

# Church Machine — Instruction Set Reference

## Church Machine Instruction Set

v1.0 — 2026-04-29 CONFIDENTIAL

### Encoding Format

All instructions are 32 bits:

```
31   27 26 23 22 19 18 15 14           0
|opcode| cond|  dst|  src|   imm15   |
| 5 bit| 4 bit| 4 bit| 4 bit|  15 bits  |
```

- **opcode** (5 bits): Instruction identifier (0–19)
- **cond** (4 bits): ARM-style condition code for conditional execution
- **dst** (4 bits): Destination register (CR0–CR15 or DR0–DR15)
- **src** (4 bits): Source register (CR0–CR15 or DR0–DR15)
- **imm15** (15 bits): Immediate value (signed or unsigned depending on instruction)

### Condition Codes

All instructions support conditional execution:

Code	Mnemonic	Meaning	Flags
0000	EQ	Equal	Z=1
0001	NE	Not equal	Z=0
0010	CS/HS	Carry set / unsigned higher or same	C=1
0011	CC/LO	Carry clear / unsigned lower	C=0
0100	MI	Minus (negative)	N=1

Code	Mnemonic	Meaning	Flags
0101	PL	Plus (positive or zero)	N=0
0110	VS	Overflow set	V=1
0111	VC	Overflow clear	V=0
1000	HI	Unsigned higher	C=1 and Z=0
1001	LS	Unsigned lower or same	C=0 or Z=1
1010	GE	Signed greater or equal	N=V
1011	LT	Signed less than	N≠V
1100	GT	Signed greater than	Z=0 and N=V
1101	LE	Signed less or equal	Z=1 or N≠V
1110	AL	Always (unconditional)	—
1111	NV	Never (reserved)	—

Append the condition suffix to any mnemonic: CALLEQ, BRANCHNE, IADDGE, etc.

## Church Domain (10 Instructions)

These instructions manipulate capabilities (Golden Tokens). They operate on Context Registers (CR0–CR15).

### LOAD (opcode 0)

LOAD CRd, CRs, #row

Loads a Golden Token from the c-list pointed to by CRs at the given row (word offset) into CRd. Requires L (load) permission on the source GT. CR6-specific M-elevation: LOAD from CR6 skips L permission check because CALL already validated E.

**Security:** mLoad validates version, seal, bounds, L permission, and F-bit.

**Transparent Suspension (NULL GT):** If the c-list slot contains a NULL GT (0x00000000) and the slot has a pet name (from programCapabilities), the simulator first attempts **inline resolution** — searching nsLabels for a case-insensitive match. If a match is found and the NS entry is valid, the GT is written in place and execution continues normally. If no match is found, the thread is **suspended transparently** (`_lazySuspended = true`): no fault is raised, a lazyResolvePending event is emitted, and the IDE

has until the deadline to supply the GT via `sim.resolvePendingSlot(slot, nsIdx)`. Only if the deadline expires does the simulator escalate to a `NULL_CAP` fault via `sim.escalateLazyResolve(slot)`. If the slot is `NULL` and has **no pet name**, a `NULL_CAP` fault is raised immediately. See `docs/transparent-suspension.md` for the full protocol. Same transparent-suspension logic applies to `ELOADCALL` and `XLOADLAMBDA`.

## SAVE (opcode 1)

SAVE CRd, CRs, #row

Saves the GT in CRs (source GT, B=1 required) to the c-list pointed to by CRd (destination c-list pointer, S permission required) at the given row (word offset).

- **CRd** — destination c-list pointer. Must have S (save) permission.
- **CRs** — source GT to save. Must have B=1 (bindable). A GT without B=1 cannot be delegated.

**Security:** mSave validates version, seal, bounds, S permission on CRd, B-bit on CRs, and F-bit on the target slot.

## CALL (opcode 2)

CALL CRs [, #method\_index]

Enters the abstraction identified by the E-GT in CRs. CALL pushes **2 words** onto the call stack: - **Word 0** — the caller's E-GT (used by RETURN to revalidate and re-derive CR6/CR14) - **Word 1** — NIA (return offset) | packed machine indicators

No DRs and no other CRs are pushed. Callee inherits DR0-DR15, CR0-CR5, CR7-CR11, CR15 (caller context); CR12/CR13 are system-wide and unchanged. Sets CR6 = callee c-list (E-only), CR14 = callee code (X-only, privileged, M=1).

