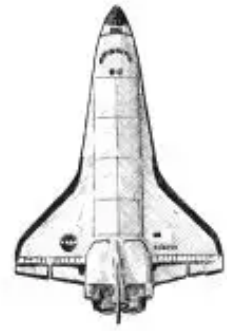
HALMAT

Instruction Set Reference

The Intermediate Language of the HAL/S-FC Compiler
Reconstructed from 630 XPL Source Files, Release 32V0



Zane Hambly

February 2026

Built with techniques from CBT Tape.
Verified against regression binary output.
180 opcodes across 9 classes.

HALMAT Instruction Set Reference

Foreword

This document is the first comprehensive reference for HALMAT, the intermediate language of the HAL/S-FC compiler. The compiler that built the flight software for the Space Shuttle.

HALMAT was never publicly documented. Not by IBM, not by Intermetrics, not by NASA. It existed only as implicit knowledge encoded in the compiler source, until Ron Burkey began the painstaking work of deciphering it.

Burkey's HALMAT documentation in the Virtual AGC project is the only prior reference of any kind. His ~850-line document established the word format, the operand qualifier system, the block structure, and thorough coverage of Class 0 (control and flow) and Class 8 (initialisation). He also wrote a Python decoder that could disassemble HALMAT binary output. Without that foundation, and without the extraordinary effort of the

Virtual AGC project in preserving and running the compiler in the first place, none of the work in this document would have been possible.

What Burkey noted as “TODO” were Classes 1 through 7: the arithmetic, comparison, and type-conversion operations that make up the computational core of the language. This document completes that picture.

I recovered the missing classes by going back to the compiler itself. The HAL/S compiler is written in XPL/I, an extended dialect of XPL (itself a strict subset of PL/I F). There are no modern parsers for this language. I built one from scratch, informed by vintage compiler construction techniques from the CBT Tape archive (the XPL BNF grammar, the XCOM bootstrap compiler, the PL/360 table-driven parser) and used it to parse every XPL source file in the HAL/S-FC compiler: all 630 files across all seven passes, at 100%.

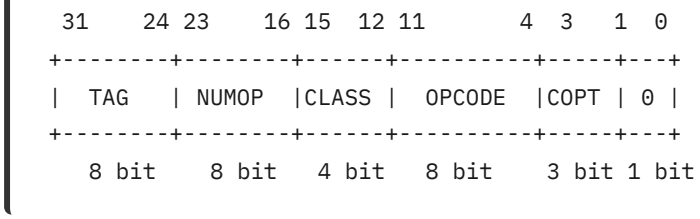
I then walked those parse trees to extract every HALMAT opcode definition, cross-reference every opcode against every compiler pass, and verify the result against actual binary output from the regression test suite, including decoding IBM System/360 hexadecimal floating-point literals and resolving symbol table entries from COMMON memory dumps.

What follows is the result: 180 opcodes across 9 classes, with word format specifications, operand encodings, emission context from the compiler source, and worked examples from disassembled binary output. Burkey’s existing documentation is incorporated throughout, merged with the extracted evidence into a unified reference.

Word Format

HALMAT operates on 32-bit words. The least significant bit distinguishes operator words from operand words.

Operator Word (bit 0 = 0)



- **TAG** -Operator-specific tag. Commonly used for loop type, error severity, or scope nesting.
- **NUMOP** -Number of operand words following this operator.
- **CLASS:OPCODE** -12-bit operation code. CLASS selects the instruction family (0–8); OPCODE selects the specific operation within that class.
- **COPT** -Compiler optimisation tag. Used by Phase 1.5 (the machine-independent optimiser) to mark common subexpressions.

Constructed in PASS1 by `HALMAT_POP(POPCODE, PIP#, COPT, TAG)`:

```
CURRENT_ATOM = SHL(TAG,24) | SHL(PIP#,16) | SHL(POPCODE&"FFF",4)
             | SHL(COPT&"7",1);
```

Operand Word (bit 0 = 1)

```
 31         16 15     8  7    4  3   1  0
+-------------+--------+------+-----+---+
|    DATA     |  TAG1  | QUAL | TAG2| 1 |
+-------------+--------+------+-----+---+
    16 bit      8 bit   4 bit  3 bit 1 bit
```

- **DATA** -16-bit operand value. Interpretation depends on QUAL.
- **TAG1** -Operand-specific tag (e.g. precision, data type modifier).
- **QUAL** -Qualifier selecting the operand type (see below).
- **TAG2** -Secondary tag (e.g. 1 if operand has HALMAT code).

Constructed in PASS1 by `HALMAT_PIP(OPERAND, QUAL, TAG1, TAG2)`:

```
CURRENT_ATOM = SHL(OPERAND,16) | SHL(TAG1,8) | SHL(QUAL&"F",4)
             | SHL(TAG2&"7",1) | "1";
```

Operand Qualifier (QUAL) Values

Q	Mnemonic	Meaning
0	-	Not used
1	SYT	Symbol table pointer
2	INL	Internal flow number reference
3	VAC	Virtual accumulator (back pointer to previous HALMAT result)
4	XPT	Extended pointer
5	LIT	Literal table pointer
6	IMD	Immediate numerical value
7	AST	Asterisk pointer
8	CSZ	Component size
9	ASZ	Array/copy size
10	OFF	Offset value

Block Structure

HALMAT is stored in blocks of 1800 words (7200 bytes). Each block is written to file unit 1 as a sequential record.

- **Word 0** -Block metadata.
- **Word 1** - `SHL(ATOM#_FAULT, 16) | 1` -pointer to the last active word in the block.
- **Words 2..N** -HALMAT operator and operand words.
- **Word N** - `XREC` operator marking end of block (TAG=1 for final block, TAG=0 otherwise).

Large-Scale Organisation

Each statement of HAL/S code generates a “paragraph” of HALMAT instructions. Since HAL/S statements are of varying complexity, such paragraphs are of varying sizes. The

final HALMAT instruction in each paragraph not within an inline function block is `SMRK`; within an inline function block it is instead `IMRK`.

The paragraphs are organized into "records" (alternately called "blocks"), which are 7200 bytes long, and thus can hold up to 1800 HALMAT instructions each. As many paragraphs as possible are put into each record, but paragraphs cannot be split across records, so there is generally a gap at the end of each block that contains no HALMAT data.

The first HALMAT instruction in a record is always `PXRC`, while the last instruction (immediately following the final paragraph in the record and preceding the end gap) is `XREC`.

The entire HAL/S source code may generate enough paragraphs to occupy several records. The `TAG` field of the `XREC` instructions at the ends of the records distinguishes the final record (`TAG` = 1) from preceding records (`TAG` = 0).

See [2, p. 115] for the original discussion. The original organisation was driven by memory constraints on the computers of the era, but the format is preserved in the regression test binaries and in the Python PASS1 port's output.

Instruction Set Reference

The 180 HALMAT opcodes fall into 9 classes. Classes 1 through 6 share a parallel structure: each class covers one HAL/S data type (bit, character, matrix, vector, scalar, integer) and provides the same set of operations (assign, arithmetic, type conversions) with matching opcode numbers. Class 0 handles programme structure and control flow. Class 7 handles comparisons and boolean connectives across all types. Class 8 handles initialisation.

Individual opcode entries document each instruction's encoding, operand layout, and compiler source references. For opcodes that participate in control flow patterns (FOR loops, IF/ELSE, CASE statements, etc.), the entry includes a cross-reference to the Control Flow Patterns section, where the full grouped behaviour is documented with verified worked examples.

Class 0: Control and Subscripting

NOP (`0x000`)

No operation

Optimisation (OPT):

```
INSERTHA.xpl:79
      78  IF DSUB_HOLE < DSUB_LOC THEN
>>>   79  OPR(DSUB_HOLE) = XNOP | SHL(DSUB_LOC - DSUB_HOLE - 1, 16);
      80  /* RESET NOP*/
PREPAREH.xpl:143
     142
>>>  143  IF OPRTR = XNOP THEN DO;
     144  CTR = CTR + OPNUM + 1;
PUSHHALM.xpl:126
     125  IF EXTN_CHK THEN D_N_INX = 0;
>>>  126  IF DISP <= ROOM AND (OPR(SMRK_CTR - ROOM) & "FFF1") = XNOP
     127  AND ^FINAL_PUSH AND ^EXTN_CHK THEN DO;
```

Notes (Burkey, HALMAT.md):

Obviously, this is a “no operation” instruction. The incomplete ref [1, p.117] available tells us this about the operator word’s fields:

- `TAG` = don’t care.
- `NUMOP` = *not* necessarily 0. Apparently you can have as many operands as you like, though of course those operations have no operational effect.
- `OP` = 0x000
- `P` = 0

EXTN (`0x001`)

Extended pointer -lists symbol-table references for structure variables (e.g. A.B.C)

Emission (PASS1):

```
ICQOUTPU.xpl:106
     105  IF SYT_TYPE(ID_LOC)=MAJ_STRUC THEN DO;
>>>  106  CALL HALMAT_POP(XEXTN,2,0,0);
     107  CALL HALMAT_PIP(ID_LOC,XSYT,0,0);
SYNTHESI.xpl:3264
    3263  IF FIXV(MP)>0 THEN DO;
>>> 3264  CALL HALMAT_TUPLE(XEXTN,0,MP,0,0);
    3265  TEMP=H1;
```

Code Generation (PASS2):

```
GENERATE.xpl:7410
    7409  DECLARE PTR BIT(16);
>>> 7410  DO WHILE NEXTPOPCODE(PTR) <= XEXTN;
    7411  PTR = NEXTCTR;
```

Optimisation (OPT):

```
RELOCAT2.xpl:105
     104  TEMP = LAST_OP(PTR);
>>>  105  IF (OPR(TEMP)&"FFF1")=XEXTN THEN DO;
     106  IF (OPR(TEMP) & "8") = 0 THEN DO;
```

Notes (Burkey, HALMAT.md):

Operator for listing the multiple symbol-table references required for referencing a structure variable. I think what this means is that if you have (say) a structure variable `A.B.C`, then that's an "extended pointer", and the symbol table will have a reference for `A`, a reference for `A.B`, and a reference for `A.B.C`. I think that the successive levels in the hierarchy of a structure are referred to as levels of qualification. Recall that in the BNF grammar for HAL/S, a construct like `A.B.C` is a `<QUAL STRUCT>`.

At any rate, in terms of execution, this acts like a `NOP`.

XREC (0x002)

End of HALMAT record/block

Emission (PASS1):

```
HALMATOU.xpl:59
      58  SAVE_ATOM=ATOMS(ATOM#_FAULT);
>>>   59  ATOMS(ATOM#_FAULT)=SHL(XXREC,4);
      60  END;
SYNTHESI.xpl:1081
    1080  ELSE IF BLOCK_MODE=0 THEN CALL ERROR(CLASS_PP,4);
>>> 1081  CALL HALMAT_POP(XXREC,0,0,1);
    1082  ATOM#_FAULT=-1;
```

Code Generation (PASS2):

```
OPTIMISE.xpl:135
     134  DO WHILE OPRTR^=XSMRK;
>>>  135  IF OPRTR=XXREC THEN RETURN;
     136  IF OPRTR = XIDEF THEN IFLAG = 1;
```

Optimisation (OPT):

```
PREPAREH.xpl:129
     128  DO WHILE OPRTR ^= XSMRK;
>>>  129  IF OPRTR=XXREC THEN DO;
     130  NOT_XREC = FALSE;
PUTHALMA.xpl:109
     108  OPR(BLOCK_PLUS_1) = XPXRC | "1 0000";
>>>  109  OPR(I) = XXREC | "8";      /* CSE TAG ON XREC FOLLOWED BY
     110  SPILLOVER BLOCK*/
RELOCATE.xpl:73
      72
>>>   73  IF CHECK_TO_XREC THEN STOPPING = XXREC;
      74  ELSE DO;
RELOCATE.xpl:83
      82  DO WHILE (OPR(I) & "FFF1") ^= STOPPING & /* DR109007FIX*/
>>>   83  (OPR(I) & "FFF1") ^= XXREC; /* DR109007 FIX*/
      84
```

IMRK (`0x003`)

Inline function statement marker

Emission (PASS1):

```
EMITSMRK.xpl:97
      96  ELSE IF T<5 THEN DO;
>>>   97  CALL HALMAT_POP(XIMRK,1,XCO_N,STATEMENT_SEVERITY);
      98  CALL HALMAT_PIP(STMT_NUM, 0, SMRK_FLAG, T>1);
```

Other References:

```
? GENCLAS0.xpl:2234
```

Notes (Burkey, HALMAT.md):

`IMRK` follows the generated code of each HAL/S source statement within an Inline Function Block. The incomplete ref [1, p.117] available tells us this about the operator word's fields:

- `TAG` = the maximum statement error severity, 0 if none. What that means exactly is TBD.
- `NUMOP` = 1.
- `OP` = 0x003.
- `P` = TBD.

