CTMM ISA Encoding — Release 1

Church-Turing Meta-Machine

Kenneth J Hamer-Hodges

May 2026

Church Machine ISA Encoding Reference

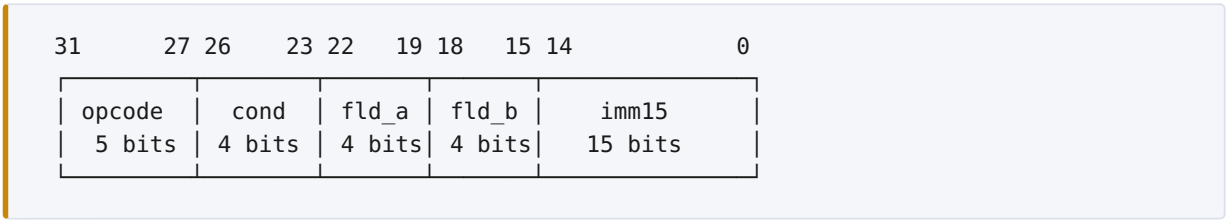
v1.0 — 2026-04-29 CONFIDENTIAL

All values verified against `simulator/assembler.js` and `simulator/simulator.js`. This document is the complete specification needed to implement an encoder with no guesswork.

---

1. Word Format

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



Encoding expression:

```
word = ((opcode & 0x1F) << 27)
     | ((cond   & 0x0F) << 23)
     | ((fld_a  & 0x0F) << 19)
     | ((fld_b  & 0x0F) << 15)
     | ( imm15  & 0x7FFF)
```

There is **no mode-select bit** separating “register” from “immediate” in the last field. Semantics are fixed per opcode — determined entirely by the opcode number, not by any flag bit in the word.

Special case: `HALT` and `NOP` both assemble to the all-zero word `0x00000000`.

2. Opcode Table — bits [31:27]

Dec	Hex	Mnemonic
0	0x00	LOAD
1	0x01	SAVE
2	0x02	CALL
3	0x03	RETURN
4	0x04	CHANGE
5	0x05	SWITCH
6	0x06	TPERM
7	0x07	LAMBDA
8	0x08	ELOADCALL
9	0x09	XLOADLAMBDA
10	0x0A	DREAD
11	0x0B	DWRITE
12	0x0C	BFEXT
13	0x0D	BFINS
14	0x0E	MCMP
15	0x0F	IADD
16	0x10	ISUB
17	0x11	BRANCH
18	0x12	SHL
19	0x13	SHR

Opcodes 20–31 are undefined (disassembler emits `???`).

3. Condition Code Table — bits [26:23]

The condition suffix is appended directly to the mnemonic. The default (always execute) is `AL = 14`; when AL is used the suffix is omitted entirely.

Dec	Suffix	Meaning
0	EQ	Equal / Z set
1	NE	Not equal / Z clear
2	CS (= HS)	Carry set / Unsigned higher or same
3	CC (= LO)	Carry clear / Unsigned lower
4	MI	Minus / negative
5	PL	Plus / positive or zero
6	VS	Overflow set
7	VC	Overflow clear
8	HI	Unsigned higher
9	LS	Unsigned lower or same
10	GE	Signed greater or equal
11	LT	Signed less than
12	GT	Signed greater than
13	LE	Signed less or equal
14	(none)	AL — always (default)
15	NV	Never (encodes, never executes)

Aliases accepted by the assembler: `HS` → 2, `LO` → 3.

Example: `IADDLT DR4, DR1, DR2` encodes opcode=0x0F, cond=11.

