib_equivalence_invariant_step15:
  fixes
```

```
    current_state :: sequential_state
  defines
    ‹next_state ≡ execute_next_instruction current_state›
  defines
    ‹step_invariant ≡ λ state .
      halt_flags state
      ∧ get_end_return state = 0
      ∧ get_end_call state = 0
      ∧ get_end_jump state = 0
      ∧ get_error state = 0
      ∧ read_register instruction_pointer_ref state = 15
      ∧ read_register r03_ref state = read_register arg00_ref state
      ∧ read_register ret00_ref state
        = of_nat (fib (unat (read_register r03_ref state)))›
  assumes
    previous_step_invariant:
      ‹get_end_return current_state = 0›
      ‹get_end_call current_state = 0›
      ‹get_end_jump current_state = 0›
      ‹get_error current_state = 0›
      ‹read_register r03_ref current_state = read_register arg00_ref current_state›

      ‹read_register ret00_ref current_state
      = of_nat (fib (unat (read_register r03_ref current_state)))›
    and previous_step: ‹read_register instruction_pointer_ref current_state = 15›
    and program_state: ‹program_memory current_state = program›
  shows
    ‹step_invariant next_state›
proof -
  have
    ‹next_state =
      current_state
      (|
        flag_state := flag_state current_state(|halt := 1|)
      |)›
    unfolding next_state_def
    using previous_step_invariant previous_step
    apply(simp)
    apply(unfold execute_next_instruction_def)
    apply(unfold fetch_instruction_def)
    apply(simp add: read_program_memory_def)
    apply(simp add: program_state)
    apply(simp add: program_def)
    apply(simp add: decode_decomp)
    apply(simp add: execute_instruction_simps)
    apply(cases ‹get_halt current_state = 1›)
    apply(simp_all add: instruction_decomp_manipulation)
    apply(simp_all add: instruction_impl_defs)
    apply(simp_all add: state_manipulation_decomp)
```

```
    apply (simp_all add: flag_normalisation)
    done
  then have
    ‹read_register instruction_pointer_ref next_state = 15›
    ‹halt_flags next_state›
    ‹read_register r03_ref next_state = read_register arg00_ref next_state›

    ‹read_register ret00_ref next_state
    = of_nat (fib (unat (read_register r03_ref next_state)))›
    using previous_step_invariant program_state previous_step
    by (simp_all add: read_register_decomp read_flag_decomp
            del: One_nat_def add_2_eq_Suc')
  then show ?thesis
    by (simp add: step_invariant_def)
qed
```

Finally, we combine all of the proofs for each step to show that the program is correct. The structure of this proof is to start with some preliminary transformation, and then show the invariant conditions are met at all steps of the program by application of the previous lemmas.

```
lemma (in Ironbark_world) fib_equivalence_invariant:
  shows
    ‹invariant
      (execute_multiple_instructions
        (initial_state(|program_memory := program|))
        num_instructions)›
proof -
  have
    ‹read_register instruction_pointer_ref (initial_state(|program_memory := program|))
    = 0›
    by (simp add: program_def initial_state_def read_register_decomp)
  moreover have
    ‹0 ∈ dom program›
    by (simp add: program_def)
  ultimately show ?thesis
    apply (subst inv_satI)
    apply (simp_all)

  proof -
    assume
      ‹read_register instruction_pointer_ref (initial_state(|program_memory := program|))
      = 0›
    then show
      ‹invariant (initial_state(|program_memory := program|))›
      apply (simp add: invariant_def)
      apply (simp add: initial_state_simps)
      done

    fix state :: ‹sequential_state›
    assume
```

```
  ‹read_register instruction_pointer_ref state ∈ dom program›
  ‹invariant state›
  ‹program_memory state = program›
then show
  ‹invariant (execute_next_instruction state)›
  ‹read_register instruction_pointer_ref (execute_next_instruction state)
  ∈ dom program›
  apply (all ‹subst (asm) program_def›)
  apply (simp)
  apply (elim disjE)

  prefer 17

  apply (simp)
  apply (elim disjE)

proof -
  fix state :: sequential_state
  assume
    ‹invariant state›
    ‹program_memory state = program›
    ‹read_register instruction_pointer_ref state = 0›
  then show
    ‹invariant (execute_next_instruction state)›