There is one operand word, as follows:

- `D` = source statement number. I presume the source statements are simply numbered sequentially in the order the compiler encounters them, but it's really TBD.
- `T1` = number used for compiler testing; normally 0.
- `Q` = don't care.
- `T2` = 0 for statements with no HALMAT code, 1 otherwise. I presume that might relate to `%MACROS`, such as direct calls to assembly-language or other non-HAL/S code.

SMRK (`0x004`)

Statement marker

Emission (PASS1):

```
EMITSMRK.xpl:91
      90  IF INLINE_LEVEL=0 THEN DO;
>>>   91  CALL HALMAT_POP(XSMRK,1,XCO_N,STATEMENT_SEVERITY);
      92  CALL HALMAT_PIP(STMT_NUM, 0, SMRK_FLAG, T>1);
```

Code Generation (PASS2):

```
OPTIMISE.xpl:134
     133  IFLAG, VDLP_DETECTED = FALSE;
>>>  134  DO WHILE OPRTR^=XSMRK;
     135  IF OPRTR=XXREC THEN RETURN;
```

Optimisation (OPT):

```
CHICKENO.xpl:2696
    2695  IF WATCH THEN OUTPUT = '******ERROR IN CTR, STATEMENT SKIPPED';
>>> 2696  DO WHILE (OPR(CTR) & "FFF1") ^= XSMRK;
    2697  CTR = CTR + 1;
DECODEPO.xpl:70
      69  OUTPUT='         HALMAT: '||MESSAGE;
>>>   70  IF (OPR(CTR) & "FFF1") = XSMRK THEN
      71  OUTPUT = '                                          STATEMENT# '
NEXTFLAG.xpl:60
      59  END;
>>>   60  DO WHILE (OPR(PTR) & "FFF1") ^= XSMRK;
      61  NUMOP_FOR_REARRANGE = NO_OPERANDS(PTR);
NEXTFLAG.xpl:67
      66  ELSE
>>>   67  DO WHILE (OPR(PTR) & "FFF1") ^= XSMRK;
      68  IF SHR(FLAG(PTR),BIT#) THEN DO;
PREPAREH.xpl:128
     127  NOT_XREC = TRUE;
>>>  128  DO WHILE OPRTR ^= XSMRK;
     129  IF OPRTR=XXREC THEN DO;
PRINTSEN.xpl:59
      58  SHR(OPR(PTR),3) THEN MSG = FORMAT(NODE(SHR(OPR(PTR),16))&"FFFF",5);
>>>   59  ELSE IF (OPR(PTR)&"FFF1")=XSMRK THEN MSG=' ST#='||SHR(OPR(PTR+1),16);
      60  ELSE MSG = '';
PRINTSEN.xpl:79
      78  CALL PRINT;
>>>   79  DO WHILE (OPR(PTR) & "FFF1") ^= XSMRK;
      80  PTR = PTR + 1;
PUTHALMA.xpl:94
      93  I = BLOCK_END - 2;
>>>   94  DO WHILE (OPR(I) & "FFF1") ^= XSMRK;
      95  I = I - 1;
REFEREN2.xpl:58
      57  OLD = PTR;
>>>   58  DO WHILE (OPR(PTR) & "FFF1") ^= XSMRK;
      59  DO FOR I = PTR + 1 TO PTR + NO_OPERANDS(PTR);
RELOCAT2.xpl:128
     127  CTR = TEMP;
>>>  128  DO WHILE (OPR(CTR)&"FFF1")^=XSMRK;
     129  DO FOR K = CTR+1 TO CTR+NO_OPERANDS(CTR);
RELOCATE.xpl:75
      74  ELSE DO;
>>>   75  STOPPING = XSMRK;
      76  IF ^ NOT_TOTAL_RELOCATE THEN
```

Other References:

> `? GENCLAS0.xpl:2234`

Notes (Burkey, HALMAT.md):

`SMRK` follows the generated code of each HAL/S source statement not contained in an Inline Function Block. The incomplete ref [1, p.117] available tells us this about the operator word's fields:

- `TAG` = the maximum statement error severity, 0 if none. What that means exactly is TBD.
- `NUMOP` = 1.
- `OP` = 0x004.
- `P` = TBD.

There is one operand word, as follows:

- `D` = source statement number. I presume the source statements are simply numbered sequentially in the order the compiler encounters them, but it's really TBD.
- `T1` = number used for compiler testing; normally 0.
- `Q` = don't care.
- `T2` = 0 for statements with no HALMAT code, 1 otherwise. I presume that might relate to `%MACROS`, such as direct calls to assembly-language or other non-HAL/S code.

PXRC (`0x005`)

Pointer to XREC -first operator in each block

Optimisation (OPT):

```
NEWHALMA.xpl:47
      46  STACKED_BLOCK#(0),BLOCK#,BLOCK_TOP = 1;
>>>   47  IF (OPR & "FFF1") = XPXRC THEN
      48  XREC_PTR = SHR(OPR(1),16);
PUTHALMA.xpl:106
     105  CALL PUSH_HALMAT(BLOCK_PLUS_1,0,1);
>>>  106  IF (OPR & "FFF1") = XPXRC THEN
     107  OPR(1) = IMD_0 | SHL(I,16);
PUTHALMA.xpl:108
     107  OPR(1) = IMD_0 | SHL(I,16);
>>>  108  OPR(BLOCK_PLUS_1) = XPXRC | "1 0000";
     109  OPR(I) = XXREC | "8";      /* CSE TAG ON XREC FOLLOWED BY
QUICKREL.xpl:73
      72
>>>   73  IF (OPR & "FFF1") = XPXRC THEN
      74  OPR(1) = OPR(1) + SHL(TOTAL,16);   /* RESET PXRC*/
```

Notes (Burkey, HALMAT.md):

PXRC is the first operator in each HALMAT record/block. The incomplete ref [1, p.118] available tells us this about the operator word's fields:

- `TAG` = don't care.
- `NUMOP` = 1.
- `OP` = 0x005.
- `P` = 0

while the operand word is:

- `D` = pointer to the index of the `XREC` for the record/block.
- `T1` , `Q` , `T2` = don't care.

IFHD (`0x007`)

IF statement header. *See Control Flow Patterns: IF/ELSE for the full IFHD/FBRA/BRA/LBL group with worked examples.*

Emission (PASS1):

```
SYNTHESI.xpl:2620
    2619  FIXL(MP)=INDENT_LEVEL;
>>> 2620  CALL HALMAT_POP(XIFHD,0,XCO_N,0);
    2621  END;
```

Notes (Burkey, HALMAT.md):

Mark beginning of an `IF` statement.

LBL (`0x008`)

Label definition. Used as exit target in IF/ELSE and other constructs. *See Control Flow Patterns: IF/ELSE.*

Emission (PASS1):

```
SYNTHESI.xpl:1648
    1647  DO;
>>> 1648  CALL HALMAT_POP(XLBL,1,XCO_N,0);
    1649  CALL HALMAT_PIP(FIXL(MP),XINL,0,0);
SYNTHESI.xpl:2502
    2501  INDENT_LEVEL=FIXL(MP);
>>> 2502  CALL HALMAT_POP(XLBL,1,XCO_N,1);
    2503  CALL HALMAT_PIP(FIXV(MP),XINL,0,0);
SYNTHESI.xpl:2569
    2568  CALL HALMAT_PIP(FL_NO,XINL,0,0);
>>> 2569  CALL HALMAT_POP(XLBL,1,XCO_N,0);
    2570  CALL HALMAT_PIP(FIXV(MP),XINL,0,0);
SYNTHESI.xpl:4269
    4268  IF NEST=0 THEN EXTERNAL_MODE=0;
>>> 4269  CALL HALMAT_POP(XLBL,1,XCO_N,0);
    4270  CALL HALMAT_PIP(FIXL(MP),XSYT,0,0);
```

Notes (Burkey, HALMAT.md):

Define a label, unless it is an unused exit label of an `IF` statement.

BRA (`0x009`)

Unconditional branch. Skips the ELSE clause in IF/ELSE constructs. *See Control Flow Patterns: IF/ELSE.*

Emission (PASS1):

```
SYNTHESI.xpl:1700
    1699  IF LABEL_MATCH THEN DO;
>>> 1700  CALL HALMAT_POP(XBRA,1,0,0);
    1701  CALL HALMAT_PIP(DO_LOC(TEMP),XINL,0,0);
SYNTHESI.xpl:1722
    1721  IF DO_INX(TEMP) THEN IF LABEL_MATCH THEN DO;
>>> 1722  CALL HALMAT_POP(XBRA,1,0,0);
    1723  CALL HALMAT_PIP(DO_LOC(TEMP)+1,XINL,0,0);
SYNTHESI.xpl:1748
    1747  ELSE IF VAR_LENGTH(I)=0 THEN VAR_LENGTH(I)=3;
>>> 1748  CALL HALMAT_POP(XBRA,1,0,0);
    1749  CALL HALMAT_PIP(I,XSYT,0,0);
SYNTHESI.xpl:2567
    2566  END;                                                  /*CR12713*/
>>> 2567  CALL HALMAT_POP(XBRA,1,0,1);
    2568  CALL HALMAT_PIP(FL_NO,XINL,0,0);
```

Notes (Burkey, HALMAT.md):

If branch not redundant, emit unconditional branch.

FBRA (`0x00A`)

Branch on false. Carries the condition result as a VAC operand. *See Control Flow Patterns: IF/ELSE for the full IFHD/FBRA/BRA/LBL group.*

Emission (PASS1):

```
SYNTHESI.xpl:2579
    2578  EMIT_IF:
>>> 2579  CALL HALMAT_POP(XFBRA,2,XCO_N,0);
    2580  CALL HALMAT_PIP(FL_NO,XINL,0,0);
```

Optimisation (OPT):

```
PREPAREH.xpl:162
     161  END;
>>>  162  ELSE IF OPRTR = XFBRA THEN DO;
     163  TMP = SHR(OPR(CTR+2), 16);
```

Notes (Burkey, HALMAT.md):

Emit code or fill in addresses in existing instructions to perform branch on false.

DCAS (`0x00B`)

DO CASE initialisation and case selection. *See Control Flow Patterns: DO CASE for the full DCAS/CLBL/ECAS group with worked examples.*

Emission (PASS1):

```
SYNTHESI.xpl:2696
    2695  IF UNARRAYED_INTEGER(MP+2) THEN CALL ERROR(CLASS_GC,1);
>>> 2696  CALL HALMAT_POP(XDCAS,2,0,FIXL(MP));
    2697  CALL EMIT_PUSH_DO(2,4,0,MP-1);
```

Notes (Burkey, HALMAT.md):

Initialize for `DO CASE` and generate standard code to perform case selection.

ECAS (`0x00C`)

End CASE - set up indirect jump table. *See Control Flow Patterns: DO CASE.*

Emission (PASS1):

```
SYNTHESI.xpl:1815
    1814  CALL HALMAT_FIX_POPTAG(FIXV(MP),1);
>>> 1815  TEMP=XECAS;
    1816  INFORMATION= '';                              /*CR12713*/
```

Notes (Burkey, HALMAT.md):

Set up table of indirect jumps to actually get to individual cases, define a label for the location after all cases.

