

# The Semantics of Power and ARM Multiprocessor Machine Code

## The HOL Specification

Jade Alglave<sup>2</sup>   Anthony Fox<sup>1</sup>   Samin Ishtiaq<sup>3</sup>   Magnus O. Myreen<sup>1</sup>  
Susmit Sarkar<sup>1</sup>   Peter Sewell<sup>1</sup>   Francesco Zappa Nardelli<sup>2</sup>

<sup>1</sup>University of Cambridge   <sup>2</sup>INRIA   <sup>3</sup>Microsoft Research Cambridge

October 14, 2008

# Brief Contents

Brief Contents	i
Full Contents	ii

I	common_coretypes	1
II	common_types	3
III	ppc_arm_axiomatic_model	6
IV	arm_coretypes	14
V	arm_types	19
VI	arm_ast	21
VII	arm_opsem	24
VIII	arm_seq_monad	36
IX	arm_event_monad	43
X	arm_decoder	51
XI	arm_program	55
XII	ppc_coretypes	58
XIII	ppc_types	60
XIV	ppc_ast	62
XV	ppc_opsem	65
XVI	ppc_decoder	74
XVII	ppc_	76
XVIII	ppc_seq_monad	78
XIX	ppc_event_monad	82
XX	ppc_program	88
	Index	91

# Full Contents

Brief Contents	i
Full Contents	ii
<b>I common_coretypes</b>	<b>1</b>
– <i>type_abbrev_proc</i> . . . . .	2
– <i>type_abbrev_program_order_index</i> . . . . .	2
– <i>iiid</i> . . . . .	2
<b>II common_types</b>	<b>3</b>
– <i>type_abbrev_Ximm</i> . . . . .	4
– <i>type_abbrev_address</i> . . . . .	4
– <i>type_abbrev_value</i> . . . . .	4
– <i>type_abbrev_eiid</i> . . . . .	4
– <i>type_abbrev_reln</i> . . . . .	4
– <i>dirn</i> . . . . .	4
– <i>location</i> . . . . .	4
– <i>synchronization</i> . . . . .	4
– <i>action</i> . . . . .	4
– <i>event</i> . . . . .	4
– <i>architecture</i> . . . . .	4
– <i>event_structure</i> . . . . .	4
– <i>type_abbrev_state_constraint</i> . . . . .	4
– <i>type_abbrev_view_orders</i> . . . . .	4
– <i>execution_witness</i> . . . . .	4
– <i>type_abbrev_eiid_state</i> . . . . .	5
– <i>next_eiid</i> . . . . .	5
– <i>initial_eiid_state</i> . . . . .	5
– <i>type_abbrev_program_word8</i> . . . . .	5
– <i>type_abbrev_run_skeleton</i> . . . . .	5
– <i>run_skeleton_wf</i> . . . . .	5
<b>III ppc_arm_axiomatic_model</b>	<b>6</b>
– <i>loc</i> . . . . .	7
– <i>value_of</i> . . . . .	7

- *proc* . . . . . 7
- *mem\_load* . . . . . 7
- *mem\_store* . . . . . 7
- *mem\_access* . . . . . 7
- *reg\_load* . . . . . 7
- *reg\_store* . . . . . 7
- *reg\_access* . . . . . 7
- *reg\_or\_mem\_location* . . . . . 7
- *reg\_or\_mem\_or\_resaddr* . . . . . 8
- *store* . . . . . 8
- *load* . . . . . 8
- *is\_barrier* . . . . . 8
- *is\_sync* . . . . . 8
- *procs* . . . . . 8
- *iiids* . . . . . 8
- *writes* . . . . . 8
- *reads* . . . . . 8
- *po* . . . . . 8
- *po\_strict* . . . . . 9
- *intra\_causality* . . . . . 9
- *po\_iico\_both* . . . . . 9
- *po\_iico\_data* . . . . . 9
- *well\_formed\_event\_structure* . . . . . 9
- *local\_register\_data\_dependency* . . . . . 9
- *address\_or\_data\_dependency\_load\_load* . . . . . 10
- *address\_or\_data\_or\_control\_dependency\_load\_store* . . . . . 10
- *preserved\_program\_order\_mem\_loc* . . . . . 10
- *preserved\_program\_order* . . . . . 10
- *preserved\_coherence\_order* . . . . . 10
- *viewed\_events* . . . . . 10
- *view\_orders\_well\_formed* . . . . . 10
- *get\_mem\_l\_stores* . . . . . 10
- *write\_serialization\_candidates* . . . . . 10
- *state\_updates* . . . . . 11
- *read\_most\_recent\_value* . . . . . 11
- *FIX* . . . . . 11
- *check\_sync\_power\_2\_05* . . . . . 11
- *check\_dmb\_arm* . . . . . 11
- *same\_granule* . . . . . 12
- *location\_res\_value* . . . . . 12
- *read\_location\_res\_value* . . . . . 12
- *check\_atomicity* . . . . . 12
- *valid\_execution* . . . . . 12

**IV arm\_coretypes 14**

- *type\_abbrev\_Aimm* . . . . . 15
- *ARMreg* . . . . . 15
- *ARMpsr* . . . . . 15

–	<i>ARMmode</i>	15
–	<i>ARMstatus</i>	15
–	<i>ARMsctlr</i>	15
–	<i>ARMcp_registers</i>	15
–	<i>ARMcondition</i>	15
–	<i>ARMexception</i>	15
–	<i>ARMversion</i>	15
–	<i>ARMextensions</i>	16
–	<i>ARMinfo</i>	16
–	<i>InstrSet</i>	16
–	<i>MemType</i>	16
–	<i>MemoryAttributes</i>	16
–	<i>FullAddress</i>	16
–	<i>AddressDescriptor</i>	16
–	<i>type_abbrev_ExclusiveMonitor</i>	16
–	<i>ExclusiveMonitors</i>	16
–	<i>MBReqDomain</i>	17
–	<i>MBReqTypes</i>	17
–	<i>word5_to_mode</i>	17
–	<i>align</i>	17
–	<i>decode_psr</i>	17
–	<i>encode_psr</i>	18
–	<i>version_number</i>	18
<b>V arm_types</b>		<b>19</b>
–	<i>arm_reg</i>	20
<b>VI arm_ast</b>		<b>21</b>
–	<i>addressing_mode1</i>	22
–	<i>addressing_mode2</i>	22
–	<i>addressing_mode3</i>	22
–	<i>ARMinstruction</i>	22
<b>VII arm_opsem</b>		<b>24</b>
–	<i>user_or_system_mode</i>	25
–	<i>word4_mode_to_reg</i>	25
–	<i>read_regm</i>	25
–	<i>write_regm</i>	25
–	<i>mode_to_psr</i>	25
–	<i>exception_to_mode</i>	25
–	<i>mode_to_word5</i>	25
–	<i>exception_to_address</i>	26
–	<i>condition_passed</i>	26
–	<i>inc_pc</i>	26
–	<i>branch_write_pc</i>	26
–	<i>LSL</i>	26

–	<i>LSR</i>	27
–	<i>ASR</i>	27
–	<i>ROR</i>	27
–	<i>immediate</i>	27
–	<i>shift_immediate</i>	27
–	<i>shift_register</i>	27
–	<i>addr_mode1</i>	27
–	<i>addr_mode2</i>	28
–	<i>ALU_arith</i>	28
–	<i>ALU_logic</i>	28
–	<i>ALU_multiply</i>	28
–	<i>ALU_multiply_long</i>	28
–	<i>ADD</i>	28
–	<i>SUB</i>	29
–	<i>AND</i>	29
–	<i>EOR</i>	29
–	<i>ORR</i>	29
–	<i>ALU</i>	29
–	<i>arithmetic</i>	29
–	<i>test_or_compare</i>	29
–	<i>rotate_mem32</i>	29
–	<i>exception_exec</i>	29
–	<i>branch_exec</i>	30
–	<i>data_processing_exec</i>	30
–	<i>multiply_exec</i>	30
–	<i>multiply_long_exec</i>	31
–	<i>load_store_exec</i>	31
–	<i>load_exclusive_exec</i>	32
–	<i>store_exclusive_exec</i>	32
–	<i>swap_exec</i>	33
–	<i>status_to_register_exec</i>	33
–	<i>breakpoint_exec</i>	33
–	<i>data_memory_barrier_exec</i>	34
–	<i>arm_execute</i>	34
<b>VIII</b>	<b>arm_seq_monad</b>	<b>36</b>
–	<i>type_abbrev_arm_state</i>	37
–	<i>AREAD_REG</i>	37
–	<i>AREAD_PSR</i>	37
–	<i>AREAD_CP</i>	37
–	<i>AREAD_MEM</i>	37
–	<i>AREAD_INFO</i>	37
–	<i>AREAD_EXCL</i>	37
–	<i>AWRITE_REG</i>	37
–	<i>AWRITE_PSR</i>	37
–	<i>AWRITE_MEM</i>	37
–	<i>AWRITE_EXCL</i>	37
–	<i>type_abbrev_M</i>	37

–	<i>constT_seq</i> . . . . .	37
–	<i>failureT_seq</i> . . . . .	37
–	<i>bindT_seq</i> . . . . .	37
–	<i>seqT_seq</i> . . . . .	37
–	<i>parT_seq</i> . . . . .	37
–	<i>lockT_seq</i> . . . . .	38
–	<i>condT_seq</i> . . . . .	38
–	<i>discardT_seq</i> . . . . .	38
–	<i>addT_seq</i> . . . . .	38
–	<i>write_reg_seq</i> . . . . .	38
–	<i>read_reg_seq</i> . . . . .	38
–	<i>write_psr_seq</i> . . . . .	38
–	<i>read_psr_seq</i> . . . . .	38
–	<i>write_flags_seq</i> . . . . .	38
–	<i>read_flags_seq</i> . . . . .	38
–	<i>read_endian_seq</i> . . . . .	38
–	<i>read_mode_seq</i> . . . . .	38
–	<i>read_instr_set_seq</i> . . . . .	38
–	<i>read_sctrl_seq</i> . . . . .	39
–	<i>read_info_seq</i> . . . . .	39
–	<i>read_version_seq</i> . . . . .	39
–	<i>set_exclusive_monitorsT_seq</i> . . . . .	39
–	<i>exclusive_monitors_passT_seq</i> . . . . .	39
–	<i>dmbT_seq</i> . . . . .	40
–	<i>write_mem8_seq</i> . . . . .	40
–	<i>write_mem16_seq</i> . . . . .	40
–	<i>write_mem32_seq</i> . . . . .	40
–	<i>read_mem8_seq</i> . . . . .	41
–	<i>read_mem16_seq</i> . . . . .	41
–	<i>read_mem32_seq</i> . . . . .	41
–	<i>constT</i> . . . . .	41
–	<i>addT</i> . . . . .	41
–	<i>lockT</i> . . . . .	41
–	<i>failureT</i> . . . . .	41
–	<i>discardT</i> . . . . .	41
–	<i>condT</i> . . . . .	41
–	<i>bindT</i> . . . . .	41
–	<i>seqT</i> . . . . .	41
–	<i>parT</i> . . . . .	41
–	<i>read_info</i> . . . . .	41
–	<i>read_version</i> . . . . .	42
–	<i>write_reg</i> . . . . .	42
–	<i>read_reg</i> . . . . .	42
–	<i>write_psr</i> . . . . .	42
–	<i>read_psr</i> . . . . .	42
–	<i>write_flags</i> . . . . .	42
–	<i>read_flags</i> . . . . .	42
–	<i>read_mode</i> . . . . .	42
–	<i>read_instr_set</i> . . . . .	42



–	<i>read_sctlr</i> . . . . .	42
–	<i>write_mem8</i> . . . . .	42
–	<i>read_mem8</i> . . . . .	42
–	<i>write_mem16</i> . . . . .	42
–	<i>read_mem16</i> . . . . .	42
–	<i>write_mem32</i> . . . . .	42
–	<i>read_mem32</i> . . . . .	42
–	<i>dmbT</i> . . . . .	42
–	<i>set_exclusive_monitorsT</i> . . . . .	42
–	<i>exclusive_monitors_passT</i> . . . . .	42
–	<i>option_apply</i> . . . . .	42
<b>IX</b>	<b>arm_event_monad</b>	<b>43</b>
–	<i>type_abbrev_M</i> . . . . .	44
–	<i>event_structure_empty</i> . . . . .	44
–	<i>event_structure_lock</i> . . . . .	44
–	<i>event_structure_union</i> . . . . .	44
–	<i>event_structure_bigunion</i> . . . . .	44
–	<i>event_structure_seq_union</i> . . . . .	44
–	<i>event_structure_control_seq_union</i> . . . . .	44
–	<i>mapT_ev</i> . . . . .	44
–	<i>choiceT_ev</i> . . . . .	45
–	<i>constT_ev</i> . . . . .	45
–	<i>discardT_ev</i> . . . . .	45
–	<i>addT_ev</i> . . . . .	45
–	<i>lockT_ev</i> . . . . .	45
–	<i>failureT_ev</i> . . . . .	45
–	<i>condT_ev</i> . . . . .	45
–	<i>bindT_ev</i> . . . . .	45
–	<i>control_seqT_ev</i> . . . . .	45
–	<i>seqT_ev</i> . . . . .	45
–	<i>parT_ev</i> . . . . .	45
–	<i>parT_unit_ev</i> . . . . .	46
–	<i>syncT_ev</i> . . . . .	46
–	<i>write_location_ev</i> . . . . .	46
–	<i>read_location_ev</i> . . . . .	46
–	<i>write_reg_ev</i> . . . . .	46
–	<i>read_reg_ev</i> . . . . .	47
–	<i>write_psr_ev</i> . . . . .	47
–	<i>read_psr_ev</i> . . . . .	47
–	<i>write_flags_ev</i> . . . . .	47
–	<i>read_flags_ev</i> . . . . .	47
–	<i>OUR_VERSION</i> . . . . .	47
–	<i>OUR_INFO</i> . . . . .	47
–	<i>OUR_MODE</i> . . . . .	47
–	<i>OUR_INSTR_SET</i> . . . . .	47
–	<i>OUR_SCTLR</i> . . . . .	47
–	<i>read_version_ev</i> . . . . .	47

–	<i>read_info_ev</i> . . . . .	47
–	<i>read_mode_ev</i> . . . . .	47
–	<i>read_instr_set_ev</i> . . . . .	47
–	<i>read_sctlr_ev</i> . . . . .	47
–	<i>set_exclusive_monitorsT_ev</i> . . . . .	48
–	<i>exclusive_monitors_passT_ev</i> . . . . .	48
–	<i>aligned32</i> . . . . .	48
–	<i>write_mem8_ev</i> . . . . .	48
–	<i>write_mem16_ev</i> . . . . .	48
–	<i>write_mem32_ev</i> . . . . .	48
–	<i>read_mem8_ev</i> . . . . .	48
–	<i>read_mem16_ev</i> . . . . .	48
–	<i>read_mem32_ev</i> . . . . .	48
–	<i>dmbT_ev</i> . . . . .	48
–	<i>constT</i> . . . . .	48
–	<i>addT</i> . . . . .	48
–	<i>lockT</i> . . . . .	48
–	<i>failureT</i> . . . . .	49
–	<i>discardT</i> . . . . .	49
–	<i>condT</i> . . . . .	49
–	<i>bindT</i> . . . . .	49
–	<i>seqT</i> . . . . .	49
–	<i>parT</i> . . . . .	49
–	<i>parT_unit</i> . . . . .	49
–	<i>read_info</i> . . . . .	49
–	<i>read_version</i> . . . . .	49
–	<i>write_reg</i> . . . . .	49
–	<i>read_reg</i> . . . . .	49
–	<i>write_psr</i> . . . . .	49
–	<i>read_psr</i> . . . . .	49
–	<i>write_flags</i> . . . . .	49
–	<i>read_flags</i> . . . . .	49
–	<i>read_mode</i> . . . . .	49
–	<i>read_instr_set</i> . . . . .	49
–	<i>read_sctlr</i> . . . . .	49
–	<i>write_mem8</i> . . . . .	50
–	<i>read_mem8</i> . . . . .	50
–	<i>write_mem16</i> . . . . .	50
–	<i>read_mem16</i> . . . . .	50
–	<i>write_mem32</i> . . . . .	50
–	<i>read_mem32</i> . . . . .	50
–	<i>dmbT</i> . . . . .	50
–	<i>set_exclusive_monitorsT</i> . . . . .	50
–	<i>exclusive_monitors_passT</i> . . . . .	50
<b>X</b>	<b>arm_decoder</b> . . . . .	<b>51</b>
–	<i>condition_decode</i> . . . . .	52
–	<i>arm_decode</i> . . . . .	52

<b>XI</b>	<b>arm_program</b>	<b>55</b>
-	<i>VERSION_MULTICORE</i> . . . . .	56
-	<i>type_abbrev_Ainstruction</i> . . . . .	56
-	<i>type_abbrev_program_Ainstruction</i> . . . . .	56
-	<i>arm_decode_program_fun</i> . . . . .	56
-	<i>arm_decode_program_rel</i> . . . . .	56
-	<i>arm_event_execute</i> . . . . .	56
-	<i>arm_execute_with_pc_check</i> . . . . .	56
-	<i>arm_event_structures_of_run_skeleton</i> . . . . .	56
-	<i>arm_semantics</i> . . . . .	56
<b>XII</b>	<b>ppc_coretypes</b>	<b>58</b>
-	<i>type_abbrev_ireg</i> . . . . .	59
-	<i>type_abbrev_freg</i> . . . . .	59
-	<i>type_abbrev_ppc_constant</i> . . . . .	59
-	<i>type_abbrev_crbit</i> . . . . .	59
-	<i>ppc_bit</i> . . . . .	59
-	<i>ppc_reg32</i> . . . . .	59
<b>XIII</b>	<b>ppc_types</b>	<b>60</b>
-	<i>ppc_reg</i> . . . . .	61
<b>XIV</b>	<b>ppc_ast</b>	<b>62</b>
-	<i>Pinstruction</i> . . . . .	63
<b>XV</b>	<b>ppc_opsem</b>	<b>65</b>
-	<i>ppc_sint_cmp</i> . . . . .	66
-	<i>ppc_uint_cmp</i> . . . . .	66
-	<i>OK_nextinstr</i> . . . . .	66
-	<i>reg_update</i> . . . . .	66
-	<i>uint_reg_update</i> . . . . .	66
-	<i>sint_reg_update</i> . . . . .	66
-	<i>uint_compare</i> . . . . .	66
-	<i>sint_compare</i> . . . . .	66
-	<i>bit_update</i> . . . . .	66
-	<i>const_low</i> . . . . .	66
-	<i>const_high</i> . . . . .	66
-	<i>conditional</i> . . . . .	66
-	<i>read_bit_word</i> . . . . .	66
-	<i>read_ireg</i> . . . . .	67
-	<i>gpr_or_zero</i> . . . . .	67
-	<i>no_carry</i> . . . . .	67
-	<i>goto_label</i> . . . . .	67
-	<i>effective_address</i> . . . . .	67
-	<i>assertT</i> . . . . .	67

–	<i>write_mem_aux</i> . . . . .	67
–	<i>store_word</i> . . . . .	67
–	<i>register_store</i> . . . . .	67
–	<i>read_mem_aux</i> . . . . .	67
–	<i>load_word</i> . . . . .	67
–	<i>register_load</i> . . . . .	67
–	<i>set_CR0_to_00xNONE</i> . . . . .	68
–	<i>ppc_exec_instr</i> . . . . .	68
<b>XVI</b>	<b>ppc_decoder</b>	<b>74</b>
–	<i>ppc_match_step</i> . . . . .	75
–	<i>ppc_decode</i> . . . . .	75
<b>XVII</b>	<b>ppc_</b>	<b>76</b>
–	<i>iiid_dummy</i> . . . . .	77
–	<i>PPC_NEXT</i> . . . . .	77
<b>XVIII</b>	<b>ppc_seq_monad</b>	<b>78</b>
–	<i>type_abbrev_ppc_state</i> . . . . .	79
–	<i>PREAD_R</i> . . . . .	79
–	<i>PREAD_S</i> . . . . .	79
–	<i>PREAD_M</i> . . . . .	79
–	<i>PWRITE_R</i> . . . . .	79
–	<i>PWRITE_S</i> . . . . .	79
–	<i>PWRITE_M</i> . . . . .	79
–	<i>PREAD_REVERSE_BIT</i> . . . . .	79
–	<i>PREAD_REVERSE_ADDRESS</i> . . . . .	79
–	<i>PWRITE_REVERSE_BIT</i> . . . . .	79
–	<i>PWRITE_REVERSE_ADDRESS</i> . . . . .	79
–	<i>type_abbrev_M</i> . . . . .	79
–	<i>constT_seq</i> . . . . .	79
–	<i>addT_seq</i> . . . . .	79
–	<i>lockT_seq</i> . . . . .	79
–	<i>syncT_seq</i> . . . . .	79
–	<i>failureT_seq</i> . . . . .	79
–	<i>seqT_seq</i> . . . . .	79
–	<i>parT_seq</i> . . . . .	79
–	<i>parT_unit_seq</i> . . . . .	80
–	<i>write_reg_seq</i> . . . . .	80
–	<i>read_reg_seq</i> . . . . .	80
–	<i>write_status_seq</i> . . . . .	80
–	<i>read_status_seq</i> . . . . .	80
–	<i>write_mem8_seq</i> . . . . .	80
–	<i>read_mem8_seq</i> . . . . .	80
–	<i>read_mem32_seq</i> . . . . .	80
–	<i>write_mem32_seq</i> . . . . .	80

-	<i>write_reserve_bit_seq</i> . . . . .	80
-	<i>read_reserve_bit_seq</i> . . . . .	80
-	<i>write_reserve_address_seq</i> . . . . .	80
-	<i>read_reserve_address_seq</i> . . . . .	81
-	<i>constT</i> . . . . .	81
-	<i>addT</i> . . . . .	81
-	<i>lockT</i> . . . . .	81
-	<i>syncT</i> . . . . .	81
-	<i>failureT</i> . . . . .	81
-	<i>control_seqT</i> . . . . .	81
-	<i>seqT</i> . . . . .	81
-	<i>parT</i> . . . . .	81
-	<i>parT_unit</i> . . . . .	81
-	<i>write_reg</i> . . . . .	81
-	<i>read_reg</i> . . . . .	81
-	<i>write_status</i> . . . . .	81
-	<i>read_status</i> . . . . .	81
-	<i>write_mem8</i> . . . . .	81
-	<i>read_mem8</i> . . . . .	81
-	<i>write_mem32</i> . . . . .	81
-	<i>read_mem32</i> . . . . .	81
-	<i>write_reserve_bit</i> . . . . .	81
-	<i>read_reserve_bit</i> . . . . .	81
-	<i>write_reserve_address</i> . . . . .	81
-	<i>read_reserve_address</i> . . . . .	81
-	<i>option_apply</i> . . . . .	81

