CTMM ISA Reference — Release 1

Church-Turing Meta-Machine

Kenneth J Hamer-Hodges

May 2026

Church Machine ISA Reference

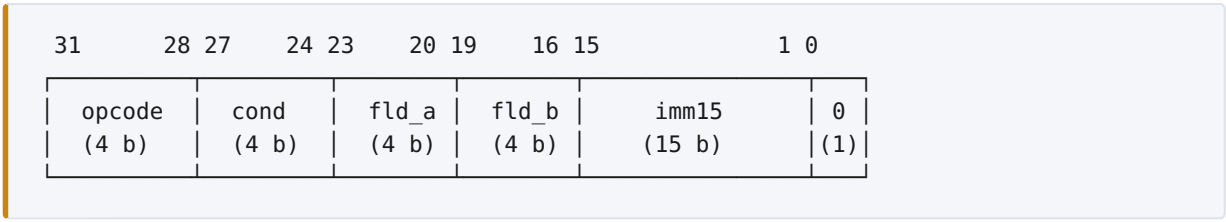
Version 1.0 — May 2026 Authoritative sources: `simulator/simulator.js`, `simulator/assembler.js`, `hardware/*.py`

This document is the single definitive specification for all 20 Church Machine instructions. Where existing documents conflict with what is stated here, this document takes precedence. Simulator/hardware deviations are called out explicitly; see `docs/HARDWARE-DEVIATIONS.md` for the full deviation register.

---

1. Instruction Word Format

Every instruction is a 32-bit word with a fixed layout:



- **Bit 0** is always 0 (word-aligned; a 1 here is a FAULT).
- **opcode** (bits 31:28): selects one of the 20 instructions (0–19). Values 20–15 are reserved and fault.
- **cond** (bits 27:24): condition under which the instruction executes (see §2).
- **fld_a** (bits 23:20): first register operand — CR or DR index depending on instruction.
- **fld_b** (bits 19:16): second register operand — CR or DR index depending on instruction.
- **imm15** (bits 15:1): 15-bit immediate; interpretation varies per instruction.

The **all-zero word** `0x00000000` (opcode=LOAD, cond=EQ, all fields zero) is accepted by the assembler as `HALT` or `NOP`. The simulator treats an all-zero instruction word encountered during normal execution as a warm reboot (not a halt and not a fault). See §4 (HALT/NOP note) for implications.

---

2. Condition Codes

The `cond` field gates execution on the current flag state. If the condition is false, the instruction is skipped (PC advances, no side effects, no faults).

Code	Mnemonic	Meaning	Flags tested
0	EQ	Equal / Zero	Z = 1
1	NE	Not equal / Non-zero	Z = 0
2	LT	Less than (signed)	N ≠ V
3	LE	Less than or equal (signed)	Z = 1 or N ≠ V
4	GT	Greater than (signed)	Z = 0 and N = V
5	GE	Greater than or equal	N = V
6	CS / CC	Carry set	C = 1
7	CC	Carry clear	C = 0
8	MI	Minus / Negative	N = 1
9	PL	Plus / Non-negative	N = 0
10	VS	Overflow set	V = 1
11	VC	Overflow clear	V = 0
12	HI	Unsigned higher	C = 1 and Z = 0
13	LS	Unsigned lower or same	C = 0 or Z = 1
14	AL	Always (unconditional)	(none)
15	NV	Never (no-op)	(none — always skip)

`AL` (always) is the normal unconditional form. `NV` is a no-op regardless of flags.

---

3. Register Files

3.1 Capability Registers (CR0–CR15)

Sixteen 64-bit capability registers. Each holds a **Guard Token (GT)**: a type-tagged, permission-bearing, hardware-verified reference to an object.

Range	Name	Notes
CR0–CR5	User CRs	General-purpose; caller context preserved by CALL
CR6	C-list root	E-permission token for current abstraction's c-list; re-derived by CALL/RETURN
CR7–CR11	User CRs	General-purpose; caller context preserved by CALL
CR12	Thread stack	Privileged; system-wide; unchanged by CALL/RETURN; only writeable via CHANGE
CR13	Interrupt handler	Privileged; system-wide; only writeable via SWITCH (hardware: PassKey gate)
CR14	Code register (CLOOMC)	Privileged; per-thread; set by CALL, re-derived by RETURN; X-only
CR15	Namespace root	Privileged; per-thread; only writeable via SWITCH (hardware: PassKey gate)

Privilege zone: CR12–CR15 cannot appear as operands in LOAD, SAVE, TPERM, LAMBDA instructions. CALL, RETURN, CHANGE, and SWITCH are the only Church-domain instructions that touch them.

3.2 Data Registers (DR0–DR15)

Sixteen 32-bit integer registers.

> **A.1 — DR0 is hardwired zero.**
>
> DR0 reads as 0 at all times. After every instruction that produces a result, the simulator unconditionally writes 0 to DR0 (`simulator.js` line 2748: `this._writeDR(0, 0)`). Writes targeting DR0 are silently discarded — the value is immediately overwritten back to 0.
>
> This enables two universal idioms, replacing MOV and load-immediate opcodes that would otherwise need their own encodings:

Idiom	Instruction	Effect
Register copy	`IADD DRd, DR0, DRs`	DRd ← DRs
Load immediate	`IADD DRd, DR0, #k`	DRd ← k (0 ≤ k ≤ 16383)

Any instruction that writes a computed result into DR0 (e.g., `IADD DR0, DR1, DR2`) always reads back 0 on the next instruction. This is not a bug — it is the intended architectural property. Do not use DR0 as a scratch register.

3.3 Permission Bits (GT word0)

The permission field of a GT encodes the following access rights:

Bit	Symbol	Meaning
30	E	Execute — may call the abstraction
29	S	Save — may store a GT into this object's c-list
28	L	Load — may load a GT from this object's c-list
27	X	Code — execute raw instructions from lump memory
26	W	Write — may write data words into lump memory
25	R	Read — may read data words from lump memory
31	B	Busy — object lock; clearable by TPERM B-modifier

Domain purity rule: X may not coexist with L, S, or E in the same GT's effective permission set. A GT that would combine X with any of L/S/E is invalid and causes TPERM to fault with `TPERM_RSV` when the combination is tested.

---

4. HALT / NOP (all-zero word)

```
Encoding: 0x00000000
Assembler aliases: HALT, NOP
```

The all-zero word is architecturally the instruction `LOAD AL, CR0, CR0, #0` — a conditional LOAD that would load CR0 from `CR0[0]`. In practice, an all-zero instruction word is used to mark the end of a code region.

Simulator behaviour: an all-zero word encountered during execution triggers a warm reboot sequence, not a halt. Execution does not pause cleanly; the boot ROM re-runs. Writers of code lumps should never allow execution to fall through to an all-zero word unless a reboot is the intended outcome.

---

5. Flag Behaviour — Quick Reference

Four flags: **N** (negative), **Z** (zero), **C** (carry), **V** (overflow).

> **A.2 — BFEXT and BFINS do write flags: N and Z reflect the result; C and V are always cleared.**
>
> Hardware (`core.py` lines 1140–1143, 1169–1172) sets N = result[31], Z = (result == 0), C = 0, V = 0 for both instructions. The simulator (`_execBfext`, `_execBfins`) matches this behaviour.
>
> This means a BFEXT result can be tested directly with a conditional branch:
>
> ```
> BFEXT  DR1, DR2, 0, 8      ; extract byte — Z = 1 if byte is zero
> BRANCH EQ, handle_zero     ; correctly tests the extracted byte
> ```
>
> Note that C and V are **cleared**, not preserved. Any preceding instruction's carry or overflow flag is lost after BFEXT or BFINS.