CLBL (`0x00D`)

Case label - jump to end of DO CASE, define flow number. TAG=1 on the last CLBL marks end-of-list. *See Control Flow Patterns: DO CASE.*

Emission (PASS1):

```
SYNTHESI.xpl:2745
    2744  INFORMATION=INFORMATION||CASE_STACK(TEMP);         /*DR109091*/
>>> 2745  CALL HALMAT_POP(XCLBL,2,XCO_N,0);
    2746  CALL HALMAT_PIP(DO_LOC(DO_LEVEL),XINL,0,0);
```

Notes (Burkey, HALMAT.md):

Generate jump to the location after the `DO CASE` statement; if this is not the last case, define the flow number of this one and link it into a list.

DTST (`0x00E`)

DO WHILE/UNTIL - generate jump around test, define loop start. TAG=0 for WHILE, TAG=1 for UNTIL. *See Control Flow Patterns: DO WHILE/DO UNTIL for the full DTST/CTST/ETST group.*

Emission (PASS1):

```
SYNTHESI.xpl:2772
    2771  IF PARSE_STACK(MP-1)=DO_TOKEN THEN DO;
>>> 2772  CALL HALMAT_POP(XDTST,1,XCO_N,TEMP);
    2773  CALL EMIT_PUSH_DO(3,3,0,MP-2);
```

Notes (Burkey, HALMAT.md):

If this is `DO UNTIL` , generate jump around test; define beginning of a loop.

ETST (`0x00F`)

End test - generate jump back to loop start, define exit label. *See Control Flow Patterns: DO WHILE/DO UNTIL.*

Emission (PASS1):

```
SYNTHESI.xpl:1820
    1819  /* DO WHILE */
>>> 1820  TEMP=XETST;
    1821  END;
```

Notes (Burkey, HALMAT.md):

Generate jump back to the beginning of the loop; define a label for the location after the loop; free temporary storage.

DFOR (`0x010`)

DO FOR loop header. *See Control Flow Patterns: FOR Loop for the full DFOR/EFOR/CFOR/AFOR group with worked examples.*

Emission (PASS1):

```
SYNTHESI.xpl:2797
    2796  IF UNARRAYED_SIMPLE(SP-1) THEN CALL ERROR(CLASS_GC,3);
>>> 2797  CALL HALMAT_POP(XDFOR,TEMP2+3,XCO_N,0);
    2798  CALL EMIT_PUSH_DO(1,5,PSEUDO_TYPE(PTR(SP))=INT_TYPE,MP-2,FIXL(MP));
SYNTHESI.xpl:2819
    2818  TEMP=PTR(MP-1);   /*<FOR KEY> PTR */
>>> 2819  CALL HALMAT_POP(XDFOR,2,XCO_N,0);
    2820  CALL EMIT_PUSH_DO(1,5,0,MP-3,FIXL(MP-1));
```

Notes (Burkey, HALMAT.md):

Set the “ `DO` type” equal to the tag field. Allocate space for temporaries if this is `DO FOR TEMPORARY` . Ref [2, PDF pp. 494-495] has a lot more to say about this.

EFOR (`0x011`)

End FOR loop - define end-of-loop label. *See Control Flow Patterns: FOR Loop.*

Emission (PASS1):

```
SYNTHESI.xpl:1811
    1810  /* DO FOR */
>>> 1811  TEMP=XEFOR;
    1812  /* DO CASE */
```

Example (HELLO.hal):

```
    66:  00010110  CTRL                  EFOR  (CTRL/EFOR, 1 ops)
         00060021    op1               INL(6)
```

Notes (Burkey, HALMAT.md):

Define label which is end of loop. See [2, PDF p. 495] for more.

CFOR (`0x012`)

Conditional FOR - emit WHILE/UNTIL test code. *See Control Flow Patterns: FOR Loop.*

Emission (PASS1):

```
SYNTHESI.xpl:2675
    2674  CALL HALMAT_FIX_POPTAG(FIXV(MPP1),SHL(INX(TEMP),4)|PTR(MPP1));
>>> 2675  CALL HALMAT_POP(XCFOR,1,0,INX(TEMP));
    2676  EMIT_WHILE:
```

Optimisation (OPT):

```
PREPAREH.xpl:171
     170  END;
>>>  171  ELSE IF OPRTR = XCFOR | OPRTR = XCTST THEN DO;
     172  TMP = SHR(OPR(CTR+1), 16);
```

Notes (Burkey, HALMAT.md):

At this point, loop header code and code to evaluate condition have been issued. Emit code to perform `WHILE/UNTIL` test. Define label of beginning of actual code to allow

skipping around `UNTIL` code on first iteration.

DSMP (`0x013`)

Simple DO - bump DO level. Scope boundary only, no loop condition. *See Control Flow Patterns: Simple DO.*

Emission (PASS1):

```
SYNTHESI.xpl:2626
    2625  FIXL(MPP1)=0;
>>> 2626  CALL HALMAT_POP(XDSMP,1,0,0);
    2627  CALL EMIT_PUSH_DO(0,1,0,MP-1);
```

Notes (Burkey, HALMAT.md):

Bump `DO LEVEL`.

ESMP (`0x014`)

End simple DO - define exit label, free temporaries. *See Control Flow Patterns: Simple DO.*

Emission (PASS1):

```
SYNTHESI.xpl:1809
    1808  /* SIMPLE DO */
>>> 1809  TEMP=XESMP;
    1810  /* DO FOR */
```

Notes (Burkey, HALMAT.md):

If anybody needed the address of the end of the loop, define it; free temporaries used in loop.

AFOR (`0x015`)

Advance FOR - update loop index, exit or continue. TAG=1 on the last AFOR in a discrete value list marks end-of-list. *See Control Flow Patterns: FOR Loop.*

Emission (PASS1):

```
SYNTHESI.xpl:2830
    2829  PTR_TOP=PTR(SP)-1;
>>> 2830  CALL HALMAT_TUPLE(XAFOR,XCO_N,SP,0,0);
    2831  FL_NO=FL_NO+1;
```

Notes (Burkey, HALMAT.md):

Like all `xFOR` opcodes, this is a part of a `FOR` -loop. Its behavior is rather complex, so I'd recommend looking at ref [2, p. 496]. In brief, it updates the loop index and then either exits the loop or else continues within the loop.

CTST (`0x016`)

WHILE/UNTIL test and skip label. *See Control Flow Patterns: DO WHILE/DO UNTIL.*

Emission (PASS1):

```
SYNTHESI.xpl:2686
    2685  TEMP=PTR(MPP1);
>>> 2686  CALL HALMAT_POP(XCTST,1,0,INX(TEMP));
    2687  GO TO EMIT_WHILE;
```

Optimisation (OPT):

```
PREPAREH.xpl:171
     170  END;
>>>  171  ELSE IF OPRTR = XCFOR | OPRTR = XCTST THEN DO;
     172  TMP = SHR(OPR(CTR+1), 16);
```

Notes (Burkey, HALMAT.md):

Generate code to perform WHILE/UNTIL test and label for skipping UNTIL test.

ADLP (`0x017`)

Arrayed DO loop -initialise tables for arrayed expression

Emission (PASS1):

```
EMITARRA.xpl:86
      85  ARRAYNESS_FLAG=0;
>>>   86  ACODE=XADLP;
      87  END EMIT_ARRAYNESS;
ICQARRA2.xpl:80
      79  IF SYT_ARRAY(ID_LOC)^=0 THEN DO;
>>>   80  IF (SYT_FLAGS(ID_LOC)&AUTO_FLAG)^=0 THEN I=XADLP;
      81  ELSE I=XIDLP;
```

Code Generation (PASS2):

```
OPTIMISE.xpl:137
     136  IF OPRTR = XIDEF THEN IFLAG = 1;
>>>  137  ELSE IF OPRTR = XADLP THEN VDLP_DETECTED = SHR(OPR(SMRK_CTR+OPNUM),8);
     138  ELSE IF OPRTR >= XREAD & OPRTR <= XWRIT THEN READCTR = SMRK_CTR;
```

Notes (Burkey, HALMAT.md):

Initialize tables for constructing do loop for arrayed expression and generate initial code in complicated cases.

DLPE (`0x018`)

DO loop end -generate end-of-loop code

Emission (PASS1):

```
EMITARRA.xpl:82
      81  END;
>>>   82  CALL HALMAT_POP(XDLPE,0,XCO_N,FCN_LV);
      83  CURRENT_ARRAYNESS=0;
ICQARRA2.xpl:91
      90  CALL HALMAT_FIX_PIP#(LAST_POP#,I-1);
>>>   91  CALL HALMAT_POP(XDLPE,0,XCO_N,0);
      92  END;
```

Optimisation (OPT):

```
PREPAREH.xpl:277
     276  CALL OPDECODE(CTR);
>>>  277  IF OPRTR = XDLPE THEN DO;
     278  SIZE = CTR - START(FUNC_LEVEL);
```

Other References:

```
? GENCLAS0.xpl:2811
? GENCLAS0.xpl:3495
```

Notes (Burkey, HALMAT.md):

Generate end of loop code and clean up.

DSUB (`0x019`)

Regular subscript specifier. *See Control Flow Patterns: Subscript Access for the full DSUB/IDLP/TSUB group.*

Emission (PASS1):

```
SYNTHESI.xpl:3339
    3338  ELSE IF IND_LINK>0 THEN DO;
>>> 3339  CALL HALMAT_TUPLE(XDSUB,0,MP,0,PSEUDO_TYPE(H1)|NAME_BIT);
    3340  INX(H1)=NEXT_ATOM#-1;
```

Code Generation (PASS2):

```
GENERATE.xpl:1501
    1500  /* AN OPERAND OF A NAME PSEUDO-FUNCTION                        */
>>> 1501  IF (HALMAT_OPCODE = XDSUB) & NAME_SUB THEN  /*DR109032*/
    1502  /* TAG3=1,5 MEANS PROCESSING THE SUBSCRIPT (NOT THE VARIABLE-CR12432 */
```

Notes (Burkey, HALMAT.md):

Specifies a regular subscript, which seems to apply to ARRAY as well as VECTOR and MATRIX. Generates all necessary code to evaluate subscript expression and put value of

the expression in an index register. Our incomplete ref [1, p. 120] tells us the following about the operator word:

- `TAG` = result type of the data after possible modification by the component subscripts. (Recall that there can be subscripts like `@DOUBLE` that alter the datatype, like "casts" in C.) The numerical values and their interpretations are TBD.
- `NUMOP` = variable.
- `OP` = 0x019.
- `P` = 0.

The first operand word differs in format from the succeeding operand words:

- `D` = direct or indirect reference to the data item referenced. I suppose the "data item referenced" means the item being subscripted.
- `T1` = don't care.
- `Q` = `ESV`. Recall that `ESV` is a generic qualifier mnemonic that can mean either `SYT` (numerical 1, pointer into symbol table) or `XPT` (numerical 4, extended pointer).
- `P` = 1 for assign context, (?) 0 otherwise. What that means is TBD.

Apparently, the Shuttle-development implementation of HAL/S allowed a maximum of 5 dimensions (i.e., 5 subscripts). Each subscript, in left-to-right order, is associated with a group of 1 to 4 operand words. So in principle there could be up to 20 operand words, aside from the one described above. All of these additional operand words are formatted using the pattern:

- `D` = operand.
- `T1` = α (defined in the table below).
- `Q` = `qual` (defined in the table below).
- `T2` = β (defined in the table below).

Where:

(Refer to Ron Burkey's HALMAT.md for the DSUB operand-format table.)

By itself, this table only allows for groups with up to 2 operand words. However, the "note" it refers to tells us that an operand word with `Q` = `CSZ` or `ASZ`, *may* be immediately followed by yet another operand word. `CSZ` and `ASZ` respectively correspond to # expressions for character strings or arrays specified with *, in which the object size (#) is not known at compile time.