**Method index dispatch** (imm15 field):

imm15	NIA
0	lump_base + 4 (word 1 — single entry point, no method table)
n > 0	hardware reads memory[lump_base + n×4]; result is lump-base-relative word offset → NIA = lump_base +

<b>imm15</b>	<b>NIA</b>
	offset×4. Result = 0 → private method → FAULT.

**PC = 0 always FAILTs** — the lump header (word 0) is never a valid entry point.

## RETURN (opcode 3)

RETURN [mask]

**Encoding:** opcode[5]=00011 | cond[4] | 0[11] | mask[12]

mask is a 12-bit literal in bits [11:0]. The mask field is currently **not implemented** in hardware — all bits are ignored. Bit 6 is **reserved and must be 0**: CR6 is always re-derived unconditionally by cload regardless of the mask. All other mask bits (0-5, 7-11) are also unimplemented and must be 0. Use bare RETURN (mask=0). Encoding a non-zero mask will produce no error on current hardware but the clearing does not occur.

**Execution order:** 1. Pop 2-word frame 2. mLoad caller’s E-GT (Word 0) — version + MAC + G-bit reset; NS split re-derives CR6 and CR14 3. Restore PC from NIA and machine indicators from Word 1 4. Apply mask — all marked CRs written to NULL in **one parallel clock edge** (mask bits fan into CR register-file write enables; zero overhead regardless of how many bits are set)

Note: CR5 is a thread-bound capability installed by CHANGE from Zone<sup>4</sup> bounds when a thread is resumed; it is not saved or restored by CALL/RETURN.

**Why the mask is programmer-declared:** GTs are first-class values — a callee may legitimately return a GT in CR0. Only the programmer knows which CRs carry return values vs. internal working state. The mask is emitted as a compile-time literal by the CLOOMC compiler from a `clear:` annotation.

DRs and non-masked CRs retain whatever values the callee left. Shared between Church and Turing domains.

If the call stack is empty, or if RETURN unwinds through the boot sentinel frame (NIA = 0x7FFF), RETURN faults with `STACK_UNDERFLOW` — not a reboot, not a halt. The “warm reboot” description in older documents is incorrect.

## CHANGE (opcode 4)

CHANGE CRd, CRs, idx

Privileged register write. Reads the GT at index `idx` in CRs's c-list and installs it into privileged register CRd (CR12–CR15 only; PRIV\_REG fault if CRd < 12).

- **CR12/CR13** (system-wide): direct write; no context switch. CR12 = thread stack; CR13 = interrupt handler.
- **CR14/CR15** (per-thread): full context switch — saves CR0–CR11, CR14, CR15, DRs, and PC into the current thread lump; loads the target thread's saved state. First activation starts at PC=0.

## SWITCH (opcode 5)

SWITCH CRn, CRs

PassKey-gated privileged register write. Validates CRs as a system-capability Abstract GT with the hardware sentinel address for the target slot, then writes it into privileged CR12–CR15. Valid targets: CR13 (Tgt=101<sub>2</sub> = interrupt handler) and CR15 (Tgt=111<sub>2</sub> = namespace root). All other Tgt values produce INVALID\_OP.

## TPERM (opcode 6)

TPERM is the single-instruction GT health check. It shares one opcode across two modes — **health check** (flag-setting, no trap) and **permission restriction** (monotonic attenuation). The mode is determined by how the standard 32-bit encoding fields are used.

### Encoding

Both modes use the standard 32-bit format:

```

31   27 26 23 22 19 18 15 14           0
|00110 | cond | dst | src |   imm15   |
  op=6  4-bit 4-bit 4-bit 15 bits

```

### Mode 1 — Health Check (flag-setting)

TPERM CRs, #preset, offset

Field	Usage
dst (4 bits)	CRs — the context register to check
src (4 bits)	Preset code — selects the permission mask to test (see table below)
imm15 (15 bits)	

Field	Usage
	Offset — hardware checks base + offset ≤ limit

**What the hardware checks (all at once, one cycle):** 1. **Permissions** — does CRs have the requested permission bits? 2. **Valid** — does the GT pass version and MAC validation? 3. **Base + Limit** — is base + offset within the GT's region?