Flag-writing summary across all 20 instructions:

Instruction	N	Z	C	V	Notes
LOAD	—	—	—	—	
SAVE	—	—	—	—	
CALL	—	—	—	—	
RETURN	—	—	—	—	
CHANGE	—	—	—	—	
SWITCH	—	—	—	—	
TPERM	✓	✓	0	0	N = !Z; C and V always cleared
LAMBDA	—	—	—	—	
ELOADCALL	—	—	—	—	
XLOADLAMBDA	—	—	—	—	
DREAD	—	—	—	—	
DWRITE	—	—	—	—	
BFEXT	✓	✓	0	0	N = result[31]; Z = (result==0); C and V cleared — see A.2
BFINS	✓	✓	0	0	N = result[31]; Z = (result==0); C and V cleared — see A.2
MCMP	✓	✓	✓	✓	Subtraction flags: a − b; no result register
IADD	✓	✓	✓	✓	Addition flags
ISUB	✓	✓	✓	✓	Subtraction flags
BRANCH	—	—	—	—	Reads flags, never writes them
SHL	✓	✓	✓	0	C = last bit shifted out (source[32-shamt], 0 if shamt=0); V

Instruction	N	Z	C	V	Notes
					always 0. Hardware confirmed (Task #857)
SHR	✓	✓	✓	0	C = last bit shifted out (source[shamt-1], 0 if shamt=0); imm[5]=0→LSR, imm[5]=1→ASR (sign-extend). V always 0. Hardware confirmed (Task #857)

6. Cross-Cutting Encoding Rules

A.3 — SHR: bit 5 selects ASR vs LSR; A.4 — BRANCH: bit 14 is the sign bit

A.3 — SHR imm15[5] = mode select.

`SHR DRd, DRs, #amt` encodes the shift amount in `imm15[4:0]` (0–31). `imm15[5]` selects the fill mode:

imm15[5]	Mode	Fill bit	Assembler suffix
0	LSR	0	(none)
1	ASR	sign bit (DRs[31])	`ASR`

Assembler syntax: `SHR DR1, DR2, #4` (LSR) or `SHR DR1, DR2, #4, ASR` (ASR). C = last bit shifted out. V = 0 always.

Simulator/hardware deviation (D-12, open): The hardware currently implements LSR only (C = 0, no ASR mode). The simulator implements both correctly. Task #857 tracks the hardware fix.

A.4 — BRANCH: bit 14 of imm15 is the sign bit.

BRANCH uses all 15 bits of the imm15 field as a signed PC-relative word offset:

```
soff  = (imm & 0x4000) ? (imm | 0xFFFF8000) : imm   ; sign-extend bit 14
target = current_PC + soff                            ; word addressing
```

Range: **−16384 to +16383 words** (±65536 bytes on byte-addressed hardware).

Hardware sign-extension (Amaranth): `Cat(immediate, immediate[14].replicate(17))` then `nia += sign_extend(imm) × 4` — equivalent; byte vs word addressing only.

Key offsets:

Offset	Effect
`#0`	**Infinite loop** — branches back to itself (target = current PC)
`#1`	Fall-through — target = current_PC + 1 = next instruction (effectively a NOP branch)
`#-1`	Step back — target = previous instruction

Assembler label resolution: labels resolve at assemble time as `offset = label_word_index − branch_word_index`. The assembler is two-pass, so forward references (label appears after the branch) are legal. After encoding, every BRANCH imm is bounds-checked to fit in −16384..+16383; out-of-range offsets are a hard assembler error.

Runtime bounds: the simulator faults immediately with BOUNDS if the target is outside the loaded memory image. Hardware detects the violation on the next instruction fetch via the CR14 code fence (`fetch_bounds_fault`).

Condition code is mandatory. There is no bare `BRANCH label` form — the condition code is always present in the word encoding. Use `BRANCH AL, label` for an unconditional jump, or the alias mnemonics: `BRANCHEQ`, `BRANCHNE`, `BRANCHLT`, `BRANCHGE`, `BRANCHGT`, `BRANCHLE`, etc.

A.5 — CALL: 1-based method index; A.6 — ELOADCALL: split imm15

A.5 — CALL imm15 is 1-based: user method index N encodes as imm15 = N + 1.

imm15 = 0 is the fast-path shorthand: NIA = lump_base + 4 (word 1, first instruction in the lump). The method table is bypassed entirely. This is the correct encoding for single-entry-point abstractions.

imm15 > 0 is a table lookup: hardware reads `memory[lump_base + imm15 × 4]` to get the callee's first instruction word offset, then sets NIA = lump_base + entry × 4. A zero table entry → PRIVATE_METHOD FAULT.

What you write	imm15 encoded	Hardware does
`CALL CRn` (no selector)	0	Fast path: NIA = lump_base + 4
`CALL CRn, 0`	1	Table entry [1]: user method index 0
`CALL CRn, 1`	2	Table entry [2]: user method index 1
`CALL CRn, N`	N + 1	Table entry [N+1]: user method index N

Assembler: named methods (`CALL CRn, MethodName`) are resolved from the registered method conventions table and encoded as `imm15 = method.index + 1`. Dot-notation (`CALL SlideRule.Multiply`) resolves the object binding and method name in one step. Numeric selector range: 0–16383 (imm15 range: 1–16384).

Disassembly note: a disassembled CALL showing imm15 = 3 means user method index 2, not 3. Always subtract 1 from the raw imm15 field to get the user-facing method selector.

A.6 — ELOADCALL imm15 is split into two fields.

```
imm15[14:8]  =  method index (7 bits, 0–127)    — same 1-based encoding as CALL
imm15[7:0]   =  c-list row   (8 bits, 0–255)    — word offset into CRsrc c-list
```

ELOADCALL atomically loads the GT at `CRsrc[row]` into the destination CR and calls it with the given method index. The method index is encoded identically to CALL (0 = fast path, N+1 = user method N).

Backward compatibility: old programs that encoded `ELOADCALL CRd, CRs, #N` with a simple row number stored the row in bits[7:0] and left bits[14:8] = 0. Method index 0 → fast path — behaviour is identical to the pre-split encoding.

Valid ranges (enforced by assembler and checked at assembly time): - c-list row: 0–255 (8 bits; values 256+ are a hard assembler error) - method index: 0–126 in user terms → 0–127 in imm15[14:8] (value 127 rejected)

A.7–A.11 — TPERM: preset table, NULL behaviour, domain-purity, B-modifier, flag invariants

TPERM preset encoding (imm15[4:0])

Bit 4 = B-modifier (see A.11). Bits [3:0] = preset code (0–15).

Code	Mnemonic	Permissions required	Reserved?
0x00	CLEAR	none	No
0x01	R	R	No
0x02	RW	R, W	No
0x03	X	X	No
0x04	RX	R, X	No
0x05	RWX	R, W, X	No
0x06	L	L	No
0x07	S	S	No
0x08	E	E	No
0x09	LS	L, S	No
0x0A	W	W only (no R)	No
0x0B–0x0F	—	—	**Reserved**

Adding 0x10 to any valid code sets the B-modifier: `TPERM CRd, RB` = code 0x11. All B-modifier variants of 0x0B–0x0F are also reserved.

A.7 — CLEAR (preset 0) always passes for any non-NULL, non-reserved GT.

`TPERM CRd, CLEAR` requires no permissions. Since the required set is empty, `required.every(p => gt.permissions[p])` is vacuously true. Z = 1 for any valid non-NULL GT regardless of what permissions it actually holds. This is the standard “GT is live and non-NULL” existence check.

A.8 — NULL GT always produces Z = 0, N = 1, no fault.

If `CR.word0 = 0` (NULL GT), TPERM immediately sets Z = 0, N = 1, C = 0, V = 0 and returns — before checking the preset code, before the domain-purity check, before anything else. No fault is raised. This applies to every preset including CLEAR: `TPERM CRd, CLEAR` on a NULL GT gives Z = 0.

Pattern to distinguish NULL from “lacks permission”:

```
TPERM  CR1, CLEAR        ; Z=1 → non-NULL; Z=0 → NULL
BRANCH EQ, not_null
```

A.9 — Domain-purity violation → hard FAULT(TPERM_RSV).

If the result permission set (intersection of preset’s required bits and the GT’s held bits) would combine X with any of L, S, or E, TPERM faults with `TPERM_RSV`. This is a hard fault — not Z = 0, not recoverable.

No built-in preset triggers this (presets are X-pure or LSE-pure, never mixed), but a GT that already combines X and L/S/E could trigger it on certain presets. In practice this guards against malformed GTs reaching code that uses them.