When `Q = CSZ` or `ASZ` and `D = 0`, the # expression is just # alone, and the operand words is not followed by an additional operand word. But if `D = 1`, then the # expression is # + something, while if `D = 2`, then the # expression is # - something, and there is an additional operand word:

- `D` = operand.
- `T1` = don't care
- `Q = EEV`. Recall that `EEV` one of those omnibus qualifier mnemonics which may be any of `SYT` (1, pointer into symbol table), `VAC` (3, virtual accumulator), `XPT` (4, extended pointer), `LIT` (5, pointer into literal table), or `IMD` (6, actual numerical value).
- `T2` = don't care

IDLP (`0x01A`)

Indexed DO loop. *See Control Flow Patterns: Subscript Access.*

Emission (PASS1):

```
ICQARRA2.xpl:81
      80  IF (SYT_FLAGS(ID_LOC)&AUTO_FLAG)^=0 THEN I=XADLP;
>>>   81  ELSE I=XIDLP;
      82  CALL HALMAT_POP(I,0,XCO_D,0);
```

Notes (Burkey, HALMAT.md):

Set up array do loop parameters to describe the arrayness as copied from the IDLP operands.

TSUB (`0x01B`)

Subscript for # (pound) operator. *See Control Flow Patterns: Subscript Access.*

Emission (PASS1):

```
SYNTHESI.xpl:3256
    3255  IF ATTACH_SUBSCRIPT THEN DO;
>>> 3256  CALL HALMAT_TUPLE(XTSUB,0,MP,0,MAJ_STRUC|NAME_BIT);
    3257  INX(H1)=NEXT_ATOM#-1;
```

Notes (Burkey, HALMAT.md):

Similar to a very stripped down DSUB. There is only one level of subscripting and that applies across the entire structure.

PCAL (`0x01D`)

Procedure call. *See Control Flow Patterns: Function/Procedure Calls.*

Emission (PASS1):

```
ENDANYFC.xpl:298
     297  IF FCN_LV=0 THEN                              /*CR13571*/
>>>  298  CALL HALMAT_TUPLE(XPCAL,0,MP,0,0);
     299  ELSE DO;
```

Code Generation (PASS2):

```
GENERATE.xpl:1518
    1517  /* PSEUDO-FUNCTION (ARG_NAME(ARG#)).                        */
>>> 1518  IF (HALMAT_OPCODE = XPCAL) | (HALMAT_OPCODE = XFCAL) THEN DO;  /* DR109046
*/
    1519  IF ARG_NAME(ARG#) THEN RETURN TRUE;                        /* DR109046 */
```

Notes (Burkey, HALMAT.md):

Check that we are not in a nested function call (*n.b.* `TAG` will be 0); make stack entry for procedure name. See [2, p. 498] for more.

FCAL (`0x01E`)

Function call. Reuses XXST/XXAR/XXND argument group with TAG=1 to distinguish from I/O. *See Control Flow Patterns: Function/Procedure Calls.*

Emission (PASS1):

```
ENDANYFC.xpl:301
     300  CALL RESET_ARRAYNESS;
>>>  301  CALL HALMAT_TUPLE(XFCAL,0,MP,0,FCN_LV);
     302  CALL SETUP_VAC(MP,PSEUDO_TYPE(PTR(MP)));
SETUPNOA.xpl:224
     223  CALL STRUCTURE_FCN;
>>>  224  CALL HALMAT_TUPLE(XFCAL,0,MP,0,FCN_LV+1);
     225  CALL SETUP_VAC(MP,PSEUDO_TYPE(PTR_TOP));
```

Code Generation (PASS2):

```
GENERATE.xpl:1518
    1517  /* PSEUDO-FUNCTION (ARG_NAME(ARG#)).                          */
>>> 1518  IF (HALMAT_OPCODE = XPCAL) | (HALMAT_OPCODE = XFCAL) THEN DO;  /* DR109046
*/
    1519  IF ARG_NAME(ARG#) THEN RETURN TRUE;                          /* DR109046 */
```

Notes (Burkey, HALMAT.md):

Check that we are nested to the proper depth in function calls; make stack entry for function name. See [2, p. 498] for more.

READ (`0x01F`)

READ statement. *See Control Flow Patterns: I/O Groups.*

Emission (PASS1):

```
SYNTHESI.xpl:1849
    1848  XSET"3";
>>> 1849  CALL HALMAT_TUPLE(XREAD(INX(PTR(MP))),0,MP,0,0);
    1850  PTR_TOP=PTR(MP)-1;
```

Code Generation (PASS2):

```
OPTIMISE.xpl:138
     137  ELSE IF OPRTR = XADLP THEN VDLP_DETECTED = SHR(OPR(SMRK_CTR+OPNUM),8);
>>>  138  ELSE IF OPRTR >= XREAD & OPRTR <= XWRIT THEN READCTR = SMRK_CTR;
     139  SMRK_CTR=SMRK_CTR+OPNUM+1;
```

Optimisation (OPT):

```
PREPAREH.xpl:292
     291  FUNC_LEVEL = NEST_LEVEL(OPTAG);
>>>  292  ELSE IF OPRTR >= XREAD & OPRTR <= XWRIT THEN
     293  READCTR = CTR;
```

Notes (Burkey, HALMAT.md):

TBD

RDAL (`0x020`)

READALL statement

Notes (Burkey, HALMAT.md):

Note: This is the first of the cases, of which all of the following below are also examples, in which `CLASS` =0 but `SUBCODE` ≠0. Thus there is an ambiguity in the two different usages of `OPCODE` (as described earlier), so one must be careful in how one uses that mnemonic.

TBD

WRIT (`0x021`)

WRITE statement. *See Control Flow Patterns: I/O Groups for the full XXST/XXAR/WRIT/XXND group.*

Code Generation (PASS2):

```
OPTIMISE.xpl:138
     137  ELSE IF OPRTR = XADLP THEN VDLP_DETECTED = SHR(OPR(SMRK_CTR+OPNUM),8);
>>>  138  ELSE IF OPRTR >= XREAD & OPRTR <= XWRIT THEN READCTR = SMRK_CTR;
     139  SMRK_CTR=SMRK_CTR+OPNUM+1;
```

Optimisation (OPT):

```
PREPAREH.xpl:292
     291  FUNC_LEVEL = NEST_LEVEL(OPTAG);
>>>  292  ELSE IF OPRTR >= XREAD & OPRTR <= XWRIT THEN
     293  READCTR = CTR;
```

Example (HELLO.hal):

```
    22:  00010210  CTRL                  WRIT  (CTRL/WRIT, 1 ops)
         00060061    op1               IMD(6)
```

Notes (Burkey, HALMAT.md):

TBD

FILE (`0x022`)

File I/O operation

Emission (PASS1):

```
SYNTHESI.xpl:1862
    1861  DO;
>>> 1862  CALL HALMAT_TUPLE(XFILE,0,MP,SP-1,FIXV(MP));
    1863  CALL HALMAT_FIX_PIPTAGS(NEXT_ATOM#-1,PSEUDO_TYPE(PTR(SP-1)),1);
SYNTHESI.xpl:1872
    1871  DO;
>>> 1872  CALL HALMAT_TUPLE(XFILE,0,SP-1,MP,FIXV(SP-1));
    1873  H1=VAL_P(PTR(MP));
```

Other References:

```
? GENCLAS0.xpl:2764
```

Notes (Burkey, HALMAT.md):

Generate code to do library call, presumably having something to do with files and not just library calls in general.

XXST (`0x025`)

Start of I/O argument list. Also used for function/procedure call arguments (TAG=1). *See Control Flow Patterns: I/O Groups and Function/Procedure Calls.*

Emission (PASS1):

```
STARTNOR.xpl:173
     172  CALL SAVE_ARRAYNESS;
>>>  173  CALL HALMAT_POP(XXXST,1,XCO_N,FCN_LV);
     174  CALL HALMAT_PIP(FIXL(MP),XSYT,0,0);
SYNTHESI.xpl:3566
    3565  IF TEMP>0 THEN DO;
>>> 3566  CALL HALMAT_POP(XXXST,1,XCO_N,TEMP+FCN_LV-1);
    3567  CALL HALMAT_PIP(FIXL(MP-1),XSYT,0,0);
SYNTHESI.xpl:3885
    3884  XSET SHL(TEMP,11);
>>> 3885  CALL HALMAT_POP(XXXST,1,XCO_N,0);
    3886  CALL HALMAT_PIP(TEMP,XIMD,0,0);
```

Optimisation (OPT):

```
PREPAREH.xpl:148
     147  ELSE
>>>  148  IF OPRTR = XSFST | OPRTR = XXXST THEN DO;
     149  NEST_LEVEL(OPTAG) = NEST_LEVEL;
PREPAREH.xpl:225
     224  CALL OPDECODE(CTR);
>>>  225  IF OPRTR = XSFST | OPRTR = XXXST THEN DO;
     226  NEST_LEVEL(OPTAG) = NEST_LEVEL;
PREPAREH.xpl:284
     283  END;
>>>  284  ELSE IF OPRTR = XXXST | OPRTR = XSFST THEN DO;
     285  NEST_LEVEL(OPTAG) = FUNC_LEVEL;
```

XXND (`0x026`)

End of I/O argument list (also ends function/procedure call arguments). *See Control Flow Patterns: I/O Groups.*

Emission (PASS1):

```
ENDANYFC.xpl:305
     304  IF MAXPTR>0 THEN MAXPTR=XCO_N;
>>>  305  CALL HALMAT_POP(XXXND,0,MAXPTR,FCN_LV);
     306  END;
SETUPNOA.xpl:226
     225  CALL SETUP_VAC(MP,PSEUDO_TYPE(PTR_TOP));
>>>  226  CALL HALMAT_POP(XXXND,0,0,FCN_LV+1);
     227  IF INLINE_LEVEL>0 THEN CALL ERROR(CLASS_PP,8);
SYNTHESI.xpl:1851
    1850  PTR_TOP=PTR(MP)-1;
>>> 1851  CALL HALMAT_POP(XXXND,0,0,0);
    1852  GO TO FIX_NOLAB;
```

Optimisation (OPT):

```
PREPAREH.xpl:155
     154  END;
>>>  155  ELSE IF OPRTR = XSFND | OPRTR = XXXND THEN DO;
     156  FLAG_LOC = START(OPTAG);
PREPAREH.xpl:235
     234  END;
>>>  235  ELSE IF OPRTR = XSFND | OPRTR = XXXND THEN DO;
     236  NEST_LEVEL = NEST_LEVEL(OPTAG);
PREPAREH.xpl:290
     289  END;
>>>  290  ELSE IF OPRTR = XXXND | OPRTR = XSFND THEN
     291  FUNC_LEVEL = NEST_LEVEL(OPTAG);
```

Example (HELLO.hal):

```
    24:  00000260  CTRL                 XXND  (CTRL/XXND, 0 ops)
```

XXAR (`0x027`)

I/O or call argument. TAG1 encodes the argument type. *See Control Flow Patterns: I/O Groups.*

Emission (PASS1):

```
SETUPCAL.xpl:202
     201  IF INLINE_LEVEL=0 THEN DO;
>>>  202  CALL HALMAT_TUPLE(XXXAR,XCO_N,SP,0,FCN_LV);
     203  CALL HALMAT_FIX_PIPTAGS(NEXT_ATOM#-1,PSEUDO_TYPE(I)|
SYNTHESI.xpl:3026
    3025  FCN_ARG=FCN_ARG+1;
>>> 3026  CALL HALMAT_TUPLE(XXXAR,XCO_N,SP,0,0);
    3027  CALL HALMAT_FIX_PIPTAGS(NEXT_ATOM#-1,PSEUDO_TYPE(PTR(SP))|
SYNTHESI.xpl:3863
    3862  IF INLINE_LEVEL>0 THEN CALL ERROR(CLASS_PP,5);
>>> 3863  CALL HALMAT_TUPLE(XXXAR,XCO_N,MP,0,0);
    3864  CALL HALMAT_FIX_PIPTAGS(NEXT_ATOM#-1,PSEUDO_TYPE(PTR(MP)),TEMP);
```

Code Generation (PASS2):

```
GENERATE.xpl:1511
    1510  /* FOR THE NAME() PSEUDO-FUNCTION (ARG_NAME(ARG_STAK_PTR)).  */
>>> 1511  IF (HALMAT_OPCODE = XXXAR) THEN DO;                            /* DR109046
*/
    1512  IF ARG_NAME(ARG_STACK_PTR) THEN RETURN TRUE;               /* DR109046 */
```

Optimisation (OPT):

```
PREPAREH.xpl:299
     298
>>>  299  IF OPRTR ^= XXXAR THEN
     300  DO FOR TEMP = CTR + 1 TO CTR+NUMOP;
```

Other References:

```
? GENCLAS0.xpl:2764
```

TDEF (0x02A)

Task definition header

Emission (PASS1):

```
SYNTHESI.xpl:4367
    4366  DO;
>>> 4367  TEMP=XTDEF;
    4368  TEMP2=TASK_MODE;
```

Notes (Burkey, HALMAT.md):

Task definition header. Sets up block definitions. Our incomplete ref [1, p.119] tells us the format of the operator word:

- `TAG` = don't care.
- `NUMOP` = 1.
- `OP` = 0x02A.
- `P` = 0.