**Flags set:** - **Z = 1**: all checks passed (permissions present, valid, in bounds) - **Z = 0**: one or more checks failed

**Faulting:** TPERM faults with TPERM\_RSV if the preset code is reserved (codes 10-12 and 15, and their B-modifier variants 0x1A-0x1C and 0x1F). Codes 13 (FRAME) and 14 (EXACT) are valid non-permission presets — they perform hardware state queries rather than GT permission checks and never fault. For all valid presets, TPERM does not fault — if a permission check fails the Z flag says so and software decides what to do via conditional execution. The actual LOAD/SAVE/CALL instructions that follow enforce safety. TPERM is the “ask first” instruction.

### Conditional Execution: Zero-Cost Try-Catch

Because every Church Machine instruction carries a 4-bit ARM condition code, TPERM + conditional suffixes give you try-catch with no branches and no overhead on the happy path:

```

TPERM CR5, RW, offset      ; dst=CR5, src=preset 2 (RW),
imm15=offset
                                ; Z=1 if all pass, Z=0 if any fail

; --- happy path (EQ suffix = fires only when Z=1) ---
readEQ DR1, CR5, offset    ; skipped if Z=0
IADDEQ DR2, DR1, 1        ; skipped if Z=0
writeEQ CR5, offset, DR2  ; skipped if Z=0
returnEQ                  ; return to caller – skipped if Z=0

; --- catch path (NE suffix = fires only when Z=0) ---
; Execution reaches here ONLY if TPERM set Z=0.
; Every EQ instruction above was silently skipped by hardware.
MOVNE DR1, #0             ; set error code (0 = failed)
returnNE                  ; return error to caller

```

The happy path does not branch, does not check errors, does not even know failure is possible. Every instruction carries EQ and the hardware silently skips it if TPERM failed. The catch path runs on the same principle in reverse: NE instructions fire only when Z=0.

Both paths execute in sequence with no branching — the condition code on every instruction determines whether the hardware executes or skips it.

### Execution trace when TPERM passes (Z=1):

TPERM → Z=1  
readEQ → executes (Z=1 matches EQ)  
IADDEQ → executes  
writeEQ → executes  
returnEQ → executes – caller gets result, never reaches catch  
MOVNE → skipped (Z=1 does not match NE)  
returnNE → skipped

### Execution trace when TPERM fails (Z=0):

TPERM → Z=0  
readEQ → skipped (Z=0 does not match EQ)  
IADDEQ → skipped  
writeEQ → skipped  
returnEQ → skipped  
MOVNE → executes (Z=0 matches NE) – sets error code  
returnNE → executes – caller gets error

No branches. No jumps. The hardware skips or executes each instruction based on the condition suffix alone.

## Mode 2 — Permission Restriction (monotonic attenuation)

TPERM CRd, #preset

Field	Usage
dst (4 bits)	CRd — the context register to attenuate
src (4 bits)	Preset code — permission mask to AND with current permissions
imm15 (15 bits)	0x7FFF — all-ones sentinel; distinguishes restriction from health-check and allows health-check at offset 0

ANDs the preset mask with CRd's current permissions. Permissions can only be removed, never added (monotonic restriction). Sets Z=1 if resulting permissions are non-zero. The attenuation is local to the cached CR; the namespace slot is not updated until a SAVE commits it. Domain purity is enforced: Turing (R, W, X) and Church (L, S, E) permissions cannot be mixed.

## Preset Table

Code	Name	Bits Checked / Operation
0	CLEAR	None (always Z=1 if GT is non-NULL)
1	R	R
2	RW	R, W
3	X	X
4	RX	R, X
5	RWX	R, W, X
6	L	L
7	S	S
8	E	E
9	LS	L, S
10	RSV3	<b>FAULT</b> (TPERM_RSV) — unconditionally reserved
11	RSV4	<b>FAULT</b> (TPERM_RSV) — unconditionally reserved
12	RSV5	<b>FAULT</b> (TPERM_RSV) — unconditionally reserved
13	FRAME	Call-stack query: Z=1 if a real return frame exists (RETURN would not underflow). No GT is read; crDst is ignored.
14	EXACT	Bit-exact identity check: Z=1 iff CRd.word0 == CRs.word0 (all 32 bits). No fault on mismatch — sets Z=0.
15	RSV1	<b>FAULT</b> (TPERM_RSV) — unconditionally reserved

**E isolation** (GT creation rule): E permission must not be combined with L or S in a single GT. E grants entry into an abstraction as a black box; L or S grant c-list visibility. A holder of both could inspect or modify the abstraction's hidden implementation — violating encapsulation. This rule is enforced by **Mint** and domain-purity checks at GT creation time — it is not a TPERM preset rule. The former LE/SE/LSE preset names (codes 10-12) were a design-history artefact; these codes are unconditionally reserved at the hardware level regardless of permission combinations.

