Machine Load (mLoad) — Release 1

Church-Turing Meta-Machine

Kenneth J Hamer-Hodges

May 2026

mLoad — The Single Trusted Gate

v1.0 — 2026-04-29 CONFIDENTIAL

mLoad is the hardware micro-operation that every Church instruction uses to read a Golden Token from memory and make it live in a Capability Register. No GT can enter a CR by any other path. mLoad is not an instruction — it is a sub-operation invoked internally by LOAD, CALL, RETURN, CHANGE, SWITCH, TPERM, LAMBDA, ELOADCALL, and XLOADLAMBDA.

Why One Gate

Every security property of the Church Machine is enforced in mLoad:

Property	Where enforced
Absent Outform objects are detected and trigger a lazy-load event	CHECK_NS stage (type check on GT)
Evicted warm-slot lumps (Inform GT, magic=0x00) trigger CODE_NOT_RESIDENT → Loader Mode 1 restore	CALL/LOAD pre-check (software, before mLoad fires)
Only L-perm holders can unlock the permitted access right(s) to the object or resource defined by a GT in their posession	CHECK_L stage
No capability can be read beyond its index range	CHECK_BOUNDS stage
No revoked GT can be re-used	CHECK_VERSION stage (gt_seq match)
No forged GT can ever load	CHECK_VERSION + seal stage (CRC-16/CCITT)
GC always knows which slots are live	RESET_GBIT to match GC cycle (0→1 or 1→0)
Thread lump always mirrors live CRs	UPDATE_THREAD stage

The eight properties in brief:

1. **Absent Outform Detection** — Lazy-load events triggered when accessing unloaded resources
2. **L-permission Control** — Only L-holders can unlock access rights to a GT
3. **Bounds Enforcement** — No GT can be read beyond its index range
4. **Revocation Prevention** — Stale tokens (version mismatch) are rejected
5. **Forgery Detection** — CRC-16/CCITT seal validates GT authenticity
6. **GC Liveness Tracking** — g-bit toggles during GC cycle to mark reachability
7. **Thread Shadow Sync** — Thread lump capability zone kept in sync with live CRs
8. **Outform Lazy-Load** — Absent remote objects trigger download and registration

Because all eight properties collapse to a single code path, the attack surface is a small, formally verifiable FSM rather than a distributed set of per-instruction permission checks. The hardware design for mLoad is arranged to perform these tests in parallel with the instruction and prevent the operation if any check fails, preventing any digital damage in advance and updating the MTBF calculations for the Abstraction. A first fault detection can be caught by simple instruction recovery code (if loaded) but a second fault by the same abstraction leads to system recovery that isolates error-prone modules (hardware or software).

Callers

Instruction	When mLoad fires	Purpose
LOAD	Every execution	Load one GT from c-list into a CR
CALL	Phase 2 (post E-perm check)	Populate both CR6 (c-list) and CR14 (code) from the E_GT, to the callee NS slots for CR6 and CR14 using the Lump header and Base location or for lazy-Load from the Memory Manager. The mLoad is the single point of Trusted Security Code (microcode implemented in open source hardware for public review on GitHub)
RETURN	Restoring caller context	Reload caller’s CR6 and CR14 from the return frame E-GT using the same rules for CALL
CHANGE	Load target thread + restore CRs	Switch thread context; restore per-thread capability set and Reload caller’s CR6 and CR14 from the return frame E-GT using the same rules for CALL in conjunction with the Thread Abstraction
SWITCH	Context switch to new Namespace Table in CR15	in conjunction with the Namespace Abstraction
TPERM	Permission transfer phase	Check the permission of a CR against the instruction conditions and index and set the capability indicators true or false for use by the next instruction
LAMBDA	Closure capture	Change the NIA (next instruction address) to the lambda code location and save the current location for a return. On CALL save the

Instruction	When mLoad fires	Purpose
		Return IA as a one word offset frame
ELOADCALL	Extended load-and-call	Load GT and immediately populate the method address for the callee entry choice
XLOADLAMBDA	Cross-domain lambda invocation	Load the lambda GT across a domain boundary into the target CR

CHANGE uses its own private `ChurchMLoad` instance; LOAD, CALL, RETURN, SWITCH, TPERM, LAMBDA, ELOADCALL, and XLOADLAMBDA share a single `u_shared_mload` arbiter in the core.