Whereas the operand word is:

- `D` = pointer to the task name in the symbol table.
- `T1` = don't care.
- `Q` = `SYL`. (Recall that the qualifier-mnemonic `SYL` has the numerical value 1 and is interpreted as "symbol table pointer".)
- `T2` = don't care.

MDEF (`0x02B`)

Program (main) definition header

Emission (PASS1):

```
SYNTHESI.xpl:4316
    4315  DO;
>>> 4316  TEMP=XMDEF;
    4317  PARMS_PRESENT=0;
```

Example (HELLO.hal):

```
    2:  000102B0  CTRL                  MDEF  (CTRL/MDEF, 1 ops)
        00010011    op1               SYT(1)
```

Notes (Burkey, HALMAT.md):

Program definition header. Sets up block definitions. Our incomplete ref [1, p.119] tells us the format of the operator word:

- `TAG` = don't care.
- `NUMOP` = 1.
- `OP` = 0x02B.
- `P` = 0.

Whereas the operand word is:

- `D` = pointer to the program name in the symbol table.
- `T1` = don't care.
- `Q` = `SYL`. (Recall that the qualifier-mnemonic `SYL` has the numerical value 1 and is interpreted as "symbol table pointer".)
- `T2` = don't care.

FDEF (`0x02C`)

Function definition header. *See Control Flow Patterns: Function/Procedure Calls.*

Emission (PASS1):

```
SYNTHESI.xpl:4474
    4473  END;
>>> 4474  TEMP=XFDEF;
    4475  TEMP2=FUNC_MODE;
```

Notes (Burkey, HALMAT.md):

Function definition header. Sets up block definitions. Our incomplete ref [1, p. 119] tells us the format of the operator word:

- `TAG` = don't care.
- `NUMOP` = 1.
- `OP` = 0x02C.

- `P` = 0.

Whereas the operand word is:

- `D` = pointer to the function name in the symbol table.
- `T1` = don't care.
- `Q` = `SYL`. (Recall that the qualifier-mnemonic `SYL` has the numerical value 1 and is interpreted as "symbol table pointer".)
- `T2` = don't care.

(Actually, the operand word is presumably described on [1, p. 120], which is among the many missing pages. I merely *assume* it has the form given above.)

PDEF (`0x02D`)

Procedure definition header. *See Control Flow Patterns: Function/Procedure Calls.*

Emission (PASS1):

```
SYNTHESI.xpl:4481
    4480  TEMP2=PROC_MODE;
>>> 4481  TEMP=XPDEF;
    4482  CALL SET_LABEL_TYPE(FIXL(MP),PROC_LABEL);
```

Notes (Burkey, HALMAT.md):

Procedure definition header. Sets up block definitions. Our incomplete ref [1, p.119] tells us the format of the operator word:

- `TAG` = don't care.
- `NUMOP` = 1.
- `OP` = 0x02D.
- `P` = 0.

Whereas the operand word is:

- `D` = pointer to the procedure name in the symbol table.
- `T1` = don't care.
- `Q` = `SYL`. (Recall that the qualifier-mnemonic `SYL` has the numerical value 1 and is interpreted as "symbol table pointer".)

- `T2` = don't care.

TBD

UDEF (`0x02E`)

Update block definition header

Emission (PASS1):

```
SYNTHESI.xpl:4389
    4388  TEMP2=UPDATE_MODE;
>>> 4389  TEMP=XUDEF;
    4390  CALL SET_LABEL_TYPE(FIXL(MP),STMT_LABEL);
```

Notes (Burkey, HALMAT.md):

Set up block definitions.

TBD

CDEF (`0x02F`)

COMPOOL definition header

Emission (PASS1):

```
SYNTHESI.xpl:4343
    4342  DO;
>>> 4343  TEMP=XCDEF;
    4344  TEMP2=CMPL_MODE;
```

Notes (Burkey, HALMAT.md):

Set up block definitions. Perhaps this is for a COMPOOL block, but that's TBD.

TBD

CLOS (`0x030`)

Close scope -end of PROGRAM, PROCEDURE, FUNCTION, etc.

Emission (PASS1):

```
SYNTHESI.xpl:3924
    3923  DO;
>>> 3924  TEMP=XCLOS;
    3925  TEMP2=0;
```

Example (HELLO.hal):

```
    83:  00010300  CTRL                  CLOS  (CTRL/CLOS, 1 ops)
         00010011    op1               SYT(1)
```

Notes (Burkey, HALMAT.md):

Check that close is at correct level and close the block.

EDCL (`0x031`)

End of declarations

Emission (PASS1):

```
SYNTHESI.xpl:4112
    4111  DO;
>>> 4112  CALL HALMAT_POP(XEDCL,0,XCO_N,0);
    4113  GO TO CHECK_DECLS;
SYNTHESI.xpl:4118
    4117  DO;
>>> 4118  CALL HALMAT_POP(XEDCL, 0, XCO_N, 1);
    4119  CHECK_DECLS:
```

Notes (Burkey, HALMAT.md):

I don't understand this enough to summarize it. See [2, p. 500].

RTRN (0x032)

RETURN statement. *See Control Flow Patterns: Function/Procedure Calls.*

Emission (PASS1):

```
SYNTHESI.xpl:1766
    1765  ELSE IF BLOCK_MODE(NEST)=UPDATE_MODE THEN CALL ERROR(CLASS_UP,2);
>>> 1766  CALL HALMAT_POP(XRTRN,0,0,0);
    1767  XSET"7";
SYNTHESI.xpl:1795
    1794  END;
>>> 1795  CALL HALMAT_TUPLE(XRTRN,0,MPP1,0,0);
    1796  CALL HALMAT_FIX_PIPTAGS(NEXT_ATOM#-1,         /*DR120223*/
```

Notes (Burkey, HALMAT.md):

For functions, generate code to return result. Generate jump to return.

TDCL (0x033)

Temporary declaration

Emission (PASS1):

```
SYNTHESI.xpl:2657
    2656  IF FIXL(MPP1)>0 THEN DO;
>>> 2657  CALL HALMAT_POP(XTDCL,1,0,0);
    2658  CALL HALMAT_PIP(FIXL(MPP1),XSYT,0,0);
SYNTHESI.xpl:5113
    5112  DO_CHAIN(DO_LEVEL)=ID_LOC;
>>> 5113  CALL HALMAT_POP(XTDCL,1,0,0);
    5114  CALL HALMAT_PIP(ID_LOC,XSYT,0,0);
```

Notes (Burkey, HALMAT.md):

TBD

WAIT (0x034)

WAIT statement (real-time)

Emission (PASS1):

```
SYNTHESI.xpl:1885
    1884  DO;
>>> 1885  CALL HALMAT_POP(XWAIT,0,0,0);
    1886  XSET"B";
SYNTHESI.xpl:1899
    1898  XSET"B";
>>> 1899  CALL HALMAT_TUPLE(XWAIT,0,SP-1,0,TEMP);
    1900  PTR_TOP=PTR(SP-1)-1;
```

Other References:

```
? GENCLAS0.xpl:2553
```

Notes (Burkey, HALMAT.md):

Allocate run time space and generate code to perform WAIT SVC.

SGNL (`0x035`)

SIGNAL statement (real-time)

Emission (PASS1):

```
SYNTHESI.xpl:1967
    1966  XSET"D";
>>> 1967  CALL HALMAT_TUPLE(XSGNL,0,MP,0,INX(PTR(MP)));
    1968  PTR_TOP=PTR(MP)-1;
```

Notes (Burkey, HALMAT.md):

Allocate run time space and generate code to perform SIGNAL, SET, or RESET SVC.

CANC (`0x036`)

CANCEL statement (real-time)

Emission (PASS1):

```
SYNTHESI.xpl:5721
    5720  DO;
>>> 5721  FIXL(MP)=XCANC;
    5722  FIXV(MP)="A000";
```

Notes (Burkey, HALMAT.md):

TBD

TERM (`0x037`)

TERMINATE statement (real-time)

Emission (PASS1):

```
SYNTHESI.xpl:5716
    5715  DO;
>>> 5716  FIXL(MP)=XTERM;
    5717  FIXV(MP)="E000";
```

Notes (Burkey, HALMAT.md):

TBD, but apparently the same thing as `CANC`.

PRIO (`0x038`)

PRIORITY statement (real-time)

Emission (PASS1):

```
SYNTHESI.xpl:1939
    1938  CALL ERROR(CLASS_RT,4,'UPDATE PRIORITY');
>>> 1939  CALL HALMAT_TUPLE(XPRIO,0,SP-1,TEMP,TEMP>0);
    1940  GO TO UPDATE_CHECK;
```

Notes (Burkey, HALMAT.md):

Allocate run time space and generate code to perform an UPDATE PRIORITY SVC.

SCHD (`0x039`)

SCHEDULE statement (real-time)

Emission (PASS1):

```
SYNTHESI.xpl:1954
    1953  XSET"9";
>>> 1954  CALL HALMAT_POP(XSCHD,PTR_TOP-REFER_LOC+1,0,INX(REFER_LOC));
    1955  DO WHILE REFER_LOC<=PTR_TOP;
```

Other References:

```
? GENCLAS0.xpl:2550
```

Notes (Burkey, HALMAT.md):

Allocate run time space and generate code to perform a SCHEDULE SVC.

ERON (`0x03C`)

ON ERROR

Emission (PASS1):

```
SYNTHESI.xpl:1981
    1980  DO;
>>> 1981  CALL HALMAT_TUPLE(XERON,0,MP,0,FIXL(MP),FIXV(MP)&"3F");
    1982  PTR_TOP=PTR(MP)-1;
SYNTHESI.xpl:1987
    1986  DO;
>>> 1987  CALL HALMAT_TUPLE(XERON,0,MP,MP+2,FIXL(MP),FIXV(MP)&"3F",0,0,
    1988  INX(PTR(MP+2)));
SYNTHESI.xpl:1995
    1994  CALL ERROR_SUB(0);
>>> 1995  CALL HALMAT_TUPLE(XERON,0,MP+2,0,3,FIXV(MP)&"3F");
    1996  PTR_TOP=PTR(MP+2)-1;
SYNTHESI.xpl:2929
    2928  CALL ERROR_SUB(1);
>>> 2929  CALL HALMAT_POP(XERON,2,0,0);
    2930  CALL HALMAT_PIP(LOC_P(PTR(MP+2)),XIMD,FIXV(MP)&"3F",0);
```

Notes (Burkey, HALMAT.md):

Generate code to manipulate runtime error ~;tack for ON ERROR and OFF ERROR statements.

ERSE (`0x03D`)

SEND ERROR

Emission (PASS1):

```
SYNTHESI.xpl:1974
    1973  CALL ERROR_SUB(2);
>>> 1974  CALL HALMAT_TUPLE(XERSE,0,MP+2,0,0,FIXV(MP)&"3F");
    1975  CALL SET_OUTER_REF(FIXV(MP),"0000");
```

Notes (Burkey, HALMAT.md):

Generate SVC instruction to perfornl SEND ERROR.