## **XIX** *ppc\_event\_monad* **82**

-	<i>type_abbrev_M</i> . . . . .	83
-	<i>event_structure_empty</i> . . . . .	83
-	<i>event_structure_lock</i> . . . . .	83
-	<i>event_structure_union</i> . . . . .	83
-	<i>event_structure_bigunion</i> . . . . .	83
-	<i>event_structure_seq_union</i> . . . . .	83
-	<i>event_structure_control_seq_union</i> . . . . .	83
-	<i>mapT_ev</i> . . . . .	83
-	<i>choiceT_ev</i> . . . . .	84
-	<i>constT_ev</i> . . . . .	84
-	<i>addT_ev</i> . . . . .	84
-	<i>lockT_ev</i> . . . . .	84
-	<i>failureT_ev</i> . . . . .	84
-	<i>condT_ev</i> . . . . .	84
-	<i>seqT_ev</i> . . . . .	84
-	<i>control_seqT_ev</i> . . . . .	84
-	<i>parT_ev</i> . . . . .	84
-	<i>parT_unit_ev</i> . . . . .	84
-	<i>write_location_ev</i> . . . . .	85
-	<i>read_location_ev</i> . . . . .	85

–	<i>syncT_ev</i> . . . . .	85
–	<i>write_reg_ev</i> . . . . .	85
–	<i>read_reg_ev</i> . . . . .	85
–	<i>write_status_ev</i> . . . . .	85
–	<i>read_status_ev</i> . . . . .	86
–	<i>aligned32</i> . . . . .	86
–	<i>write_mem8_ev</i> . . . . .	86
–	<i>read_mem8_ev</i> . . . . .	86
–	<i>write_mem32_ev</i> . . . . .	86
–	<i>read_mem32_ev</i> . . . . .	86
–	<i>write_reserve_bit_ev</i> . . . . .	86
–	<i>read_reserve_bit_ev</i> . . . . .	86
–	<i>write_reserve_address_ev</i> . . . . .	86
–	<i>read_reserve_address_ev</i> . . . . .	86
–	<i>constT</i> . . . . .	86
–	<i>addT</i> . . . . .	86
–	<i>lockT</i> . . . . .	86
–	<i>syncT</i> . . . . .	87
–	<i>failureT</i> . . . . .	87
–	<i>seqT</i> . . . . .	87
–	<i>control_seqT</i> . . . . .	87
–	<i>parT</i> . . . . .	87
–	<i>parT_unit</i> . . . . .	87
–	<i>write_reg</i> . . . . .	87
–	<i>read_reg</i> . . . . .	87
–	<i>write_status</i> . . . . .	87
–	<i>read_status</i> . . . . .	87
–	<i>write_mem8</i> . . . . .	87
–	<i>read_mem8</i> . . . . .	87
–	<i>write_mem32</i> . . . . .	87
–	<i>read_mem32</i> . . . . .	87
–	<i>write_reserve_bit</i> . . . . .	87
–	<i>read_reserve_bit</i> . . . . .	87
–	<i>write_reserve_address</i> . . . . .	87
–	<i>read_reserve_address</i> . . . . .	87
<b>XX</b>	<b>ppc_program</b> . . . . .	<b>88</b>
–	<i>type_abbrev_program_Pinstruction</i> . . . . .	89
–	<i>ppc_decode_program_fun</i> . . . . .	89
–	<i>ppc_decode_program_rel</i> . . . . .	89
–	<i>ppc_event_execute</i> . . . . .	89
–	<i>ppc_execute_with_pc_check</i> . . . . .	89
–	<i>ppc_event_structures_of_run_skeleton</i> . . . . .	89
–	<i>ppc_semantics</i> . . . . .	89
	<b>Index</b> . . . . .	<b>91</b>

# Part I

## **common\_coretypes**

*iid*

2

**type\_abbrev** proc : num

**type\_abbrev** program\_order\_index : num

*iid* = { proc : proc; *poi* : program\_order\_index }



**Part II**  
**common\_types**

**type\_abbrev** Ximm : word32

**type\_abbrev** address : Ximm

**type\_abbrev** value : Ximm

**type\_abbrev** eiid : num

**type\_abbrev** reln : 'a#'a → bool

dirn = R | W

location = LOCATION\_REG **of** proc 'reg  
 | LOCATION\_MEM **of** address  
 | LOCATION\_RES **of** proc  
 | LOCATION\_RES\_ADDR **of** proc

synchronization = SYNC

action = ACCESS **of** dirn 'reg location value  
 | BARRIER **of** synchronization

event =⟦ eiid : eiid;  
 iid : iid;  
 action : 'reg action⟧

architecture = POWER205 | ARMv7

event\_structure =⟦ events : ('reg event)set;  
 intra\_causality\_data : ('reg event)reln;  
 intra\_causality\_control : ('reg event)reln;  
 atomicity : ('reg event)set set;  
 arch : architecture;  
 granule\_size\_exponent : num⟧

**type\_abbrev** state\_constraint : ('reg location) → value option

**type\_abbrev** view\_orders : proc → ('reg event)reln

execution\_witness =  
 ⟦ initial\_state : ('reg state\_constraint);  
 vo : ('reg view\_orders);  
 write\_serialization : ('reg event reln)⟧

**type\_abbrev** *eiid\_state* : *eiid* set

(*next\_eiid* : *eiid\_state* → (*eiid*#*eiid\_state*)set) *eiids* = {(*eiid*, *eiids* ∪ {*eiid*}) | *eiid* | ¬(*eiid* ∈ *eiids*)}

(*initial\_eiid\_state* : *eiid\_state*) = {}

**type\_abbrev** *program\_word8* : (address → word8 option)

**type\_abbrev** *run\_skeleton* : (proc → (program\_order\_index → address option))

(*run\_skeleton\_wf* : address set → *run\_skeleton* → bool) *addrs* *rs* =  
 (∀ *p i i'*. (¬((*rs p i'*) = NONE)) ∧ (*i* < *i'*)) ⇒ (¬((*rs p i*) = NONE))) ∧  
 (∀ *p i a*. (*rs p i* = SOME *a*) ⇒ *a* ∈ *addrs*) ∧  
**finite**{*p* | ∃ *i a*. *rs p i* = SOME *a*}

## Part III

# ppc\_arm\_axiomatic\_model

loc  $e =$

**case**  $e.action$  **of**  
 ACCESS  $d\ l\ v \rightarrow$  SOME  $l$   
 ||  $_ \rightarrow$  NONE

value\_of  $e =$

**case**  $e.action$  **of**  
 ACCESS  $d\ l\ v \rightarrow$  SOME  $v$   
 ||  $_ \rightarrow$  NONE

proc  $e = e.iid.proc$

mem\_load  $e =$

**case**  $e.action$  **of**  
 ACCESS R(LOCATION\_MEM  $a$ ) $v \rightarrow$  **T**  
 ||  $_ \rightarrow$  **F**

mem\_store  $e =$

**case**  $e.action$  **of**  
 ACCESS W(LOCATION\_MEM  $a$ ) $v \rightarrow$  **T**  
 ||  $_ \rightarrow$  **F**

mem\_access  $e =$

**case**  $e.action$  **of**  
 ACCESS  $d$ (LOCATION\_MEM  $a$ ) $v \rightarrow$  **T**  
 ||  $_ \rightarrow$  **F**

reg\_load  $e =$

**case**  $e.action$  **of**  
 ACCESS R(LOCATION\_REG  $p\ r$ ) $v \rightarrow$  **T**  
 ||  $_ \rightarrow$  **F**

reg\_store  $e =$

**case**  $e.action$  **of**  
 ACCESS W(LOCATION\_REG  $p\ r$ ) $v \rightarrow$  **T**  
 ||  $_ \rightarrow$  **F**

reg\_access  $e =$

**case**  $e.action$  **of**  
 ACCESS  $d$ (LOCATION\_REG  $p\ r$ ) $v \rightarrow$  **T**  
 ||  $_ \rightarrow$  **F**

$reg\_or\_mem\_location\ l =$   
**case**  $l$  **of**  
 (LOCATION\_REG  $p\ r$ )  $\rightarrow \mathbf{T}$   
 $\parallel$  (LOCATION\_MEM  $a$ )  $\rightarrow \mathbf{T}$   
 $\parallel$  (LOCATION\_RES  $p$ )  $\rightarrow \mathbf{F}$   
 $\parallel$  (LOCATION\_RES\_ADDR  $p$ )  $\rightarrow \mathbf{F}$

$reg\_or\_mem\_or\_resaddr\ l =$   
**case**  $l$  **of**  
 (LOCATION\_REG  $p\ r$ )  $\rightarrow \mathbf{T}$   
 $\parallel$  (LOCATION\_MEM  $a$ )  $\rightarrow \mathbf{T}$   
 $\parallel$  (LOCATION\_RES  $p$ )  $\rightarrow \mathbf{F}$   
 $\parallel$  (LOCATION\_RES\_ADDR  $p$ )  $\rightarrow \mathbf{T}$

$store\ e =$   
**case**  $e.action$  **of**  
 ACCESS W  $l\ v$   $\rightarrow \mathbf{T}$   
 $\parallel$   $\_ \rightarrow \mathbf{F}$

$load\ e =$   
**case**  $e.action$  **of**  
 ACCESS R  $l\ v$   $\rightarrow \mathbf{T}$   
 $\parallel$   $\_ \rightarrow \mathbf{F}$

$is\_barrier\ e =$   
**case**  $e.action$  **of**  
 BARRIER  $bar$   $\rightarrow \mathbf{T}$   
 $\parallel$   $\_ \rightarrow \mathbf{F}$

$is\_sync\ e =$   
**case**  $e.action$  **of**  
 BARRIER SYNC  $\rightarrow \mathbf{T}$   
 $\parallel$   $\_ \rightarrow \mathbf{F}$

$procs\ E = \{proc\ e \mid e \in E.events\}$

$iiids\ E = \{e.iiid \mid e \in E.events\}$

$writes\ E = \{e \mid e \in E.events \wedge \exists l\ v.e.action = ACCESS\ W\ l\ v\}$

$reads\ E = \{e \mid e \in E.events \wedge \exists l\ v.e.action = ACCESS\ R\ l\ v\}$

$$\text{po } E = \{(e_1, e_2) \mid (e_1.\text{iiid}.\text{proc} = e_2.\text{iiid}.\text{proc}) \wedge \\ e_1.\text{iiid}.\text{poi} \leq e_2.\text{iiid}.\text{poi} \wedge \\ e_1 \in E.\text{events} \wedge e_2 \in E.\text{events}\}$$

$$\text{po\_strict } E = \{(e_1, e_2) \mid (e_1.\text{iiid}.\text{proc} = e_2.\text{iiid}.\text{proc}) \wedge \\ e_1.\text{iiid}.\text{poi} < e_2.\text{iiid}.\text{poi} \wedge \\ e_1 \in E.\text{events} \wedge e_2 \in E.\text{events}\}$$

$$\text{intra\_causality } E = E.\text{intra\_causality\_data} \cup \\ E.\text{intra\_causality\_control}$$

$$\text{po\_iico\_both } E = \text{po\_strict } E \cup E.\text{intra\_causality\_data} \cup \\ E.\text{intra\_causality\_control}$$

$$\text{po\_iico\_data } E = \text{po\_strict } E \cup E.\text{intra\_causality\_data}$$

$$\text{well\_formed\_event\_structure } E = \\ (\forall e_1 e_2 \in (E.\text{events}).(e_1.\text{eiid} = e_2.\text{eiid}) \wedge (e_1.\text{iiid} = e_2.\text{iiid}) \\ \implies (e_1 = e_2)) \wedge \\ (\text{DOM}(\text{intra\_causality } E) \subseteq E.\text{events} \wedge \\ \text{range}(\text{intra\_causality } E) \subseteq E.\text{events} \wedge \\ \text{acyclic}(\text{intra\_causality } E) \wedge \\ (\forall (e_1, e_2) \in (\text{intra\_causality } E).(e_1.\text{iiid} = e_2.\text{iiid})) \wedge \\ (\forall (e_1 \in \text{writes } E) e_2. \\ \neg(e_1 = e_2) \wedge \\ (e_2 \in \text{writes } E \vee e_2 \in \text{reads } E) \wedge \\ (e_1.\text{iiid} = e_2.\text{iiid}) \wedge \\ (\text{loc } e_1 = \text{loc } e_2) \wedge \\ (\exists p r. \text{loc } e_1 = \text{SOME } (\text{LOCATION\_REG } p r)) \\ \implies \\ (e_1, e_2) \in (\text{intra\_causality } E)^+ \vee \\ (e_2, e_1) \in (\text{intra\_causality } E)^+) \wedge \\ \text{PER } E.\text{events } E.\text{atomicity} \wedge \\ (\forall es \in (E.\text{atomicity}).\forall e_1 e_2 \in es.(e_1.\text{iiid} = e_2.\text{iiid}))$$

$$\text{local\_register\_data\_dependency } E p = \\ \{(e_1, e_2) \mid \\ (e_1, e_2) \in \text{po\_iico\_data } E \wedge \\ (\text{proc } e_1 = p) \wedge (\text{proc } e_2 = p) \wedge \\ (\exists r v_1 v_2. \\ (e_1.\text{action} = \text{ACCESS W}(\text{LOCATION\_REG } p r)v_1) \wedge \\ (e_2.\text{action} = \text{ACCESS R}(\text{LOCATION\_REG } p r)v_2) \wedge \\ \neg(\exists e_3 v_3.(e_1, e_3) \in \text{po\_iico\_data } E \wedge \\ (e_3, e_2) \in \text{po\_iico\_data } E \wedge \\ (e_3.\text{action} = \text{ACCESS W}(\text{LOCATION\_REG } p r)v_3))))))\}$$

address\_or\_data\_dependency\_load\_load  $E p =$   
 $\{(e_1, e_2) \mid$   
 $(\text{mem\_load } e_1 \wedge \text{mem\_load } e_2 \wedge (e_1, e_2) \in \text{po } E \wedge$   
 $(\text{proc } e_1 = p) \wedge (\text{proc } e_2 = p) \wedge$   
 $(e_1, e_2) \in ((E.\text{intra\_causality\_data} \cup$   
 $\text{local\_register\_data\_dependency } E p)^+))\}$

address\_or\_data\_or\_control\_dependency\_load\_store  $E p =$   
 $\{(e_1, e_2) \mid$   
 $(\text{mem\_load } e_1 \wedge \text{mem\_store } e_2 \wedge (e_1, e_2) \in \text{po } E \wedge$   
 $(\text{proc } e_1 = p) \wedge (\text{proc } e_2 = p) \wedge$   
 $(e_1, e_2) \in ((E.\text{intra\_causality\_data} \cup$   
 $E.\text{intra\_causality\_control} \cup$   
 $\text{local\_register\_data\_dependency } E p)^+))\}$

preserved\_program\_order\_mem\_loc  $E p =$   
 $\{(e_1, e_2) \mid (e_1, e_2) \in \text{po\_iico\_data } E \wedge$   
 $(\text{proc } e_1 = p) \wedge (\text{proc } e_2 = p) \wedge (\text{loc } e_1 = \text{loc } e_2) \wedge$   
 $\text{mem\_access } e_1 \wedge \text{mem\_access } e_2\}$

preserved\_program\_order  $E p =$   
 $\{(e_1, e_2) \mid (e_1, e_2) \in E.\text{intra\_causality\_data} \wedge$   
 $(\text{proc } e_1 = p) \wedge (\text{proc } e_2 = p)\} \cup$   
 $\{(e_1, e_2) \mid (e_1, e_2) \in E.\text{intra\_causality\_control} \wedge$   
 $(\text{proc } e_1 = p) \wedge (\text{proc } e_2 = p)\} \cup$   
 $\text{address\_or\_data\_dependency\_load\_load } E p \cup$   
 $\text{address\_or\_data\_or\_control\_dependency\_load\_store } E p \cup$   
 $\text{preserved\_program\_order\_mem\_loc } E p$

preserved\_coherence\_order  $E p =$   
 $\{(e_1, e_2) \mid (e_1, e_2) \in \text{po\_iico\_data } E \wedge$   
 $(\text{proc } e_1 = p) \wedge (\text{proc } e_2 = p) \wedge (\text{loc } e_1 = \text{loc } e_2) \wedge$   
 $(\text{mem\_store } e_1 \wedge \text{mem\_store } e_2)\}$

viewed\_events  $E p =$   
 $\{e \mid e \in E.\text{events} \wedge ((\text{proc } e = p) \vee \text{mem\_store } e)\}$

view\_orders\_well\_formed  $E vo =$   
 $\forall p \in (\text{procs } E).$   
 $\text{strict\_linear\_order}(vo p)(\text{viewed\_events } E p)$

get\_mem\_lstores  $E l =$   
 $\{e \mid e \in E.\text{events} \wedge \text{mem\_store } e \wedge (\text{loc } e = \text{SOME } l)\}$



write\_serialization\_candidates  $E$   $cand =$   
 $(\forall (e_1, e_2) \in cand.$   
 $\quad \exists l. e_1 \in (\text{get\_mem\_l\_stores } E l) \wedge$   
 $\quad \quad e_2 \in (\text{get\_mem\_l\_stores } E l)) \wedge$   
 $(\forall l. \text{strict\_linear\_order}(cand|_{(\text{get\_mem\_l\_stores } E l)}$   
 $\quad (\text{get\_mem\_l\_stores } E l)))$

state\_updates  $E$   $vo$   $e =$   
 $\{ew \mid ew \in (\text{writes } E) \wedge (ew, e) \in vo(\text{proc } e) \wedge$   
 $\quad (\text{loc } ew = \text{loc } e)\}$

read\_most\_recent\_value  $E$   $initial\_state$   $vo =$   
 $\forall e \in (E.\text{events}). \forall v.$   
 $\quad ((e.\text{action} = \text{ACCESS R } l v) \wedge \text{reg\_or\_mem\_or\_resaddr } l)$   
 $\quad \implies$   
 $\quad (\text{if } (\text{state\_updates } E \text{ } vo e) = \{\} \text{ then}$   
 $\quad \quad (\text{SOME } v = \text{initial\_state } l)$   
 $\quad \text{else}$   
 $\quad \quad ((\text{SOME } v) \in \{\text{value\_of } ew \mid$   
 $\quad \quad \quad ew \in \text{maximal\_elements}(\text{state\_updates } E \text{ } vo e)$   
 $\quad \quad \quad (vo(\text{proc } e))\}))$

**FIX**  $f$   $x = \text{biginter}\{X \mid (f X \cup x) \subseteq X\}$

check\_sync\_power\_2\_05  $E$   $vos =$   
 $\forall es \in (E.\text{events}). (es.\text{action} = \text{BARRIER SYNC}) \implies$   
 $\text{let } group\_A = \{e \mid ((e, es) \in \text{po } E \vee (e, es) \in vos(\text{proc } es)) \wedge \text{mem\_access } e\} \text{ in}$   
 $\text{let } group\_B\_base = \{e \mid (es, e) \in \text{po } E \wedge \text{mem\_access } e\} \text{ in}$   
 $\text{let } group\_B\_ind B_0 =$   
 $\quad \{e \mid \text{mem\_access } e \wedge$   
 $\quad \quad (\neg(\text{proc } e = \text{proc } es)) \wedge$   
 $\quad \quad \exists er. \text{mem\_load } er \wedge (er, e) \in vos(\text{proc } er) \wedge (\text{proc } er = \text{proc } e) \wedge$   
 $\quad \quad \quad \exists ew. \text{mem\_store } ew \wedge ew \in B_0 \wedge (ew, er) \in vos(\text{proc } er) \wedge (\text{loc } er = \text{loc } ew) \wedge$   
 $\quad \quad \quad (\neg(\exists ew'. (ew, ew') \in vos(\text{proc } er) \wedge (ew', er) \in vos(\text{proc } er) \wedge$   
 $\quad \quad \quad \quad (\text{loc } ew' = \text{loc } er) \wedge \text{mem\_store } ew'))\} \text{ in}$   
 $\text{let } group\_B = \text{FIX } group\_B\_ind group\_B\_base \text{ in}$   
 $\forall p \in (\text{procs } E). \forall ea \in group\_A. \forall eb \in group\_B. (ea \in \text{viewed\_events } E p \wedge eb \in \text{viewed\_events } E p) \implies$   
 $\quad \text{if } (p = es.\text{iiid}.\text{proc}) \text{ then } ((ea, es) \in vos p \wedge (es, eb) \in vos p) \text{ else } (ea, eb) \in vos p$

check\_dmb\_arm  $E$   $vos =$   
 $\forall es \in (E.\text{events}). (es.\text{action} = \text{BARRIER SYNC}) \implies$   
 $\text{let } group\_A\_base = \{e \mid ((e, es) \in vos(\text{proc } es)) \wedge \text{mem\_access } e\} \text{ in}$   
 $\text{let } group\_A\_ind A_0 = \{er \mid \text{mem\_load } er \wedge$   
 $\quad \exists e \in A_0. (er, e) \in vos(\text{proc } e)\} \text{ in}$   
 $\text{let } group\_B\_base = \{e \mid (es, e) \in \text{po } E \wedge \text{mem\_access } e\} \text{ in}$   
 $\text{let } group\_B\_ind B_0 = \{e \mid \text{mem\_access } e \wedge$

$\exists ew. \text{mem\_store } ew \wedge ew \in B_0 \wedge (ew, e) \in \text{vos}(\text{proc } e)\} \mathbf{in}$   
**let**  $group\_A = \text{FIX } group\_A\_ind \ group\_A\_base \mathbf{in}$   
**let**  $group\_B = \text{FIX } group\_B\_ind \ group\_B\_base \mathbf{in}$   
 $\forall p \in (\text{procs } E). \forall ea \in group\_A. \forall eb \in group\_B. (ea \in \text{viewed\_events } E \ p \wedge eb \in \text{viewed\_events } E \ p) \implies$   
**if**  $(p = \text{es.iiid.proc}) \mathbf{then} ((ea, es) \in \text{vos } p \wedge (es, eb) \in \text{vos } p) \mathbf{else} (ea, eb) \in \text{vos } p$

**same\_granule**  $E \ a \ a' =$   
**let**  $f \ x = x // (1w \ll E.\text{granule\_size\_exponent}) \mathbf{in} (f \ a = f \ a')$

**location\_res\_value**  $E \ vo \ e =$   
**let**  $prior\_reservations = \{ew \mid ew \in (\text{writes } E) \wedge$   
 $(ew, e) \in \text{vo}(\text{proc } e) \wedge$   
 $(\text{loc } ew = \text{SOME } (\text{LOCATION\_RES\_ADDR}(\text{proc } e)))\} \mathbf{in}$   
**if**  $(prior\_reservations = \{\}) \mathbf{then} \ 0w \mathbf{else}$   
**let**  $reservation = \text{maximalElements}$   
 $prior\_reservations(\text{vo}(\text{proc } e)) \mathbf{in}$   
**let**  $intervening\_writes = \{ew \mid$   
 $\exists ew' \in reservation. \exists a'. \exists a \ v.$   
 $(ew.\text{action} = \text{ACCESS } W(\text{LOCATION\_MEM } a)v) \wedge$   
 $(ew', ew) \in \text{vo}(\text{proc } e) \wedge (ew, e) \in \text{vo}(\text{proc } e) \wedge$   
 $(\text{value\_of } ew' = \text{SOME } a') \wedge$   
 $\text{same\_granule } E \ a \ a'\} \mathbf{in}$   
**if**  $intervening\_writes = \{\} \mathbf{then} \ 0w \mathbf{else} \ 1w$