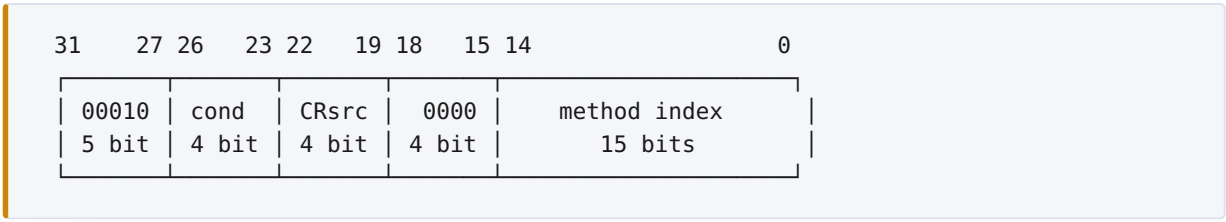
---

4. Per-Opcode Field Usage

Capability register (CR) instructions

Op	Mnemonic	fld_a	fld_b	imm15
0	LOAD	CR dst	CR base	unsigned word offset into c-list (0–32767)
1	SAVE	CR dst (c-list, S perm)	CR src (GT, B=1)	unsigned word offset into c-list (0–32767)
2	CALL	CR src	0	method index (15 bits; see note below)
3	RETURN	0	0	12-bit mask in bits [11:0] — bit N=1 clears CR_N to NULL after frame pop; bit 6 reserved (must be 0); mask=0 → no scrub
4	CHANGE	CR dst	0	NS slot index (unsigned)
5	SWITCH	0	CR src	new permission — lower 3 bits (0–7)
6	TPERM	CR dst	0	5-bit preset code (see §6)
7	LAMBDA	CR dst	0	0
8	ELOADCALL	CR dst	CR src	bits[14:8] = method index (7 bits, 0–127); bits[7:0] = c-list row (8 bits, 0–255)
9	XLOADLAMBDA	CR dst	CR src	unsigned word offset into c-list (0–32767)

CALL — method index encoding

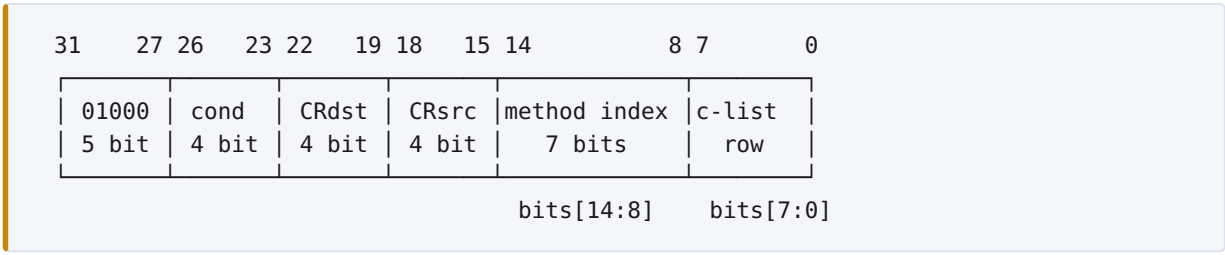


method index	Behaviour
0	Single entry point — no method table. NIA = lump_base + 4 (word 1).
n > 0	Hardware reads `memory[lump_base + n×4]` (lump-base-relative word offset). NIA = lump_base + table_entry×4. Table entry = 0 → private method → FAULT.

PC=0 (lump header) is always a FAULT — the lump header word is never an executable instruction.

Backward compatibility: all programs assembled before this revision encode imm15=0 (method index 0) → NIA = lump_base + 4. Behaviour is unchanged for single-entry-point abstractions.

ELOADCALL — split imm15



`c-list row` = word offset into the c-list pointed to by CRsrc (0–255 entries, matches max `cc` field of lump header). `method index` = passed to the CALL phase after the GT is loaded (same semantics as CALL imm15). Existing programs have bits[14:8]=0 → method index 0 → backward compatible.