A.10 — TPERM flag invariants: N = !Z, C = 0, V = 0 always.

These three relationships hold unconditionally after every TPERM that does not fault. The flag table in §5 already captures this. Key implication: C and V are **always cleared** by TPERM — any preceding instruction’s carry or overflow is lost.

A.11 — B-modifier (imm15 bit 4): clears the Busy bit on a passing test.

When bit 4 of the imm15 field is set and the permission test passes (Z = 1), TPERM clears bit 31 of `CR.word0` (the B “Busy” bit) **in place** — no namespace write, no SAVE needed. The change is local to the CR until a SAVE commits it.

If the test fails (Z = 0), the B bit is left unchanged regardless of the modifier. The modifier has no effect on flags.

```
TPERM  CR2, EB          ; test for E permission; if Z=1, clear B bit
atomically
BRANCH EQ, call_ok      ; Z=1: abstraction is callable and now marked un-busy
```

Reserved preset + B-modifier: codes 0x1B–0x1F are reserved; behaviour matches A-13 — both hardware and simulator fault with `TPERM_RSV` (D-3 closed; simulator aligned with hardware as of Task #873).

A.12–A.14 — Call stack: CALL frame layout, LAMBDA SZ=0, M-window writeback

A.12 — CALL pushes exactly 2 words to thread memory; no CRs or DRs.

When CALL executes, it writes two words into the current thread's lump memory at the stack pointer (STO):

Word offset (from STO)	Contents
STO	Frame word: packed (returnPC[14:0] | sz[12] | flags[11:8] | savedSTO[7:0])
STO − 1	Caller's E-GT (CR6 value before the call)

`sz = 1` distinguishes CALL frames from LAMBDA frames (sz = 0). No capability registers or data registers are written to thread memory — the callee inherits all DRs and CRs from the caller (with the exception of CR6 and CR14, which are replaced by the callee's c-list and code tokens).

The JS-side simulator call stack (`callStack[]`) additionally holds a snapshot of all saved registers for state inspection, but this is not part of the hardware frame format.

A.13 — LAMBDA pushes a SZ=0 (1-word) frame; RETURN identifies it by sz.

LAMBDA writes only the frame word (sz = 0) to thread memory — no E-GT slot. RETURN distinguishes frame types by the sz field:

sz	Frame type	Pop size	Return address source
0	LAMBDA	1 word	`lambdaReturnPC` cache (no memory read)
1	CALL	2 words	Frame word in thread memory

The leaf-lambda fast path: when `lambdaActive = 1` and the frame popped is SZ=0, RETURN restores PC from the cached `lambdaReturnPC` register without a memory read. This gives O(1) RETURN at any recursion depth.

LAMBDA CR6 idempotent re-entry (D-9): re-executing `LAMBDA CR6` while `lambdaActive = 1` is non-faulting (same return address overwrites the same register). `LAMBDA CRn` (n ≠ 6) while `lambdaActive = 1` → FAULT.

A.14 — M-window writeback fires at every CALL and every RETURN.

When an abstraction is called via an Abstract GT (M-bit = 1), hardware tracks modified namespace state in a 3-register M-window (DR11–DR13). Before the callee's frame is pushed (CALL) or before the caller's frame is popped (RETURN), the M-window writeback fires: DR11–DR13 are written back to the CR15 namespace entry and the M-bit is cleared.

If the writeback fails (e.g. NULL DR11, invalid state), CALL and RETURN both fault with `INVALID_OP` before any frame manipulation occurs. The frame stack is never left in a partially-modified state.

A.15–A.16 — BFINS source masking; LOAD short form

A.15 — BFINS uses only the low `width` bits of DRs.

The value inserted is `DRs & ((1 << width) - 1)`. Upper bits of DRs beyond `width` are discarded before insertion. The destination word is modified as:

```
mask     = ((1 << width) - 1) << pos
new_word = (old_DRd & ~mask) | ((DRs & ((1<<width)-1)) << pos)
```

There is no alignment requirement — `pos` and `width` may be any values satisfying `width ≥ 1` and `pos + width ≤ 32`.

A.16 — LOAD (and SAVE, ELOADCALL, XLOADLAMBDA) short form uses CR6 implicitly.

Two-operand forms resolve the named abstraction from the assembler's NS binding table and substitute CR6 as the c-list base:

```
LOAD  CRd, SlideRule        →  LOAD  CRd, CR6, <slot>
SAVE  CRd, SlideRule        →  SAVE  CRd, CR6, <slot>
ELOADCALL  CRd, SlideRule   →  ELOADCALL  CRd, CR6, <slot>
XLOADLAMBDA  CRd, SlideRule →  XLOADLAMBDA  CRd, CR6, <slot>
```

CR6 is the c-list root by architectural convention. The slot is looked up from the namespace binding established by a prior `LOAD CRd, Name` (which registers the name → CR mapping in `nsLoaded`). Using the short form for a name that has never been loaded is a hard assembler error.

A.17 — TPERM Mode 2: capability attenuation (`imm15 = 0x7FFF`)

A.17 — `TPERM CRd, CRs, 0x7FFF` attenuates CRd's GT to the permission subset held in CRs.

When `imm15 = 0x7FFF` (all 15 immediate bits set), TPERM enters **Mode 2 (attenuation)** instead of the normal preset-test path. The two-register form uses:

Field	Role
`CRd`	Source GT — the full-authority capability being attenuated (read and written)
`CRs`	Permission template — GT whose permission bits specify the desired narrower set

Semantics:

1. If `CRd.word0 = 0` (NULL source GT): Z = 0, N = 1, C = 0, V = 0. Return. No fault.
2. Parse permission bits from `CRd` (source) and `CRs` (requested subset).
3. **Subset validation:** for every permission bit set in `CRs`, verify the same bit is also set in `CRd`. If any bit in `CRs` is absent from `CRd`, the caller is

attempting to expand authority beyond what they hold — Z = 0, N = 1, C = 0, V = 0. `CRd` is **not** modified. Return. No hard fault.

4. **Construct attenuated GT:** copy `CRd`'s NS index, version (`gt_seq`), and GT type; replace the permission bits with those from `CRs`.
5. Write the attenuated GT to `CRd`.
6. Z = 1, N = 0, C = 0, V = 0.

Encoding:

```
TPERM CRd, CRs, 0x7FFF
  imm15[14:0] = 0x7FFF  (sentinel — all bits set)
  fld_a       = CRd index
  fld_b       = CRs index
```

Identity attenuation (`CRs` has the same permissions as `CRd`) is valid and succeeds (Z = 1). The result is a logically equivalent GT written to `CRd` (same permissions and NS metadata).

NULL `CRs` (template GT word = 0) requests zero permissions (CLEAR). This is a valid strict subset of any non-NULL source: Z = 1, and `CRd` receives a zero-permission GT with the source's NS metadata.

B-modifier and domain-purity checks do not apply to Mode 2. Those checks (A.9, A.11) are in the normal preset path (Mode 1) which is bypassed when `imm15 = 0x7FFF`.

Flag invariant: N = !Z, C = 0, V = 0 always (same as Mode 1 — see A.10).

Use case — handing a restricted capability to untrusted code:

```
; CR0 holds an R,W,E GT for a shared buffer.
; We want to hand CR1 an R-only copy so the callee cannot write.
LOAD  CR1, CR0          ; CR1 = copy of source GT (same perms — temporary)
; ... set up a template GT with only R in CR1 (e.g. by constructing from
scratch)
TPERM CR0, CR1, 0x7FFF ; attenuation: CR0 gets R-only, source NS metadata
BRANCH NE, fault        ; Z=0 → expansion attempted (should not happen here)
CALL  CR0               ; call callee with attenuated R-only token
```

7. Instruction Reference — All 20 Opcodes

Encoding note: The 32-bit word layout used throughout this section is:

```
bits[31:27]  op     (5 bits)  — opcode 0–19; 20–30 reserved; 31=lump header
bits[26:23]  cond   (4 bits)  — condition code (see §2)
bits[22:19]  fld_a  (4 bits)  — first operand register (CR or DR)
bits[18:15]  fld_b  (4 bits)  — second operand register (CR or DR)
bits[14:0]   imm15  (15 bits) — immediate / index field
```

This matches `assembler.js` line 924: `((op & 0x1F) << 27) | ((cond & 0xF) << 23) | ((fld_a & 0xF) << 19) | ((fld_b & 0xF) << 15) | (imm & 0x7FFF)`. The diagram in §1 shows opcode as 4 bits — that diagram is a simplification; the correct width is 5 bits.