CHANGE Private mLoad — Security Proof

Why CHANGE Cannot Share the Arbiter

Every other mLoad caller fires the sub-operation exactly once per instruction. CHANGE fires it in a multi-step sequence:

1. **LOAD_THREAD** — one mLoad to validate the target thread GT and load it into CR8 (the incoming thread identity)
2. **RESTORE_CALL / RESTORE_NEXT loop** — up to fifteen sequential mLoad calls, one per CR0–CR14 that appears in the restore mask

If CHANGE shared `u_shared_mload` it would hold the arbiter locked for up to **sixteen consecutive mLoad pipelines**. Every other instruction that needs mLoad (LOAD, CALL, RETURN, SWITCH, …) would be stalled for the entire duration of the context switch. CHANGE therefore owns a private instance so its restoration loop runs independently without blocking the rest of the CPU.

Identity of the Private Instance

The private instance is not a fork, simplified copy, or alternate implementation. It is the same class:

```
# hardware/change.py, line 52
u_mload = ChurchMLoad()
m.submodules.u_mload = u_mload
```

`ChurchMLoad()` with no arguments inherits the global `ENABLE_SEAL_CHECK` flag (from `hw_types.py`), exactly as the shared instance does. The FSM state machine, all signal widths, all combinational checks, and all memory access patterns are **bit-for-bit identical by construction** — there is no divergent code path.

Static Wiring Choices

CHANGE hard-wires two inputs to the private mLoad (lines 104–106 of `change.py`):

Input	Value wired by CHANGE	Shared-arbiter value
`sub_direct`	`0` (always)	varies per caller
`sub_direct_gt`	`0` (always)	varies per caller
`sub_m_elevated`	`1` (always)	varies per caller

`sub_direct = 0` means CHANGE never bypasses the c-list read. Every GT loaded by CHANGE's private mLoad is read from memory through the full FETCH_GT stage — no GT is injected from a register without a memory round-trip.

`sub_m_elevated = 1` means the CHECK_L stage inside mLoad will pass regardless of whether the source CR holds an L-perm GT. The justification for this is structural and is covered in the next section.

M-Elevation Justification

M-elevation only bypasses the CHECK_L stage. All other stages run unchanged. The L-perm check is not dropped — it is **externalised** to the CHANGE FSM itself and performed before the first mLoad ever fires:

```
CHANGE FSM — before any mLoad call
  LATCH_CRN state (change.py line 163):
    if ~crn_has_l_perm:          # explicit L-perm check on cr_src
        fault → PERM_L
        → FAULT
    else:
        → SAVE_DR                # only reaches mLoad if L-perm confirmed
```

CHANGE verifies L-permission on `cr_src` (the c-list GT supplied by the instruction) before entering SAVE_DR. No path reaches LOAD_THREAD or the RESTORE loop unless the source CR has already passed the L-perm gate.

In the RESTORE loop (RESTORE_CALL), the source is CR8 — the thread GT just installed by LOAD_THREAD. That GT was validated by the immediately preceding mLoad pipeline (full FSM, seal check, version check, bounds check). It cannot be tampered with between the two calls because CR8 is a hardware register written only by mLoad's CR write port.

M-elevation is therefore sound: the L-perm check is not absent, it is done at the earliest structurally correct point (CHANGE FSM for cr_src; previous mLoad pipeline for CR8), and it cannot be bypassed by user code because `sub_m_elevated` is an internal hardware signal not reachable from the ISA.

Stage-by-Stage Comparison Table

The table below compares every FSM stage of the shared mLoad against CHANGE’s private instance for both call sites within CHANGE.