    ‹read_register instruction_pointer_ref (execute_next_instruction state)
    ∈ dom program›
    apply (simp_all add: invariant_def fib_equivalence_invariant_step0)
    apply (simp add: program_def)
    done

next

  fix state :: sequential_state
  assume
    ‹invariant state›
    ‹program_memory state = program›
    ‹read_register instruction_pointer_ref state = 1›
  then show
    ‹invariant (execute_next_instruction state)›

    ‹read_register instruction_pointer_ref (execute_next_instruction state)
    ∈ dom program›
    apply (simp_all add: invariant_def fib_equivalence_invariant_step1)
    apply (simp add: program_def)
    done

next
```

```
  fix state :: sequential_state
  assume
    ‹invariant state›
    ‹program_memory state = program›
    ‹read_register instruction_pointer_ref state = 2›
  then show
    ‹invariant (execute_next_instruction state)›

    ‹read_register instruction_pointer_ref (execute_next_instruction state)
    ∈ dom program›
    apply(simp_all add: invariant_def fib_equivalence_invariant_step2)
    apply(simp add: program_def)
    done

next

  fix state :: sequential_state
  assume
    ‹invariant state›
    ‹program_memory state = program›
    ‹read_register instruction_pointer_ref state = 3›
  then show
    ‹invariant (execute_next_instruction state)›

    ‹read_register instruction_pointer_ref (execute_next_instruction state)
    ∈ dom program›
    apply(simp_all add: invariant_def fib_equivalence_invariant_step3)
    apply(simp add: program_def)
    done

next

  fix state :: sequential_state
  assume
    ‹invariant state›
    ‹program_memory state = program›
    ‹read_register instruction_pointer_ref state = 4›
  then show
    ‹invariant (execute_next_instruction state)›

    ‹read_register instruction_pointer_ref (execute_next_instruction state)
    ∈ dom program›
    apply(simp_all add: invariant_def fib_equivalence_invariant_step4)
    apply(simp add: program_def)
    done

next

  fix state :: sequential_state
```

```
  assume
    ‹invariant state›
    ‹program_memory state = program›
    ‹read_register instruction_pointer_ref state = 5›
  then show
    ‹invariant (execute_next_instruction state)›

    ‹read_register instruction_pointer_ref (execute_next_instruction state)
    ∈ dom program›
    apply(simp_all add: invariant_def fib_equivalence_invariant_step5)
    apply(simp add: program_def)
    done

next

  fix state :: sequential_state
  assume
    ‹invariant state›
    ‹program_memory state = program›
    ‹read_register instruction_pointer_ref state = 6›
  then show
    ‹invariant (execute_next_instruction state)›

    ‹read_register instruction_pointer_ref (execute_next_instruction state)
    ∈ dom program›
    using fib_equivalence_invariant_step6
    apply(simp_all add: invariant_def)
    apply(simp_all add: fib_equivalence_invariant_step6)
    apply(simp add: program_def)
    done

next

  fix state :: sequential_state
  assume
    ‹invariant state›
    ‹program_memory state = program›
    ‹read_register instruction_pointer_ref state = 7›
  then show
    ‹invariant (execute_next_instruction state)›

    ‹read_register instruction_pointer_ref (execute_next_instruction state)
    ∈ dom program›
    using fib_equivalence_invariant_step7[of ‹state›]
    apply(simp_all add: invariant_def)
    apply(elim conjE)
    apply(simp)
    apply(elim conjE disjE)
    apply(simp_all)
```