Permission abbreviations used below: R=Read, W=Write, X=Execute, L=Load, S=Store, E=Enter, B=Busy. “GT” = Golden Token (capability). NULL GT = word0=0.

LOAD — Load Capability (opcode 0, 0x00)

```
Syntax:  LOAD CRd, CRs, #row
         LOAD CRd, Name          (shorthand: CRs = CR6, row = NS slot — see A.16)
Encoding: op[4:0]=0x00 | cond[4] | CRd[4] | CRs[4] | row[15]
```

Semantics: Loads the 32-bit GT word from the c-list addressed by CRs at word offset `row` into `CRd`. The full NS entry is then read and installed into CRd’s three-word cache (word0=GT, word1=NS Word 1, word2=NS seals).

- **CRs = CR6** (c-list root): mLoad validates version and CRC seal; L permission check is skipped because CALL already validated E before the call frame was created.
- **CRs ≠ CR6**: mLoad validates L permission on CRs itself, reads the lump header to locate the c-list region, then validates the slot.
- **Abstract GT in slot**: installed directly into CRd without an NS table lookup (hardware device handle path).
- **Outform GT in slot**: triggers lazy-load (Mode 1 or Mode 2); NS entry is promoted Outform→Inform on completion.

Flags: N — Z — C — V (no flag writes)

Faults: | Fault | Condition | |——-|———-| | `NULL_CAP` | CRs is NULL, or c-list slot at `row` is 0 | | `PERM` | CRs lacks L permission (except when CRs = CR6) | | `BOUNDS` | `row` is outside the c-list range | | `SEAL` | NS entry CRC-16 validation fails | | `F_BIT` | Source GT is in a Far namespace (F=1) | | `CODE_NOT_RESIDENT` | Outform lazy-load failed |

Example: Load the abstraction at c-list row 5 from CR6 into CR0.

```
LOAD AL, CR0, CR6, #5    ; opcode=0, cond=14, fld_a=0, fld_b=6, imm=5
                          ; encoding: 0x07030005
```

SAVE — Save Capability (opcode 1, 0x01)

```
Syntax:  SAVE CRd, CRs, #row
         SAVE CRd, Name          (shorthand: CRs = CR6, row = NS slot — see A.16)
Encoding: op[4:0]=0x01 | cond[4] | CRd[4] | CRs[4] | row[15]
```

Field roles (note: reversed vs. what “destination” usually implies): - **CRd** — the c-list pointer. Must hold a GT with **S (Save) permission**. This is the c-list that will be written. - **CRs** — the GT to save. Must have **B = 1** (Bindable bit). A GT with B = 0 cannot be delegated.

Semantics: Writes the GT held in CRs into the c-list pointed to by CRd at word offset `row`. mSave validates: version seal on CRd, S permission on CRd, B=1 on CRs, and the F-bit policy on the target slot. The namespace entry is not touched — only the c-list word is written.

Flags: N — Z — C — V (no flag writes)

Faults: | Fault | Condition | |——-|———–| | `NULL_CAP` | CRd or CRs is NULL | | `PERM` | CRd lacks S permission, or CRs has B = 0 | | `BOUNDS` | `row` is outside the c-list range | | `SEAL` | CRd NS entry CRC fails | | `F_BIT` | Target slot has F = 1 (far-bit, cross-namespace) |

Example: Save the GT in CR1 to c-list row 3 of CR0’s c-list.

```
SAVE AL, CR0, CR1, #3    ; CRd=CR0 (S perm), CRs=CR1 (B=1), row=3
                          ; opcode=1, cond=14, fld_a=0, fld_b=1, imm=3
                          ; encoding: 0x0F008003
```

CALL — Enter Abstraction (opcode 2, 0x02)

```
Syntax:  CALL CRs [, #method_index]
         CALL CRs, MethodName     (named method — assembler resolves to 1-based
index)
         CALL Name.Method         (dot-notation shorthand)
Encoding: op[4:0]=0x02 | cond[4] | CRs[4] | 0[4] | method[15]
```

Method index encoding (see A.5): - `imm15 = 0` → fast path: NIA = lump_base + 4 (word 1, skips method table) - `imm15 = N` → hardware reads `memory[lump_base + N×4]`; result is lump-relative word offset; 0 entry = PRIVATE_METHOD FAULT

Semantics: 1. M-window writeback fires (if M-flag set); faults `INVALID_OP` if writeback fails. 2. mLoad validates E permission on CRs GT; derives callee lump base, c-list address, and code address. 3. Push 2-word call frame to thread stack (see A.12): - `memory[STO]` = frame word (packed returnPC, sz=1, flags, savedSTO) - `memory[STO-1]` = caller's E-GT (CR6 value before call) 4. CR6 ← callee c-list GT (E-only). CR14 ← callee code GT (X-only, privileged). 5. PC ← NIA (lump_base + method entry, or lump_base + 4 for fast path).

Callee inherits DR0–DR15, CR0–CR5, CR7–CR11 from the caller. CR12 and CR13 are system-wide (unchanged). CR15 is per-thread (unchanged by CALL itself — restored by RETURN).

Flags: N — Z — C — V (no flag writes)

Faults: | Fault | Condition | |——-|———–| | `NULL_CAP` | CRs is NULL | | `PERM` | CRs lacks E permission (unless Abstract GT — bypasses E check) | | `SEAL` | NS entry CRC fails | | `STACK_OVERFLOW` | STO would write into heap/DR zone | | `F_BIT` | Far-namespace GT (checked before M-window writeback) | | `PRIVATE_METHOD` | Method table entry at `imm15` is 0 | | `INVALID_OP` | M-window writeback failed |

Example: Enter the abstraction in CR1 via fast path (single entry point).

```
CALL AL, CR1              ; imm=0 → NIA = lump_base + 4
                           ; opcode=2, cond=14, fld_a=1, fld_b=0, imm=0
                           ; encoding: 0x17080000
```

RETURN — Return from Abstraction (opcode 3, 0x03)

```
Syntax:  RETURN [#mask]
Encoding: op[4:0]=0x03 | cond[4] | 0[4] | 0[4] | mask[12] | 0[3]
```

`mask` is a 12-bit field in `imm15[11:0]`. Bit N = 1 → CR_N is cleared to NULL after context restore. Bit 6 is architecturally reserved (CR6 is always restored from the frame E-GT; clearing it via mask is undefined). Bare `RETURN` (mask = 0) is fully backward-compatible.

Semantics (see A.12, A.13): 1. M-window writeback fires; faults `INVALID_OP` if writeback fails. 2. Call stack is checked; empty stack or sentinel frame → `STACK_UNDERFLOW`. 3. All M-bits are reset across all CRs (`_resetAllMBits`). 4. Frame is popped: - **SZ = 1 (CALL frame):** caller's CRs and DRs are restored from the saved snapshot; returnPC comes from the frame word. - **SZ = 0 (LAMBDA frame):** PC is restored from `lambdaReturnPC` cache without a memory read (O(1) fast path). `lambdaActive` is cleared. 5. mask is applied: each bit-N=1 CRs is written to NULL in one parallel operation. 6. PC ← returnPC.

Flags: N — Z — C — V (no flag writes; caller's flags are restored from the saved snapshot)

Faults: | Fault | Condition | |——-|———–| | `STACK_UNDERFLOW` | Call stack empty, or RETURN through boot sentinel frame (NIA = 0x7FFF) | | `INVALID_OP` | M-window writeback failed before frame pop |

Example: Return, clearing CR0 and CR1 (mask = 0b000000000011 = 0x003).

```
RETURN AL, #0x003        ; mask bits 0 and 1 set → CR0, CR1 ← NULL
                          ; opcode=3, cond=14, fld_a=0, fld_b=0, imm=3
                          ; encoding: 0x1F000003
```

Bare return (no clearing):

```
RETURN AL                ; mask=0 → no CRs cleared
                          ; encoding: 0x1F000000
```

CHANGE — Thread Context Switch (opcode 4, 0x04)

```
Syntax:  CHANGE CR12, CR12, #idx
Encoding: op[4:0]=0x04 | cond[4] | CRd[4] | CRs[4] | idx[15]
```

Assembler restriction: The assembler only permits `CRd = CR12` and `CRs = CR12` (both operands reference the thread-stack register). All other privilege-zone registers (CR13–CR15) are blocked at the assembler level. The hardware is broader: CRd may be any of CR12–CR15 and drives different behaviour.