MSHP (`0x040`)

Matrix shaping function

Emission (PASS1):

```
ENDANYFC.xpl:429
     428  ARG#=PTR(MP);
>>>  429  TEMP2=XMSHP(FCN_LOC(FCN_LV));
     430  IF FCN_LOC(FCN_LV)>=2 THEN DO;  /*  INTEGER AND SCALAR */
```

Notes (Burkey, HALMAT.md):

Allocate runtime temporary and then generate code to perform shaping.

VSHP (`0x041`)

Vector shaping function

Notes (Burkey, HALMAT.md):

Identical to MSHP with ROW=I.

SSHP (`0x042`)

Scalar shaping function

Notes (Burkey, HALMAT.md):

Essentially the same as MSHP.

ISHP (`0x043`)

Integer shaping function

Notes (Burkey, HALMAT.md):

Identical to MSHP (?) with OPTYPE=INTEGER.

SFST (`0x045`)

Subscript range -first element

Emission (PASS1):

```
STARTNOR.xpl:196
     195  CALL SAVE_ARRAYNESS;
>>>  196  CALL HALMAT_POP(XSFST,0,XCO_N,FCN_LV);
     197  END;
SYNTHESI.xpl:1599
    1598  CALL SAVE_ARRAYNESS;
>>> 1599  CALL HALMAT_POP(XSFST,0,XCO_N,FCN_LV);
    1600  VAL_P(PTR_TOP)=LAST_POP#;
```

Optimisation (OPT):

```
PREPAREH.xpl:148
     147  ELSE
>>>  148  IF OPRTR = XSFST | OPRTR = XXXST THEN DO;
     149  NEST_LEVEL(OPTAG) = NEST_LEVEL;
PREPAREH.xpl:225
     224  CALL OPDECODE(CTR);
>>>  225  IF OPRTR = XSFST | OPRTR = XXXST THEN DO;
     226  NEST_LEVEL(OPTAG) = NEST_LEVEL;
PREPAREH.xpl:284
     283  END;
>>>  284  ELSE IF OPRTR = XXXST | OPRTR = XSFST THEN DO;
     285  NEST_LEVEL(OPTAG) = FUNC_LEVEL;
```

Notes (Burkey, HALMAT.md):

Set up call stack for shaping function call.

SFND (`0x046`)

Subscript range -final element

Emission (PASS1):

```
ENDANYFC.xpl:488
     487  CALL SETUP_VAC(MP,PSEUDO_TYPE(ARG#));
>>>  488  CALL HALMAT_POP(XSFND,0,XCO_N,FCN_LV);
     489  END_ARITH_SHAPERS:
ENDANYFC.xpl:563
     562  CALL SETUP_VAC(MP,I);
>>>  563  CALL HALMAT_POP(XSFND,0,XCO_N,FCN_LV);
     564  CALL RESET_ARRAYNESS;
```

Optimisation (OPT):

```
PREPAREH.xpl:155
     154  END;
>>>  155  ELSE IF OPRTR = XSFND | OPRTR = XXXND THEN DO;
     156  FLAG_LOC = START(OPTAG);
PREPAREH.xpl:235
     234  END;
>>>  235  ELSE IF OPRTR = XSFND | OPRTR = XXXND THEN DO;
     236  NEST_LEVEL = NEST_LEVEL(OPTAG);
PREPAREH.xpl:290
     289  END;
>>>  290  ELSE IF OPRTR = XXXND | OPRTR = XSFND THEN
     291  FUNC_LEVEL = NEST_LEVEL(OPTAG);
```

Notes (Burkey, HALMAT.md):

Pop up call stack.

SFAR (`0x047`)

Subscript range -array

Emission (PASS1):

```
SETUPCAL.xpl:306
     305  END;
>>>  306  CALL HALMAT_TUPLE(XSFAR,XCO_N,SP,0,FCN_LV);
     307  CALL HALMAT_FIX_PIPTAGS(NEXT_ATOM#-1,PSEUDO_TYPE(I),0);
SETUPCAL.xpl:334
     333  ELSE DO;
>>>  334  CALL HALMAT_TUPLE(XSFAR,XCO_N,SP,0,FCN_LV,PSEUDO_TYPE(I),
     335  SHR(BI_FLAGS(FCN_LOC(FCN_LV)),4)&"1");
```

Other References:

```
? GENCLAS0.xpl:2764
```

Notes (Burkey, HALMAT.md):

Stack argument for later processing by shaping functions.

BFNC (`0x04A`)

Built-in function call

Emission (PASS1):

```
ENDANYFC.xpl:413
     412  END;
>>>  413  CALL HALMAT_POP(XBFNC,ARG#,0,FCN_LOC(FCN_LV));
     414  DO I = MAXPTR TO MAXPTR + ARG# - 1;
SETUPNOA.xpl:210
     209  BI_LOC(FIXL(MP)),10 ) );          /*CR13335*/
>>>  210  CALL HALMAT_POP(XBFNC,0,0,FIXL(MP));
     211  CALL SETUP_VAC(MP,SHR(BI_INFO,24));
```

Optimisation (OPT):

```
TWINHALM.xpl:42
      41  OP = OPR(PTR);
>>>   42  IF (OP & "FFF1") ^= XBFNC THEN RETURN FALSE;
      43  RETURN OP >= SINCOS AND OP < SINCOS + TWIN#;
```

Notes (Burkey, HALMAT.md):

Generate code to call (or perform in-line) built-in function.

LFNC (`0x04B`)

Library function call

Emission (PASS1):

```
ENDANYFC.xpl:559
     558  IF FCN_ARG(FCN_LV)>1 THEN CALL ERROR(CLASS_FN,4,VAR(MP));
>>>  559  CALL HALMAT_POP(XLFNC,1,0,FCN_LV);
     560  CALL HALMAT_PIP(FCN_LOC(FCN_LV),XIMD,I,0);
```

Notes (Burkey, HALMAT.md):

Get runtime temporary and generate code to perform library call for list-type built-in functions.

TNEQ (`0x04D`)

Name comparison -not equal (task/event)

Notes (Burkey, HALMAT.md):

Set up stack entries for the structures to be compared. Set up branch points one way or the other depending on whether this is TNEQ or TEQU. Generate code to compare the entirety of the two structures.

TEQU (`0x04E`)

Name comparison -equal (task/event)

Emission (PASS1):

```
SYNTHESI.xpl:2311
    2310  DO ;
>>> 2311  TEMP=XTEQU(REL_OP);
    2312  VAR(MP)='STRUCTURE';
```

TASN (`0x04F`)

Task/event name assignment

Other References:

```
? GENCLAS0.xpl:2810
```

Notes (Burkey, HALMAT.md):

Generate code to copy the structure .

IDEF (0x051)

Inline function definition header

Emission (PASS1):

```
SYNTHESI.xpl:4183
    4182  VAR_LENGTH(I)=TEMP;
>>> 4183  CALL HALMAT_POP(XIDEF,1,0,INLINE_LEVEL);
    4184  CALL HALMAT_PIP(I,XSYT,0,0);
```

Code Generation (PASS2):

```
OPTIMISE.xpl:136
     135  IF OPRTR=XXREC THEN RETURN;
>>>  136  IF OPRTR = XIDEF THEN IFLAG = 1;
     137  ELSE IF OPRTR = XADLP THEN VDLP_DETECTED = SHR(OPR(SMRK_CTR+OPNUM),8);
```

Optimisation (OPT):

```
PREPAREH.xpl:175
     174  END;
>>>  175  ELSE IF OPRTR = XIDEF THEN DO;
     176  IFLAG = 1;
```

Notes (Burkey, HALMAT.md):

Generate code to save whatever is necessary, open the block, and set aside space to receive inline function result.

ICLS (0x052)

Inline function close

Emission (PASS1):

```
SYNTHESI.xpl:1633
    1632  TEMP2=INLINE_LEVEL;
>>> 1633  TEMP=XICLS;
    1634  GRAMMAR_FLAGS(STACK_PTR(SP))=GRAMMAR_FLAGS(STACK_PTR(SP))|INLINE_FLAG;
```

Optimisation (OPT):

```
PREPAREH.xpl:183
     182  END;
>>>  183  ELSE IF OPRTR = XICLS THEN DO;
     184  IF FUNC_LEVEL = OPTAG THEN DO;
```

Notes (Burkey, HALMAT.md):

Close block to finish off inline function.

NNEQ (`0x055`)

Name not-equal comparison

Code Generation (PASS2):

```
GENERATE.xpl:1476
    1475  /* NAME COMPARISON OR A NAME ASSIGNMENT                    */
>>> 1476  IF (HALMAT_OPCODE = XNNEQ) | (HALMAT_OPCODE = XNEQU) |
    1477  (HALMAT_OPCODE = XNASN) THEN RETURN TRUE;
```

NEQU (`0x056`)

Name equality comparison

Emission (PASS1):

```
SYNTHESI.xpl:2319
    2318  CALL NAME_COMPARE(MP,SP,CLASS_C,4);
>>> 2319  TEMP=XNEQU(REL_OP);
    2320  VAR(MP)='NAME';
```

Code Generation (PASS2):

```
GENERATE.xpl:1476
    1475  /* NAME COMPARISON OR A NAME ASSIGNMENT                    */
>>> 1476  IF (HALMAT_OPCODE = XNNEQ) | (HALMAT_OPCODE = XNEQU) |
    1477  (HALMAT_OPCODE = XNASN) THEN RETURN TRUE;
```

NASN (`0x057`)

Name assignment

Emission (PASS1):

```
SYNTHESI.xpl:2446
    2445  CALL NAME_COMPARE(MP,SP,CLASS_AV,5);
>>> 2446  CALL HALMAT_TUPLE(XNASN,0,SP,MP,0);
    2447  IF COPINESS(MP,SP)>2 THEN CALL ERROR(CLASS_AA,1);
```

Code Generation (PASS2):

```
GENERATE.xpl:1477
    1476  IF (HALMAT_OPCODE = XNNEQ) | (HALMAT_OPCODE = XNEQU) |
>>> 1477  (HALMAT_OPCODE = XNASN) THEN RETURN TRUE;
    1478
```

Notes (Burkey, HALMAT.md):

Generate code to compare the two NAME operands and jump accordingly.

PMHD (`0x059`)

Percent macro header

Emission (PASS1):

```
SYNTHESI.xpl:2001
    2000  DO;
>>> 2001  CALL HALMAT_POP(XPMHD,0,0,FIXL(MP));
    2002  CALL HALMAT_POP(XPMIN,0,0,FIXL(MP));
SYNTHESI.xpl:2042
    2041  XSET (PC_STMT_TYPE_BASE + FIXL(MP));
>>> 2042  CALL HALMAT_POP(XPMHD,0,0,FIXL(MP));
    2043  DELAY_CONTEXT_CHECK=TRUE;
```

Notes (Burkey, HALMAT.md):

Initialize for %MACRO.

PMAR (`0x05A`)

Percent macro argument

Emission (PASS1):

```
SYNTHESI.xpl:2014
    2013  CALL ERROR(CLASS_XM, 2, VAR(MP));
>>> 2014  CALL HALMAT_TUPLE(XPMAR, 0, MPP1, 0, 0, PSEUDO_TYPE(PTR(MPP1)));
    2015  PTR_TOP=PTR(MPP1)-1;
SYNTHESI.xpl:2049
    2048  DO;
>>> 2049  CALL HALMAT_TUPLE(XPMAR, 0, MPP1, 0, 0, PSEUDO_TYPE(PTR(MPP1)));
    2050  PTR_TOP=PTR(MPP1)-1;
```

Code Generation (PASS2):

```
GENERATE.xpl:1483
    1482  /* %NAMEADD MACRO.                                        */
>>> 1483  IF (HALMAT_OPCODE = XPMAR) THEN DO ;
    1484  IF (PMINDEX = NCOPY_TAG) | ( (PMINDEX = NADD_TAG) & /*DR111390*/
```

Notes (Burkey, HALMAT.md):

Put %MACRO argument into argument stack.