Data register (DR) / mixed instructions

Op	Mnemonic	fld_a	fld_b	imm15
10	DREAD	DR dst	CR base	unsigned offset
11	DWRITE	DR src	CR base	unsigned offset
12	BFEXT	DR dst	CR base	`(pos & 0x1F) << 5 | (width & 0x1F)` — bits [9:5]=pos, [4:0]=width
13	BFINS	DR src	CR base	`(pos & 0x1F) << 5 | (width & 0x1F)` — bits [9:5]=pos, [4:0]=width
14	MCMP	DR op1	DR op2	0 — result goes to condition flags only, no writeback
15	IADD	DR dst	DR src1	reg: `imm15[3:0]`=DR src2; imm: `0x4000|(val&0x3FFF)` (bit 14 selects mode)
16	ISUB	DR dst	DR src1	reg: `imm15[3:0]`=DR src2; imm: `0x4000|(val&0x3FFF)` (bit 14 selects mode)
17	BRANCH	0	0	signed 15-bit PC-relative offset; bit 14 is the sign bit
18	SHL	DR dst	DR src	`imm15[4:0]` = shift amount (0–31)
19	SHR	DR dst	DR src	`imm15[5]`=1 for ASR / 0 for LSR; `imm15[4:0]`=shift amount

Instruction-specific notes

CALL — Two execution modes selected by `imm15` (method index):

- **index = 0** (no method table): NIA = lump_base + 4 (word 1, single entry point). Backward-compatible with all existing programs.
- **index n > 0** (hardware method-table dispatch): hardware reads the 32-bit word at byte address `lump_base + n×4`; the word holds a lump-base-relative WORD offset baked in by the compiler (private method = 0 → FAULT; first public method index = methodTableSize + 1). NIA = lump_base + entry×4.
- **PC=0 always FAULTs** — the lump header (word 0) is never a valid entry point.
- **ELOADCALL backward compat**: existing programs have `imm15 = small c-list row` (bits[14:8] = 0 → method index 0 → single entry point). Fully backward-compatible.

BRANCH — `imm15` is a **signed PC-relative offset**. Execution: `target_PC = current_PC + sign_extend(imm15)`. Bit 14 of imm15 is the sign bit. Sign-extend: `soff = (imm & 0x4000) ? (imm | 0xFFFF8000) : imm`.

The assembler resolves label names to PC-relative offsets via `label_word - current_word` (confirmed from `assembler.js` line: `imm = this.labels[branchToken] - addr`). Numeric literals (e.g. `BRANCH -5`) are also encoded as-is. Both forms produce correct relative offsets.

MCMP — No destination field. fld_a and fld_b both hold DR operand numbers. The comparison result is written only to the condition flags register. imm15 is always zero.

IADD / ISUB — two encoding forms for the third operand:

The assembler distinguishes register vs. immediate via the third operand's syntax:

```
IADD DR4, DR1, DR2      ; register form  — third operand is a DR number
IADD DR4, DR1, #42      ; immediate form — third operand is a signed literal
```

- **Register form**: `imm15 = DR_src2_index` (4 bits, 0–15 in imm15[3:0]; bits [14:4] = 0).
- **Immediate form**: bit 14 of imm15 is set to 1 as a mode flag; bits [13:0] hold the 14-bit signed immediate (`imm15 = 0x4000 | (value & 0x3FFF)`).

Decoders must check `imm15 & 0x4000` to distinguish: 1 → immediate form, 0 → register form.

DR0 is hardwired to 0 — the simulator writes 0 to DR0 after every instruction. `IADD DRn, DR0, DRk` computes `DRn = 0 + DRk = DRk` (a register copy). `IADD DRn, DR0, #42` loads the literal 42 directly into DRn via immediate form.

BFEXT / BFINS — imm packing: `imm = (pos << 5) | width`, occupying bits [9:0] of imm15. Bits [14:10] are unused (zero).

SHR ASR — arithmetic (sign-extending) right shift: set `imm15[5] = 1`.
Logical right shift: `imm15[5] = 0`. Shift amount is `imm15[4:0]`.