Semantics by destination register (hardware): - **CR12 or CR13** (system-wide): Load GT directly from c-list at `CRs[idx]`; no per-thread save/restore. - **CR14 or CR15** (per-thread): Full context switch — save the current thread's per-thread registers (CR0–CR11, CR14, CR15, DR0–DR15, STO, PC, flags) to the current thread lump; load the incoming thread's saved state from its lump. First activation of a thread starts at PC = 0.

Flags: N — Z — C — V (no flag writes)

Faults: | Fault | Condition | |——-|———–| | `PRIV_REG` | CRd < 12 (destination is not a privileged register) | | `PERM` | CRs lacks required permission for the target type | | `NULL_CAP` | Source c-list slot is NULL |

Example: Switch to thread at NS index 7 (both operands are CR12 per assembler convention).

```
CHANGE AL, CR12, CR12, #7   ; opcode=4, cond=14, fld_a=12, fld_b=12, imm=7
                              ; encoding: 0x27660007
```

SWITCH — PassKey-Gated Privileged Write (opcode 5, 0x05)

```
Syntax:  SWITCH CRn, #target
Encoding: op[4:0]=0x05 | cond[4] | 0[4] | CRn[4] | target[3] | 0[12]
```

`target` is a 3-bit value in `imm15[2:0]`. Valid target values: `5` (CR13, interrupt handler) and `7` (CR15, namespace root).

Hardware semantics (reference: `hardware/switch.py`): 1. **Target validity check**: `target` must be 5 or 7 — any other value → `FAULT(INVALID_OP)`. 2. **Source range check**: CRn index (from fld_b) must be in range 0–7. 3. **PassKey type check**: `CRn` must hold an Abstract GT (`gt_type == 0b11`). Any Inform, Outform, or NULL GT → `FAULT(INVALID_OP)`. 4. **Sentinel check**: `CRn.word1_location` must equal the hardware-reserved sentinel for the chosen target: `0xFFFFFFFE` for CR13, `0xFFFFFFFF` for CR15. Mismatch → `FAULT(INVALID_OP)`. 5. On all checks passing: `ChurchMLoad` writes the source capability into the target privileged register with `m_elevated = 1`; G-bit is reset. **The source CR is not cleared** — this is a one-way install, not a swap.

Simulator semantics (`_execSwitch`, line ~3836): performs an **atomic CR swap** (`CRn ↔ CR[target]`) with only a NULL check. No PassKey check, no target-validity check, no sentinel check, no mLoad. See D-11 in `HARDWARE-DEVIATIONS.md`.

Flags: N — Z — C — V (no flag writes)

Faults (hardware only): | Fault | Condition | |——-|———–| | `INVALID_OP` | target ≠ 5 and target ≠ 7 | | `INVALID_OP` | CRn not an Abstract GT | | `INVALID_OP` | CRn sentinel address does not match target | | `NULL_CAP` | CRn is NULL (simulator only) |

Example: Install a PassKey into CR15 (namespace root).

```
SWITCH AL, CR2, #7       ; CRn=CR2 (must be Abstract GT with sentinel 0xFFFFFFFF)
                          ; opcode=5, cond=14, fld_a=0, fld_b=2, imm=7
                          ; encoding: 0x2F010007
```

TPERM — Test/Attenuate Permission (opcode 6, 0x06)

```
Syntax:  TPERM CRd, preset
         TPERM CRd, presetB        (B-modifier variant, e.g. EB, RWB)
Encoding: op[4:0]=0x06 | cond[4] | CRd[4] | 0[4] | preset[5] | 0[10]
```

`preset` is a 5-bit value in `imm15[4:0]`. Bit 4 = B-modifier; bits [3:0] = preset code:

Code	Name	Required permissions
0x00	CLEAR	none (vacuously true)
0x01	R	R
0x02	RW	R, W
0x03	X	X
0x04	RX	R, X
0x05	RWX	R, W, X
0x06	L	L
0x07	S	S
0x08	E	E
0x09	LS	L, S
0x0A	W	W only (no R)
0x0B–0x0F	—	**Reserved**

B-modifier variants (bit 4 = 1): add 0x10 to any valid code (e.g. `EB = 0x18`). The B-modifier clears the Busy bit in CRd when the permission test passes.

Semantics (in order): 1. **NULL check**: if `CRd.word0 == 0` → Z=0, N=1, C=0, V=0, return immediately (no fault). 2. **Reserved preset check**: if preset code (bits[3:0]) is `0x0B–0x0F` → `FAULT(TPERM_RSV)`. Both hardware and simulator fault here (D-3 closed — simulator aligned with hardware, Task #873). 3. **Domain-purity check**: if the intersection of the preset's required permissions and the GT's held permissions would combine X with any of L/S/E → `FAULT(TPERM_RSV)`. 4. **Permission test**: `hasAll = required_permissions.every(p => GT.permissions[p])`. 5. **B-modifier**: if bit 4 is set and `hasAll == true` → clear bit 31 of `CRd.word0` (the Busy bit) in place. 6. Set flags: Z = hasAll, N = !hasAll, C = 0, V = 0.

Flags: N = !Z, Z = (all required permissions present), C = 0 (always), V = 0 (always)

Faults: | Fault | Condition | |——-|———–| | `TPERM_RSV` | Result permissions would combine X with L/S/E (domain-purity violation) |

Reserved preset codes (0x0B–0x0F) fault with `TPERM_RSV` in both hardware and simulator (D-3 closed — aligned as of Task #873).

Example: Test for E permission; clear B-bit if present.

```
TPERM AL, CR1, EB        ; preset EB = 0x18 (E=0x08 | B-modifier=0x10)
BRANCH EQ, call_ok       ; Z=1 → CR1 has E permission and B was cleared
                          ; opcode=6, cond=14, fld_a=1, fld_b=0, imm=0x18
                          ; encoding: 0x37080018
```

Test for R permission (existence of a readable data object):

```
TPERM AL, CR2, R         ; preset R = 0x01
                          ; opcode=6, cond=14, fld_a=2, fld_b=0, imm=1
                          ; encoding: 0x37100001
```

Mode 2 (attenuation sentinel): `imm15 = 0x7FFF` triggers permission attenuation rather than a preset test — see A.17 in §6.

LAMBDA — Execute Code Object in Caller Scope (opcode 7, 0x07)

```
Syntax:  LAMBDA CRn
Encoding: op[4:0]=0x07 | cond[4] | CRn[4] | 0[4] | 0[15]
```

Semantics: 1. mLoad validates X permission on CRn's GT. Faults if NULL or lacks X. 2. M-window cache is flushed (`_flushLambdaCache`). 3. A **SZ=0 frame** (1 word) is written to the thread stack: `memory[STO] = frameWord` (see A.13). No E-GT word is pushed. 4. `lambdaActive` ← 1. `lambdaReturnPC` ← PC + 1 (instruction after LAMBDA). 5. PC ← NIA derived from CRn's lump (code executes within the **caller's c-list context** — CR6 is not changed). CR14 is also unchanged; the lambda shares the caller's code fence.

LAMBDA does not switch c-lists. The callee inherits the caller's CR6. This is the defining difference from CALL.

Idempotent re-entry rule (D-9): `LAMBDA CR6` while `lambdaActive = 1` is permitted and idempotent (same return address overwrites the same register). `LAMBDA CRn` (n ≠ 6) while `lambdaActive = 1` is a FAULT.