PMIN (`0x05B`)

Percent macro invocation

Emission (PASS1):

```
SYNTHESI.xpl:2002
    2001  CALL HALMAT_POP(XPMHD,0,0,FIXL(MP));
>>> 2002  CALL HALMAT_POP(XPMIN,0,0,FIXL(MP));
    2003  XSET (PC_STMT_TYPE_BASE + FIXL(MP));
SYNTHESI.xpl:2017
    2016  DELAY_CONTEXT_CHECK=FALSE;
>>> 2017  CALL HALMAT_POP(XPMIN,0,0,FIXL(MP));
    2018  ASSIGN_ARG_LIST = FALSE;  /* RESTORE LOCK GROUP CHECKING */
```

Code Generation (PASS2):

```
GENERATE.xpl:1493
    1492  /* %NAMEADD MACRO.                                        */
>>> 1493  IF (HALMAT_OPCODE = XPMIN) THEN DO ;
    1494  IF (TAG = NCOPY_TAG) | ( (TAG = NADD_TAG) &         /*DR111390*/
GENERATE.xpl:6479
    6478
>>> 6479  IF HALMAT_OPCODE = XPMIN & PMINDEX = COPY_TAG /*%COPY*/
    6480  & ^HIGHOPT & ^NAME_VAR(OP) THEN
```

Notes (Burkey, HALMAT.md):

Generate inline code to perform a %MACRO.

Class 1: Bit Operations

Burkey's Notes on Class 1

The following notes are from Burkey's HALMAT.md. They represent early analysis based on ref [2] (the Intermetrics documentation) rather than verified compiler source. Individual opcode entries below supersede these where they overlap, but the high-level notes on TAG semantics and event operations are preserved here for context.

TAG ≠ 0: Event Operations

For event operation. Generated only for real time statements. The code is not built to evaluate the expression. Rather, the events and operators are put together into an argument for an SVC call. The supervisor will actually evaluate the expression.

SUBCODE Dispatch

When TAG = 0, PASS2 dispatches on the SUBCODE field (COPT bits 3:1):

SUBCODE	Opcodes	Notes (Burkey)
0	BASN, BAND, BOR, BNOT, BCAT	Core bit operations
1	BTOB, BTOQ	“Just process subscripts” [2, p. 504]
2	CTOB	“Generate code to transform from character to bit string and then process subscripts” [2, p. 504]
5	STOB, STOQ	Force into accumulator as integer, process subscripts [2, p. 504]
6	ITOB, ITOQ	“Just handle subscripts” [2, p. 504]

BASN (`0x101`)

Bit assignment

No references found in compiler source. This opcode may only be used via computed values or array indexing.

BAND (`0x102`)

Bit AND

Emission (PASS1):

```
SYNTHESI.xpl:2224
    2223  ELSE DO;
>>> 2224  TEMP = XBAND ;
    2225  DO_BIT_FACTOR:
```

BOR (`0x103`)

Bit OR

Emission (PASS1):

```
SYNTHESI.xpl:2245
    2244  ELSE DO;
>>> 2245  TEMP=XBOR;
    2246  GO TO DO_BIT_FACTOR;
```

BNOT (`0x104`)

Bit NOT

Emission (PASS1):

```
SYNTHESI.xpl:2198
    2197  ELSE DO;
>>> 2198  CALL HALMAT_TUPLE(XBNOT,0,SP,0,INX(PTR(SP)));
    2199  CALL SETUP_VAC(SP,BIT_TYPE);
SYNTHESI.xpl:2206
    2205  DO;
>>> 2206  CALL HALMAT_TUPLE(XBNOT,0,SP,0,0);
    2207  CALL SETUP_VAC(SP,BIT_TYPE);
```

BCAT (`0x105`)

Bit concatenation

Emission (PASS1):

```
SYNTHESI.xpl:2184
    2183  END;
>>> 2184  CALL HALMAT_TUPLE(XBCAT,0,MP,SP,0);
    2185  CALL SETUP_VAC(MP,BIT_TYPE,TEMP);
```

BTOB (`0x121`)

Bit to bit conversion

Emission (PASS1):

```
ENDANYFC.xpl:527
     526  ELSE PSEUDO_LENGTH(PTR_TOP)=BIT_LENGTH_LIM;
>>>  527  ARG#=XBTOB(PSEUDO_TYPE(MAXPTR)-BIT_TYPE);
     528  END;
```

BTOQ (`0x122`)

Bit to bit conversion (qualified)

Emission (PASS1):

```
ENDSUBBI.xpl:161
     160  END;
>>>  161  CALL HALMAT_TUPLE(XBTOQ(PSEUDO_TYPE(TEMP)-BIT_TYPE),0,MPP1,0,T);
     162  CALL SETUP_VAC(MP,BIT_TYPE,FIX_DIM);
```

CTOB (`0x141`)

Character to bit conversion

No references found in compiler source. This opcode may only be used via computed values or array indexing.

CTOQ (`0x142`)

Character to bit conversion (qualified)

No references found in compiler source. This opcode may only be used via computed values or array indexing.

STOB (`0x1A1`)

Scalar to bit conversion

No references found in compiler source. This opcode may only be used via computed values or array indexing.

STOQ (`0x1A2`)

Scalar to bit conversion (qualified)

No references found in compiler source. This opcode may only be used via computed values or array indexing.

ITOB (`0x1C1`)

Integer to bit conversion

No references found in compiler source. This opcode may only be used via computed values or array indexing.

ITOQ (`0x1C2`)

Integer to bit conversion (qualified)

No references found in compiler source. This opcode may only be used via computed values or array indexing.

Class 2: Character Operations

CASN (`0x201`)

Character assignment

No references found in compiler source. This opcode may only be used via computed values or array indexing.

CCAT (`0x202`)

Character concatenation (||)

Emission (PASS1):

```
SYNTHESI.xpl:2420
    2419  DO_CHAR_CAT:
>>> 2420  CALL HALMAT_TUPLE(XCCAT,0,MP,SP,0);
    2421  CALL SETUP_VAC(MP,CHAR_TYPE);
```

BTOC (`0x221`)

Bit to character conversion

Emission (PASS1):

```
ARITHTOC.xpl:80
      79  END;
>>>   80  CALL HALMAT_TUPLE(XBTOC(TEMP-BIT_TYPE),0,I,0,0);
      81  CALL SETUP_VAC(I,CHAR_TYPE);
ENDANYFC.xpl:343
     342  IF PSEUDO_TYPE(MAXPTR)^=CHAR_TYPE THEN DO;
>>>  343  CALL HALMAT_TUPLE(XBTOC(PSEUDO_TYPE(MAXPTR)-BIT_TYPE),
     344  0,SP-1,0,0);
ENDANYFC.xpl:350
     349  IF PSEUDO_TYPE(MAXPTR+1)^=CHAR_TYPE THEN DO;
>>>  350  CALL HALMAT_TUPLE(XBTOC(PSEUDO_TYPE(MAXPTR+1)-BIT_TYPE)
     351  ,0,SP,0,0);
ENDANYFC.xpl:515
     514  END;                                  /*DR111376*/
>>>  515  ARG#=XBTOC(PSEUDO_TYPE(MAXPTR)-BIT_TYPE);
     516  END;
```

CTOC (`0x241`)

Character to character conversion

No references found in compiler source. This opcode may only be used via computed values or array indexing.

STOC (`0x2A1`)

Scalar to character conversion

No references found in compiler source. This opcode may only be used via computed values or array indexing.

ITOC (`0x2C1`)

Integer to character conversion

No references found in compiler source. This opcode may only be used via computed values or array indexing.

Class 3: Matrix Arithmetic

MASN (0x301)

Matrix assignment

Code Generation (PASS2):

```
GENERATE.xpl:8205
    8204  IF PART > 0 THEN DO;
>>> 8205  IF OPCODE = XMASN THEN
    8206  CALL FORCE_NUM(FIXARG3, PART, 8);
GENERATE.xpl:8264
    8263  IF DATATYPE(TYPE(OP0))=MATRIX & DEL(OP0)^=0 |
>>> 8264  DATATYPE(TYPE(OP1))=MATRIX & DEL(OP1)^=0 THEN OPCODE = XMASN;
    8265  ELSE OPCODE = XXASN;
```

MTRA (0x329)

Matrix transpose

Emission (PASS1):

```
SYNTHESI.xpl:1436
    1435  IF VAR(SP)='T' THEN DO;
>>> 1436  CALL HALMAT_TUPLE(XMTRA,0,MP,0,0);
    1437  CALL SETUP_VAC(MP,TEMP,SHL(TEMP2,8)|SHR(TEMP2,8));
```

Other References:

```
? GENCLAS0.xpl:4525
```

MTOM (0x341)

Matrix to matrix conversion

Emission (PASS1):

```
PRECSCAL.xpl:114
     113  IF (P_TEMP&"F")^=0 THEN DO;
>>>  114  CALL HALMAT_TUPLE(XMTOM(P_TYPE-MAT_TYPE),0,MP,0,
     115  P_TEMP&"F");
```

MNEG (0x344)

Matrix negation

Emission (PASS1):

```
SYNTHESI.xpl:1109
    1108  TEMP=PSEUDO_TYPE(PTR(SP));
>>> 1109  CALL HALMAT_TUPLE(XMNEG(TEMP-MAT_TYPE),0,SP,0,0);
    1110  CALL SETUP_VAC(SP,TEMP,PSEUDO_LENGTH(PTR(SP)));
```

MADD (0x362)

Matrix addition

Emission (PASS1):

```
ADDANDSU.xpl:132
     131  DO CASE MODE;
>>>  132  MODE=XMADD(TEMP-MAT_TYPE);
     133  MODE=XMSUB(TEMP-MAT_TYPE);
CHECKSUB.xpl:162
     161  IF (MODE&"F")=XIMD THEN DO;
>>>  162  IF NEWSIZE="10" THEN NEWSIZE=XMADD(INT_TYPE-MAT_TYPE);
     163  ELSE NEWSIZE=XMSUB(INT_TYPE-MAT_TYPE);
```

MSUB (0x363)

Matrix subtraction

Emission (PASS1):

```
ADDANDSU.xpl:133
     132  MODE=XMADD(TEMP-MAT_TYPE);
>>>  133  MODE=XMSUB(TEMP-MAT_TYPE);
     134  END;
CHECKSUB.xpl:163
     162  IF NEWSIZE="10" THEN NEWSIZE=XMADD(INT_TYPE-MAT_TYPE);
>>>  163  ELSE NEWSIZE=XMSUB(INT_TYPE-MAT_TYPE);
     164  CALL HALMAT_POP(NEWSIZE,2,0,0);
```

MMPR (0x368)

Matrix-matrix product

Emission (PASS1):

```
MULTIPLY.xpl:198
     197  IF TEMP^=TEMP2 THEN CALL ERROR(CLASS_EM,3);
>>>  198  CALL HALMAT_TUPLE(XMMPR,0,I,J,0);
     199  CALL SETUP_VAC(K,MAT_TYPE,(PSEUDO_LENGTH(PTR(I))&"FF00")|
```

MDET (0x371)

Matrix determinant

Other References:

```
? GENCLAS0.xpl:4538
```

MIDN (0x373)

Matrix identity

Other References:

```
? GENCLAS3.xpl:106
```

VVPR (0x387)

Vector-vector outer product (yielding matrix)

Emission (PASS1):

```
MULTIPLY.xpl:175
     174  DO ;
>>>  175  CALL HALMAT_TUPLE(XVVPR,0,I,J,0);
     176  CALL SETUP_VAC(K,MAT_TYPE,SHL(PSEUDO_LENGTH(PTR(I)),8)+
```

MSPR (0x3A5)

Matrix-scalar product

Emission (PASS1):

```
MULTIPLY.xpl:156
     155  CALL MATCH_SIMPLES(0,J);
>>>  156  CALL HALMAT_TUPLE(XMSPR,0,I,J,0);
     157  CALL SETUP_VAC(K,MAT_TYPE,PSEUDO_LENGTH(PTR(I)));
```

MSDV (0x3A6)

Matrix-scalar division

Emission (PASS1):

```
SYNTHESI.xpl:1140
    1139  TEMP=PSEUDO_TYPE(PTR(MP));
>>> 1140  CALL HALMAT_TUPLE(XMSDV(TEMP-MAT_TYPE),0,MP,SP,0);
    1141  CALL SETUP_VAC(MP,TEMP);
```

MINV (`0x3CA`)

Matrix inverse

Emission (PASS1):

```
SYNTHESI.xpl:1448
    1447  IF (TEMP2&"FF")^=SHR(TEMP2,8) THEN CALL ERROR(CLASS_EM,4);
>>> 1448  CALL HALMAT_TUPLE(XMINV,0,MP,SP,0);
    1449  CALL SETUP_VAC(MP,TEMP);
```

Other References:

```
? GENCLAS0.xpl:4529
? GENCLAS3.xpl:110
```

Class 4: Vector Arithmetic

VASN (`0x401`)

Vector assignment

No references found in compiler source. This opcode may only be used via computed values or array indexing.