**read\_location\_res\_value**  $E \ initial\_state \ vo =$   
 $\forall e \in (E.\text{events}). \forall p \ v.$   
 $(e.\text{action} = \text{ACCESS } R(\text{LOCATION\_RES } p)v) \implies$   
 $(v = \text{location\_res\_value } E \ vo \ e)$

**check\_atomicity**  $E \ vo =$   
 $\forall p \in (\text{procs } E). \forall es \in (E.\text{atomicity}).$   
 $\forall e_1 \ e_2 \in es. (e_1, e_2) \in (vo \ p) \implies$   
 $\forall e. (e_1, e) \in (vo \ p) \wedge (e, e_2) \in (vo \ p) \implies e \in es$

**valid\_execution**  $E \ X =$   
 $\text{view\_orders\_well\_formed } E \ X.\text{vo} \wedge$   
 $\text{read\_most\_recent\_value } E \ X.\text{initial\_state } X.\text{vo} \wedge$   
 $X.\text{write\_serialization} \in \text{write\_serialization\_candidates } E \wedge$   
 $(\forall p \in (\text{procs } E).$   
 $\text{preserved\_coherence\_order } E \ p \subseteq X.\text{write\_serialization} \wedge$   
 $X.\text{write\_serialization} \subseteq X.\text{vo } p \wedge$   
 $\text{preserved\_program\_order } E \ p \subseteq X.\text{vo } p \wedge$   
 $(\text{*no intervening writes in local register data dependency*})$   
 $\text{local\_register\_data\_dependency } E \ p \subseteq X.\text{vo } p \wedge$   
 $(\forall (e_1, e_2) \in (\text{local\_register\_data\_dependency } E \ p).$   
 $\neg(\exists e_3. (e_1, e_3) \in X.\text{vo } p \wedge (e_3, e_2) \in X.\text{vo } p \wedge$

$$\begin{aligned}
& (\text{loc } e_3 = \text{loc } e_1 \wedge \text{store } e_3))) \wedge \\
& (*\text{no intervening writes before a reg read from initial state*}) \\
& (\forall e \in (E.\text{events}).(\text{reg\_load } e \wedge \\
& \quad (\neg(\exists e_0.(e_0, e) \in \text{po\_iico\_both } E \wedge \text{reg\_store } e_0 \wedge \\
& \quad \quad (\text{loc } e_0 = \text{loc } e)))) \implies \\
& \quad (\neg(\exists e_0.(e_0, e) \in X.\text{vo}(\text{proc } e) \wedge \text{reg\_store } e_0 \wedge \\
& \quad \quad (\text{loc } e_0 = \text{loc } e)))) \wedge \\
& (*\text{no intervening writes after a reg write to the final state*}) \\
& (\forall e \in (E.\text{events}).(\text{reg\_store } e \wedge \\
& \quad (\neg(\exists e_1.(e, e_1) \in \text{po\_iico\_both } E \wedge \text{reg\_store } e_1 \wedge \\
& \quad \quad (\text{loc } e_1 = \text{loc } e)))) \implies \\
& \quad (\neg(\exists e_1.(e, e_1) \in X.\text{vo}(\text{proc } e) \wedge \text{reg\_store } e_1 \wedge \\
& \quad \quad (\text{loc } e_1 = \text{loc } e)))) \wedge \\
& (\text{case } E.\text{arch} \text{ of} \\
& \quad \text{POWER205} \rightarrow \text{check\_sync\_power\_2\_05 } E X.\text{vo} \\
& \quad \parallel \text{ARMV7} \rightarrow \text{check\_dmb\_arm } E X.\text{vo}) \wedge \\
& \text{read\_location\_res\_value } E X.\text{initial\_state } X.\text{vo} \wedge \\
& \text{check\_atomicity } E X.\text{vo}
\end{aligned}$$

## Part IV

# arm\_coretypes

**type\_abbrev** Aimm : word32

ARMreg =

```
R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7 |
R8 | R9 | R10 | R11 | R12 | R13 | R14 | R15 |
R8_FIQ | R9_FIQ | R10_FIQ | R11_FIQ | R12_FIQ | R13_FIQ | R14_FIQ |
                                     R13_IRQ | R14_IRQ |
                                     R13_SVC | R14_SVC |
                                     R13_ABT | R14_ABT |
                                     R13_UND | R14_UND
```

ARMpsr =

```
CPSR | SPSR_FIQ | SPSR_IRQ | SPSR_SVC | SPSR_ABT | SPSR_UND
```

ARMmode = USR | FIQ | IRQ | SVC | ABT | UND | SYS

ARMstatus =

```
{ N : bool; Z : bool; C : bool; V : bool;
  Q : bool; IT : word8; J : bool; Reserved : word4; GE : word4;
  E : bool; A : bool;
  I : bool; F : bool; T : bool; M : word5 }
```

ARMsctlr =

```
{ TE : bool; AFE : bool; TRE : bool; NMFI : bool; EE : bool;
  VE : bool; U : bool; FI : bool; HA : bool; RR : bool;
  V : bool; I : bool; Z : bool; SW : bool; B : bool;
  C : bool; A : bool; M : bool }
```

ARMcp\_registers = { SCTLR : ARMsctlr }

ARMcondition =

```
EQ | CS | MI | VS | HI | GE | GT | AL |
NE | CC | PL | VC | LS | LT | LE | NV
```

ARMexception =

```
EXCEPTION_RESET |
EXCEPTION_UNDEFINED |
EXCEPTION_SUPERVISOR |
EXCEPTION_PREFETCH_ABORT |
EXCEPTION_DATA_ABORT |
EXCEPTION_ADDRESS |
EXCEPTION_INTERRUPT |
EXCEPTION_FAST
```

```

ARMversion =
  ARMv4 | ARMv4T |
  ARMv5T | ARMv5TE |
  ARMv6 | ARMv6K | ARMv6T2 |
  ARMv7_A | ARMv7_R | ARMv7_M

```

```

ARMextensions =
  EXTENSION_THUMBEE | EXTENSION_VFP | EXTENSION_ADVANVEDSIMD |
  EXTENSION_SECURITY | EXTENSION_JAZELLE | EXTENSION_MULTIPROCESSING

```

```

ARMinfo =
  ⟨ version : ARMversion;
    extensions : ARMextensions set ⟩

```

```

InstrSet =
  INSTRSET_ARM | INSTRSET_THUMB | INSTRSET_JAZELLE | INSTRSET_THUMBEE

```

```

MemType =
  MEMTYPE_NORMAL | MEMTYPE_DEVICE | MEMTYPE_STRINGLYORDERED

```

```

MemoryAttributes =
  ⟨ type : MemType;
    innerattrs : word2;
    outerattrs : word2;
    shareable : bool;
    outershareable : bool ⟩

```

```

FullAddress =
  ⟨ physicaladdress : word32;
    physicaladdressex : word8;
    NS : bool(* F = Secure; T = Non-secure *) ⟩

```

```

AddressDescriptor =
  ⟨ memattrs : MemoryAttributes;
    paddress : FullAddress ⟩

```

**type\_abbrev** ExclusiveMonitor : FullAddress#iid#num → bool

```

ExclusiveMonitors =
  ⟨ TranslateAddress :
    word32#bool#bool → AddressDescriptor;
    (* TranslateAddress(VA,ispriv,iswrite) converts a virtual address to an address descriptor. Implementation depends on the memory architecture. *)
    MarkExclusiveGlobal :

```

```

(FullAddress#iid#num) → ExclusiveMonitor → ExclusiveMonitor;
MarkExclusiveLocal :
(FullAddress#iid#num) → ExclusiveMonitor → ExclusiveMonitor;
ClearExclusiveLocal : iid → ExclusiveMonitor#ExclusiveMonitor →
    ExclusiveMonitor#ExclusiveMonitor;
IsExclusiveLocal : ExclusiveMonitor;
IsExclusiveGlobal : ExclusiveMonitor}

```

```

MBReqDomain =
MBREQDOMAIN_FULLSYSTEM |
MBREQDOMAIN_OUTERSHAREABLE |
MBREQDOMAIN_INNERSHAREABLE |
MBREQDOMAIN_NONSHAREABLE

```

```

MBReqTypes = MBREQTYPES_ALL | MBREQTYPES_WRITES

```

```

word5_to_mode(m : word5) =

```

```

case m of
  16w → SOME USR
  || 17w → SOME FIQ
  || 18w → SOME IRQ
  || 19w → SOME SVC
  || 23w → SOME ABT
  || 27w → SOME UND
  || 31w → SOME SYS
  || _ → NONE

```

```

align(w : 'a word, n : num) : 'a word =
n2w(n * (w2n w div n))

```

```

decode_psr(psr : word32) =
{ N := psr[31];
  Z := psr[30];
  C := psr[29];
  V := psr[28];
  Q := psr[27];
  IT := ((15 >< 10)psr : word6)@@((26 >< 25)psr : word2);
  J := psr[24];
  Reserved := (23 >< 20)psr;
  GE := (19 >< 16)psr;
  E := psr[9];
  A := psr[8];
  I := psr[7];
  F := psr[6];
  T := psr[5];
  M := (4 >< 0)psr}

```

```

encode_psr(psr : ARMstatus) : word32 =
word_modify(λx b.
if x < 5 then psr.M[x] else
if x = 5 then psr.T else
if x = 6 then psr.F else
if x = 7 then psr.I else
if x = 8 then psr.A else
if x = 9 then psr.E else
if x < 16 then psr.IT[(x - 8)] else
if x < 20 then psr.GE[(x - 16)] else
if x < 24 then psr.Reserved[(x - 20)] else
if x = 24 then psr.J else
if x < 27 then psr.IT[(x - 25)] else
if x = 27 then psr.Q else
if x = 28 then psr.V else
if x = 29 then psr.C else
if x = 30 then psr.Z else
(* x = 31 *)psr.N)0w

```

```

(version_number ARMv4 = 4) ∧
(version_number ARMv4T = 4) ∧
(version_number ARMv5T = 5) ∧
(version_number ARMv5TE = 5) ∧
(version_number ARMv6 = 6) ∧
(version_number ARMv6K = 6) ∧
(version_number ARMv6T2 = 6) ∧
(version_number ARMv7_A = 7) ∧
(version_number ARMv7_R = 7) ∧
(version_number ARMv7_M = 7)

```



Part V  
arm\_types

*arm\_reg*

20

arm\_reg = REG32 of ARMreg | REGPSR of ARMpsr

**Part VI**  
**arm\_ast**



| DSP\_ADD\_SUBTRACT **of** word2 word4 word4 word4  
| DSP\_MULTIPLY **of** word4 word4 word4 bool word4

## Part VII

### arm\_opsem

user\_or\_system\_mode  $m = (m = \text{USR}) \vee (m = \text{SYS})$

```
word4_mode_to_reg( $w : \text{word4}, m$ ) =
num2ARMreg(let  $n = \mathbf{w2n} \ w$  in
(if  $(n = 15) \vee \text{user\_or\_system\_mode } m \vee$ 
 $(m = \text{FIQ}) \wedge n < 8 \vee$ 
 $\neg(m = \text{FIQ}) \wedge n < 13$ 
then
 $n$ 
else case  $m$  of
   $\text{FIQ} \rightarrow n + 8$ 
  ||  $\text{IRQ} \rightarrow n + 10$ 
  ||  $\text{SVC} \rightarrow n + 12$ 
  ||  $\text{ABT} \rightarrow n + 14$ 
  ||  $\text{UND} \rightarrow n + 16$ 
  ||  $\_ \rightarrow \text{ARB}$ ))
```

read\_regm  $ii \ x = \text{read\_reg } ii(\text{word4\_mode\_to\_reg } x)$

write\_regm  $ii \ x = \text{write\_reg } ii(\text{word4\_mode\_to\_reg } x)$

```
mode_to_psr  $mode =$ 
case  $mode$  of
   $\text{USR} \rightarrow \text{CPSR}$ 
  ||  $\text{FIQ} \rightarrow \text{SPSR\_FIQ}$ 
  ||  $\text{IRQ} \rightarrow \text{SPSR\_IRQ}$ 
  ||  $\text{SVC} \rightarrow \text{SPSR\_SVC}$ 
  ||  $\text{ABT} \rightarrow \text{SPSR\_ABT}$ 
  ||  $\text{UND} \rightarrow \text{SPSR\_UND}$ 
  ||  $\text{SYS} \rightarrow \text{CPSR}$ 
```

```
exception_to_mode  $type =$ 
case  $type$  of
   $\text{EXCEPTION\_RESET} \rightarrow \text{SVC}$ 
  ||  $\text{EXCEPTION\_UNDEFINED} \rightarrow \text{UND}$ 
  ||  $\text{EXCEPTION\_SUPERVISOR} \rightarrow \text{SVC}$ 
  ||  $\text{EXCEPTION\_ADDRESS} \rightarrow \text{SVC}$ 
  ||  $\text{EXCEPTION\_PREFETCH\_ABORT} \rightarrow \text{ABT}$ 
  ||  $\text{EXCEPTION\_DATA\_ABORT} \rightarrow \text{ABT}$ 
  ||  $\text{EXCEPTION\_INTERRUPT} \rightarrow \text{IRQ}$ 
  ||  $\text{EXCEPTION\_FAST} \rightarrow \text{FIQ}$ 
```

```
mode_to_word5  $mode : \text{word5} =$ 
case  $mode$  of
   $\text{USR} \rightarrow 16w$ 
  ||  $\text{FIQ} \rightarrow 17w$ 
```

```

|| IRQ → 18w
|| SVC → 19w
|| ABT → 23w
|| UND → 27w
|| SYS → 31w

```

```

exception_to_address high_vectors type : word32 =
(if high_vectors then 0xFFFF0000w else 0w) +
n2w(4 * ARMexception2num type)

```

```

condition_passed(N, Z, C, V)cond =
case cond of
  EQ → Z
|| NE → ¬Z
|| CS → C
|| CC → ¬C
|| MI → N
|| PL → ¬N
|| VS → V
|| VC → ¬V
|| HI → C ∧ ¬Z
|| LS → ¬C ∨ Z
|| GE → N = V
|| LT → ¬(N = V)
|| GT → ¬Z ∧ (N = V)
|| LE → Z ∨ ¬(N = V)
|| AL → T
|| NV → F

```

```

inc_pc ii = read_reg ii R15 >>= (λx. write_reg ii R15(x + 4w))

```

```

branch_write_pc ii(addr : word32) =
(read_version ii!! read_instr_set ii) >>=
(λ(version, iset).
  if iset = INSTRSET_ARM then
    if version < 6 ∧ ((1 >< 0)addr = 0w : word2)
    then
      failureT
    else
      write_reg ii R15((31 ≠ 2)addr)
  else
    write_reg ii R15((31 ≠ 1)addr))

```

```

LSL(m : word32)(n : word8)c =
if n = 0w then (c, m) else
(n <= +32w ∧ m[(32 - w2n n)], m << w2n n)

```



```
LSR(m : word32)(n : word8)c =
if n = 0w then LSL m 0w c else
(n <= +32w ∧ m[(w2n n - 1)], m >>> w2n n)
```

```
ASR(m : word32)(n : word8)c =
if n = 0w then LSL m 0w c else
(m[(min 31(w2n n - 1))], m >> w2n n)
```

```
ROR(m : word32)(n : word8)c =
if n = 0w then LSL m 0w c else
(m[(w2n((w2w n) : word5) - 1)], m# >> w2n n)
```

```
immediate ii(rot : word4)(imm : word8) =
(inc_pc ii!! read_flags ii) >>=
(λ(unit, (n, z, c, v)). constT(ROR(w2w imm)(2w * w2w rot)c))
```

```
shift_immediate ii(shift : word5)(sh : word2)rm =
(read_flags ii!! read_mode ii) >>=
(λ((n, z, c, v), m).
  read_regm ii(rm, m) >>=
  (λrm. inc_pc ii >> -
    constT
    case sh of
      0w → LSL rm(w2w shift)c
    || 1w → LSR rm(if shift = 0w then 32w else w2w shift)c
    || 2w → ASR rm(if shift = 0w then 32w else w2w shift)c
    || _ → if shift = 0w then
      word_rrx(c, rm)
    else
      ROR rm(w2w shift)c))
```

```
shift_register ii rs(sh : word2)rm =
(read_flags ii!! read_mode ii) >>=
(λ((n, z, c, v), m).
  read_regm ii(rs, m) >>=
  (λrs. inc_pc ii >> -
    read_regm ii(rm, m) >>=
    (λrm. constT
      case sh of
        0w → LSL rm(w2w rs)c
      || 1w → LSR rm(w2w rs)c
      || 2w → ASR rm(w2w rs)c
      || _ → ROR rm(w2w rs)c)))
```

```

(addr_mode1 ii(MODE1_IMMEDIATE rot imm) =
immediate ii rot imm) ∧
(addr_mode1 ii(MODE1_SHIFT_IMMEDIATE shift sh rm) =
shift_immediate ii shift sh rm) ∧
(addr_mode1 ii(MODE1_SHIFT_REGISTER rs sh rm) =
shift_register ii rs sh rm)

```

```

(addr_mode2 ii u w rn(MODE2_IMMEDIATE offset) =
read_mode ii >>=
(λm.
  read_regm ii(rn, m) >>=
  (λrn.inc_pc ii >> -
    let address = if u then rn + w2w offset else rn - w2w offset in
    constT(if w then address else rn, address)))) ∧
(addr_mode2 ii u w rn(MODE2_SHIFT_IMMEDIATE shift sh rm) =
read_mode ii >>=
(λm.
  read_regm ii(rn, m) >>=
  (λrn.shift_immediate ii shift sh rm >>=
    (λ(c_shift, offset).
      let address = if u then rn + offset else rn - offset in
      constT(if w then address else rn, address))))))

```

```

ALU_arith op(rn : word32)(op2 : word32) =
let sign = word_msb rn
and (q, r) = DIVMOD_2EXP 32(op(w2n rn)(w2n op2)) in
let res = (n2w r) : word32 in
((word_msb res, r = 0, ODD q,
  (word_msb op2 = sign) ∧ ¬(word_msb res = sign)), res)

```

```

ALU_logic(res : word32) = ((word_msb res, res = 0w, F, F), res)

```

```

ALU_multiply a rn rs rm =
let res : word32 = rm * rs + (if a then rn else 0w) in
(word_msb res, res = 0w, res)

```

```

ALU_multiply_long s a(rdhi : word32)(rdlo : word32)(rs : word32)(rm : word32) =
let res : word64 =
  (if a then rdhi@rdlo else 0w : word64) +
  (if s then sw2sw rm * sw2sw rs else w2w rm * w2w rs)
in
let reshi = (63 >< 32)res : word32
and reslo = (31 >< 0)res : word32
in
(word_msb res, res = 0w, reshi, reslo)

```

ADD  $a\ b\ c = \text{ALU\_arith}(\lambda x\ y.\ x + y + (\text{if } c \text{ then } 1 \text{ else } 0))a\ b$

SUB  $a\ b\ c = \text{ADD } a(-b)c$

AND  $a\ b = \text{ALU\_logic}(a\&\&b)$

EOR  $a\ b = \text{ALU\_logic}(a\?\?b)$

ORR  $a\ b = \text{ALU\_logic}(a\!\!b)$

ALU( $opc : \text{word4}$ ) $rn\ op2\ c =$

**case**  $opc$  **of**

$0w \rightarrow \text{AND } rn\ op2$   
 ||  $1w \rightarrow \text{EOR } rn\ op2$   
 ||  $2w \rightarrow \text{SUB } rn\ op2\ \mathbf{T}$   
 ||  $4w \rightarrow \text{ADD } rn\ op2\ \mathbf{F}$   
 ||  $3w \rightarrow \text{SUB } op2\ rn\ \mathbf{T}$   
 ||  $5w \rightarrow \text{ADD } rn\ op2\ c$   
 ||  $6w \rightarrow \text{SUB } rn\ op2\ c$   
 ||  $7w \rightarrow \text{SUB } op2\ rn\ c$   
 ||  $8w \rightarrow \text{AND } rn\ op2$   
 ||  $9w \rightarrow \text{EOR } rn\ op2$   
 ||  $10w \rightarrow \text{SUB } rn\ op2\ \mathbf{T}$   
 ||  $11w \rightarrow \text{ADD } rn\ op2\ \mathbf{F}$   
 ||  $12w \rightarrow \text{ORR } rn\ op2$   
 ||  $13w \rightarrow \text{ALU\_logic } op2$   
 ||  $14w \rightarrow \text{AND } rn(-op2)$   
 ||  $\_ \rightarrow \text{ALU\_logic}(\neg op2)$

arithmetic( $opcode : \text{word4}$ ) =

$(opcode[2] \vee opcode[1]) \wedge (\neg(opcode[3]) \vee \neg(opcode[2]))$

test\_or\_compare( $opcode : \text{word4}$ ) =  $((3 - -2)opcode = 2w)$

rotate\_mem32( $oareg : \text{word2}$ )( $w : \text{word32}$ ) =  $w\#\ \gg\ (8 * \mathbf{w2n}\ oareg)$

exception\_exec  $ii\ type =$

discardT

$((\text{read\_reg } ii\ \mathbf{R15}!! \text{read\_sctlr } ii!! \text{read\_psr } ii\ \mathbf{CPSR}) \gg =$

$(\lambda(pc, sctlr, cpsr).$

**(let**  $mode = \text{exception\_to\_mode } type$  **in**

$\text{write\_regm } ii(14w, mode)(pc - 4w)!!$

$\text{branch\_write\_pc } ii(\text{exception\_to\_address } sctlr.V\ type)!!$

$\text{write\_psr } ii(\text{mode\_to\_psr } mode)cpsr!!$

```

write_psr ii CPSR(cpsr
  ⟨ I := T;
    F := (type ∈ {EXCEPTION_RESET; EXCEPTION_FAST} ∨ cpsr.F);
    A := ((type = EXCEPTION_RESET) ∨ cpsr.A); IT := 0w; J := F;
    T := sctlr.TE; E := sctlr.EE;
    M := mode_to_word5 mode⟩))))

```

```

branch_exec ii(BRANCH l offset) =
read_reg ii R15 >>=
(λpc.
  let target = pc + sw2sw offset << 2 in
  if ¬(word_msb target = word_msb(pc - 8w)) then
    failureT
  else
    if l then
      discardT(read_mode ii >>=
        (λm.
          write_regm ii(14w, m)(pc - 4w)!!
          branch_write_pc ii target))
    else
      branch_write_pc ii target)

```

```

data_processing_exec ii(DATA_PROCESSING opcode s rn rd op2) =
(addr_model ii op2!! read_flags ii!! read_mode ii) >>=
(λ((c_shift, opnd2), (n, z, c, v), m).
  read_regm ii(rn, m) >>=
  (λrn.
    let ((n_alu, z_alu, c_alu, v_alu), res) = ALU opcode rn opnd2 c_shift in
    discardT
      (condT(¬test_or_compare opcode)
        (write_regm ii(rd, m)res)!!
      condT s
        (if (rd = 15w) ∧ ¬(test_or_compare opcode) then
          read_psr ii(mode_to_psr m) >>=
          (λspsr. write_psr ii CPSR spsr)
        else
          write_flags ii
            (if arithmetic opcode then
              (n_alu, z_alu, c_alu, v_alu)
            else
              (n_alu, z_alu, c_shift, v))))))

```

```

multiply_exec ii(MULTIPLY a s rd rn rs rm) =
if (rd = 15w) ∨ (rm = 15w) ∨ (rs = 15w) ∨ (rd = rm) then
  failureT
else
  discardT((read_flags ii!! read_mode ii) >>=
  (λ((n, z, c, v), m).