Flags: N — Z — C — V (no flag writes)

Faults: | Fault | Condition | |——-|———–| | `NULL_CAP` | CRn is NULL | | `PERM` | CRn's GT lacks X permission | | `BOUNDS` | STO would write into heap/DR zone (stack overflow) | | `FAULT` | `LAMBDA CRn` (n ≠ 6) while `lambdaActive = 1` |

Example: Execute reduction code referenced by CR3.

```
LAMBDA AL, CR3           ; opcode=7, cond=14, fld_a=3, fld_b=0, imm=0
                          ; encoding: 0x3F180000
```

ELOADCALL — Fused Load + Call (opcode 8, 0x08)

```
Syntax:  ELOADCALL CRd, CRs, #row [, #method]
         ELOADCALL CRd, Name [, Method]    (shorthand — see A.6 and A.16)
Encoding: op[4:0]=0x08 | cond[4] | CRd[4] | CRs[4] | method[7] | row[8]
```

`imm15[14:8]` = method index (7 bits, 1-based; 0 = fast path). `imm15[7:0]` = c-list row (8 bits, 0–255).

Semantics: Atomic LOAD + CALL. No intermediate CR state is visible between the two phases. 1. mLoad validates CRs's c-list for the row slot (L permission check as per LOAD semantics). 2. GT at `CRs[row]` is loaded; Outform GTs are promoted Outform→Inform. 3. mLoad validates E permission on the loaded GT. 4. CRd ← loaded GT. 5. Hardware CALL phase executes: push 2-word frame, set CR6/CR14, jump to callee entry. Method index is decoded from `imm15[14:8]` using the same 1-based encoding as CALL.

Backward compatibility: programs that encode `imm15[14:8] = 0` get the fast-path (NIA = lump_base + 4) — identical to pre-split behaviour.

Flags: N — Z — C — V (no flag writes)

Faults: All faults from LOAD and CALL apply. Additionally: | Fault | Condition | |——-| ———–| | `BOUNDS` | `row` > 255 (assembler-enforced; hardware checks via c-list bounds) |

Example: Load abstraction from c-list row 2 of CR6 into CR0, then call it via fast path.

```
ELOADCALL AL, CR0, CR6, #2    ; row=2, method=0 (fast path)
                                ; opcode=8, cond=14, fld_a=0, fld_b=6, imm=0x0002
                                ; encoding: 0x43030002
```

XLOADLAMBDA — Fused Load + Lambda (opcode 9, 0x09)

```
Syntax:  XLOADLAMBDA CRd, CRs, #row
         XLOADLAMBDA CRd, Name       (shorthand — see A.16)
Encoding: op[4:0]=0x09 | cond[4] | CRd[4] | CRs[4] | row[15]
```

Semantics: Atomic LOAD + LAMBDA. Equivalent to `LOAD CRd, CRs, #row` followed by `LAMBDA CRd`, but the two operations are indivisible — no intermediate CR state is visible.

1. Load GT from `CRs[row]` into CRd (same as LOAD — L permission check, mLoad validation).

2. LAMBDA phase: mLoad validates X permission on the loaded GT; push SZ=0 frame; set `lambdaActive`, `lambdaReturnPC`; PC jumps to callee code within the **caller's c-list** (CR6 unchanged).

XLOADLAMBDA is used to invoke shared code objects (reducers, helpers) referenced in an abstraction's c-list without switching context.

Flags: N — Z — C — V (no flag writes)

Faults: All faults from LOAD and LAMBDA apply.

Example: Load and execute the reduction at c-list row 7.

```
XLOADLAMBDA AL, CR1, CR6, #7  ; opcode=9, cond=14, fld_a=1, fld_b=6, imm=7
                              ; encoding: 0x4B038007
```

DREAD — Data Read (opcode 10, 0x0A)

```
Syntax:  DREAD DRd, CRs, #offset
Encoding: op[4:0]=0x0A | cond[4] | DRd[4] | CRs[4] | offset[15]
```

Semantics: Read a 32-bit word from the data object referenced by CRs at word offset `offset` into `DRd`.

- **CRs requires R permission** (exception: when CRs = CR14, X permission is accepted in place of R — this allows reading embedded `WORD` constants from the current code lump via the code register).
- mLoad validates: permission, version, CRC seal, and that `lump_base + offset` lies within the GT's bounds.
- `DRd` is written via `_writeDR`, which enforces the DR0 hardwired-zero rule (A.1).
- **Abstract GT intercept**: if CRs holds an Abstract GT (type = 0b11), the read is routed to the Abstract Manager — MMIO device registers (LED, UART, TIMER, BUTTON) are accessed this way.

Flags: N — Z — C — V (no flag writes)

Faults: | Fault | Condition | |——-|———–| | `NULL_CAP` | CRs is NULL | | `PERM` | CRs lacks R permission (or X when CRs = CR14) | | `BOUNDS` | `lump_base + offset` is outside the GT's allowed range | | `SEAL` | NS entry CRC fails |

Example: Read word 3 from the data object in CR2 into DR1.

```
DREAD AL, DR1, CR2, #3   ; opcode=10, cond=14, fld_a=1, fld_b=2, imm=3
                         ; encoding: 0x57090003
```

Read inline data constant from current lump (CR14 path):

```
data_val:  WORD 0x1234
; ...
DREAD AL, DR0, CR14, #data_val   ; offset = label word index; X permission accepted
```

DWRITE — Data Write (opcode 11, 0x0B)

```
Syntax:  DWRITE DRd, CRs, #offset
Encoding: op[4:0]=0x0B | cond[4] | DRd[4] | CRs[4] | offset[15]
```

Semantics: Write the 32-bit value in `DRd` to the data object referenced by CRs at word offset `offset`.

- **CRs requires W permission**. mLoad validates: W permission, version, CRC, bounds.
- The value written is `DR[DRd] >>> 0` (unsigned 32-bit).
- **Thread sync**: if `DRd < 16`, the written value is also mirrored to the DR home slot in the thread lump (`thread_base + 1 + DRd`) so the thread image stays coherent across context switches.
- **Abstract GT intercept**: if CRs holds an Abstract GT (type = 0b11), the write is routed to the Abstract Manager (MMIO device write).
- **CR14 restriction**: CR14 is X-only; DWRITE to CR14 will fail the W-permission check.

Flags: N — Z — C — V (no flag writes)

Faults: | Fault | Condition | |——-|———–| | `NULL_CAP` | CRs is NULL | | `PERM` | CRs lacks W permission | | `BOUNDS` | `lump_base + offset` outside GT's allowed range | | `SEAL` | NS entry CRC fails | | `NULL_CAP` | CR12 (thread stack) is NULL when `DRd < 16` (cannot sync DR home) |

Example: Write DR0 to offset 0 of the data object in CR1.

```
DWRITE AL, DR0, CR1, #0   ; opcode=11, cond=14, fld_a=0, fld_b=1, imm=0
                          ; encoding: 0x5F008000
```

BFEXT — Bit-Field Extract (opcode 12, 0x0C)

```
Syntax:  BFEXT DRd, DRs, #pos, #width
Encoding: op[4:0]=0x0C | cond[4] | DRd[4] | DRs[4] | pos[5] | width[5] | 0[5]
```

`imm15[9:5]` = `pos` (5-bit, 0–31 — LSB of the extracted field). `imm15[4:0]` = `width` (5-bit, 1–32 — number of bits to extract).

Semantics: Extract `width` bits from `DRs` starting at bit `pos` (LSB) into `DRd`, zero-extended to 32 bits.

```
mask  = (1 << width) - 1
DRd   = (DRs >>> pos) & mask        ; bits[pos + width - 1 : pos]
```

Constraints: `width ≥ 1` and `pos + width ≤ 32` (faults `BOUNDS` otherwise).

Flags are written (see A.2): N = result[31], Z = (result == 0), C = 0, V = 0. Since a single-byte field can never have bit 31 set, N is effectively always 0 for widths < 32.