```
    apply (elim conjE)
    apply (simp)
    apply (elim conjE disjE)
    apply (simp_all add: program_def)
    done

next

  fix state :: sequential_state
  assume
    ‹invariant state›
    ‹program_memory state = program›
    ‹read_register instruction_pointer_ref state = 8›
  then show
    ‹invariant (execute_next_instruction state)›

    ‹read_register instruction_pointer_ref (execute_next_instruction state)
    ∈ dom program›
    apply (simp_all add: invariant_def fib_equivalence_invariant_step8)
    apply (simp add: program_def)
    done

next

  fix state :: sequential_state
  assume
    ‹invariant state›
    ‹program_memory state = program›
    ‹read_register instruction_pointer_ref state = 9›
  then show
    ‹invariant (execute_next_instruction state)›

    ‹read_register instruction_pointer_ref (execute_next_instruction state)
    ∈ dom program›
    apply (simp_all add: invariant_def fib_equivalence_invariant_step9)
    apply (simp add: program_def)
    done

next

  fix state :: sequential_state
  assume
    ‹invariant state›
    ‹program_memory state = program›
    ‹read_register instruction_pointer_ref state = 10›
  then show
    ‹invariant (execute_next_instruction state)›

    ‹read_register instruction_pointer_ref (execute_next_instruction state)
```

```
    ∈ dom program›
    apply(simp_all add: invariant_def fib_equivalence_invariant_step10)
    apply(simp add: program_def)
    done

next

  fix state :: sequential_state
  assume
    ‹invariant state›
    ‹program_memory state = program›
    ‹read_register instruction_pointer_ref state = 11›
  then show
    ‹invariant (execute_next_instruction state)›

    ‹read_register instruction_pointer_ref (execute_next_instruction state)
    ∈ dom program›
    apply(simp_all add: invariant_def)
    apply(simp_all add: fib_equivalence_invariant_step11)
    apply(simp add: program_def)
    done

next

  fix state :: sequential_state
  assume
    ‹invariant state›
    ‹program_memory state = program›
    ‹read_register instruction_pointer_ref state = 12›
  then show
    ‹invariant (execute_next_instruction state)›

    ‹read_register instruction_pointer_ref (execute_next_instruction state)
    ∈ dom program›
    apply(simp_all add: invariant_def)
    apply(simp_all add: fib_equivalence_invariant_step12)
    apply(simp add: program_def)
    done

next

  fix state :: sequential_state
  assume
    ‹invariant state›
    ‹program_memory state = program›
    ‹read_register instruction_pointer_ref state = 13›
  then show
    ‹invariant (execute_next_instruction state)›
```

```
      ‹read_register instruction_pointer_ref (execute_next_instruction state)
      ∈ dom program›
      apply(simp_all add: invariant_def)
      apply(simp_all add: fib_equivalence_invariant_step13)
      apply(simp add: program_def)
      done

  next

    fix state :: sequential_state
    assume
      ‹invariant state›
      ‹program_memory state = program›
      ‹read_register instruction_pointer_ref state = 14›
    then show
      ‹invariant (execute_next_instruction state)›

      ‹read_register instruction_pointer_ref (execute_next_instruction state)
      ∈ dom program›
      apply(simp_all add: invariant_def)
      apply(simp_all add: fib_equivalence_invariant_step14)
      apply(simp add: program_def)
      done

  next

    fix state :: sequential_state
    assume
      ‹invariant state›
      ‹program_memory state = program›
      ‹read_register instruction_pointer_ref state = 15›
    then show
      ‹invariant (execute_next_instruction state)›

      ‹read_register instruction_pointer_ref (execute_next_instruction state)
      ∈ dom program›
      apply(simp_all add: invariant_def)
      apply(simp_all add: fib_equivalence_invariant_step15)
      apply(simp add: program_def)
      done
  qed
 qed
qed
end
```

G Top Level File

theory *top_proof*

imports
"Ironbark/add_example"
"Ironbark/fibonacci_invariant_piecemeal"
"Ironbark/fibonacci_function"
"Ironbark_execution.security_properties"

begin

Here we import all the top level theory files into a single file.

end