FSM Stage	Shared mLoad	CHANGE LOAD_THREAD	CHANGE RESTORE loop
FETCH_SRC	Reads `cr_src` from register file	Reads `cr_src` from register file	Reads CR8 from register file
CHECK_L	✓ Fails on no-L or null	Bypassed (M-elevated) — L-perm confirmed by CHANGE FSM upstream	Bypassed (M-elevated) — CR8 installed by validated mLoad
CHECK_BOUNDS	✓ `index < clist_limit`	✓ Runs unconditionally	✓ Runs unconditionally
FETCH_GT	✓ Memory read: c-list at `src.loc + index×4`	✓ Memory read: c-list at `src.loc + index×4`	✓ Memory read: c-list at CR8.loc + cr_index×4
CHECK_NS	✓ `slot_id < NS_size`	✓ Runs unconditionally	✓ Runs unconditionally
FETCH_LOC	✓ Memory read: NS base + `slot_id×16`	✓ Memory read: NS base + `slot_id×16`	✓ Memory read: NS base + `slot_id×16`
FETCH_W2	✓ Memory read: NS +4	✓ Memory read: NS +4	✓ Memory read: NS +4
FETCH_W3	✓ Memory read: NS +8 (if seal enabled)	✓ Memory read: NS +8 (if seal enabled)	✓ Memory read: NS +8 (if seal enabled)
CHECK_VERSION	✓ gt_seq match + CRC-16/CCITT	✓ Runs unconditionally	✓ Runs unconditionally
RESET_GBIT	✓ Writes g_bit=0 to NS +8	✓ Runs unconditionally	✓ Runs unconditionally
UPDATE_THREAD	✓ Shadows CR0–CR7 to thread lump	✓ Runs unconditionally	✓ Runs unconditionally
COMPLETE / FAULT	✓ Writes CR or asserts fault	✓ Identical	✓ Identical

Every security-critical stage — bounds, version, seal, g-bit, thread shadow — executes on every loop iteration without exception.

Formal Security Invariant

> **Invariant TCB-CHANGE-1 (Private mLoad Equivalence):** The `ChurchMLoad` instance inside `ChurchChange` is the same class as the shared arbiter. Its FSM is structurally identical. The sole deviation is the permanent assertion of `sub_m_elevated`, which skips only the CHECK_L stage. This is architecturally sound because: (a) the CHANGE FSM performs an equivalent L-perm check on the source CR before any mLoad fires; (b) subsequent loop iterations use a source CR (CR8) that was just validated by the immediately preceding mLoad pipeline; and (c) `sub_m_elevated` is a hardware-internal signal unreachable from user code. All remaining security properties — bounds enforcement, revocation prevention (gt_seq), forgery detection (CRC-16/CCITT seal), GC liveness (g-bit reset), and thread-lump shadow sync — are enforced on every single iteration of the CHANGE restoration loop without modification or bypass. CHANGE's private mLoad is therefore part of the minimal Trusted Security Code Base, not an exception to it.

Inputs

Signal	Width	Description
`sub_start`	1	Pulse to begin a mLoad
`sub_cr_src`	4	Source CR number — holds the c-list GT (CR6 normally)
`sub_cr_dst`	4	Destination CR number — where the loaded GT lands
`sub_index`	16	Index into the c-list pointed to by CR_src
`sub_direct`	1	**Direct mode**: skip c-list read; use `sub_direct_gt` as the GT
`sub_direct_gt`	32	Raw GT word used when `sub_direct = 1`
`sub_m_elevated`	1	**M-elevation**: skip L-perm check (used by CALL, CHANGE internally)
`cr15_namespace`	128	Live copy of CR15 — provides NS table base and size

Outputs

Signal	Width	Description
`sub_busy`	1	High while FSM is not IDLE
`sub_done`	1	Pulses high on COMPLETE — GT is in CR_dst
`sub_fault`	1	Pulses high on FAULT
`sub_fault_type`	4	`FaultType` code (see table below)
`cr_wr_addr`	4	CR number being written
`cr_wr_data`	128	Full 4-word capability being written
`cr_wr_en`	1	Write-enable for the register file
`gbit_reset_done`	1	Pulses high after the G-bit write-back (GC signal)
`thread_wr_en`	1	Write-enable for thread lump shadow copy
`thread_wr_idx`	4	CR index (0–7) being shadowed to the lump
`thread_wr_data`	32	GT word being written to thread lump shadow

Memory Reads

mLoad performs three or four sequential memory reads:

```
1. C-list GT word (if not direct mode)
   Address: CR_src.word1_location + (index × 4)
   Width:   32 bits — this is GT.word0 of the capability being loaded

2. NS entry word0 — Location (code base address)
   Address: CR15.word1_location + (slot_id × 16)
   Width:   32 bits

3. NS entry word1 — word1_w2 (limit_offset | gt_seq)
   Address: NS_entry_base + 4
   Width:   32 bits

4. NS entry word2 — word2_w3 (crc | g_bit)      [seal check enabled only]
   Address: NS_entry_base + 8
   Width:   32 bits
```

NS entry addresses use a **12-byte stride**: `slot_id * 12` (3 words × 4 bytes). The NS table base comes from `CR15.word1_location`.