Flags: N = result[31], Z = (result == 0), C = 0, V = 0

Faults: | Fault | Condition | |——-|———–| | `BOUNDS` | `width == 0` or `pos + width > 32` |

Example: Extract byte 0 (bits [7:0]) of DR2 into DR1.

```
BFEXT AL, DR1, DR2, #0, #8  ; pos=0, width=8
                              ; imm = (0 << 5) | 8 = 0x0008
                              ; opcode=12, cond=14, fld_a=1, fld_b=2, imm=8
                              ; encoding: 0x67090008
```

BFINS — Bit-Field Insert (opcode 13, 0x0D)

```
Syntax:  BFINS DRd, DRs, #pos, #width
Encoding: op[4:0]=0x0D | cond[4] | DRd[4] | DRs[4] | pos[5] | width[5] | 0[5]
```

Same imm15 layout as BFEXT: `imm15[9:5]` = pos, `imm15[4:0]` = width.

Semantics: Insert the low `width` bits of `DRs` into `DRd` at bit position `pos`. Bits of DRs beyond `width` are discarded (see A.15).

```
mask    = ((1 << width) - 1) << pos
DRd_new = (DRd & ~mask) | ((DRs & ((1 << width) - 1)) << pos)
DRd     ← DRd_new
```

Flags reflect the **entire new value of DRd** (not just the inserted bits): N = DRd_new[31], Z = (DRd_new == 0), C = 0, V = 0.

Flags: N = DRd_new[31], Z = (DRd_new == 0), C = 0, V = 0

Faults: | Fault | Condition | |——-|———–| | `BOUNDS` | `width == 0` or `pos + width > 32` |

Example: Insert the low 4 bits of DR3 into DR2 at bit position 8.

```
BFINS AL, DR2, DR3, #8, #4  ; pos=8, width=4
                            ; imm = (8 << 5) | 4 = 0x0104
                            ; opcode=13, cond=14, fld_a=2, fld_b=3, imm=0x104
                            ; encoding: 0x6F118104
```

MCMP — Integer Compare (opcode 14, 0x0E)

```
Syntax:  MCMP DRd, DRs
Encoding: op[4:0]=0x0E | cond[4] | DRd[4] | DRs[4] | 0[15]
```

Semantics: Compute `DRd − DRs` and set condition flags. **No result register is written.** This is a pure flag-setting instruction.

```
temp = DRd - DRs
N = temp[31]
Z = (temp == 0)
C = (DRd_unsigned >= DRs_unsigned)   ; borrow flag (C=1 means no borrow)
V = signed overflow of DRd - DRs
```

The flag semantics match ISUB exactly (they share `_setSubFlags`). MCMP is the canonical compare instruction for use before a conditional BRANCH or conditional suffix.

Flags: N, Z, C, V — all written (subtraction semantics)

Faults: None

Example: Compare DR1 with DR2, then branch if equal.

```
MCMP AL, DR1, DR2        ; flags set from DR1 - DR2
BRANCH EQ, equal_path    ; branch if Z=1 (DR1 == DR2)
                          ; opcode=14, cond=14, fld_a=1, fld_b=2, imm=0
                          ; encoding: 0x77090000
```

IADD — Integer Add (opcode 15, 0x0F)

```
Syntax:  IADD DRd, DRs, DRt      (register form)
         IADD DRd, DRs, #k        (immediate form, 0 ≤ k ≤ 16383)
Encoding (register): op[4:0]=0x0F | cond[4] | DRd[4] | DRs[4] | 0[11] | DRt[4]
Encoding (immediate): op[4:0]=0x0F | cond[4] | DRd[4] | DRs[4] | 1[1] | k[14]
```

`imm15[14] = 1` → immediate mode; `imm15[13:0]` = k (0–16383, unsigned).
`imm15[14] = 0` → register mode; `imm15[3:0]` = DRt index.

Semantics: - Register form: `DRd = DRs + DRt` (unsigned 32-bit addition with flag capture) - Immediate form: `DRd = DRs + k`

All arithmetic is 33-bit wide internally; the low 32 bits are stored in DRd; bit 32 drives the C flag.

Idioms enabled by DR0 = 0 (A.1): - `IADD DRd, DR0, #k` → `DRd ← k` (load small immediate, 0–16383) - `IADD DRd, DR0, DRs` → `DRd ← DRs` (register copy)

Note on negative immediate: `#-1` is not directly encodable (immediate is unsigned 14 bits: 0–16383). To load −1, use `ISUB DRd, DR0, #1`.

Flags: N = result[31], Z = (result == 0), C = carry out (bit 32), V = signed overflow

Faults: None

Example: Add DR1 and the constant 10, store in DR2.

```
IADD AL, DR2, DR1, #10   ; immediate form: imm = 0x4000 | 10 = 0x400A
                           ; opcode=15, cond=14, fld_a=2, fld_b=1, imm=0x400A
                           ; encoding: 0x7F10C00A
```

Load immediate 42 into DR5:

```
IADD AL, DR5, DR0, #42   ; DR0 = 0 always; result = 0 + 42 = 42
                           ; encoding: 0x7F28000A... (imm=0x4000|42=0x402A)
```

ISUB — Integer Subtract (opcode 16, 0x10)

```
Syntax:  ISUB DRd, DRs, DRt      (register form)
         ISUB DRd, DRs, #k        (immediate form, 0 ≤ k ≤ 16383)
Encoding (register): op[4:0]=0x10 | cond[4] | DRd[4] | DRs[4] | 0[11] | DRt[4]
Encoding (immediate): op[4:0]=0x10 | cond[4] | DRd[4] | DRs[4] | 1[1] | k[14]
```

Immediate encoding identical to IADD: `imm15[14] = 1` → immediate, `imm15[13:0]` = k.

Semantics: - Register form: `DRd = DRs − DRt` - Immediate form: `DRd = DRs − k`

Carry flag semantics: `C = 1` when `DRs_unsigned ≥ DRt_unsigned` (no borrow). This matches ARM convention: C is the complement of the borrow bit.

Idiom: `ISUB DRd, DR0, #1` → `DRd ← −1` (all-ones, 0xFFFFFFFF), since DR0 = 0.

Flags: N = result[31], Z = (result == 0), C = (DRs ≥ DRt unsigned), V = signed overflow

Faults: None

Example: Subtract 1 from the loop counter in DR3.

```
ISUB AL, DR3, DR3, #1    ; DR3 ← DR3 - 1; sets Z=1 when DR3 reaches 0
                           ; opcode=16, cond=14, fld_a=3, fld_b=3, imm=0x4001
                           ; encoding: 0x831BC001
```

BRANCH — Conditional PC-Relative Branch (opcode 17, 0x11)

```
Syntax:  BRANCH[cond] #offset        (signed integer offset)
         BRANCH[cond] label           (assembler resolves to signed offset)
Encoding: op[4:0]=0x11 | cond[4] | 0[4] | 0[4] | soff[15]
```

`imm15` is interpreted as a **signed 15-bit offset** with bit 14 as the sign bit (see A.4):

```
soff = (imm & 0x4000) ? (imm | 0xFFFF8000) : imm    ; sign-extend bit 14
PC   ← PC + soff                                      ; word addressing
```

Range: −16384 to +16383 words. BRANCH always requires a condition suffix — use `BRANCH AL` for unconditional. Condition code is encoded in bits[26:23] as usual; the instruction word has no “bare BRANCH” form.

The condition is tested on entry to this instruction. If the condition is false, BRANCH is skipped like any other instruction (PC += 1, no flags changed, no side effects).

Key offsets (see A.4): - `#0` → infinite loop (target = current PC) - `#1` → fall-through (target = PC + 1 = next instruction; effectively a NOP branch) - `#-1` → step back to previous instruction

Flags: None written (flags are read, never modified)

Faults: | Fault | Condition | |——-|———–| | `BOUNDS` | `PC + soff` is outside the loaded memory image (simulator) or outside CR14 code fence (hardware) |

Example: Loop back to `loop_top` while DR1 ≠ 0.

```
loop_top:
    ISUB AL, DR1, DR1, #1
    BRANCH NE, loop_top   ; soff = loop_top_word − branch_word (negative)
                           ; cond=1 (NE), opcode=17
```

Unconditional forward jump by 3 words:

```
BRANCH AL, #3             ; opcode=17, cond=14, fld_a=0, fld_b=0, imm=3
                           ; encoding: 0x8B000003
```

SHL — Shift Left (opcode 18, 0x12)

```
Syntax:  SHL DRd, DRs, #shamt     (0 ≤ shamt ≤ 31)
Encoding: op[4:0]=0x12 | cond[4] | DRd[4] | DRs[4] | 0[10] | shamt[5]
```

`imm15[4:0]` = shift amount (5 bits, 0–31). `imm15[14:5]` must be 0.