---

5. Register Numbering

Both register files use the same 4-bit encoding 0–15 in fld_a / fld_b:

Register	Assembly syntax	Field value	Notes
CR0–CR15	`CR0` … `CR15`	0–15 in fld_a or fld_b	
DR0	`DR0`	0 in fld_a, fld_b, or imm15[3:0]	**Hardwired zero** — the simulator writes 0 to DR0 after every instruction. It cannot hold a value across instructions.
DR1–DR15	`DR1` … `DR15`	1–15 in fld_a, fld_b, or imm15[3:0]	General-purpose

CRs and DRs are **separate register files**. The opcode determines which file is accessed. The same 4-bit encoding is reused in both fields independently; there is no shared namespace at the hardware level.

The DR0 = 0 invariant is enforced unconditionally by the simulator (`this.dr[0] = 0` after every step), making it usable as a zero source for address arithmetic and register copies.

---

6. TPERM Preset Codes — bits [4:0] of imm15 (opcode 6 only)

Bit 4 (`imm & 0x10`) is the **Bound (B) flag**. Bits [3:0] select the base permission preset. The assembled word uses the numeric code directly.

Code	Name	Code	Name
0x00	CLEAR	0x10	B
0x01	R	0x11	RB
0x02	RW	0x12	RWB
0x03	X	0x13	XB
0x04	RX	0x14	RXB
0x05	RWX	0x15	RWXB
0x06	L	0x16	LB
0x07	S	0x17	SB
0x08	E	0x18	EB
0x09	LS	0x19	LSB
0x0A	(rsv)	0x1A	(rsv)
0x0B	(rsv)	0x1B	(rsv)
0x0C	(rsv)	0x1C	(rsv)
0x0D	(rsv)	0x1D	(rsv)
0x0E	(rsv)	0x1E	(rsv)
0x0F	(rsv)	0x1F	(rsv)

Permission bit key: **R**=Read, **W**=Write, **X**=Execute, **L**=Load (load capability), **S**=Save (store capability), **E**=Enter (enter enclave / abstraction), **B**=Bound (capability is bounds-checked).

If a numeric value is given instead of a named preset, the assembler uses the lower 5 bits of that number directly (`imm & 0x1F`).

Quick-Reference: Encoding Pseudocode

```
def encode(opcode, cond=14, fld_a=0, fld_b=0, imm15=0):
    return (
        ((opcode & 0x1F) << 27) |
        ((cond   & 0x0F) << 23) |
        ((fld_a  & 0x0F) << 19) |
        ((fld_b  & 0x0F) << 15) |
        ( imm15  & 0x7FFF)
    )

def branch_offset(from_pc, to_pc):
    """Compute the signed PC-relative imm15 for a BRANCH instruction."""
    offset = to_pc - from_pc
    return offset & 0x7FFF   # truncate to 15 bits (sign preserved in bit 14)

# Examples
IADD_DR1_DR0_DR2 = encode(opcode=15, fld_a=1, fld_b=0, imm15=2)
# DR1 = DR0 + DR2  (DR0 is always 0, so this copies DR2 into DR1)

# BRANCH: always PC-relative. Encode as offset from the branch instruction's PC.
# branch at PC=10, target at PC=3:  offset = 3 - 10 = -7
BRANCH_back_7    = encode(opcode=17, imm15=branch_offset(from_pc=10, to_pc=3))
# branch at PC=5, target at PC=12: offset = 12 - 5 = +7
BRANCH_forward_7 = encode(opcode=17, imm15=branch_offset(from_pc=5, to_pc=12))
# Conditional branch: BRANCHLT (branch if less-than)
BRANCHLT_back_2  = encode(opcode=17, cond=11, imm15=branch_offset(10, 8))

TPERM_CR0_RWX   = encode(opcode=6,  fld_a=0, imm15=0x05)
MCMP_DR1_DR2    = encode(opcode=14, fld_a=1, fld_b=2)
HALT_or_NOP     = 0x00000000
```

Confidential — Kenneth Hamer-Hodges — April 2026