**FRAME (preset 13)**: Call-stack query. Tests whether a genuine caller return frame exists on the hardware call stack — i.e., whether RETURN would succeed without underflowing into the boot sentinel. Z=1 if a real frame is present; Z=0 if only the sentinel remains or the stack is empty. No GT is read (crDst is

ignored). No stack frame is pushed. This is the only safe way to pre-check RETURN: a CALL to query depth would itself consume a frame, making the reading stale before it arrives.

```
TPERM CR0, FRAME      ; Z=1 if caller frame exists
BRANCH.Z no_caller    ; skip RETURN if nothing to return to
RETURN
no_caller:
; root-level exit path
```

**EXACT (preset 14):** Credential identity check. Compares CRd.word0 (the 32-bit GT word) against CRs.word0 bit-for-bit. Sets Z=1 if they are identical, Z=0 otherwise. EXACT never faults — it is a pure comparison operator. Use it to verify that a credential has not been attenuated or substituted.

**B-modifier** (bit 4 of preset code): Adding B (0x10) clears the B-bit in the GT on a passing check — e.g. TPERM CR5, EB (code 0x18) checks E permission and, if it passes, clears the Bind bit. Named B-variants: RB, RWB, XB, RXB, RWXB, LB, SB, EB, LSB.

**NULL GT rule:** If the GT in CRd is NULL (word0=0), TPERM always sets Z=0 with no fault.

## Design Rationale

TPERM is the single gateway for inspecting and restricting GT metadata — permissions, validity, type, stack indicators, and bounds. Keeping all metadata operations in one opcode minimises opcode usage and silicon cost while providing a uniform interface. The two modes coexist in the same encoding: imm15 = 0x7FFF (all fifteen bits set) is the restriction sentinel — it performs monotonic attenuation with no bounds check. Any other imm15 value (0-32766) is a health-check offset, including 0 (test the base address). The all-ones sentinel was chosen over 0 precisely to allow offset=0 as a valid health-check target. The flag-setting (no-trap) design enables the conditional execution try-catch pattern that gives the happy path zero overhead.

## LAMBDA (opcode 7)

LAMBDA CRd

Applies the code object referenced by CRd in the current scope. Requires X (execute) permission. Does not switch c-lists — executes target code with caller's capabilities. Saves current PC as lambda return point. Machine-status fast path available for single-instruction targets.

## **ELOADCALL (opcode 8)**

ELOADCALL CRd, CRs, #row [, #method\_index]

Fused LOAD + CALL. Loads a GT from CRs's c-list at the given row (word offset), then immediately enters it. Atomic: no intermediate CR state is visible between the LOAD and the CALL.

imm15 split: bits[7:0] = c-list row (word offset into the c-list of CRsrc, 0-255); bits[14:8] = method index passed to the hardware CALL phase (same semantics as CALL imm15, 0-127).

Existing programs encode bits[14:8]=0 → method index 0 → NIA = lump\_base + 4 (backward compatible).

## **XLOADLAMBDA (opcode 9)**

XLOADLAMBDA CRd, CRs, #row

Fused LOAD + LAMBDA. Loads a GT from CRs's c-list at the given row (word offset), then immediately applies it. Equivalent to LOAD CRd, CRs, #row followed by LAMBDA CRd, but atomic.

---

## **Turing Domain (10 Instructions + shared RETURN)**

These instructions process data. They operate on Data Registers (DR0-DR15) and access DATA objects via R/W-permissioned GTs.

### **DREAD (opcode 10)**

DREAD DRd, CRs, #offset

Reads a 32-bit word from the DATA object referenced by CRs at the given offset into DRd. Requires R (read) permission.

### **DWRITE (opcode 11)**

DWRITE CRd, DRs, #offset

Writes DRs to the DATA object referenced by CRd at the given offset. Requires W (write) permission.

### **BFEXT (opcode 12)**

BFEXT DRd, DRs, #width, #lsb

Extracts a bitfield from DRs. Width and LSB position are encoded in the immediate field.

### **BFINS (opcode 13)**

BFINS DRd, DRs, #width, #lsb

Inserts a bitfield from DRs into DRd at the specified position.