Semantics: Logical shift left. Zero-fill from the right.

```
DRd    = (DRs << shamt) & 0xFFFFFFFF    ; 32-bit result
C      = (shamt > 0) ? DRs[32 - shamt] : 0
                                          ; last bit shifted out (source[32-shamt])
N      = DRd[31]
Z      = (DRd == 0)
V      = 0 always
```

Hardware confirmed: C is the last bit shifted out of DRs, gated on shamt > 0 (Task #857, D-12).

Flags: N = result[31], Z = (result == 0), C = last bit shifted out (0 if shamt = 0), V = 0

Faults: None

Example: Shift DR2 left by 4, store in DR1.

```
SHL AL, DR1, DR2, #4     ; opcode=18, cond=14, fld_a=1, fld_b=2, imm=4
                           ; encoding: 0x97090004
```

SHR — Shift Right (opcode 19, 0x13)

```
Syntax:  SHR DRd, DRs, #shamt              (LSR — logical, zero-fill)
         SHR DRd, DRs, #shamt, ASR         (ASR — arithmetic, sign-extend)
Encoding: op[4:0]=0x13 | cond[4] | DRd[4] | DRs[4] | 0[9] | mode[1] | shamt[5]
```

`imm15[4:0]` = shift amount (0–31). `imm15[5]` = mode select: 0 = LSR, 1 = ASR (see A.3).

Semantics:

```
; LSR (mode=0):
DRd = DRs >>> shamt            ; zero-fill from the left
C   = (shamt > 0) ? DRs[shamt - 1] : 0

; ASR (mode=1):
DRd = sign_extend(DRs) >> shamt   ; sign-fill from the left (bit 31 propagated)
DRd = DRd & 0xFFFFFFFF
C   = (shamt > 0) ? DRs[shamt - 1] : 0
N   = DRd[31]
Z   = (DRd == 0)
V   = 0 always
```

Hardware deviation (D-12, **CLOSED**): the hardware previously implemented LSR only (imm[5] ignored, C = 0). Task #857 fixed the hardware to match the simulator. Both now implement ASR and the C flag correctly.

Flags: N = result[31], Z = (result == 0), C = last bit shifted out (0 if shamt = 0), V = 0

Faults: None

Example: Logical right shift DR2 by 3.

```
SHR AL, DR1, DR2, #3     ; LSR, mode=0; imm = (0 << 5) | 3 = 0x03
                           ; opcode=19, cond=14, fld_a=1, fld_b=2, imm=3
                           ; encoding: 0x9F090003
```

Arithmetic right shift DR2 by 3 (sign-preserving):

```
SHR AL, DR1, DR2, #3, ASR   ; ASR, mode=1; imm = (1 << 5) | 3 = 0x23
                              ; encoding: 0x9F090023
```

8. Quick-Reference Appendix

8.1 Opcode Table

Dec	Hex	Mnemonic	Domain	fld_a	fld_b	imm15 summary	Flags written	Fau
0	0x00	LOAD	Church	CRd	CRs	c-list row (0–32767)	—	NUL BOU
1	0x01	SAVE	Church	CRd(S)	CRs(B)	c-list row (0–32767)	—	NUL BOU
2	0x02	CALL	Church	CRs	0	method index (0=fast, N+1=user N)	—	NUL PRIV STAC
3	0x03	RETURN	Church	0	0	mask[11:0] — CRs to NULL on return	—	STAC
4	0x04	CHANGE	Church	CRd	CRs	NS index (0–32767)	—	PRIV NUL
5	0x05	SWITCH	Church	0	CRn	target[2:0] (5=CR13, 7=CR15)	—	INVA
6	0x06	TPERM	Church	CRd	0	preset[4:0] (bit4=B-mod, [3:0]=code)	N=!Z Z C=0 V=0	TPER
7	0x07	LAMBDA	Church	CRn	0	0 (unused)	—	NUL BOU
8	0x08	ELOADCALL	Church	CRd	CRs	method[14:8] | row[7:0]	—	NUL BOU
9	0x09	XLOADLAMBDA	Church	CRd	CRs	c-list row (0–32767)	—	NUL BOU
10	0x0A	DREAD	Turing	DRd	CRs	word offset (0–32767)	—	NUL BOU
11	0x0B	DWRITE	Turing	DRd	CRs	word offset (0–32767)	—	NUL BOU
12	0x0C	BFEXT	Turing	DRd	DRs	pos[9:5] | width[4:0]	N Z C=0 V=0	BOU (pos
13	0x0D	BFINS	Turing	DRd	DRs	pos[9:5] | width[4:0]	N Z C=0 V=0	BOU (pos
14	0x0E	MCMP	Turing	DRd	DRs	0 (unused)	N Z C V	—
15	0x0F	IADD	Turing	DRd	DRs	bit14=0→DRt[3:0]; bit14=1→imm[13:0]	N Z C V	—

Dec	Hex	Mnemonic	Domain	fld_a	fld_b	imm15 summary	Flags written	Faul
16	0x10	ISUB	Turing	DRd	DRs	bit14=0→DRt[3:0]; bit14=1→imm[13:0]	N Z C V	—
17	0x11	BRANCH	Turing	0	0	signed offset (−16384..+16383)	—	BOU
18	0x12	SHL	Turing	DRd	DRs	shamt[4:0] (0–31)	N Z C V=0	—
19	0x13	SHR	Turing	DRd	DRs	mode[5] | shamt[4:0] (mode: 0=LSR, 1=ASR)	N Z C V=0	—

8.2 Assembler Encoding Formula

```
word = ((opcode & 0x1F) << 27)
     | ((cond   & 0x0F) << 23)
     | ((fld_a  & 0x0F) << 19)
     | ((fld_b  & 0x0F) << 15)
     | ((imm    & 0x7FFF))
```

8.3 Flag Conditions (for BRANCH and conditional suffixes)

Condition	Code	Flags tested	Use after
EQ	0	Z = 1	MCMP, ISUB, IADD
NE	1	Z = 0	MCMP, ISUB, IADD
LT	2	N ≠ V	MCMP signed
LE	3	Z=1 or N≠V	MCMP signed
GT	4	Z=0 and N=V	MCMP signed
GE	5	N = V	MCMP signed
CS/HS	6	C = 1	IADD carry
CC/LO	7	C = 0	ISUB borrow
MI	8	N = 1	IADD/ISUB
PL	9	N = 0	IADD/ISUB
VS	10	V = 1	IADD/ISUB
VC	11	V = 0	IADD/ISUB
HI	12	C=1 and Z=0	ISUB unsigned
LS	13	C=0 or Z=1	ISUB unsigned
AL	14	(always)	unconditional
NV	15	(never)	static no-op

8.4 Closed Design Decisions (Group E)

These questions are now resolved. Recorded here to prevent the decisions from being re-opened.

ID	Instruction	Decision
E-1	IADD / ISUB	Immediate is unsigned 0–16383. `#-1` cannot be encoded directly; use `ISUB DRd, DR0, #1`. **Closed.**
E-2	RETURN mask	The entire 12-bit mask field is not implemented in current hardware — all bits ignored. Bit 6 is reserved and must be zero (CR6 always re-derived unconditionally by cload). Assembler warns on any non-zero mask (Task #888). Use bare `RETURN`. **Closed — D-2 updated.**
E-3	DREAD CR14	X-in-place-of-R is CR14-specific only. No broader X→R substitution applies. **Closed.**
E-4	CHANGE operand restriction	Assembler convention only, not an ISA rule. Hardware `change.py` accepts any CR12–CR15 destination. The assembler restriction is a toolchain safety guard. **Closed.**

Deviation flags: SWITCH (D-11, closed — simulator now matches hardware). SHR/SHL carry+ASR (D-12, closed). TPERM reserved-preset fault (D-3, closed). TPERM Mode 2 (C.3, Task #874, closed). All deviations closed.