```

```

(read_regm ii(rn, m)!!
 read_regm ii(rs, m)!!
 read_regm ii(rm, m)) >>=
( $\lambda$ (rn, rs, rm).
  let (n_alu, z_alu, res) = ALU_multiply a rn rs rm in
  inc_pc ii!!
  write_regm ii(rd, m)res!!
  condT s(read_version ii >>=
( $\lambda$ version.
  let c_flag = if version = 4 then ARB else c in
  write_flags ii(n_alu, z_alu, c_flag, v))))))

multiply_long_exec ii(MULTIPLY_LONG u a s rdhi rdlo rs rm) =
if (rdhi = 15w)  $\vee$  (rdlo = 15w)  $\vee$  (rm = 15w)  $\vee$  (rs = 15w)  $\vee$ 
  (rdhi = rm)  $\vee$  (rdlo = rm)  $\vee$  (rdhi = rdlo) then
failureT
else
discardT((read_flags ii!! read_mode ii) >>=
( $\lambda$ ((n, z, c, v), m).
  (read_regm ii(rdhi, m)!!
  read_regm ii(rdlo, m)!!
  read_regm ii(rs, m)!!
  read_regm ii(rm, m)) >>=
( $\lambda$ (rdhi', rdlo', rs, rm).
  let (n_alu, z_alu, reshi, reslo) =
    ALU_multiply_long u a rdhi' rdlo' rs rm
  in
  inc_pc ii!!
  write_regm ii(rdhi, m)reshi!!
  write_regm ii(rdlo, m)reslo!!
  condT s(read_version ii >>=
( $\lambda$ version.
  let c_flag = if version = 4 then ARB else c in
  write_flags ii(n_alu, z_alu, c_flag, v))))))

load_store_exec ii(LOAD_STORE p u b w l rn rd op2) =
let wb = p  $\implies$  w in
if wb  $\wedge$  (rn = rd) then
  failureT
else
  discardT((addr_mode2 ii u w rn op2!! read_mode ii) >>=
  ( $\lambda$ (address, wb_address), m).
    if l then
      condT wb(write_regm ii(rn, m)wb_address)!!
      (if b then
        read_mem8 ii address >>= ( $\lambda$ d. constT(w2w d))
      else
        read_mem32 ii address >>=

```

```

      ( $\lambda d$ . constT(rotate_mem32( $w2w$  address) $d$ ))) >>=
      ( $\lambda data$ . write_regm  $ii(rd, m) data$ )
    else
      read_regm  $ii(rd, m) >>=$ 
      ( $\lambda rd$ .
        if  $b$  then
          write_mem8  $ii$  address( $w2w rd$ )
        else
          write_mem32  $ii$  address  $rd$ !!
      condT  $wb$ (write_regm  $ii(rn, m)wb\_address$ )))

load_exclusive_exec  $ii$ (LOAD_EXCLUSIVE  $rn rt imm8$ ) =
read_version  $ii >>=$ 
( $\lambda version$ .
  if  $version < 6$  then
    exception_exec  $ii$  EXCEPTION_UNDEFINED
  else
    if ( $rt = 15w$ )  $\vee$  ( $rn = 15w$ ) then
      failureT
    else
      read_mode  $ii >>=$ 
      ( $\lambda m$ .
        read_regm  $ii(rn, m) >>=$ 
        ( $\lambda rn$ .let address =  $rn + w2w imm8$  in
          set_exclusive_monitorsT  $ii$ (address, 4) >> -
          read_mem32  $ii$  address >>=
          ( $\lambda d$ . write_regm  $ii(rt, m)d$ ))))))

store_exclusive_exec  $ii$ (STORE_EXCLUSIVE  $rn rd rt imm8$ ) =
read_version  $ii >>=$ 
( $\lambda version$ .
  if  $version < 6$  then
    exception_exec  $ii$  EXCEPTION_UNDEFINED
  else
    if ( $rd = 15w$ )  $\vee$  ( $rt = 15w$ )  $\vee$  ( $rn = 15w$ )  $\vee$ 
      ( $rd = rt$ )  $\vee$  ( $rd = rt$ )
    then
      failureT
    else
      read_mode  $ii >>=$ 
      ( $\lambda m$ .
        read_regm  $ii(rn, m) >>=$ 
        ( $\lambda rn$ .let address =  $rn + w2w imm8$  in
          (exclusive_monitors_passT  $ii$ (address, 4) >>=
          ( $\lambda pass$ .
            if  $pass$  then
              read_regm  $ii(rt, m) >>=$ 
              ( $\lambda rt$ .

```

```

        discardT
        (write_mem32 ii address rt!!
         write_regm ii(rd, m)0w))
    else
        write_regm ii(rd, m)1w))))))

swap_exec ii(SWAP b rn rd rm) =
if (rd = 15w) ∨ (rn = 15w) ∨ (rm = 15w) ∨ (rn = rm) ∨ (rn = rd) then
failureT
else
lockT(read_mode ii >>=
(λm.
  (read_regm ii(rn, m)!! read_regm ii(rm, m)) >>=
   (λ(address, rm).
    (if b then
      read_mem8 ii address >>= (λd. constT(w2w d))
    else
      read_mem32 ii address >>=
      (λd. constT(rotate_mem32(w2w address)d))) >>=
      (λdata. discardT
       (inc_pc ii!!
        write_mem32 ii address rm!!
        write_regm ii(rd, m)data))))))

status_to_register_exec ii(STATUS_TO_REGISTER r rd) =
read_psr ii CPSR >>=
(λcpsr.
  let mode = word5_to_mode cpsr.M in let m = the mode in
  if (rd = 15w) ∨ IS_NONE mode ∨ user_or_system_mode m ∧ r then
  failureT
  else
  (if r then
    read_psr ii(mode_to_psr m) >>= (λspsr. constT(encode_psr spsr))
  else
    constT(encode_psr cpsr&&
            0b11111000_11111111_00000011_11011111w)) >>=
    (λspsr. discardT(inc_pc ii!! write_regm ii(rd, m)psr)))

breakpoint_exec ii cond =
if ¬(cond = AL) then
failureT
else
read_version ii >>=
(λversion.
  if version = 4 then
  exception_exec ii EXCEPTION_UNDEFINED
  else
  exception_exec ii EXCEPTION_PREFETCH_ABORT)

```

```

data_memory_barrier_exec ii(DATA_MEMORY_BARRIER option) =
read_version ii >>=
( $\lambda$ version.
  if version < 7 then
    exception_exec ii EXCEPTION_UNDEFINED
  else
    discardT(inc_pc ii!! dmbT ii
      (case option of
        0b0010w  $\rightarrow$  (MBREQDOMAIN_OUTERSHAREABLE, MBREQTYPES_WRITES)
        || 0b0011w  $\rightarrow$  (MBREQDOMAIN_OUTERSHAREABLE, MBREQTYPES_ALL)
        || 0b0110w  $\rightarrow$  (MBREQDOMAIN_NONSHAREABLE, MBREQTYPES_ALL)
        || 0b0111w  $\rightarrow$  (MBREQDOMAIN_NONSHAREABLE, MBREQTYPES_ALL)
        || 0b1010w  $\rightarrow$  (MBREQDOMAIN_INNERSHAREABLE, MBREQTYPES_WRITES)
        || 0b1011w  $\rightarrow$  (MBREQDOMAIN_INNERSHAREABLE, MBREQTYPES_ALL)
        || 0b1110w  $\rightarrow$  (MBREQDOMAIN_FULLSYSTEM, MBREQTYPES_WRITES)
        || _  $\rightarrow$  (MBREQDOMAIN_FULLSYSTEM, MBREQTYPES_ALL))))

```

```

arm_execute ii(cond, inst) =
read_flags ii >>=
( $\lambda$ flags.
  if condition_passed flags cond then
    case inst of
      BRANCH l offset  $\rightarrow$ 
        branch_exec ii inst
      || SWAP b rn rd rm  $\rightarrow$ 
        swap_exec ii inst
      || DATA_PROCESSING opcode s rn rd op2  $\rightarrow$ 
        data_processing_exec ii inst
      || LOAD_STORE p u b w l rn rd op2_1  $\rightarrow$ 
        load_store_exec ii inst
      || LOAD_EXCLUSIVE rn rt imm8  $\rightarrow$ 
        load_exclusive_exec ii inst
      || STORE_EXCLUSIVE rn rd rt imm8  $\rightarrow$ 
        store_exclusive_exec ii inst
      || MULTIPLY_LONG u a s rdhi rdlo rs rm  $\rightarrow$ 
        multiply_long_exec ii inst
      || MULTIPLY a s rd rn rs rm  $\rightarrow$ 
        multiply_exec ii inst
      || STATUS_TO_REGISTER r rd  $\rightarrow$ 
        status_to_register_exec ii inst
      || BREAKPOINT number16  $\rightarrow$ 
        breakpoint_exec ii cond
      || DATA_MEMORY_BARRIER option  $\rightarrow$ 
        data_memory_barrier_exec ii inst
      || SUPERVISOR_CALL number24  $\rightarrow$ 
        exception_exec ii EXCEPTION_SUPERVISOR
      || UNDEFINED  $\rightarrow$ 
        exception_exec ii EXCEPTION_UNDEFINED
      || _  $\rightarrow$ 

```



```
        failureT  
else  
    inc_pc ii)
```

## Part VIII

# arm\_seq\_monad

**type\_abbrev** arm\_state : (ARMreg  $\rightarrow$  word32)#(\* - general-purpose registers \*)  
 (ARMpsr  $\rightarrow$  ARMstatus)#(\* - program-status registers \*)  
 ARMcp\_registers#(\* - co-processor registers \*)  
 (word32  $\rightarrow$  word8 option)#(\* - unsegmented memory \*)  
 ARMinfo#(\* - info on ISA version and extensions \*)  
 ExclusiveMonitors(\* - for synchronization & semaphores \*)

AREAD\_REG  $x \hat{\text{arm\_state}} = r \ x$

AREAD\_PSR  $x \hat{\text{arm\_state}} = p \ x$

AREAD\_CP  $\hat{\text{arm\_state}} = c$

AREAD\_MEM  $x \hat{\text{arm\_state}} = m \ x$

AREAD\_INFO  $\hat{\text{arm\_state}} = v$

AREAD\_EXCL  $\hat{\text{arm\_state}} = e$

AWRITE\_REG  $i \ x \hat{\text{arm\_state}} = ((i = +x)r, p, c, m, v, e) : \text{arm\_state}$

AWRITE\_PSR  $i \ x \hat{\text{arm\_state}} = (r, (i = +x)p, c, m, v, e) : \text{arm\_state}$

AWRITE\_MEM  $i \ x \hat{\text{arm\_state}} = (r, p, c, (i = +x)m, v, e) : \text{arm\_state}$

AWRITE\_EXCL  $x \hat{\text{arm\_state}} = (r, p, c, m, v, x) : \text{arm\_state}$

**type\_abbrev** M : arm\_state  $\rightarrow$  ('a#arm\_state) option

(constT\_seq : 'a  $\rightarrow$  'a M)x =  $\lambda y.$ SOME (x, y)

(failureT\_seq : 'a M) =  $\lambda y.$ NONE

(bindT\_seq : 'a M  $\rightarrow$  ('a  $\rightarrow$  'b M)  $\rightarrow$  'b M)s f =  
 $\lambda y.$ case s y of NONE  $\rightarrow$  NONE || SOME (z, t)  $\rightarrow$  f z t

(seqT\_seq : 'a M  $\rightarrow$  'b M  $\rightarrow$  'b M)s f =  
 bindT\_seq s( $\lambda x.$ f)

(parT\_seq : 'a M  $\rightarrow$  'b M  $\rightarrow$  ('a#'b)M)s t =  
 bindT\_seq s( $\lambda x.$  bindT\_seq t( $\lambda y.$  constT\_seq(x, y)))

(lockT\_seq : 'a M → 'a M) s = s

(condT\_seq : bool → unit M → unit M) b s =  
**if** b **then** s **else** constT\_seq()

(discardT\_seq : 'a M → unit M) s =  
 seqT\_seq s(constT\_seq())

(addT\_seq : 'a → 'b M → ('a#'b)M) x s =  
 bindT\_seq s(λz. constT\_seq(x, z))

(write\_reg\_seq ii r x) : unit M =  
 λs.SOME ((), AWRITE\_REG r x s)

(read\_reg\_seq ii r) : Aimm M =  
 λs.SOME (**let** x = AREAD\_REG r s **in if** r = R15 **then** x + 8w **else** x, s)

(write\_psr\_seq ii r x) : unit M =  
 λs.**case** word5\_to\_mode x.M **of**  
   NONE → NONE  
   || SOME m → SOME ((), AWRITE\_PSR r x s)

(read\_psr\_seq ii r) : ARMstatus M =  
 λs.SOME (AREAD\_PSR r s, s)

(write\_flags\_seq ii(n, z, c, v)) : unit M =  
 λs.SOME ((),  
   **let** cpsr = AREAD\_PSR CPSR s ⟨ N := n; Z := z; C := c; V := v ⟩ **in**  
   AWRITE\_PSR CPSR cpsr s)

(read\_flags\_seq ii) : (bool#bool#bool#bool)M =  
 λs.SOME (**let** cpsr = AREAD\_PSR CPSR s **in** (cpsr.N, cpsr.Z, cpsr.C, cpsr.V), s)

(read\_endian\_seq ii) : bool M =  
 λs.SOME ((AREAD\_PSR CPSR s).E, s)

(read\_mode\_seq ii) : ARMmode M =  
 λs.**case** word5\_to\_mode(AREAD\_PSR CPSR s).M **of**  
   NONE → NONE  
   || SOME m → SOME (m, s)

```

(read_instr_set_seq ii) : InstrSet M =
λs.SOME (let cpsr = AREAD_PSR CPSR s in
  (case (cpsr.J, cpsr.T) of
    (F, F) → INSTRSET_ARM
  || (F, T) → INSTRSET_THUMB
  || (T, F) → INSTRSET_JAZELLE
  || (T, T) → INSTRSET_THUMBEE), s)

```

```

(read_sctlr_seq ii) : ARMsctlr M =
λs.SOME ((AREAD_CP s).SCTLR, s)

```

```

(read_info_seq ii) : ARMinfo M =
λs.SOME (AREAD_INFO s, s)

```

```

(read_version_seq ii) : num M =
λs.SOME (version_number(AREAD_INFO s).version, s)

```

```

(set_exclusive_monitorsT_seq ii(a : word32, size : num)) : unit M =
(λs.case word5_to_mode(AREAD_PSR CPSR s).M of
  NONE → NONE
|| SOME m → SOME ((),
  let monitor = AREAD_EXCL s in
  let memaddrdesc = monitor.TranslateAddress(a, ¬(m = USR), F)
  in
  AWRITE_EXCL
  (monitor
    ( IsExclusiveGlobal :=
      (if memaddrdesc.memattrs.shareable then
        monitor.MarkExclusiveGlobal
        (memaddrdesc.paddress, ii, size)
        monitor.IsExclusiveGlobal
      else monitor.IsExclusiveGlobal);
      IsExclusiveLocal :=
        monitor.MarkExclusiveLocal(memaddrdesc.paddress, ii, size)
        monitor.IsExclusiveLocal))s))

```

```

(exclusive_monitors_passT_seq ii(a : word32, size : num)) : bool M =
λs.case word5_to_mode(AREAD_PSR CPSR s).M of
  NONE → NONE
|| SOME m →
  let monitor = AREAD_EXCL s in
  let memaddrdesc = monitor.TranslateAddress(a, ¬(m = USR), F) in
  let local_pass = monitor.IsExclusiveLocal
    (memaddrdesc.paddress, ii, size) in
  let passed =
    if memaddrdesc.memattrs.shareable then

```

```

        monitor.IsExclusiveLocal(memaddrdesc.paddress, ii, size) ∧
        local_pass
    else local_pass
in
    SOME (passed,
        if passed then
            AWRITE_EXCL
            (let (local, global) =
                monitor.ClearExclusiveLocal ii
                (monitor.IsExclusiveLocal, monitor.IsExclusiveGlobal)
            in
                monitor
                ⟨⟨ IsExclusiveLocal := local;
                    IsExclusiveGlobal := global ⟩⟩s
            else s)

```

```

(dmbT_seq : iid → MReqDomain#MReqTypes → unit M) ii x =
λs.SOME ((), s)

```

```

(write_mem8_seq ii a x) : unit M =
(λs.case AREAD_MEM a s of
    NONE → NONE
  || SOME y → SOME ((), AWRITE_MEM a(SOME x)s))

```

```

(write_mem16_seq ii a(x : word16)) : unit M =
if a[0] then
failureT_seq
else
bindT_seq(read_endian_seq ii)
(λe.let l = word2bytes 4 x in
    discardT_seq
    (if e then
        parT_seq(write_mem8_seq ii a(EL 1 l))
                (write_mem8_seq ii(a + 1w)(EL 0 l))
    else
        parT_seq(write_mem8_seq ii a(EL 0 l))
                (write_mem8_seq ii(a + 1w)(EL 1 l))))

```

```

(write_mem32_seq ii a(x : word32)) : unit M =
bindT_seq(read_endian_seq ii)
(λe.let aa = align(a, 4) and l = word2bytes 4 x in
    discardT_seq
    (if e then
        parT_seq(write_mem8_seq ii aa(EL 3 l))
                (parT_seq(write_mem8_seq ii(aa + 1w)(EL 2 l))
                    (parT_seq(write_mem8_seq ii(aa + 2w)(EL 1 l))
                        (write_mem8_seq ii(aa + 3w)(EL 0 l))))
    else
        parT_seq(write_mem8_seq ii aa(EL 3 l))
                (parT_seq(write_mem8_seq ii(aa + 1w)(EL 2 l))
                    (parT_seq(write_mem8_seq ii(aa + 2w)(EL 1 l))
                        (write_mem8_seq ii(aa + 3w)(EL 0 l))))

```

```

    else
      parT_seq(write_mem8_seq ii aa(EL 0 l))
      (parT_seq(write_mem8_seq ii(aa + 1w)(EL 1 l))
       (parT_seq(write_mem8_seq ii(aa + 2w)(EL 2 l))
        (write_mem8_seq ii(aa + 3w)(EL 3 l))))))

(read_mem8_seq ii a) : word8 M =
(λs.case AREAD_MEM a s of NONE → NONE || SOME x → SOME (x, s))

(read_mem16_seq ii a) : word16 M =
if a[0] then
  failureT_seq
else
  bindT_seq(parT_seq(read_mem8_seq ii a)(read_mem8_seq ii(a + 1w)))
  (λ(b0, b1). constT_seq(b1@@b0))

(read_mem32_seq ii a) : word32 M =
  bindT_seq
  (let aa = align(a, 4) in
    parT_seq(read_mem8_seq ii aa)
    (parT_seq(read_mem8_seq ii(aa + 1w))
     (parT_seq(read_mem8_seq ii(aa + 2w))
      (read_mem8_seq ii(aa + 3w))))))
  (λ(b0, b1, b2, b3). constT_seq((b3@@b2@@b1@@b0)))

(constT : 'a → 'a M) = constT_seq

(addT : 'a → 'b M → ('a##'b)M) = addT_seq

(lockT : unit M → unit M) = lockT_seq

(failureT : unit M) = failureT_seq

(discardT : 'a M → unit M) = discardT_seq

(condT : bool → unit M → unit M) = condT_seq

(bindT : 'a M → ('a → 'b M) → 'b M) = bindT_seq

(seqT : 'a M → 'b M → 'b M) = seqT_seq

(parT : 'a M → 'b M → ('a##'b)M) = parT_seq

```

```

(read_info : iiid → ARMinfo M) = read_info_seq
(read_version : iiid → num M) = read_version_seq
(write_reg : iiid → ARMreg → Aimm → unit M) = write_reg_seq
(read_reg : iiid → ARMreg → Aimm M) = read_reg_seq
(write_psr :
  iiid → ARMpsr → ARMstatus → unit M) = write_psr_seq
(read_psr : iiid → ARMpsr → ARMstatus M) = read_psr_seq
(write_flags :
  iiid → bool#bool#bool#bool → unit M) = write_flags_seq
(read_flags :
  iiid → (bool#bool#bool#bool)M) = read_flags_seq
(read_mode : iiid → ARMmode M) = read_mode_seq
(read_instr_set : iiid → InstrSet M) = read_instr_set_seq
(read_sctlr : iiid → ARMsctlr M) = read_sctlr_seq
(write_mem8 : iiid → word32 → word8 → unit M) = write_mem8_seq
(read_mem8 : iiid → word32 → word8 M) = read_mem8_seq
(write_mem16 :
  iiid → word32 → word16 → unit M) = write_mem16_seq
(read_mem16 : iiid → word32 → word16 M) = read_mem16_seq
(write_mem32 :
  iiid → word32 → word32 → unit M) = write_mem32_seq
(read_mem32 : iiid → word32 → Aimm M) = read_mem32_seq
(dmbT :
  iiid → MBReqDomain#MBReqTypes → unit M) = dmbT_seq
(set_exclusive_monitorsT :
  iiid → (word32#num) → unit M) = set_exclusive_monitorsT_seq
(exclusive_monitors_passT :
  iiid → (word32#num) → bool M) = exclusive_monitors_passT_seq
option_apply x f = if x = NONE then NONE else f(the x)

```



## Part IX

### arm\_event\_monad

**type\_abbrev** M : iid\_state → ((iid\_state#'a#(arm\_reg event\_structure))set)

event\_structure\_empty = { events := {}; intra\_causality\_data := {}; intra\_causality\_control := {}; atomicity := {} }

event\_structure\_lock es = { events := es.events; intra\_causality\_data := es.intra\_causality\_data; intra\_causality\_control := es.intra\_causality\_control; atomicity := es.atomicity }

event\_structure\_union es<sub>1</sub> es<sub>2</sub> =  
 { events := es<sub>1</sub>.events ∪ es<sub>2</sub>.events;  
 intra\_causality\_data := es<sub>1</sub>.intra\_causality\_data ∪ es<sub>2</sub>.intra\_causality\_data;  
 intra\_causality\_control := es<sub>1</sub>.intra\_causality\_control ∪ es<sub>2</sub>.intra\_causality\_control;  
 atomicity := es<sub>1</sub>.atomicity ∪ es<sub>2</sub>.atomicity }

event\_structure\_bigunion(ess : (arm\_reg event\_structure)set) =  
 { events := **bigunion**{ es.events | es ∈ ess };  
 intra\_causality\_data := **bigunion**{ es.intra\_causality\_data | es ∈ ess };  
 intra\_causality\_control := **bigunion**{ es.intra\_causality\_control | es ∈ ess };  
 atomicity := **bigunion**{ es.atomicity | es ∈ ess } }

event\_structure\_seq\_union es<sub>1</sub> es<sub>2</sub> =  
 { events := es<sub>1</sub>.events ∪ es<sub>2</sub>.events;  
 intra\_causality\_data := es<sub>1</sub>.intra\_causality\_data  
 ∪ es<sub>2</sub>.intra\_causality\_data  
 ∪ { (e<sub>1</sub>, e<sub>2</sub>)  
 | e<sub>1</sub> ∈ (maximal\_elements es<sub>1</sub>.events es<sub>1</sub>.intra\_causality\_data)  
 ∧ e<sub>2</sub> ∈ (minimal\_elements es<sub>2</sub>.events es<sub>2</sub>.intra\_causality\_data) };  
 intra\_causality\_control := es<sub>1</sub>.intra\_causality\_control  
 ∪ es<sub>2</sub>.intra\_causality\_control;  
 atomicity := es<sub>1</sub>.atomicity ∪ es<sub>2</sub>.atomicity }

event\_structure\_control\_seq\_union es<sub>1</sub> es<sub>2</sub> =  
 { events := es<sub>1</sub>.events ∪ es<sub>2</sub>.events;  
 intra\_causality\_data := es<sub>1</sub>.intra\_causality\_data  
 ∪ es<sub>2</sub>.intra\_causality\_data;  
 intra\_causality\_control := es<sub>1</sub>.intra\_causality\_control  
 ∪ es<sub>2</sub>.intra\_causality\_control  
 ∪ { (e<sub>1</sub>, e<sub>2</sub>)  
 | e<sub>1</sub> ∈ (maximal\_elements es<sub>1</sub>.events es<sub>1</sub>.intra\_causality\_control)  
 ∧ e<sub>2</sub> ∈ (minimal\_elements es<sub>2</sub>.events es<sub>2</sub>.intra\_causality\_control) };  
 atomicity := es<sub>1</sub>.atomicity ∪ es<sub>2</sub>.atomicity }

(mapT\_ev : ('a → 'b) → 'a M → 'b M) f s =

λ iid\_next : iid\_state.