FSM — State by State

```
IDLE
  │  sub_start asserted
  ▼
FETCH_SRC ─── direct_mode ──► CHECK_NS
  │                              (GT already in result_cap from sub_direct_gt)
  │ src_cap ← CR_src
  ▼
CHECK_L
  ├── src_is_null ──────────────► FAULT  (NULL_CAP)
  ├── ~has_l_perm & ~m_elevated ─► FAULT  (PERM_L)
  └── pass ─────────────────────► CHECK_BOUNDS

CHECK_BOUNDS
  ├── index ≥ clist_limit ──────► FAULT  (BOUNDS)
  └── pass ─────────────────────► FETCH_GT

FETCH_GT                      [memory read: c-list word at src.word1_location +
index×4]
  └── result_cap.word0_gt ← mem ─► CHECK_NS

CHECK_NS
  ├── slot_id ≥ NS_size ────────► FAULT  (BOUNDS)
  └── pass ─────────────────────► FETCH_LOC

FETCH_LOC                     [memory read: NS base + slot_id×16]
  └── result_cap.word1_location ← mem ─► FETCH_W2

FETCH_W2                      [memory read: NS base + slot_id×16 + 4]
  └── result_cap.word2_w2 ← mem
      ├── seal check disabled ──► UPDATE_THREAD
      └── seal check enabled ───► FETCH_W3

FETCH_W3                      [memory read: NS base + slot_id×16 + 8]
  └── result_cap.word3_w3 ← mem, ns_w3_saved ← mem ─► CHECK_VERSION

CHECK_VERSION
  ├── gt_seq mismatch ──────────► FAULT  (VERSION)
  ├── CRC-16/CCITT mismatch ────► FAULT  (SEAL)
  └── pass ─────────────────────► RESET_GBIT

RESET_GBIT                    [memory write: NS +8 with g_bit cleared to 0]
  │  gbit_reset_done asserted (GC integration pulse)
  └──────────────────────────────► UPDATE_THREAD

UPDATE_THREAD
  │  if cr_dst ≤ 7: thread_wr_en, write GT word to thread lump shadow
  └──────────────────────────────► COMPLETE

COMPLETE
  │  cr_wr_en asserted — full 128-bit result_cap written to CR_dst
  └──────────────────────────────► IDLE

FAULT
  └──────────────────────────────► IDLE
```

Capability Written to CR_dst

When mLoad completes successfully, a full 128-bit `CAP_REG` is assembled from three separate memory reads and written to the destination CR:

Word	Source	Content
`word0_gt` (32 b)	C-list entry	GT word: slot_id, gt_seq, gt_type, perms, b_flag
`word1_location` (32 b)	NS entry +0	Code / data base address
`word2_w2` (32 b)	NS entry +4	`limit_offset[20:0]` · `gt_seq[6:0]` · spare
`word3_w3` (32 b)	NS entry +8	`crc[15:0]` · `g_bit` · spare (seal-check mode only)

`word3_w3` is only populated when `ENABLE_SEAL_CHECK = True`; it is zero otherwise.

Seal Check — CRC-16/CCITT

When `ENABLE_SEAL_CHECK = True` (the default), mLoad recomputes a 16-bit CRC in hardware and compares it to `word2_w3.crc`:

Input bit-stream (89 bits, MSB-first):

```
GT[24:0]         — 25 bits  (slot_id, gt_seq, gt_type; perms and b_flag excluded)
word1_location   — 32 bits  (code base address)
word1_w2         — 32 bits  (limit_offset | gt_seq)
──────────────────────────────
Total            89 bits
```

Algorithm: CRC-16/CCITT — polynomial `0x1021`, init `0xFFFF`, no input reflection, no output XOR.