### **MCMP (opcode 14)**

MCMP DRd, DRs

Compares DRd and DRs, setting condition flags (N, Z, C, V) without storing a result. Used before conditional instructions.

### **IADD (opcode 15)**

IADD DRd, DRs, #imm

Integer addition. DRd = DRs + imm. Sets condition flags.

### **ISUB (opcode 16)**

ISUB DRd, DRs, #imm

Integer subtraction. DRd = DRs - imm. Sets condition flags.

### **BRANCH (opcode 17)**

BRANCH #offset  
BRANCHEQ #offset  
BRANCHNE #offset

Branches to PC + offset (signed). Always conditional-compatible. Offset is relative to the current instruction.

### **SHL (opcode 18)**

SHL DRd, DRs, #amount

Logical shift left. DRd = DRs << amount.

### **SHR (opcode 19)**

SHR DRd, DRs, #amount

Logical shift right. DRd = DRs >> amount.

---

# Assembly Syntax

## Labels

```
loop:  
    IADD DR1, DR1, #1  
    BRANCHNE loop
```

## Comments

```
LOAD CR0, CR6, #4    ; Load Salvation from c-list slot 4  
CALL CR0             -- Enter Salvation (imm=0: fast-path, NIA =  
    lump word 1)
```

Both ; and -- introduce comments.

## Register Names

- Context registers: CR0 through CR15
- Data registers: DR0 through DR15

## Immediate Values

- Decimal: #42
- Hexadecimal: #0x2A

- **Negative: #-1**

## Three-Tier Fault Recovery (Task #1077)

When the simulator detects a fault, it attempts recovery in order before halting:

### Tier 1 — .catch Method on the Faulting Abstraction

Before escalating, the simulator checks whether the abstraction currently executing (derived from CR14) has a .catch entry in its method dispatch table. If present, .catch is invoked with a structured fault record (live machine state — not a snapshot). The handler may return { handled: true } to suppress the halt and advance PC past the faulting instruction.

### Structured fault record fields (Task #1077):

<b>Field</b>	<b>Type</b>	<b>Description</b>
type	string	Fault type mnemonic (e.g. 'PERM_R')
message	string	Human-readable fault description
faultCode	number   null	Numeric hardware fault code (from ChurchSimulator.FAULT_CODES)
faultingMnemonic	string   null	Instruction name that triggered the fault
involvedGT	number   null	Golden Token value, if relevant
pipelineStage	string   null	mLoad pipeline stage where fault occurred
faultingAbstractionSlot	number   null	NS slot of the executing lump
faultingAbstractionLabel	string   null	Human label of the executing lump
tier	1   2   3   null	Which recovery tier handled the fault
catchInvoked	boolean	Whether .catch was called (Tier 1)
irqInvoked	boolean	Whether Scheduler.IRQ was dispatched (Tier 2)
tier3Recovery	boolean	Whether Tier 3 CHANGE-to-CR13 was performed

## **Tier 2 — Scheduler.IRQ (Hidden ELOADCALL)**

If .catch is absent or returns handled: false, the fault escalates to Scheduler.IRQ (NS slot 8, method index 5). This is a hidden ELOADCALL — it does not consume a code word. The Scheduler.IRQ handler succeeds only if a faultRecoveryHandler is registered on the scheduler state (sim.\_schedulerState.faultRecoveryHandler). Default is null (safe: halts on fault, preserving pre-Task-#1077 behaviour for all existing programs).

## **Tier 3 — Double-Fault (CHANGE to CR13)**

If a fault occurs while the Scheduler.IRQ frame is already active (irqState.irqActive == true), a double-fault is declared. The simulator performs a CHANGE to the GT burned into CR13 by PP250 (simulated as \_returnToBoot()). The machine does not permanently halt; it resets to boot state.

## **Scheduler.pause and Timer Interrupts**

Scheduler.pause(duration) arms the hardware alarm (irqState.timerArmed = true, irqState.timerDeadline = stepCount + duration). On each step() call, before fetching the next instruction, the simulator checks whether the deadline has elapsed. When it fires, a hidden ELOADCALL Scheduler.IRQ is injected with reason='TIMER'. The IRQ is masked while irqActive is true to prevent nesting.

```
; Suspend calling thread for 100 simulation steps:  
IADD DR1, DR0, #100    ; duration in DR1  
Scheduler.pause()     ; arms timer; thread state → sleeping
```

---

Confidential — Kenneth Hamer-Hodges — April 2026