**let** t = s iid\_next **in**  
 { (iid\_next', f x, es)  
 | (iid\_next', x, es) ∈ t }

(choiceT\_ev : 'a M → 'a M → 'a M) s s' =  
 λ eiid\_next : eiid\_state. s eiid\_next ∪ s' eiid\_next

(constT\_ev : 'a → 'a M) x = λ eiid\_next. {(eiid\_next, x, event\_structure\_empty)}

(discardT\_ev : 'a M → unit M) s =  
 λ eiid\_next. let (t : (eiid\_state# 'a# (arm\_reg event\_structure))set) = s eiid\_next in  
 image(λ (eiid\_next', v, es). (eiid\_next', (), es)) t

(addT\_ev : 'a → 'b M → ('a# 'b) M) x s =  
 λ eiid\_next. let (t : (eiid\_state# 'b# (arm\_reg event\_structure))set) = s eiid\_next in  
 image(λ (eiid\_next', v, es). (eiid\_next', (x, v), es)) t

(lockT\_ev : 'a M → 'a M) s =  
 λ eiid\_next. let (t : (eiid\_state# 'a# (arm\_reg event\_structure))set) = s eiid\_next in  
 image(λ (eiid\_next', v, es). (eiid\_next', v, event\_structure\_lock es)) t

(failureT\_ev : 'a M) = λ eiid\_next. {}

(condT\_ev : bool → unit M → unit M) b s =  
 if b then s else constT\_ev()

(bindT\_ev : 'a M → ('a → 'b M) → 'b M) s f =  
 λ eiid\_next : eiid\_state.  
 let t = s eiid\_next in  
 bigunion { let t' = f x eiid\_next' in  
 {(eiid\_next'', x', event\_structure\_seq\_union es es')  
 | (eiid\_next'', x', es') ∈ t'}  
 | (eiid\_next', x, es) ∈ t }

(controlseqT\_ev : 'a M → ('a → 'b M) → 'b M) s f =  
 λ eiid\_next : eiid\_state.  
 let t = s eiid\_next in  
 bigunion { let t' = f x eiid\_next' in  
 {(eiid\_next'', x', event\_structure\_controlseq\_union es es')  
 | (eiid\_next'', x', es') ∈ t'}  
 | (eiid\_next', x, es) ∈ t }

(seqT\_ev : 'a M → 'b M → 'b M) s s' =  
 bindT\_ev s (λ x. s')

(parT\_ev : 'a M → 'b M → ('a##'b)M)s s' =  
 λeiiid\_next : eiiid\_state.

**let** t = s eiiid\_next **in**  
**bigunion**{**let** t' = s' eiiid\_next' **in**  
   {(eiiid\_next'', (x, x'), event\_structure\_union es es')  
   | (eiiid\_next'', x', es') ∈ t'}  
 | (eiiid\_next', x, es) ∈ t}

(parT\_unit\_ev : unit M → unit M → unit M)s s' =  
 λeiiid\_next : eiiid\_state.

**let** t = s eiiid\_next **in**  
**bigunion**{**let** t' = s' eiiid\_next' **in**  
   {(eiiid\_next'', (), event\_structure\_union es es')  
   | (eiiid\_next'', (), es') ∈ t'}  
 | (eiiid\_next', (), es) ∈ t}

(syncT\_ev ii) : unit M =

λeiiid\_next.{(eiiid\_next',  
 (),  
 ⌊ events := {⌊ eiiid := eiiid';  
   iidd := ii;  
   action := BARRIER SYNC⌋};  
 intra\_causality\_data := {};  
 intra\_causality\_control := {};  
 atomicity := {}}) | (eiiid', eiiid\_next') ∈ next\_eiid eiiid\_next}

(write\_location\_ev ii l x) : unit M =

λeiiid\_next.{(eiiid\_next',  
 (),  
 ⌊ events := {⌊ eiiid := eiiid';  
   iidd := ii;  
   action := ACCESS W l x⌋};  
 intra\_causality\_data := {};  
 intra\_causality\_control := {};  
 atomicity := {}}) | (eiiid', eiiid\_next') ∈ next\_eiid eiiid\_next}

(read\_location\_ev ii l) : value M =

λeiiid\_next.{(eiiid\_next',  
 x,  
 ⌊ events := {⌊ eiiid := eiiid';  
   iidd := ii;  
   action := ACCESS R l x⌋};  
 intra\_causality\_data := {};  
 intra\_causality\_control := {};  
 atomicity := {}})  
 | x ∈ UNIV ∧ (eiiid', eiiid\_next') ∈ next\_eiid eiiid\_next}

```
(write_reg_ev ii r x) : unit M =
write_location_ev ii(LOCATION_REG ii.proc(REG32 r))x
```

```
(read_reg_ev ii r) : value M =
read_location_ev ii(LOCATION_REG ii.proc(REG32 r))
```

```
(write_psr_ev ii r x) : unit M =
write_location_ev ii(LOCATION_REG ii.proc(REGPSR r))(encode_psr x)
```

```
(read_psr_ev ii r) : ARMstatus M =
bindT_ev(read_location_ev ii(LOCATION_REG ii.proc(REGPSR r)))( $\lambda psrw$ .constT_ev(decode_psr psrw))
```

```
(write_flags_ev ii(bn, bz, bc, bv)) : unit M =
lockT_ev(bindT_ev(read_psr_ev ii CPSR)
          ( $\lambda psr$ .write_psr_ev ii CPSR(psr { N := bn; Z := bz; C := bc; V := bv })))
```

```
(read_flags_ev ii) : (bool#bool#bool#bool)M =
bindT_ev(read_psr_ev ii CPSR)( $\lambda psr$ .constT_ev(psr.N, psr.Z, psr.C, psr.V))
```

```
OUR_VERSION = ARMv7_A
```

```
OUR_INFO = { version := OUR_VERSION; extensions := {} }
```

```
OUR_MODE = USR
```

```
OUR_INSTR_SET = INSTRSET_ARM
```

```
OUR_SCTLR = { TE := T; AFE := T; TRE := T; NMFI := T; EE := T; VE := T; U := T; FI := T; HA := T; RR := T; ... }
```

```
(read_version_ev ii) : num M =
constT_ev(version_number OUR_VERSION)
```

```
(read_info_ev ii) : ARMinfo M =
constT_ev(OUR_INFO)
```

```
(read_mode_ev ii) : ARMmode M =
constT_ev(OUR_MODE)
```

```
(read_instr_set_ev ii) : InstrSet M =
constT_ev(OUR_INSTR_SET)
```

```
(read_sctlr_ev ii) : ARMsctlr M =
constT_ev(OUR_SCTLR)
```

```
(set_exclusive_monitorsT_ev ii(a : word32, size : num)) : unit M =
failureT_ev
```

```
(exclusive_monitors_passT_ev ii(a : word32, size : num)) : bool M =
failureT_ev
```

```
aligned32 a = ((a&&3w) = 0w)
```

```
(write_mem8_ev ii a x) : unit M =
failureT_ev
```

```
(write_mem16_ev ii a(x : word16)) : unit M =
failureT_ev
```

```
(write_mem32_ev ii a(x : word32)) : unit M =
if aligned32 a then
write_location_ev ii(LOCATION_MEM a)x
else
failureT_ev
```

```
(read_mem8_ev ii a) : word8 M =
failureT_ev
```

```
(read_mem16_ev ii a) : word16 M =
failureT_ev
```

```
(read_mem32_ev ii a) : word32 M =
if aligned32 a then
read_location_ev ii(LOCATION_MEM a)
else
failureT_ev
```

```
(dmbT_ev ii(d, t)) : unit M =
syncT_ev ii
```

```
(constT : 'a → 'a M) = constT_ev
```

```
(addT : 'a → 'b M → ('a##'b)M) = addT_ev
```

(lockT : unit M → unit M) = lockT\_ev

(failureT : unit M) = failureT\_ev

(discardT : 'a M → unit M) = discardT\_ev

(condT : bool → unit M → unit M) = condT\_ev

(bindT : 'a M → ('a → 'b M) → 'b M) = bindT\_ev

(seqT : 'a M → 'b M → 'b M) = seqT\_ev

(parT : 'a M → 'b M → ('a#'#b)M) = parT\_ev

(parT\_unit : unit M → unit M → unit M) = parT\_unit\_ev

(read\_info : iid → ARMinfo M) = read\_info\_ev

(read\_version : iid → num M) = read\_version\_ev

(write\_reg : iid → ARMreg → Aimm → unit M) = write\_reg\_ev

(read\_reg : iid → ARMreg → Aimm M) = read\_reg\_ev

(write\_psr : iid → ARMpsr → ARMstatus → unit M) = write\_psr\_ev

(read\_psr : iid → ARMpsr → ARMstatus M) = read\_psr\_ev

(write\_flags :  
iid → bool#bool#bool#bool → unit M) = write\_flags\_ev

(read\_flags :  
iid → (bool#bool#bool#bool)M) = read\_flags\_ev

(read\_mode : iid → ARMmode M) = read\_mode\_ev

(read\_instr\_set : iid → InstrSet M) = read\_instr\_set\_ev

(read\_sctlr : iid → ARMsctlr M) = read\_sctlr\_ev

(write\_mem8 : *iid* → word32 → word8 → unit M) = write\_mem8\_ev

(read\_mem8 : *iid* → word32 → word8 M) = read\_mem8\_ev

(write\_mem16 : *iid* → word32 → word16 → unit M) = write\_mem16\_ev

(read\_mem16 : *iid* → word32 → word16 M) = read\_mem16\_ev

(write\_mem32 : *iid* → word32 → word32 → unit M) = write\_mem32\_ev

(read\_mem32 : *iid* → word32 → Aimm M) = read\_mem32\_ev

(dmbT :  
  *iid* → MBReqDomain#MBReqTypes → unit M) = dmbT\_ev

(set\_exclusive\_monitorsT :  
  *iid* → (word32#num) → unit M) = set\_exclusive\_monitorsT\_ev

(exclusive\_monitors\_passT :  
  *iid* → (word32#num) → bool M) = exclusive\_monitors\_passT\_ev



**Part X**  
**arm\_decoder**

```

condition_decode(cond : word4) =
let n = w2n((3 - -1)cond) in
num2ARMcondition(if cond[0] then n + 8 else n)

```

```

arm_decode_version(ireg : word32) =
let b n = ireg[n]
and i2 n = (n + 1 >< n)ireg : word2
and i3 n = (n + 2 >< n)ireg : word3
and i4 n = (n + 3 >< n)ireg : word4
and i5 n = (n + 4 >< n)ireg : word5
and i8 n = (n + 7 >< n)ireg : word8
and i12 n = (n + 11 >< n)ireg : word12
and i16 n = (n + 15 >< n)ireg : word16
and i24 n = (23 >< 0)ireg : word24 in
let cond = i4 28 and r = i4
in
if cond = 15w then
if version < 5 then
(AL, UNPREDICTABLE)
else
case (b 27, b 26, b 25, b 24, b 23, b 22, b 21, b 20, b 7, b 6, b 5, b 4) of
(F, T, F, T, F, T, T, T, F, T, F, T) →
(AL, DATA_MEMORY_BARRIER(i4 0))
|| (F, _26, _25, _24, _23, _22, _21, _20, _7, _6, _5, _4) → (AL, UNDEFINED)
|| (T, F, F, _24, _23, _22, _21, _20, _7, _6, _5, _4) → (AL, UNDEFINED)
|| (T, F, T, b24, _23, _22, _21, _20, _7, _6, _5, _4) →
(AL, BRANCH_LINK_EXCHANGE1 b24(i24 0))
|| (T, T, F, b24, _23, _22, _21, _20, _7, _6, _5, _4) →
(AL,
CP_LOAD_STORE2 b24(b 23)(b 22)(b 21)(b 20)(r 16)(r 12)
(i4 8)(i8 0))
|| (T, T, T, F, _23, _22, _21, _20, _7, _6, _5, F) →
(AL, CP_DATA_PROCESSING2(i4 20)(r 16)(r 12)(i4 8)(i3 5)(r 0))
|| (T, T, T, F, _23, _22, _21, _20, _7, _6, _5, T) →
(AL, CP_TRANSFER2(i3 21)(b 20)(r 16)(r 12)(i4 8)(i3 5)(r 0))
|| (T, T, T, T, _23, _22, _21, _20, _7, _6, _5, _4) → (AL, UNDEFINED)
else
(condition_decode cond,
case (b 27, b 26, b 25, b 24, b 23, b 22, b 21, b 20, b 7, b 6, b 5, b 4) of
(* v — "Miscellaneous instructions" —————-v *)
(F, F, F, T, F, b22, F, F, F, F, F, F) →
STATUS_TO_REGISTER b22(r 12)
|| (F, F, F, T, F, b22, T, F, F, F, F, F) →
REGISTER_TO_STATUS b22(i4 16)(r 0)
|| (F, F, F, T, F, F, T, F, F, F, F, T) →
BRANCH_EXCHANGE(r 0)
|| (F, F, F, T, F, T, T, F, F, F, F, T) →
COUNT_LEADING_ZEROES(r 12)(r 0)
|| (F, F, F, T, F, F, T, F, F, F, T, T) →

```

```

    BRANCH_LINK_EXCHANGE2(r 0)
|| (F, F, F, T, F, _22, _21, F, F, T, F, T) →
    DSP_ADD_SUBTRACT(i2 21)(r 16)(r 12)(r 0)
|| (F, F, F, T, F, F, T, F, F, T, T, T) →
    BREAKPOINT(i12 8@@i4 0)
|| (F, F, F, T, F, _22, _21, F, T, b6, _5, F) →
    DSP_MULTIPLY(r 16)(r 12)(r 8)b6(r 0)
(* ^-----^ *)
|| (F, F, F, _24, _23, _22, _21, b20, _7, _6, _5, F) →
    DATA_PROCESSING(i4 21)b20(r 16)(r 12)
    (MODEL_SHIFT_IMMEDIATE(i5 7)(i2 5)(r 0))
|| (F, F, F, _24, _23, _22, _21, b20, F, _6, _5, T) →
    DATA_PROCESSING(i4 21)b20(r 16)(r 12)
    (MODEL_SHIFT_REGISTER(r 8)(i2 5)(r 0))
(* v — "Multiplies and extra load/store instruction" —v *)
|| (F, F, F, F, F, F, b21, b20, T, F, F, T) →
    MULTIPLY b21 b20(r 16)(r 12)(r 8)(r 0)
|| (F, F, F, F, T, b22, b21, b20, T, F, F, T) →
    MULTIPLY_LONG b22 b21 b20(r 16)(r 12)(r 8)(r 0)
|| (F, F, F, T, F, b22, F, F, T, F, F, T) →
    SWAP b22(r 16)(r 12)(r 0)
|| (F, F, F, T, T, F, F, F, T, F, F, T) →
    STORE_EXCLUSIVE(r 16)(r 12)(r 0)0w
|| (F, F, F, T, T, F, F, T, T, F, F, T) →
    LOAD_EXCLUSIVE(r 16)(r 12)0w
|| (F, F, F, b24, b23, F, b21, b20, T, F, T, T) →
    LOAD_STORE_HALFWORD b24 b23 b21 b20(r 16)(r 12)ARB
    (MODE3_REGISTER(r 0))
|| (F, F, F, b24, b23, T, b21, b20, T, F, T, T) →
    LOAD_STORE_HALFWORD b24 b23 b21 b20(r 16)(r 12)ARB
    (MODE3_IMMEDIATE(i4 8@@i4 0))
|| (F, F, F, b24, b23, F, b21, F, T, T, b5, T) →
    LOAD_STORE_TWO_WORDS b24 b23 b21(r 16)(r 12)b5
    (MODE3_REGISTER(r 0))
|| (F, F, F, b24, b23, F, b21, T, T, T, b5, T) →
    LOAD_STORE_HALFWORD b24 b23 b21 T(r 16)(r 12)b5
    (MODE3_REGISTER(r 0))
|| (F, F, F, b24, b23, T, b21, F, T, T, b5, T) →
    LOAD_STORE_TWO_WORDS b24 b23 b21(r 16)(r 12)b5
    (MODE3_IMMEDIATE(i4 8@@i4 0))
|| (F, F, F, b24, b23, T, b21, T, T, T, b5, T) →
    LOAD_STORE_HALFWORD b24 b23 b21 T(r 16)(r 12)b5
    (MODE3_IMMEDIATE(i4 8@@i4 0))
(* ^-----^ *)
|| (F, F, T, T, F, _22, F, F, _7, _6, _5, _4) →
    if version < 4 then UNPREDICTABLE else UNDEFINED
|| (F, F, T, T, F, b22, T, F, _7, _6, _5, _4) →
    IMMEDIATE_TO_STATUS b22(i4 16)(i4 8)(i8 0)
|| (F, F, T, _24, _23, _22, _21, b20, _7, _6, _5, _4) →

```

```

DATA_PROCESSING(i4 21) b20(r 16)(r 12)
(MODE1_IMMEDIATE(i4 8)(i8 0))
|| (F, T, F, b24, b23, b22, b21, b20, -7, -6, -5, -4) →
LOAD_STORE b24 b23 b22 b21 b20(r 16)(r 12)
(MODE2_IMMEDIATE(i12 0))
|| (F, T, T, b24, b23, b22, b21, b20, -7, -6, -5, F) →
LOAD_STORE b24 b23 b22 b21 b20(r 16)(r 12)
(MODE2_SHIFT_IMMEDIATE(i5 7)(i2 5)(r 0))
|| (F, T, T, -24, -23, -22, -21, -20, -7, -6, -5, T) →
UNDEFINED
|| (T, F, F, b24, b23, b22, b21, b20, -7, -6, -5, -4) →
LOAD_STORE_MULTIPLE b24 b23 b22 b21 b20(r 16)(i16 0)
|| (T, F, T, b24, -23, -22, -21, -20, -7, -6, -5, -4) →
BRANCH b24(i24 0)
|| (T, T, F, b24, b23, b22, b21, b20, -7, -6, -5, -4) →
CP_LOAD_STORE b24 b23 b22 b21 b20(r 16)(r 12)(i4 8)(i8 0)
|| (T, T, T, F, -23, -22, -21, -20, -7, -6, -5, F) →
CP_DATA_PROCESSING(i4 20)(r 16)(r 12)(i4 8)(i3 5)(r 0)
|| (T, T, T, F, -23, -22, -21, b20, -7, -6, -5, T) →
CP_TRANSFER(i3 21) b20(r 16)(r 12)(i4 8)(i3 5)(r 0)
|| (T, T, T, T, -23, -22, -21, -20, -7, -6, -5, -4) →
SUPERVISOR_CALL(i24 0)
|| _ → if version < 4 then UNPREDICTABLE else UNDEFINED)

```

## Part XI

### **arm\_program**

VERSION\_MULTICORE = 7

**type\_abbrev** Ainstruction : (ARMcondition#ARMinstruction)

**type\_abbrev** program\_Ainstruction : (address → (Ainstruction#num) option)

(arm\_decode\_program\_fun : program\_word8 → address → (Ainstruction#num) option) prog\_word8 a =  
**if** (a&&3w = 0w) **then** NONE  
**else**  
**let** w0 = prog\_word8(a + 0w) **in**  
**let** w1 = prog\_word8(a + 1w) **in**  
**let** w2 = prog\_word8(a + 2w) **in**  
**let** w3 = prog\_word8(a + 3w) **in**  
**if mem** NONE [w0; w1; w2; w3] **then** NONE **else**  
**let** (raw\_instruction : word32) = (**the** w0)@@(((**the** w1)@@(((**the** w2)@@(**the** w3)) : word16)) : word24) **in**  
**let** i = arm\_decode VERSION\_MULTICORE raw\_instruction **in**  
 SOME (i, 4)

(arm\_decode\_program\_rel : program\_word8 → program\_Ainstruction → bool)  
 prog\_word8 prog\_Xinst =  
 ∀a. **case** prog\_Xinst a **of**  
 NONE → **T**  
 | SOME (inst, n) → arm\_decode\_program\_fun prog\_word8 a = SOME (inst, n)

arm\_event\_execute = arm\_event\_opsem \$arm\_execute

arm\_execute\_with\_pc\_check ii inst pc =  
**let** s = (arm\_event\_execute ii inst){} **in**  
 {E  
 | ∃eid\_next x.  
 (eid\_next, x, E) ∈ s ∧  
 ∀v ev. ((ev.action = (ACCESS R(LOCATION\_REG ii.proc(REG32 R15))v)) ∧  
 ev ∈ E.events) ⇒ (v = pc)  
 }

(arm\_event\_structures\_of\_run\_skeleton : program\_Ainstruction → run\_skeleton → (arm\_reg event\_structure)set)  
 prog\_Ainstruction rs =  
**let** Ess = {arm\_execute\_with\_pc\_check(⟦ proc := p; poi := i ⟧ inst pc | ∃n.  
 (rs p i = SOME pc) ∧ (SOME (inst, n) = prog\_Ainstruction pc)}  
**in**  
 {event\_structure\_bigunion Es | Es ∈ all\_choices Ess}