Verification:

```
computed_crc == NS_entry.word2_w3[15:0]
AND
GT.gt_seq == NS_entry.word1_w2[27:21]   (gt_seq field)
```

A mismatch in either check routes to FAULT with `FaultType.VERSION` or `FaultType.SEAL` respectively. The two checks happen in the single `CHECK_VERSION` state — version first, then seal, so a stale GT always reports VERSION before the seal is even evaluated.

The 16-bit seal is computed by the IDE at upload time and stored in `word2_w3`. The hardware re-derives it on every access; there is no way to cache a pass result.

G-bit and GC Integration

At `RESET_GBIT`, mLoad writes back NS entry `+8` with the `g_bit` field forced to zero. The `gbit_reset_done` output pulses high for one cycle, telling the GC unit that this NS slot was just accessed.

GC's mark phase sets `g_bit = 1` on every NS entry in the scan range. Every successful mLoad clears `g_bit = 0` on the accessed entry. GC's sweep phase collects entries where `g_bit` is still `1` — i.e. entries that were not accessed since the last mark pass.

There is no separate reachability graph. mLoad is the reachability signal.

Thread Lump Shadow Update

At `UPDATE_THREAD`, if `cr_dst ≤ 7`, mLoad writes the 32-bit GT word (`result_cap.word0_gt`) to the thread lump at offset `cr_dst`.

This keeps the thread lump's Capabilities zone (words +244…+255) in sync with the live register file for CR0–CR7. When CHANGE suspends a thread, the GT zone is already current — no separate save sweep is needed for those registers.

CR8–CR15 (system and privileged registers) are not shadowed here; they are handled separately by the CHANGE instruction itself.

L Permission and M Elevation

Normally `sub_cr_src` must hold an L-perm GT — only a c-list GT (CR6) is expected to have L. If `src_gt.perms[PERM_L] = 0` the check fails with `PERM_L`.

M-elevation (`sub_m_elevated = 1`) bypasses this check. It is asserted only by microcode (CALL, CHANGE) when the instruction itself supplies the source GT through a known-trusted path. User code can never set M; it is invisible to the ISA.

Direct Mode

When `sub_direct = 1`, the c-list read and CHECK_L / CHECK_BOUNDS stages are skipped entirely. The GT word in `sub_direct_gt` is used directly as `result_cap.word0_gt`, and the FSM jumps straight to `CHECK_NS`. This is used by CALL when it already holds the validated callee GT from the E-perm check and wants to load the NS entry without re-reading the c-list.

Fault Types

Code	Name	Cause
`0x7`	`NULL_CAP`	`CR_src` holds a null GT (gt_type = 00)
`0x4`	`PERM_L`	`CR_src` lacks L permission and M is not elevated
`0x8`	`BOUNDS`	`index ≥ c-list limit` **or** `slot_id ≥ NS table size`
`0x9`	`VERSION`	`GT.gt_seq ≠ NS_entry.word1_w2.gt_seq`
`0xA`	`SEAL`	CRC-16/CCITT recomputed ≠ stored seal

On any FAULT the destination CR is not modified and `sub_fault_type` holds the first failing check. The FSM returns to IDLE in one cycle.

Relevant Files

File	Role
`hardware/mload.py`	Amaranth FSM — the complete mLoad implementation
`hardware/hw_types.py`	`FaultType`, `PERM_*`, `GT_TYPE_*`, `ENABLE_SEAL_CHECK`
`hardware/layouts.py`	`GT_LAYOUT`, `CAP_REG_LAYOUT`, `WORD2_LAYOUT`, `WORD3_LAYOUT`
`hardware/boot_rom.py`	`crc16_ccitt()` — the Python reference implementation of the seal
`hardware/core.py`	`u_shared_mload` arbiter; CHANGE's private mLoad instance
`hardware/change.py`	Uses its own `ChurchMLoad` submodule for thread-switch restores
`docs/figures/mload-validation-pipeline.html`	Block-diagram figure of the pipeline

Confidential — Kenneth Hamer-Hodges — April 2026