(arm\_semantics : program\_word8 → (arm\_reg state\_constraint) →  
(run\_skeleton#program\_Ainstruction#(((arm\_reg event\_structure)#((arm\_reg execution\_witness)set))set))set)  
prog\_word8 initial\_state =

**let**  $x1 = \{(rs, prog\_Xinst) \mid rs, prog\_Xinst \mid \text{run\_skeleton\_wf}(DOMAIN\ prog\_Xinst\ rs \wedge$   
arm\_decode\_program\_rel prog\_word8 prog\_Xinst) **in**

**let**  $x2 = \{(rs, prog\_Xinst, Es) \mid (rs, prog\_Xinst) \in x1 \wedge$   
( $Es = \text{arm\_event\_structures\_of\_run\_skeleton prog\_Xinst rs}$ ) **in**

**let**  $x3 = \{(rs, prog\_Xinst, \{(E, Xs) \mid E \in Es \wedge$   
( $Xs = \{X \mid \text{valid\_execution } E\ X \wedge$   
( $X.initial\_state = initial\_state\}$ ))})

$\mid (rs, prog\_Xinst, Es) \in x2\}$  **in**  
 $x3$

## Part XII

# ppc\_coretypes



**type\_abbrev** ireg : word5

**type\_abbrev** freg : word5

**type\_abbrev** ppc\_constant : word16

**type\_abbrev** crbit : word2

**ppc\_bit** = PPC\_CARRY(\* carry bit of the status register \*)  
| PPC\_CR0 **of** word2(\* bit i of the condition register \*)

**ppc\_reg32** = PPC\_IR **of** word5(\* integer registers \*)  
| PPC\_LR(\* link register (return address) \*)  
| PPC\_CTR(\* count register, used for some branches \*)  
| PPC\_PC(\* program counter \*)

Part XIII  
ppc\_types

*ppc\_reg*

61

`ppc_reg = REG32 of ppc_reg32 | REGBIT of ppc_bit`

## Part XIV

### ppc\_ast

Pinstruction =

**PADD of ireg ireg ireg**(\* integer addition \*)  
 | **PADDI of ireg ireg ppc\_constant**(\* add immediate \*)  
 | **PADDIS of ireg ireg ppc\_constant**(\* add immediate high \*)  
 | **PADDZE of ireg ireg**(\* add Carry bit \*)  
 | **PAND\_ of ireg ireg ireg**(\* bitwise and \*)  
 | **PANDC of ireg ireg ireg**(\* bitwise and-complement \*)  
 | **PANDL of ireg ireg ppc\_constant**(\* and immediate and set conditions \*)  
 | **PANDIS\_ of ireg ireg ppc\_constant**(\* and immediate high and set conditions \*)  
 | **PB of word24**(\* unconditional branch \*)  
 | **PBCTR**(\* branch to contents of register CTR \*)  
 | **PBCTRL**(\* branch to contents of CTR and link \*)  
 | **PBF of crbit 14 word**(\* branch if false \*)  
 | **PBL of word24**(\* branch and link \*)  
 | **PBS of word24**(\* branch to symbol \*)  
 | **PBLR**(\* branch to contents of register LR \*)  
 | **PBT of crbit 14 word**(\* branch if true \*)  
 | **PCMPLW of ireg ireg**(\* unsigned integer comparison \*)  
 | **PCMPLWI of ireg ppc\_constant**(\* same, with immediate argument \*)  
 | **PCMPW of ireg ireg**(\* signed integer comparison \*)  
 | **PCMPWI of ireg ppc\_constant**(\* same, with immediate argument \*)  
 | **PCROW of crbit crbit crbit**(\* or between condition bits \*)  
 | **PDIVW of ireg ireg ireg**(\* signed division \*)  
 | **PDIVWU of ireg ireg ireg**(\* unsigned division \*)  
 | **PEQV of ireg ireg ireg**(\* bitwise not-xor \*)  
 | **PEXTSB of ireg ireg**(\* 8-bit sign extension \*)  
 | **PEXTSH of ireg ireg**(\* 16-bit sign extension \*)  
 | **PFABS of freg freg**(\* float absolute value \*)  
 | **PFADD of freg freg freg**(\* float addition \*)  
 | **PFCMPI of freg freg**(\* float comparison \*)  
 | **PFCTI of ireg freg**(\* float-to-int conversion \*)  
 | **PFDIV of freg freg freg**(\* float division \*)  
 | **PFMADD of freg freg freg freg**(\* float multiply-add \*)  
 | **PFMR of freg freg**(\* float move \*)  
 | **PFMSUB of freg freg freg freg**(\* float multiply-sub \*)  
 | **PFMUL of freg freg freg**(\* float multiply \*)  
 | **PFNEG of freg freg**(\* float negation \*)  
 | **PFRRSP of freg freg**(\* float round to single precision \*)  
 | **PFSUB of freg freg freg**(\* float subtraction \*)  
 | **PICTF of freg ireg**(\* int-to-float conversion \*)  
 | **PIUCTF of freg ireg**(\* unsigned int-to-float conversion \*)  
 | **PLBZ of ireg ppc\_constant ireg**(\* load 8-bit unsigned word32 \*)  
 | **PLBZX of ireg ireg ireg**(\* same, with 2 index regs \*)  
 | **PLFD of freg ppc\_constant ireg**(\* load 64-bit float \*)  
 | **PLFDX of freg ireg ireg**(\* same, with 2 index regs \*)  
 | **PLFS of freg ppc\_constant ireg**(\* load 32-bit float \*)  
 | **PLFSX of freg ireg ireg**(\* same, with 2 index regs \*)  
 | **PLHA of ireg ppc\_constant ireg**(\* load 16-bit signed word32 \*)  
 | **PLHAX of ireg ireg ireg**(\* same, with 2 index regs \*)

| PLHZ of ireg ppc\_constant ireg(\* load 16-bit unsigned word32 \*)  
 | PLHZX of ireg ireg ireg(\* same, with 2 index regs \*)  
 | PLWARX of ireg ireg ireg(\* load 32-bit word and reserve index \*)  
 | PLWZ of ireg ppc\_constant ireg(\* load 32-bit word32 \*)  
 | PLWZX of ireg ireg ireg(\* same, with 2 index regs \*)  
 | PMFCRBIT of ireg crbit(\* move condition bit to reg \*)  
 | PMFLR of ireg(\* move LR to reg \*)  
 | PMR of ireg ireg(\* integer move \*)  
 | PMTCTR of ireg(\* move ireg to CTR \*)  
 | PMTLR of ireg(\* move ireg to LR \*)  
 | PMULLI of ireg ireg ppc\_constant(\* integer multiply immediate \*)  
 | PMULLW of ireg ireg ireg(\* integer multiply \*)  
 | PNAND of ireg ireg ireg(\* bitwise not-and \*)  
 | PNOR of ireg ireg ireg(\* bitwise not-or \*)  
 | POR of ireg ireg ireg(\* bitwise or \*)  
 | PORC of ireg ireg ireg(\* bitwise or-complement \*)  
 | PORI of ireg ireg ppc\_constant(\* or with immediate \*)  
 | PORIS of ireg ireg ppc\_constant(\* or with immediate high \*)  
 | PRLWINM of ireg ireg word5 word5 word5(\* rotate and mask \*)  
 | PSLW of ireg ireg ireg(\* shift left \*)  
 | PSRAW of ireg ireg ireg(\* shift right signed \*)  
 | PSRAWI of ireg ireg word5(\* shift right signed immediate \*)  
 | PSRW of ireg ireg ireg(\* shift right unsigned \*)  
 | PSTB of ireg ppc\_constant ireg(\* store 8-bit word \*)  
 | PSTBX of ireg ireg ireg(\* same, with 2 index regs \*)  
 | PSTFD of freg ppc\_constant ireg(\* store 64-bit float \*)  
 | PSTFDX of freg ireg ireg(\* same, with 2 index regs \*)  
 | PSTFIS of freg ppc\_constant ireg(\* store 32-bit float \*)  
 | PSTFSX of freg ireg ireg(\* same, with 2 index regs \*)  
 | PSTH of ireg ppc\_constant ireg(\* store 16-bit word \*)  
 | PSTHX of ireg ireg ireg(\* same, with 2 index regs \*)  
 | PSTW of ireg ppc\_constant ireg(\* store 32-bit word \*)  
 | PSTWCX of ireg ireg ireg(\* store word conditional indexed \*)  
 | PSTWX of ireg ireg ireg(\* store 32-bit word, with 2 index regs \*)  
 | PSUBFC of ireg ireg ireg(\* reversed integer subtraction \*)  
 | PSUBFIC of ireg ireg ppc\_constant(\* integer subtraction from immediate \*)  
 | PSYNC(\* synchronize \*)  
 | PXOR of ireg ireg ireg(\* bitwise xor \*)  
 | PXORI of ireg ireg ppc\_constant(\* bitwise xor with immediate \*)  
 | PXORIS of ireg ireg ppc\_constant(\* bitwise xor with immediate high \*)

## Part XV

### ppc\_opsem

```
ppc_sint_cmp ii(a : word32)(b : word32) =
  (parT_unit(write_status ii(PPC_CR0 0w)(SOME (a < b)))
   (parT_unit(write_status ii(PPC_CR0 1w)(SOME (b < a)))
    (parT_unit(write_status ii(PPC_CR0 2w)(SOME (a = b)))
     (write_status ii(PPC_CR0 3w)NONE))))
```

```
ppc_uint_cmp ii(a : word32)(b : word32) =
  (parT_unit(write_status ii(PPC_CR0 0w)(SOME (a < +b)))
   (parT_unit(write_status ii(PPC_CR0 1w)(SOME (b < +a)))
    (parT_unit(write_status ii(PPC_CR0 2w)(SOME (a = b)))
     (write_status ii(PPC_CR0 3w)NONE))))
```

```
OK_nextinstr ii f =
  parT_unit f(seqT(read_reg ii PPC_PC)(λx. write_reg ii PPC_PC(x + 4w)))
```

```
reg_update ii rd f s1 s2 =
  seqT(parT s1 s2)(λ(x, y). write_reg ii(PPC_IR rd)(f x y))
```

```
uint_reg_update ii rd f s1 s2 =
  seqT(parT s1 s2)
    (λ(x, y). parT_unit(write_reg ii(PPC_IR rd)(f x y))(ppc_uint_cmp ii(f x y)0w))
```

```
sint_reg_update ii rd f s1 s2 =
  seqT(parT s1 s2)
    (λ(x, y). parT_unit(write_reg ii(PPC_IR rd)(f x y))(ppc_sint_cmp ii(f x y)0w))
```

```
uint_compare ii s1 s2 =
  control_seqT(parT s1 s2)(λ(x, y). ppc_uint_cmp ii x y)
```

```
sint_compare ii s1 s2 =
  control_seqT(parT s1 s2)(λ(x, y). ppc_sint_cmp ii x y)
```

```
bit_update ii bd(f : bool → bool → bool)s1 s2 =
  seqT(parT s1 s2)(λ(x, y). write_status ii bd(SOME (f x y)))
```

```
const_low w = constT((w2w : word16 → word32)w)
```

```
const_high w = constT((w2w : word16 → word32)w << 16)
```

```
conditional x y z = if x then y else z
```

```
read_bit_word ii bit =
  seqT(read_status ii bit)(λx. constT(conditional x 1w 0w))
```



```
read_ireg ii rd = read_reg ii(PPC_IR rd)
```

```
gpr_or_zero ii d = if d = 0w then const_low 0w else read_ireg ii d
```

```
no_carry ii = write_status ii PPC_CARRY NONE
```

```
goto_label ii l =
seqT(read_reg ii PPC_PC)(λx. write_reg ii PPC_PC(x + sw2sw l * 4w))
```

```
effective_address s1 s2 = seqT(parT s1 s2)(λ(x : word32, y : word32). constT(x + y))
```

```
assertT b f = seqT(if b then constT() else failureT)(λx.f)
```

```
(write_mem_aux ii addr[] = constT()) ∧
(write_mem_aux ii addr (b ∈ bytes) =
parT_unit(write_mem8 ii addr b)
      (write_mem_aux ii(addr + 1w)bytes))
```

```
store_word ii size addr value =
assertT((address_aligned size addr) ∧ (size = 4))
      (write_mem32 ii addr value)
```

```
register_store ii size rd s1 s2 =
seqT(parT(effective_address s1 s2)(read_ireg ii rd))
      (λ(addr, x). store_word ii size addr x)
```

```
read_mem_aux ii size addr =
if size = 1 then
seqT(read_mem8 ii addr)
      (λx. constT((bytes2word[x] : word32))
else if size = 2 then
seqT(parT(read_mem8 ii addr)(read_mem8 ii(addr + 1w)))
      (λ(x0, x1). constT(bytes2word[x1; x0]))
else
seqT(parT(parT(read_mem8 ii(addr + 0w))(read_mem8 ii(addr + 1w)))
      (parT(read_mem8 ii(addr + 2w))(read_mem8 ii(addr + 3w))))
      (λ((x0, x1), (x2, x3)). constT(bytes2word[x3; x2; x1; x0]))
```

```
load_word ii size addr =
assertT((address_aligned size addr) ∧ (size = 2))
      (read_mem32 ii addr)
```

```

register_load ii size rd s1 s2 =
  seqT(effective_address s1 s2)
    (λaddr. seqT(load_word ii size addr)
      (write_reg ii(PPC_IR rd)))

set_CR0_to_00xNONE ii b =
  (parT_unit(write_status ii(PPC_CR0 0w)(SOME F))
  (parT_unit(write_status ii(PPC_CR0 1w)(SOME F))
  (parT_unit(write_status ii(PPC_CR0 2w)(SOME b)
    (write_status ii(PPC_CR0 3w)NONE))))

(ppc_exec_instr ii(PADD rd R1 R2) =
  OK_nextinstr ii(reg_update ii rd$ + (read_ireg ii R1)(read_ireg ii R2))) ∧

(ppc_exec_instr ii(PADDI rd R1 cst) =
  OK_nextinstr ii(reg_update ii rd$ + (gpr_or_zero ii R1)(const_low cst))) ∧

(ppc_exec_instr ii(PADDIS rd R1 cst) =
  OK_nextinstr ii(reg_update ii rd$ + (gpr_or_zero ii R1)(const_high cst))) ∧

(ppc_exec_instr ii(PADDZE rd R1) =
  OK_nextinstr ii(reg_update ii rd$ + (read_ireg ii R1)(read_bit_word ii PPC_CARRY))) ∧

(ppc_exec_instr ii(PAND_ rd R1 R2) =
  OK_nextinstr ii(sint_reg_update ii rd$&&(read_ireg ii R1)(read_ireg ii R2))) ∧

(ppc_exec_instr ii(PANDC rd R1 R2) =
  OK_nextinstr ii(reg_update ii rd(λx y.x&&&¬y)(read_ireg ii R1)(read_ireg ii R2))) ∧

(ppc_exec_instr ii(PANDL_ rd R1 cst) =
  OK_nextinstr ii(sint_reg_update ii rd$&&(read_ireg ii R1)(const_low cst))) ∧

(ppc_exec_instr ii(PANDIS_ rd R1 cst) =
  OK_nextinstr ii(sint_reg_update ii rd$&&(read_ireg ii R1)(const_high cst))) ∧

(ppc_exec_instr ii(PB lbl) =
  goto_label ii lbl) ∧

(ppc_exec_instr ii(PBCTR) =
  seqT(read_reg ii PPC_CTR)(write_reg ii PPC_PC)) ∧

(ppc_exec_instr ii(PBCTRL) =
  seqT(parT(read_reg ii PPC_PC)(read_reg ii PPC_CTR))
    (λ(pc, ctr). parT_unit(write_reg ii PPC_PC ctr)(write_reg ii PPC_LR(pc + 4w)))) ∧

(ppc_exec_instr ii(PBF bit lb1) =
  seqT(read_status ii(PPC_CR0 bit))
    (λb.if b then goto_label ii lb1 else OK_nextinstr ii(constT())))) ∧

```

$$\begin{aligned}
&(\text{ppc\_exec\_instr } ii(\text{PBL } ident) = \\
&\quad \text{seqT}(\text{read\_reg } ii \text{ PPC\_PC}) \\
&\quad (\lambda x. \text{parT\_unit}(\text{write\_reg } ii \text{ PPC\_PC}(x + \text{sw2sw } ident * 4w))(\text{write\_reg } ii \text{ PPC\_LR}(x + 4w)))) \wedge \\
&(\text{ppc\_exec\_instr } ii(\text{PBS } ident) = \\
&\quad \text{goto\_label } ii \text{ ident}) \wedge \\
&(\text{ppc\_exec\_instr } ii(\text{PBLR}) = \\
&\quad \text{seqT}(\text{read\_reg } ii \text{ PPC\_LR})(\text{write\_reg } ii \text{ PPC\_PC})) \wedge \\
&(\text{ppc\_exec\_instr } ii(\text{PBT } bit \text{ lb1}) = \\
&\quad \text{control\_seqT}(\text{read\_status } ii(\text{PPC\_CR0 } bit)) \\
&\quad (\lambda b. \text{if } \neg b \text{ then goto\_label } ii \text{ lb1 else OK\_nextinstr } ii(\text{constT()}))) \wedge \\
&(\text{ppc\_exec\_instr } ii(\text{PCMPLW } R1 \text{ R2}) = \\
&\quad \text{OK\_nextinstr } ii(\text{uint\_compare } ii(\text{read\_ireg } ii \text{ R1})(\text{read\_ireg } ii \text{ R2}))) \wedge \\
&(\text{ppc\_exec\_instr } ii(\text{PCMPLWI } R1 \text{ cst}) = \\
&\quad \text{OK\_nextinstr } ii(\text{uint\_compare } ii(\text{read\_ireg } ii \text{ R1})(\text{const\_low } cst))) \wedge \\
&(\text{ppc\_exec\_instr } ii(\text{PCMPW } R1 \text{ R2}) = \\
&\quad \text{OK\_nextinstr } ii(\text{sint\_compare } ii(\text{read\_ireg } ii \text{ R1})(\text{read\_ireg } ii \text{ R2}))) \wedge \\
&(\text{ppc\_exec\_instr } ii(\text{PCMPWI } R1 \text{ cst}) = \\
&\quad \text{OK\_nextinstr } ii(\text{sint\_compare } ii(\text{read\_ireg } ii \text{ R1})(\text{const\_low } cst))) \wedge \\
&(\text{ppc\_exec\_instr } ii(\text{PCROR } bd \text{ b}_1 \text{ b}_2) = \\
&\quad \text{OK\_nextinstr } ii(\text{bit\_update } ii(\text{PPC\_CR0 } bd)((\text{read\_status } ii(\text{PPC\_CR0 } b_1)) \vee (\text{read\_status } ii(\text{PPC\_CR0 } b_2))))) \\
&(\text{ppc\_exec\_instr } ii(\text{PDIVW } rd \text{ R1 } R2) = \text{failureT}) \wedge \\
&(\text{ppc\_exec\_instr } ii(\text{PDIVWU } rd \text{ R1 } R2) = \text{failureT}) \wedge \\
&(\text{ppc\_exec\_instr } ii(\text{PEQV } rd \text{ R1 } R2) = \\
&\quad \text{OK\_nextinstr } ii(\text{reg\_update } ii \text{ rd}(\lambda x \text{ y. } \neg(x??y))(\text{read\_ireg } ii \text{ R1})(\text{read\_ireg } ii \text{ R2}))) \wedge \\
&(\text{ppc\_exec\_instr } ii(\text{PEXTSB } rd \text{ R1}) = \\
&\quad \text{OK\_nextinstr } ii(\text{reg\_update } ii \text{ rd}(\lambda x \text{ y. sw2sw}((w2w \text{ x}) : \text{word8})) \\
&\quad (\text{read\_ireg } ii \text{ R1})(\text{constT()}))) \wedge \\
&(\text{ppc\_exec\_instr } ii(\text{PEXTSH } rd \text{ R1}) = \\
&\quad \text{OK\_nextinstr } ii(\text{reg\_update } ii \text{ rd}(\lambda x \text{ y. sw2sw}((w2w \text{ x}) : \text{word16})) \\
&\quad (\text{read\_ireg } ii \text{ R1})(\text{constT()}))) \wedge \\
&(\text{ppc\_exec\_instr } ii(\text{PFABS } rd \text{ R1}) = \text{failureT}) \wedge \\
&(\text{ppc\_exec\_instr } ii(\text{PFADD } rd \text{ R1 } R2) = \text{failureT}) \wedge
\end{aligned}$$

$(\text{ppc\_exec\_instr } ii(\text{PFCMPU } R1 \ R2) = \text{failureT}) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PFCTI } rd \ R1) = \text{failureT}) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PFDIV } rd \ R1 \ R2) = \text{failureT}) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PFMADD } rd \ R1 \ R2 \ R3) = \text{failureT}) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PFMR } rd \ R1) = \text{failureT}) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PFMSUB } rd \ R1 \ R2 \ R3) = \text{failureT}) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PFMUL } rd \ R1 \ R2) = \text{failureT}) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PFNEG } rd \ R1) = \text{failureT}) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PFRSP } rd \ R1) = \text{failureT}) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PFSUB } rd \ R1 \ R2) = \text{failureT}) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PICTF } rd \ R1) = \text{failureT}) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PIUCTF } rd \ R1) = \text{failureT}) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PLBZ } rd \ cst \ R1) =$   
 $\text{OK\_nextinstr } ii(\text{register\_load } ii \ 1 \ rd(\text{read\_ireg } ii \ R1)(\text{const\_low } cst))) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PLBZX } rd \ R1 \ R2) =$   
 $\text{OK\_nextinstr } ii(\text{register\_load } ii \ 1 \ rd(\text{read\_ireg } ii \ R1)(\text{read\_ireg } ii \ R2))) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PLFD } rd \ cst \ R1) = \text{failureT}) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PLFDX } rd \ R1 \ R2) = \text{failureT}) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PLFS } rd \ cst \ R1) = \text{failureT}) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PLFSX } rd \ R1 \ R2) = \text{failureT}) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PLHA } rd \ cst \ R1) =$   
 $\text{OK\_nextinstr } ii(\text{register\_load } ii \ 2 \ rd(\text{read\_ireg } ii \ R1)(\text{const\_low } cst))) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PLHAX } rd \ R1 \ R2) =$   
 $\text{OK\_nextinstr } ii(\text{register\_load } ii \ 2 \ rd(\text{read\_ireg } ii \ R1)(\text{read\_ireg } ii \ R2))) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PLHZ } rd \ cst \ R1) =$   
 $\text{OK\_nextinstr } ii(\text{register\_load } ii \ 2 \ rd(\text{read\_ireg } ii \ R1)(\text{const\_low } cst))) \wedge$   
 $(\text{ppc\_exec\_instr } ii(\text{PLHZX } rd \ R1 \ R2) =$   
 $\text{OK\_nextinstr } ii(\text{register\_load } ii \ 2 \ rd(\text{read\_ireg } ii \ R1)(\text{read\_ireg } ii \ R2))) \wedge$

$(\text{ppc\_exec\_instr } ii(\text{PLWARX } rd \ ra \ rb) =$   
 $\text{OK\_nextinstr } ii($   
 $\quad \text{seqT(effective\_address(gpr\_or\_zero } ii \ ra)(\text{read\_ireg } ii \ rb))$   
 $\quad (\lambda ea. \text{lockT(parT\_unit(write\_reserve\_bit } ii \ \mathbf{T})($   
 $\quad \quad \text{parT\_unit(write\_reserve\_address } ii \ ea)$   
 $\quad \quad (\text{seqT(load\_word } ii \ 4 \ ea)(\text{write\_reg } ii(\text{PPC\_IR } rd)))))) \wedge$

$(\text{ppc\_exec\_instr } ii(\text{PLWZ } rd \ cst \ R1) =$   
 $\text{OK\_nextinstr } ii(\text{register\_load } ii \ 4 \ rd(\text{read\_ireg } ii \ R1)(\text{const\_low } cst))) \wedge$

$(\text{ppc\_exec\_instr } ii(\text{PLWZX } rd \ R1 \ R2) =$   
 $\text{OK\_nextinstr } ii(\text{register\_load } ii \ 4 \ rd(\text{read\_ireg } ii \ R1)(\text{read\_ireg } ii \ R2))) \wedge$

$(\text{ppc\_exec\_instr } ii(\text{PMFCRBIT } v162 \ v163) = \text{failureT}) \wedge$

$(\text{ppc\_exec\_instr } ii(\text{PMFLR } rd) =$   
 $\text{OK\_nextinstr } ii(\text{seqT(read\_reg } ii \ \text{PPC\_LR})(\text{write\_reg } ii(\text{PPC\_IR } rd)))) \wedge$

$(\text{ppc\_exec\_instr } ii(\text{PMR } rd \ R1) =$   
 $\text{OK\_nextinstr } ii(\text{seqT(read\_ireg } ii \ R1)(\text{write\_reg } ii(\text{PPC\_IR } rd)))) \wedge$

$(\text{ppc\_exec\_instr } ii(\text{PMTCTR } R1) =$   
 $\text{OK\_nextinstr } ii(\text{seqT(read\_ireg } ii \ R1)(\text{write\_reg } ii \ \text{PPC\_CTR}))) \wedge$

$(\text{ppc\_exec\_instr } ii(\text{PMTLR } R1) =$   
 $\text{OK\_nextinstr } ii(\text{seqT(read\_ireg } ii \ R1)(\text{write\_reg } ii \ \text{PPC\_LR}))) \wedge$

$(\text{ppc\_exec\_instr } ii(\text{PMULLI } rd \ R1 \ cst) =$   
 $\text{OK\_nextinstr } ii(\text{reg\_update } ii \ rd \$ * (\text{read\_ireg } ii \ R1)(\text{const\_low } cst))) \wedge$

$(\text{ppc\_exec\_instr } ii(\text{PMULLW } rd \ R1 \ R2) =$   
 $\text{OK\_nextinstr } ii(\text{reg\_update } ii \ rd \$ * (\text{read\_ireg } ii \ R1)(\text{read\_ireg } ii \ R2))) \wedge$

$(\text{ppc\_exec\_instr } ii(\text{PNAND } rd \ R1 \ R2) =$   
 $\text{OK\_nextinstr } ii(\text{reg\_update } ii \ rd (\lambda x \ y. \neg(x \& \& y))(\text{read\_ireg } ii \ R1)(\text{read\_ireg } ii \ R2))) \wedge$

$(\text{ppc\_exec\_instr } ii(\text{PNOR } rd \ R1 \ R2) =$   
 $\text{OK\_nextinstr } ii(\text{reg\_update } ii \ rd (\lambda x \ y. \neg(x!!y))(\text{read\_ireg } ii \ R1)(\text{read\_ireg } ii \ R2))) \wedge$

$(\text{ppc\_exec\_instr } ii(\text{POR } rd \ R1 \ R2) =$   
 $\text{OK\_nextinstr } ii(\text{reg\_update } ii \ rd \$!!(\text{read\_ireg } ii \ R1)(\text{read\_ireg } ii \ R2))) \wedge$

$(\text{ppc\_exec\_instr } ii(\text{PORC } rd \ R1 \ R2) =$   
 $\text{OK\_nextinstr } ii(\text{reg\_update } ii \ rd (\lambda x \ y. x!!\neg y)(\text{read\_ireg } ii \ R1)(\text{read\_ireg } ii \ R2))) \wedge$

$(\text{ppc\_exec\_instr } ii(\text{PORI } rd \ R1 \ cst) =$   
 $\text{OK\_nextinstr } ii(\text{reg\_update } ii \ rd \$!!(\text{read\_ireg } ii \ R1)(\text{const\_low } cst))) \wedge$

```

(ppc_exec_instr ii(PORIS rd R1 cst) =
OK_nextinstr ii(reg_update ii rd$(read_ireg ii R1)(const_high cst))) ∧

(ppc_exec_instr ii(PRLWINM rd R1 sh mb me) = failureT) ∧

(ppc_exec_instr ii(PSLW rd R1 R2) =
OK_nextinstr ii(reg_update ii rd(λx y.x ≪ w2n((w2w y) : word6))(read_ireg ii R1)(read_ireg ii R2))) ∧

(ppc_exec_instr ii(PSRAW rd R1 R2) =
OK_nextinstr ii(parT_unit(reg_update ii rd(λx y.x >>> w2n((w2w y) : word6))(read_ireg ii R1)(read_ireg ii R2))
(no_carry ii))) ∧

(ppc_exec_instr ii(PSRAWI rd R1 sh) =
OK_nextinstr ii(parT_unit(reg_update ii rd(λx : word32 y : word32.x >>> w2n((w2w y) : word6))(read_ireg ii R1)
(no_carry ii))) ∧

(ppc_exec_instr ii(PSRW rd R1 R2) =
OK_nextinstr ii(reg_update ii rd(λx y.x ≫ w2n((w2w y) : word6))(read_ireg ii R1)(read_ireg ii R2))) ∧

(ppc_exec_instr ii(PSTB rd cst R1) =
OK_nextinstr ii(register_store ii 1 rd(read_ireg ii R1)(const_low cst))) ∧

(ppc_exec_instr ii(PSTBX rd R1 R2) =
OK_nextinstr ii(register_store ii 1 rd(read_ireg ii R1)(read_ireg ii R2))) ∧

(ppc_exec_instr ii(PSTFD rd cst R1) = failureT) ∧

(ppc_exec_instr ii(PSTFDX rd R1 R2) = failureT) ∧

(ppc_exec_instr ii(PSTFS rd cst R1) = failureT) ∧

(ppc_exec_instr ii(PSTFSX rd R1 R2) = failureT) ∧

(ppc_exec_instr ii(PSTH rd cst R1) =
OK_nextinstr ii(register_store ii 2 rd(read_ireg ii R1)(const_low cst))) ∧

(ppc_exec_instr ii(PSTHX rd R1 R2) =
OK_nextinstr ii(register_store ii 2 rd(read_ireg ii R1)(read_ireg ii R2))) ∧

(ppc_exec_instr ii(PSTWCX rs ra rb) =
OK_nextinstr ii(
seqT(parT(effective_address(gpr_or_zero ii ra)(read_ireg ii rb))
(read_reserve_bit ii))
(λ(ea, reserve).
if reserve then
lockT(parT_unit(seqT(read_ireg ii rs)(λx.store_word ii 4 ea x))
(parT_unit(set_CR0_to_00xNONE ii T)
(write_reserve_bit ii F)))
else

```

$$(\text{set\_CR0\_to\_00xNONE } ii \mathbf{T})) \wedge$$

$$\begin{aligned} &(\text{ppc\_exec\_instr } ii(\text{PSTW } rd \text{ } cst \text{ } R1) = \\ &\text{OK\_nextinstr } ii(\text{register\_store } ii \text{ } 4 \text{ } rd(\text{read\_ireg } ii \text{ } R1)(\text{const\_low } cst))) \wedge \end{aligned}$$

$$\begin{aligned} &(\text{ppc\_exec\_instr } ii(\text{PSTWX } rd \text{ } R1 \text{ } R2) = \\ &\text{OK\_nextinstr } ii(\text{register\_store } ii \text{ } 4 \text{ } rd(\text{read\_ireg } ii \text{ } R1)(\text{read\_ireg } ii \text{ } R2))) \wedge \end{aligned}$$

$$\begin{aligned} &(\text{ppc\_exec\_instr } ii(\text{PSUBFC } rd \text{ } R1 \text{ } R2) = \\ &\text{OK\_nextinstr } ii(\text{parT\_unit}(\text{reg\_update } ii \text{ } rd\$ - (\text{read\_ireg } ii \text{ } R2)(\text{read\_ireg } ii \text{ } R1)) \\ &\quad (\text{no\_carry } ii))) \wedge \end{aligned}$$

$$\begin{aligned} &(\text{ppc\_exec\_instr } ii(\text{PSUBFIC } rd \text{ } R1 \text{ } cst) = \\ &\text{OK\_nextinstr } ii(\text{parT\_unit}(\text{reg\_update } ii \text{ } rd\$ - (\text{const\_low } cst)(\text{read\_ireg } ii \text{ } R1)) \\ &\quad (\text{no\_carry } ii))) \wedge \end{aligned}$$

$$\begin{aligned} &(\text{ppc\_exec\_instr } ii(\text{PSYNC}) = \\ &\text{OK\_nextinstr } ii(\text{syncT } ii)) \wedge \end{aligned}$$

$$\begin{aligned} &(\text{ppc\_exec\_instr } ii(\text{PXOR } rd \text{ } R1 \text{ } R2) = \\ &\text{OK\_nextinstr } ii(\text{reg\_update } ii \text{ } rd\$??(\text{read\_ireg } ii \text{ } R1)(\text{read\_ireg } ii \text{ } R2))) \wedge \end{aligned}$$

$$\begin{aligned} &(\text{ppc\_exec\_instr } ii(\text{PXORI } rd \text{ } R1 \text{ } cst) = \\ &\text{OK\_nextinstr } ii(\text{reg\_update } ii \text{ } rd\$??(\text{read\_ireg } ii \text{ } R1)(\text{const\_low } cst))) \wedge \end{aligned}$$

$$\begin{aligned} &(\text{ppc\_exec\_instr } ii(\text{PXORIS } rd \text{ } R1 \text{ } cst) = \\ &\text{OK\_nextinstr } ii(\text{reg\_update } ii \text{ } rd\$??(\text{read\_ireg } ii \text{ } R1)(\text{const\_high } cst))) \end{aligned}$$

Part XVI  
ppc\_decoder



```

ppc_match_step name =
if name = "0" then DF else
if name = "1" then DT else
if mem name ["A"; "B"; "C"; "D"; "S"; "BI"; "crbA"; "crbB"; "crbD"; "SH"; "MB"; "ME"] then
  assign_drop name 5
  else if mem name ["BD"] then
    assign_drop name 14
  else if mem name ["SIMM"; "UIMM"; "d"] then
    assign_drop name 16
  else if mem name ["LI"] then
    assign_drop name 24
  else if mem name ["AA"; "Rc"; "OE"; "y"; "z"] then
    assign_drop name 1
  else
    option_fail

```

```

ppc_decode = match_list ppc_match_step(REVERSE o tokenise)(λk x.SOME (k(fst x)))^ppc_syntax

```

## Part XVII

ppc\_

```
iiid_dummy = { proc := 0; poi := 0 }
```

```
PPC_NEXT s =
let pc = PREAD_R PPC_PC s in
let w0 = PREAD_M(pc + 0w)s in
let w1 = PREAD_M(pc + 1w)s in
let w2 = PREAD_M(pc + 2w)s in
let w3 = PREAD_M(pc + 3w)s in
let raw_instruction = w2bits(the w0) ++ w2bits(the w1) ++ w2bits(the w2) ++ w2bits(the w3) in
let i = ppc_decode raw_instruction in
let s' = ppc_exec_instr iiid_dummy(the i)s in
if ¬(pc && 3w = 0w) ∨ mem NONE [w0; w1; w2; w3] ∨ (i = NONE) ∨ (s' = NONE) then NONE
else SOME (snd(the s'))
```

## Part XVIII

### ppc\_seq\_monad

**type\_abbrev** ppc\_state : (ppc\_reg32 → word32)#(ppc\_bit → bool option)#(word32 → word8 option)#bool#word32

PREAD\_R rd((r, s, m, rb, ra) : ppc\_state) = r rd

PREAD\_S rd((r, s, m, rb, ra) : ppc\_state) = s rd

PREAD\_M rd((r, s, m, rb, ra) : ppc\_state) = m rd

PWRITE\_R rd x(r, s, m, rb, ra) = ((rd = +x)r, s, m, rb, ra) : ppc\_state

PWRITE\_S rd x(r, s, m, rb, ra) = (r, (rd = +x)s, m, rb, ra) : ppc\_state

PWRITE\_M rd x(r, s, m, rb, ra) = (r, s, (rd = +x)m, rb, ra) : ppc\_state

PREAD\_REVERSE\_BIT((r, s, m, rb, ra) : ppc\_state) = rb

PREAD\_REVERSE\_ADDRESS((r, s, m, rb, ra) : ppc\_state) = ra

PWRITE\_REVERSE\_BIT x((r, s, m, rb, ra) : ppc\_state) = ((r, s, m, x, ra) : ppc\_state)

PWRITE\_REVERSE\_ADDRESS x((r, s, m, rb, ra) : ppc\_state) = ((r, s, m, rb, x) : ppc\_state)

**type\_abbrev** M : ppc\_state → ('a#ppc\_state) option

(constT\_seq : 'a → 'a M)x = λy.SOME (x, y)

(addT\_seq : 'a → 'b M → ('a##'b)M)x s =  
λy.case s y of NONE → NONE || SOME (z, t) → SOME ((x, z), t)

(lockT\_seq : 'a M → 'a M)s = s

(syncT\_seq : iid → unit M)x = constT\_seq()

(failureT\_seq : 'a M) = λy.NONE

(seqT\_seq : 'a M → ('a → 'b M) → 'b M)s f =  
λy.case s y of NONE → NONE || SOME (z, t) → f z t

```
(parT_seq : 'a M → 'b M → ('a#/'b)M)s t =
λy.case s y of NONE → NONE || SOME (a, z) →
  case t z of NONE → NONE || SOME (b, x) → SOME ((a, b), x)
```

```
(parT_unit_seq : unit M → unit M → unit M)s t =
λy.case s y of NONE → NONE || SOME (a, z) →
  case t z of NONE → NONE || SOME (b, x) → SOME ((), x)
```

```
(write_reg_seq ii r x) : unit M =
λs.SOME ((), PWRITE_R r x s)
```

```
(read_reg_seq ii r) : word32 M =
λs.SOME (PREAD_R r s, s)
```

```
(write_status_seq ii f x) : unit M =
(λs.SOME ((), PWRITE_S f x s))
```

```
(read_status_seq ii f) : bool M =
(λs.case PREAD_S f s of NONE → NONE || SOME b → SOME (b, s))
```

```
(write_mem8_seq ii a x) : unit M =
(λs.case PREAD_M a s of NONE → NONE || SOME y → SOME ((), PWRITE_M a(SOME x)s))
```

```
(read_mem8_seq ii a) : word8 M =
(λs.case PREAD_M a s of NONE → NONE || SOME x → SOME (x, s))
```

```
(read_mem32_seq ii a) : word32 M =
seqT_seq(parT_seq(read_mem8_seq ii(a + 0w))(parT_seq(read_mem8_seq ii(a + 1w))
  (parT_seq(read_mem8_seq ii(a + 2w))(read_mem8_seq ii(a + 3w))))))
  (λ(x0, x1, x2, x3).constT_seq(bytes2word[x0; x1; x2; x3]))
```

```
(write_mem32_seq ii a w) : unit M =
(let bs = word2bytes 4 w in
  parT_unit_seq(write_mem8_seq ii(a + 0w)(EL 0 bs))(parT_unit_seq(write_mem8_seq ii(a + 1w)(EL 1 bs))
  (parT_unit_seq(write_mem8_seq ii(a + 2w)(EL 2 bs))(write_mem8_seq ii(a + 3w)(EL 3 bs))))))
```

```
(write_reserve_bit_seq(ii : iid)x) : unit M =
λs.SOME ((), PWRITE_REVERSE_BIT x s)
```

```
(read_reserve_bit_seq(ii : iid)) : bool M =
λs.SOME (PREAD_REVERSE_BIT s, s)
```

```

(write_reserve_address_seq(ii : iiid)x) : unit M =
λs.SOME ((), PWRITE_REVERSE_ADDRESS x s)

(read_reserve_address_seq(ii : iiid)) : word32 M =
λs.SOME (PREAD_REVERSE_ADDRESS s, s)

(constT : 'a → 'a M) = constT_seq

(addT : 'a → 'b M → ('a#'#'b)M) = addT_seq

(lockT : unit M → unit M) = lockT_seq

(syncT : iiid → unit M) = syncT_seq

(failureT : unit M) = failureT_seq

(control_seqT : 'a M → ('a → 'b M) → 'b M) = seqT_seq

(seqT : 'a M → ('a → 'b M) → 'b M) = seqT_seq

(parT : 'a M → 'b M → ('a#'#'b)M) = parT_seq

(parT_unit : unit M → unit M → unit M) = parT_unit_seq

(write_reg : iiid → ppc_reg32 → word32 → unit M) = write_reg_seq

(read_reg : iiid → ppc_reg32 → word32 M) = read_reg_seq

(write_status : iiid → ppc_bit → bool option → unit M) = write_status_seq

(read_status : iiid → ppc_bit → bool M) = read_status_seq

(write_mem8 : iiid → word32 → word8 → unit M) = write_mem8_seq

(read_mem8 : iiid → word32 → word8 M) = read_mem8_seq

(write_mem32 : iiid → word32 → word32 → unit M) = write_mem32_seq

(read_mem32 : iiid → word32 → word32 M) = read_mem32_seq

(write_reserve_bit : iiid → bool → unit M) = write_reserve_bit_seq

(read_reserve_bit : iiid → bool M) = read_reserve_bit_seq

(write_reserve_address : iiid → word32 → unit M) = write_reserve_address_seq

(read_reserve_address : iiid → word32 M) = read_reserve_address_seq

option_apply x f = if x = NONE then NONE else f(the x)

```

## Part XIX

# ppc\_event\_monad



**type\_abbrev** M : iid\_state → ((iid\_state#'*a*#(ppc\_reg event\_structure))set)

event\_structure\_empty = { events := {}; intra\_causality\_data := {}; intra\_causality\_control := {}; atomicity := {} }

event\_structure\_lock es = { events := es.events; intra\_causality\_data := es.intra\_causality\_data; intra\_causality\_control := es.intra\_causality\_control; atomicity := es.atomicity }

event\_structure\_union es<sub>1</sub> es<sub>2</sub> =  
 { events := es<sub>1</sub>.events ∪ es<sub>2</sub>.events;  
 intra\_causality\_data := es<sub>1</sub>.intra\_causality\_data ∪ es<sub>2</sub>.intra\_causality\_data;  
 intra\_causality\_control := es<sub>1</sub>.intra\_causality\_control ∪ es<sub>2</sub>.intra\_causality\_control;  
 atomicity := es<sub>1</sub>.atomicity ∪ es<sub>2</sub>.atomicity }

event\_structure\_bigunion(ess : (ppc\_reg event\_structure)set) =  
 { events := **bigunion**{es.events | es ∈ ess};  
 intra\_causality\_data := **bigunion**{es.intra\_causality\_data | es ∈ ess};  
 intra\_causality\_control := **bigunion**{es.intra\_causality\_control | es ∈ ess};  
 atomicity := **bigunion**{es.atomicity | es ∈ ess} }

event\_structure\_seq\_union es<sub>1</sub> es<sub>2</sub> =  
 { events := es<sub>1</sub>.events ∪ es<sub>2</sub>.events;  
 intra\_causality\_data := es<sub>1</sub>.intra\_causality\_data  
 ∪ es<sub>2</sub>.intra\_causality\_data  
 ∪ { (e<sub>1</sub>, e<sub>2</sub>)  
 | e<sub>1</sub> ∈ (maximal\_elements es<sub>1</sub>.events es<sub>1</sub>.intra\_causality\_data)  
 ∧ e<sub>2</sub> ∈ (minimal\_elements es<sub>2</sub>.events es<sub>2</sub>.intra\_causality\_data) };  
 intra\_causality\_control := es<sub>1</sub>.intra\_causality\_control  
 ∪ es<sub>2</sub>.intra\_causality\_control;  
 atomicity := es<sub>1</sub>.atomicity ∪ es<sub>2</sub>.atomicity }

event\_structure\_control\_seq\_union es<sub>1</sub> es<sub>2</sub> =  
 { events := es<sub>1</sub>.events ∪ es<sub>2</sub>.events;  
 intra\_causality\_data := es<sub>1</sub>.intra\_causality\_data  
 ∪ es<sub>2</sub>.intra\_causality\_data;  
 intra\_causality\_control := es<sub>1</sub>.intra\_causality\_control  
 ∪ es<sub>2</sub>.intra\_causality\_control  
 ∪ { (e<sub>1</sub>, e<sub>2</sub>)  
 | e<sub>1</sub> ∈ (maximal\_elements es<sub>1</sub>.events es<sub>1</sub>.intra\_causality\_control)  
 ∧ e<sub>2</sub> ∈ (minimal\_elements es<sub>2</sub>.events es<sub>2</sub>.intra\_causality\_control) };  
 atomicity := es<sub>1</sub>.atomicity ∪ es<sub>2</sub>.atomicity }

(mapT\_ev : ('a → 'b) → 'a M → 'b M) f s =  
 λ iid\_next : iid\_state.

**let** t = s iid\_next **in**  
 { (iid\_next', f x, es)  
 | (iid\_next', x, es) ∈ t }

(choiceT\_ev : 'a M → 'a M → 'a M) s s' =  
 λ eiid\_next : eiid\_state. s eiid\_next ∪ s' eiid\_next

(constT\_ev : 'a → 'a M) x = λ eiid\_next. {(eiid\_next, x, event\_structure\_empty)}

(addT\_ev : 'a → 'b M → ('a#'#b)M) x s =  
 λ eiid\_next. let (t : (eiid\_state#'#b#'#(ppc\_reg event\_structure))set) = s eiid\_next in  
 image(λ(eiid\_next', v, es).(eiid\_next', (x, v), es))t

(lockT\_ev : 'a M → 'a M) s =  
 λ eiid\_next. let (t : (eiid\_state#'#a#'#(ppc\_reg event\_structure))set) = s eiid\_next in  
 image(λ(eiid\_next', v, es).(eiid\_next', v, event\_structure\_lock es))t

(failureT\_ev : 'a M) = λ eiid\_next. {}

(condT\_ev : bool → unit M → unit M) b s =  
 if b then s else constT\_ev()

(seqT\_ev : 'a M → ('a → 'b M) → 'b M) s f =  
 λ eiid\_next : eiid\_state.

let t = s eiid\_next in  
 bigunion{let t' = f x eiid\_next' in  
 {(eiid\_next'', x', event\_structure\_seq\_union es es')  
 | (eiid\_next'', x', es') ∈ t'}  
 | (eiid\_next', x, es) ∈ t}

(control\_seqT\_ev : 'a M → ('a → 'b M) → 'b M) s f =  
 λ eiid\_next : eiid\_state.

let t = s eiid\_next in  
 bigunion{let t' = f x eiid\_next' in  
 {(eiid\_next'', x', event\_structure\_control\_seq\_union es es')  
 | (eiid\_next'', x', es') ∈ t'}  
 | (eiid\_next', x, es) ∈ t}

(parT\_ev : 'a M → 'b M → ('a#'#b)M) s s' =  
 λ eiid\_next : eiid\_state.

let t = s eiid\_next in  
 bigunion{let t' = s' eiid\_next' in  
 {(eiid\_next'', (x, x'), event\_structure\_union es es')  
 | (eiid\_next'', x', es') ∈ t'}  
 | (eiid\_next', x, es) ∈ t}

(parT\_unit\_ev : unit M → unit M → unit M) s s' =  
 λeiiid\_next : eiiid\_state.

**let** t = s eiiid\_next **in**  
**bigunion**{**let** t' = s' eiiid\_next' **in**  
 {(eiiid\_next'', (), event\_structure\_union es es')  
 | (eiiid\_next'', (), es') ∈ t'}  
 | (eiiid\_next', (), es) ∈ t}

(write\_location\_ev ii l x) : unit M =  
 λeiiid\_next.{(eiiid\_next',  
 (),  
 { events := { { eiid := eiiid';  
 iiii := ii;  
 action := ACCESS W l x } };  
 intra\_causality\_data := {};  
 intra\_causality\_control := {};  
 atomicity := {} } ) | (eiiid', eiiid\_next') ∈ next\_eiid eiiid\_next}

(read\_location\_ev ii l) : value M =  
 λeiiid\_next.{(eiiid\_next',  
 x,  
 { events := { { eiid := eiiid';  
 iiii := ii;  
 action := ACCESS R l x } };  
 intra\_causality\_data := {};  
 intra\_causality\_control := {};  
 atomicity := {} } )  
 | x ∈ UNIV ∧ (eiiid', eiiid\_next') ∈ next\_eiid eiiid\_next}

(syncT\_ev ii) : unit M =  
 λeiiid\_next.{(eiiid\_next',  
 (),  
 { events := { { eiid := eiiid';  
 iiii := ii;  
 action := BARRIER SYNC } };  
 intra\_causality\_data := {};  
 intra\_causality\_control := {};  
 atomicity := {} } ) | (eiiid', eiiid\_next') ∈ next\_eiid eiiid\_next}

(write\_reg\_ev ii r x) : unit M =  
 write\_location\_ev ii (LOCATION\_REG ii. proc(REG32 r)) x

(read\_reg\_ev ii r) : value M =  
 read\_location\_ev ii (LOCATION\_REG ii. proc(REG32 r))

```

(write_status_ev ii f x) : unit M =
case x of
SOME b → write_location_ev ii(LOCATION_REG ii.proc(REGBIT f))(if b then 1w else 0w)
|| NONE → choiceT_ev(write_location_ev ii(LOCATION_REG ii.proc(REGBIT f))1w)
                (write_location_ev ii(LOCATION_REG ii.proc(REGBIT f))0w)

```

```

(read_status_ev ii f) : bool M =
seqT_ev(read_location_ev ii(LOCATION_REG ii.proc(REGBIT f)))(λw. constT_ev(w[0]))

```

```

aligned32 a = ((a&&3w) = 0w)

```

```

(write_mem8_ev ii a x) = failureT_ev

```

```

(read_mem8_ev ii a) = failureT_ev

```

```

(write_mem32_ev ii a(x : word32)) : unit M =
if aligned32 a then
write_location_ev ii(LOCATION_MEM a)x
else
failureT_ev

```

```

(read_mem32_ev ii a) : word32 M =
if aligned32 a then
read_location_ev ii(LOCATION_MEM a)
else
failureT_ev

```

```

(write_reserve_bit_ev(ii : iid)x) : unit M =
write_location_ev ii(LOCATION_RES ii.proc)(if x then 1w else 0w)

```

```

(read_reserve_bit_ev(ii : iid)) : bool M =
seqT_ev(read_location_ev ii(LOCATION_RES ii.proc))(λw. constT_ev(w[0]))

```

```

(write_reserve_address_ev(ii : iid)x) : unit M =
write_location_ev ii(LOCATION_RES_ADDR ii.proc)x

```

```

(read_reserve_address_ev(ii : iid)) : word32 M =
read_location_ev ii(LOCATION_RES_ADDR ii.proc)

```

```

(constT : 'a → 'a M) = constT_ev

```

```

(addT : 'a → 'b M → ('a#'b)M) = addT_ev

```

(lockT : unit M → unit M) = lockT\_ev

(syncT : iid → unit M) = syncT\_ev

(failureT : unit M) = failureT\_ev

(seqT : 'a M → ('a → 'b M) → 'b M) = seqT\_ev

(control\_seqT : 'a M → ('a → 'b M) → 'b M) = control\_seqT\_ev

(parT : 'a M → 'b M → ('a#'b)M) = parT\_ev

(parT\_unit : unit M → unit M → unit M) = parT\_unit\_ev

(write\_reg : iid → ppc\_reg32 → word32 → unit M) = write\_reg\_ev

(read\_reg : iid → ppc\_reg32 → word32 M) = read\_reg\_ev

(write\_status : iid → ppc\_bit → bool option → unit M) = write\_status\_ev

(read\_status : iid → ppc\_bit → bool M) = read\_status\_ev

(write\_mem8 : iid → word32 → word8 → unit M) = write\_mem8\_ev

(read\_mem8 : iid → word32 → word8 M) = read\_mem8\_ev

(write\_mem32 : iid → word32 → word32 → unit M) = write\_mem32\_ev

(read\_mem32 : iid → word32 → word32 M) = read\_mem32\_ev

(write\_reserve\_bit : iid → bool → unit M) = write\_reserve\_bit\_ev

(read\_reserve\_bit : iid → bool M) = read\_reserve\_bit\_ev

(write\_reserve\_address : iid → word32 → unit M) = write\_reserve\_address\_ev

(read\_reserve\_address : iid → word32 M) = read\_reserve\_address\_ev

**Part XX**

**ppc\_program**

**type\_abbrev** program\_Pinstruction : (address  $\rightarrow$  (Pinstruction#num) option)

```
(ppc_decode_program_fun : program_word8  $\rightarrow$  address  $\rightarrow$  (Pinstruction#num) option) prog_word8 a =
if (a&&3w = 0w) then NONE
else
let w0 = prog_word8(a + 0w) in
let w1 = prog_word8(a + 1w) in
let w2 = prog_word8(a + 2w) in
let w3 = prog_word8(a + 3w) in
if mem NONE [w0; w1; w2; w3] then NONE else
let raw_instruction = w2bits(the w0) ++ w2bits(the w1) ++ w2bits(the w2) ++ w2bits(the w3) in
let io = ppc_decode raw_instruction in
case io of
  NONE  $\rightarrow$  NONE
  || SOME i  $\rightarrow$  SOME (i, 4)
```

```
(ppc_decode_program_rel : program_word8  $\rightarrow$  program_Pinstruction  $\rightarrow$  bool)
  prog_word8 prog_Xinst =
 $\forall a$ . case prog_Xinst a of
  NONE  $\rightarrow$  T
  || SOME (inst, n)  $\rightarrow$  ppc_decode_program_fun prog_word8 a = SOME (inst, n)
```

ppc\_event\_execute = ppc\_event\_opsem \$ppc\_exec\_instr

```
ppc_execute_with_pc_check ii inst pc =
let s = (ppc_event_execute ii inst){} in
  {E
    |  $\exists iid\_next$  x.
      (iid_next, x, E)  $\in$  s  $\wedge$ 
       $\forall v$  ev. ((ev.action = (ACCESS R (LOCATION_REG ii. proc (REG32 PPC_PC)) v))  $\wedge$ 
        ev  $\in$  E.events)  $\implies$  (v = pc)
  }
```

```
(ppc_event_structures_of_run_skeleton : program_Pinstruction  $\rightarrow$  run_skeleton  $\rightarrow$  (ppc_reg event_structure)set)
  prog_Pinstruction rs =
let Ess = {ppc_execute_with_pc_check[ proc := p; poi := i] inst pc |  $\exists n$ .
  (rs p i = SOME pc)  $\wedge$  (SOME (inst, n) = prog_Pinstruction pc)}
in
  {event_structure_bigunion Es | Es  $\in$  all_choices Ess}
```

```
(ppc_semantics : program_word8  $\rightarrow$  (ppc_reg state_constraint)  $\rightarrow$ 
(run_skeleton#program_Pinstruction#(((ppc_reg event_structure)#((ppc_reg execution_witness)set))set))set)
prog_word8 initial_state =
```

```
let x1 = {(rs, prog_Xinst) | rs, prog_Xinst | run_skeleton_wf(DOMAIN prog_Xinst)rs  $\wedge$ 
  ppc_decode_program_rel prog_word8 prog_Xinst} in
```

**let**  $x2 = \{(rs, prog\_Xinst, Es) \mid (rs, prog\_Xinst) \in x1 \wedge$   
 $(Es = ppc\_event\_structures\_of\_run\_skeleton\ prog\_Xinst\ rs)\}$  **in**

**let**  $x3 = \{(rs, prog\_Xinst, \{(E, Xs) \mid E \in Es \wedge$   
 $(Xs = \{X \mid \text{valid\_execution}\ E\ X \wedge$   
 $(X.initial\_state = initial\_state)\})\})$

$\mid (rs, prog\_Xinst, Es) \in x2\}$  **in**  
 $x3$



# Index

*action*, 4  
*ADD*, 28  
*addr\_mode1*, 27  
*addr\_mode2*, 28  
*address\_or\_data\_dependency\_load\_load*, 10  
*address\_or\_data\_or\_control\_dependency\_load\_store*, 10  
*AddressDescriptor*, 16  
*addressing\_mode1*, 22  
*addressing\_mode2*, 22  
*addressing\_mode3*, 22  
*addT*, 41, 48, 81, 86  
*addT\_ev*, 45, 84  
*addT\_seq*, 38, 79  
*align*, 17  
*aligned32*, 48, 86  
*ALU*, 29  
*ALU\_arith*, 28  
*ALU\_logic*, 28  
*ALU\_multiply*, 28  
*ALU\_multiply\_long*, 28  
*AND*, 29  
*architecture*, 4  
*AREAD\_CP*, 37  
*AREAD\_EXCL*, 37  
*AREAD\_INFO*, 37  
*AREAD\_MEM*, 37  
*AREAD\_PSR*, 37  
*AREAD\_REG*, 37  
*arithmetic*, 29  
*arm\_decode*, 52  
*arm\_decode\_program\_fun*, 56  
*arm\_decode\_program\_rel*, 56  
*arm\_event\_execute*, 56  
*arm\_event\_structures\_of\_run\_skeleton*, 56  
*arm\_execute*, 34  
*arm\_execute\_with\_pc\_check*, 56  
*arm\_reg*, 20  
*arm\_semantics*, 56  
*ARMcondition*, 15  
*ARMcpr\_registers*, 15  
*ARMexception*, 15  
*ARMextensions*, 16  
*ARMinfo*, 16  
*ARMinstruction*, 22  
*ARMmode*, 15  
*ARMpsr*, 15  
*ARMreg*, 15  
*ARMsctlr*, 15  
*ARMstatus*, 15  
*ARMversion*, 15  
*ASR*, 27  
*assertT*, 67  
*AWRITE\_EXCL*, 37  
*AWRITE\_MEM*, 37  
*AWRITE\_PSR*, 37  
*AWRITE\_REG*, 37  
  
*bindT*, 41, 49  
*bindT\_ev*, 45  
*bindT\_seq*, 37  
*bit\_update*, 66  
*branch\_exec*, 30  
*branch\_write\_pc*, 26  
*breakpoint\_exec*, 33  
  
*check\_atomicity*, 12  
*check\_dmb\_arm*, 11  
*check\_sync\_power\_2\_05*, 11  
*choiceT\_ev*, 45, 84  
*condition\_decode*, 52  
*condition\_passed*, 26  
*conditional*, 66  
*condT*, 41, 49  
*condT\_ev*, 45, 84  
*condT\_seq*, 38  
*const\_high*, 66  
*const\_low*, 66  
*constT*, 41, 48, 81, 86  
*constT\_ev*, 45, 84  
*constT\_seq*, 37, 79

- control\_seqT*, 81, 87
- control\_seqT\_ev*, 45, 84
- data\_memory\_barrier\_exec*, 34
- data\_processing\_exec*, 30
- decode\_psr*, 17
- dirn*, 4
- discardT*, 41, 49
- discardT\_ev*, 45
- discardT\_seq*, 38
- dmbT*, 42, 50
- dmbT\_ev*, 48
- dmbT\_seq*, 40
- effective\_address*, 67
- encode\_psr*, 18
- EOR*, 29
- event*, 4
- event\_structure*, 4
- event\_structure\_bigunion*, 44, 83
- event\_structure\_control\_seq\_union*, 44, 83
- event\_structure\_empty*, 44, 83
- event\_structure\_lock*, 44, 83
- event\_structure\_seq\_union*, 44, 83
- event\_structure\_union*, 44, 83
- exception\_exec*, 29
- exception\_to\_address*, 26
- exception\_to\_mode*, 25
- exclusive\_monitors\_passT*, 42, 50
- exclusive\_monitors\_passT\_ev*, 48
- exclusive\_monitors\_passT\_seq*, 39
- ExclusiveMonitors*, 16
- execution\_witness*, 4
- failureT*, 41, 49, 81, 87
- failureT\_ev*, 45, 84
- failureT\_seq*, 37, 79
- FIX*, 11
- FullAddress*, 16
- get\_mem\_l\_stores*, 10
- goto\_label*, 67
- gpr\_or\_zero*, 67
- iiid*, 2
- iiid\_dummy*, 77
- iiids*, 8
- immediate*, 27
- inc\_pc*, 26
- initial\_iiid\_state*, 5
- InstrSet*, 16
- intra\_causality*, 9
- is\_barrier*, 8
- is\_sync*, 8
- load*, 8
- load\_exclusive\_exec*, 32
- load\_store\_exec*, 31
- load\_word*, 67
- loc*, 7
- local\_register\_data\_dependency*, 9
- location*, 4
- location\_res\_value*, 12
- lockT*, 41, 48, 81, 86
- lockT\_ev*, 45, 84
- lockT\_seq*, 38, 79
- LSL*, 26
- LSR*, 27
- mapT\_ev*, 44, 83
- MBReqDomain*, 17
- MBReqTypes*, 17
- mem\_access*, 7
- mem\_load*, 7
- mem\_store*, 7
- MemoryAttributes*, 16
- MemType*, 16
- mode\_to\_psr*, 25
- mode\_to\_word5*, 25
- multiply\_exec*, 30
- multiply\_long\_exec*, 31
- next\_iiid*, 5
- no\_carry*, 67
- OK\_nextinstr*, 66
- option\_apply*, 42, 81
- ORR*, 29
- OUR\_INFO*, 47
- OUR\_INSTR\_SET*, 47
- OUR\_MODE*, 47
- OUR\_SCTLR*, 47
- OUR\_VERSION*, 47
- parT*, 41, 49, 81, 87
- parT\_ev*, 45, 84
- parT\_seq*, 37, 79
- parT\_unit*, 49, 81, 87
- parT\_unit\_ev*, 46, 84
- parT\_unit\_seq*, 80

- Pinstruction*, 63
- po*, 8
- po\_iico\_both*, 9
- po\_iico\_data*, 9
- po\_strict*, 9
- ppc\_bit*, 59
- ppc\_decode*, 75
- ppc\_decode\_program\_fun*, 89
- ppc\_decode\_program\_rel*, 89
- ppc\_event\_execute*, 89
- ppc\_event\_structures\_of\_run\_skeleton*, 89
- ppc\_exec\_instr*, 68
- ppc\_execute\_with\_pc\_check*, 89
- ppc\_match\_step*, 75
- PPC\_NEXT*, 77
- ppc\_reg*, 61
- ppc\_reg32*, 59
- ppc\_semantics*, 89
- ppc\_sint\_cmp*, 66
- ppc\_uint\_cmp*, 66
- PREAD\_M*, 79
- PREAD\_R*, 79
- PREAD\_REVERSE\_ADDRESS*, 79
- PREAD\_REVERSE\_BIT*, 79
- PREAD\_S*, 79
- preserved\_coherence\_order*, 10
- preserved\_program\_order*, 10
- preserved\_program\_order\_mem\_loc*, 10
- proc*, 7
- procs*, 8
- PWRITE\_M*, 79
- PWRITE\_R*, 79
- PWRITE\_REVERSE\_ADDRESS*, 79
- PWRITE\_REVERSE\_BIT*, 79
- PWRITE\_S*, 79
- read\_bit\_word*, 66
- read\_endian\_seq*, 38
- read\_flags*, 42, 49
- read\_flags\_ev*, 47
- read\_flags\_seq*, 38
- read\_info*, 41, 49
- read\_info\_ev*, 47
- read\_info\_seq*, 39
- read\_instr\_set*, 42, 49
- read\_instr\_set\_ev*, 47
- read\_instr\_set\_seq*, 38
- read\_ireg*, 67
- read\_location\_ev*, 46, 85
- read\_location\_res\_value*, 12
- read\_mem16*, 42, 50
- read\_mem16\_ev*, 48
- read\_mem16\_seq*, 41
- read\_mem32*, 42, 50, 81, 87
- read\_mem32\_ev*, 48, 86
- read\_mem32\_seq*, 41, 80
- read\_mem8*, 42, 50, 81, 87
- read\_mem8\_ev*, 48, 86
- read\_mem8\_seq*, 41, 80
- read\_mem\_aux*, 67
- read\_mode*, 42, 49
- read\_mode\_ev*, 47
- read\_mode\_seq*, 38
- read\_most\_recent\_value*, 11
- read\_psr*, 42, 49
- read\_psr\_ev*, 47
- read\_psr\_seq*, 38
- read\_reg*, 42, 49, 81, 87
- read\_reg\_ev*, 47, 85
- read\_reg\_seq*, 38, 80
- read\_regm*, 25
- read\_reserve\_address*, 81, 87
- read\_reserve\_address\_ev*, 86
- read\_reserve\_address\_seq*, 81
- read\_reserve\_bit*, 81, 87
- read\_reserve\_bit\_ev*, 86
- read\_reserve\_bit\_seq*, 80
- read\_sctlr*, 42, 49
- read\_sctlr\_ev*, 47
- read\_sctlr\_seq*, 39
- read\_status*, 81, 87
- read\_status\_ev*, 86
- read\_status\_seq*, 80
- read\_version*, 42, 49
- read\_version\_ev*, 47
- read\_version\_seq*, 39
- reads*, 8
- reg\_access*, 7
- reg\_load*, 7
- reg\_or\_mem\_location*, 7
- reg\_or\_mem\_or\_resaddr*, 8
- reg\_store*, 7
- reg\_update*, 66
- register\_load*, 67
- register\_store*, 67
- ROR*, 27
- rotate\_mem32*, 29
- run\_skeleton\_wf*, 5

- same\_granule*, 12
- seqT*, 41, 49, 81, 87
- seqT\_ev*, 45, 84
- seqT\_seq*, 37, 79
- set\_CR0\_to\_00xNONE*, 68
- set\_exclusive\_monitorsT*, 42, 50
- set\_exclusive\_monitorsT\_ev*, 48
- set\_exclusive\_monitorsT\_seq*, 39
- shift\_immediate*, 27
- shift\_register*, 27
- sint\_compare*, 66
- sint\_reg\_update*, 66
- state\_updates*, 11
- status\_to\_register\_exec*, 33
- store*, 8
- store\_exclusive\_exec*, 32
- store\_word*, 67
- SUB*, 29
- swap\_exec*, 33
- synchronization*, 4
- syncT*, 81, 87
- syncT\_ev*, 46, 85
- syncT\_seq*, 79
  
- test\_or\_compare*, 29
- type\_abbrev\_address*, 4
- type\_abbrev\_Aimm*, 15
- type\_abbrev\_Ainstruction*, 56
- type\_abbrev\_arm\_state*, 37
- type\_abbrev\_crbit*, 59
- type\_abbrev\_eiid*, 4
- type\_abbrev\_eiid\_state*, 5
- type\_abbrev\_ExclusiveMonitor*, 16
- type\_abbrev\_freq*, 59
- type\_abbrev\_ireg*, 59
- type\_abbrev\_M*, 37, 44, 79, 83
- type\_abbrev\_ppc\_constant*, 59
- type\_abbrev\_ppc\_state*, 79
- type\_abbrev\_proc*, 2
- type\_abbrev\_program\_Ainstruction*, 56
- type\_abbrev\_program\_order\_index*, 2
- type\_abbrev\_program\_Pinstruction*, 89
- type\_abbrev\_program\_word8*, 5
- type\_abbrev\_reln*, 4
- type\_abbrev\_run\_skeleton*, 5
- type\_abbrev\_state\_constraint*, 4
- type\_abbrev\_value*, 4
- type\_abbrev\_view\_orders*, 4
- type\_abbrev\_Ximm*, 4
  
- uint\_compare*, 66
- uint\_reg\_update*, 66
- user\_or\_system\_mode*, 25
  
- valid\_execution*, 12
- value\_of*, 7
- VERSION\_MULTICORE*, 56
- version\_number*, 18
- view\_orders\_well\_formed*, 10
- viewed\_events*, 10
  
- well\_formed\_event\_structure*, 9
- word4\_mode\_to\_reg*, 25
- word5\_to\_mode*, 17
- write\_flags*, 42, 49
- write\_flags\_ev*, 47
- write\_flags\_seq*, 38
- write\_location\_ev*, 46, 85
- write\_mem16*, 42, 50
- write\_mem16\_ev*, 48
- write\_mem16\_seq*, 40
- write\_mem32*, 42, 50, 81, 87
- write\_mem32\_ev*, 48, 86
- write\_mem32\_seq*, 40, 80
- write\_mem8*, 42, 50, 81, 87
- write\_mem8\_ev*, 48, 86
- write\_mem8\_seq*, 40, 80
- write\_mem\_aux*, 67
- write\_psr*, 42, 49
- write\_psr\_ev*, 47
- write\_psr\_seq*, 38
- write\_reg*, 42, 49, 81, 87
- write\_reg\_ev*, 46, 85
- write\_reg\_seq*, 38, 80
- write\_regm*, 25
- write\_reserve\_address*, 81, 87
- write\_reserve\_address\_ev*, 86
- write\_reserve\_address\_seq*, 80
- write\_reserve\_bit*, 81, 87
- write\_reserve\_bit\_ev*, 86
- write\_reserve\_bit\_seq*, 80
- write\_serialization\_candidates*, 10
- write\_status*, 81, 87
- write\_status\_ev*, 85
- write\_status\_seq*, 80
- writes*, 8