VTOV (`0x441`)

Vector to vector conversion

No references found in compiler source. This opcode may only be used via computed values or array indexing.

VNEG (`0x444`)

Vector negation

No references found in compiler source. This opcode may only be used via computed values or array indexing.

MVPR (0x46C)

Matrix-vector product

Emission (PASS1):

```
MULTIPLY.xpl:190
     189  IF TEMP^=PSEUDO_LENGTH(PTR(J)) THEN CALL ERROR(CLASS_EV,2);
>>>  190  CALL HALMAT_TUPLE(XMVPR,0,I,J,0);
     191  CALL SETUP_VAC(K,VEC_TYPE,SHR(PSEUDO_LENGTH(PTR(I)),8));
```

Other References:

```
? GENCLAS4.xpl:29
```

VMPR (0x46D)

Vector-matrix product

Emission (PASS1):

```
MULTIPLY.xpl:183
     182  IF TEMP^=PSEUDO_LENGTH(PTR(I)) THEN CALL ERROR(CLASS_EV,3);
>>>  183  CALL HALMAT_TUPLE(XVMPR,0,I,J,0);
     184  CALL SETUP_VAC(K,VEC_TYPE,PSEUDO_LENGTH(PTR(J))&"FF");
```

VADD (0x482)

Vector addition

No references found in compiler source. This opcode may only be used via computed values or array indexing.

VSUB (0x483)

Vector subtraction

No references found in compiler source. This opcode may only be used via computed values or array indexing.

VCRS (0x48B)

Vector cross product

Emission (PASS1):

```
MULTIPLY.xpl:167
     166  DO ;
>>>  167  CALL HALMAT_TUPLE(XVCRS,0,I,J,0);
     168  CALL SETUP_VAC(K,VEC_TYPE,3);
```

VSPR (0x4A5)

Vector-scalar product

Emission (PASS1):

```
MULTIPLY.xpl:148
     147  CALL MATCH_SIMPLES(0,J);
>>>  148  CALL HALMAT_TUPLE(XVSPR,0,I,J,0);
     149  CALL SETUP_VAC(K,VEC_TYPE,PSEUDO_LENGTH(PTR(I)));
```

Class 5: Scalar Arithmetic

SASN (0x501)

Scalar assignment

Code Generation (PASS2):

```
GENERATE.xpl:8854
    8853  THEN OPCODE = XPASN;
>>> 8854  ELSE OPCODE = XSASN;
    8855  CALL ASSIGN_CLEAR(LEFTOP, 1);
```

Other References:

```
? GENCLAS8.xpl:376
```

BTOS (0x521)

Bit to scalar conversion

Emission (PASS1):

```
ENDANYFC.xpl:438
     437  DO CASE FCN_LOC(FCN_LV)-2;
>>>  438  TEMP=XBTOS(PSEUDO_TYPE(MAXPTR)-BIT_TYPE);
     439  TEMP=XBTOI(PSEUDO_TYPE(MAXPTR)-BIT_TYPE);
```

CTOS (0x541)

Character to scalar conversion

No references found in compiler source. This opcode may only be used via computed values or array indexing.

SIEX (0x571)

Scalar-integer exponentiation

Emission (PASS1):

```
SYNTHESI.xpl:1482
    1481  ELSE IF PSEUDO_FORM(I)^=XLIT THEN DO;
>>> 1482  TEMP2=XSIEX;
    1483  GO TO REGULAR_EXP;
SYNTHESI.xpl:1488
    1487  IF TEMP<0 THEN DO;
>>> 1488  TEMP2=XSIEX;
    1489  GO TO REGULAR_EXP;
```

SPEX (0x572)

Scalar power expression

Emission (PASS1):

```
SYNTHESI.xpl:1472
    1471  POWER_FAIL:
>>> 1472  TEMP2=XSPEX(TEMP-SCALAR_TYPE);
    1473  IF PSEUDO_TYPE(I)<SCALAR_TYPE THEN CALL ERROR(CLASS_E,3);
```

VDOT (0x58E)

Vector dot product

Emission (PASS1):

```
MULTIPLY.xpl:162
     161  CALL VECTOR_COMPARE(I,J,CLASS_EV,1);
>>>  162  CALL HALMAT_TUPLE(XVDOT,0,I,J,0);
     163  CALL SETUP_VAC(K,SCALAR_TYPE);
```

STOS (0x5A1)

Scalar to scalar conversion

No references found in compiler source. This opcode may only be used via computed values or array indexing.

SADD (`0x5AB`)

Scalar addition

No references found in compiler source. This opcode may only be used via computed values or array indexing.

SSUB (`0x5AC`)

Scalar subtraction

No references found in compiler source. This opcode may only be used via computed values or array indexing.

SSPR (`0x5AD`)

Scalar-scalar product

Emission (PASS1):

```
MULTIPLY.xpl:140
     139  CALL MATCH_SIMPLES(I,J);
>>>  140  CALL HALMAT_TUPLE(XSSPR(PSEUDO_TYPE(PTR(I))-SCALAR_TYPE),0,I,J,0);
     141  CALL SETUP_VAC(K,PSEUDO_TYPE(PTR(I)));
```

SSDV (`0x5AE`)

Scalar-scalar division

No references found in compiler source. This opcode may only be used via computed values or array indexing.

SEXP (`0x5AF`)

Scalar exponentiation

Emission (PASS1):

```
SYNTHESI.xpl:1454
    1453  CALL ERROR(CLASS_EV,4);
>>> 1454  TEMP2=XSEXP;
    1455  GO TO FINISH_EXP;
SYNTHESI.xpl:1475
    1474  IF PSEUDO_TYPE(I)^=INT_TYPE THEN DO;
>>> 1475  TEMP2=XSEXP;
    1476  REGULAR_EXP:
```

SNEG (`0x5B0`)

Scalar negation

No references found in compiler source. This opcode may only be used via computed values or array indexing.

ITOS (`0x5C1`)

Integer to scalar conversion

Emission (PASS1):

```
ENDANYFC.xpl:379
     378  IF PSEUDO_TYPE(MAXPTR)=INT_TYPE THEN DO;
>>>  379  CALL HALMAT_TUPLE(XITOS,0,SP-1,0,0);
     380  CALL SETUP_VAC(SP-1,SCALAR_TYPE);
ENDANYFC.xpl:383
     382  IF ARG#>=2 THEN IF PSEUDO_TYPE(MAXPTR+1)=INT_TYPE THEN DO;
>>>  383  CALL HALMAT_TUPLE(XITOS, 0, SP, 0, 0);
     384  CALL SETUP_VAC(SP, SCALAR_TYPE);
ENDANYFC.xpl:387
     386  IF ARG#=3 THEN IF PSEUDO_TYPE(MAXPTR+2)=INT_TYPE THEN DO;
>>>  387  CALL HALMAT_TUPLE(XITOS, 0, SP-2, 0, 0);
     388  CALL SETUP_VAC(SP-2, SCALAR_TYPE);
MATCHSIM.xpl:75
      74  IF T2=INT_TYPE THEN LOC1=LOC2;
>>>   75  CALL HALMAT_TUPLE(XITOS,0,LOC1,0,0);
      76  CALL SETUP_VAC(LOC1,SCALAR_TYPE);
UNARRAY2.xpl:66
      65  IF PSEUDO_TYPE(PTR(LOC))=INT_TYPE THEN DO;
>>>   66  CALL HALMAT_TUPLE(XITOS,0,LOC,0,0);
      67  CALL SETUP_VAC(LOC,SCALAR_TYPE);
```

Class 6: Integer Arithmetic

Classes 1 through 6 share a parallel naming convention. Each class covers one HAL/S data type and provides the same basic operations: assignment (xASN), arithmetic (xADD, xSUB, xPR for multiply, xDV for divide), negation (xNEG), and type conversions (xTOy). The opcode numbers within each class are assigned independently, but the operation names follow a consistent pattern: replace the type prefix to find the equivalent in another class (e.g., IASN, SASN, VASN, MASN are all assignment).

IASN (`0x601`)

Integer assignment.

Field	Value
NUMOP	2
op1	Source value: VAC(n), SYT(n), or LIT(n)
op2	Destination variable: SYT(n)

Operand order is (source, destination) - verified across 10+ compiled examples. The source is typically a VAC back-pointer to the result of a prior arithmetic operation.

Binary evidence (test_simple_do.hal, addr 24):

```
00026010  INT  IASN  (2 ops)
  op1: VAC(21)      -- result of IADD at addr 21
  op2: SYT(2)=X     -- destination variable X
```

This encodes `X = X + Y` (the IADD produced the sum, IASN stores it).

BTOI (`0x621`)

Bit to integer conversion

Emission (PASS1):

```
ENDANYFC.xpl:439
     438  TEMP=XBTOS(PSEUDO_TYPE(MAXPTR)-BIT_TYPE);
>>>  439  TEMP=XBTOI(PSEUDO_TYPE(MAXPTR)-BIT_TYPE);
     440  END;
```

CTOI (`0x641`)

Character to integer conversion

No references found in compiler source. This opcode may only be used via computed values or array indexing.

STOI (`0x6A1`)

Scalar to integer conversion

Emission (PASS1):

```
CHECKSUB.xpl:155
     154  IF PSEUDO_TYPE(NEXT_SUB)=SCALAR_TYPE THEN DO;
>>>  155  CALL HALMAT_POP(XSTOI,1,0,0);
     156  SHARP_ELIM:
ENDANYFC.xpl:356
     355  ELSE IF PSEUDO_TYPE(MAXPTR+1)=SCALAR_TYPE THEN DO;
>>>  356  CALL HALMAT_TUPLE(XSTOI,0,SP,0,0);
     357  CALL SETUP_VAC(SP,INT_TYPE);
ENDANYFC.xpl:395
     394  IF PSEUDO_TYPE(MAXPTR)=SCALAR_TYPE THEN DO;
>>>  395  CALL HALMAT_TUPLE(XSTOI,0,SP-1,0,0);
     396  CALL SETUP_VAC(SP-1,INT_TYPE);
ENDANYFC.xpl:399
     398  IF ARG#=2 THEN IF PSEUDO_TYPE(MAXPTR+1)=SCALAR_TYPE THEN DO;
>>>  399  CALL HALMAT_TUPLE(XSTOI,0,SP,0,0);
     400  CALL SETUP_VAC(SP,INT_TYPE);
UNARRAYE.xpl:66
      65  IF PSEUDO_TYPE(PTR(LOC))=SCALAR_TYPE THEN DO;
>>>   66  CALL HALMAT_TUPLE(XSTOI,0,LOC,0,0);
      67  CALL SETUP_VAC(LOC,INT_TYPE);
```

ITOI (0x6C1)

Integer to integer conversion

No references found in compiler source. This opcode may only be used via computed values or array indexing.

IADD (0x6CB)

Integer addition. Result available as VAC for subsequent operations.

Field	Value
NUMOP	2
op1	First addend: SYT(n) or LIT(n)
op2	Second addend: SYT(n) or LIT(n)

The compiler does not emit direct references to IADD - the opcode value is computed at runtime from the expression type and operation. PASS1 calls `HALMAT_POP` with a base opcode that is adjusted by the type, which is why no static cross-reference appears.

Binary evidence (test_while.hal, addr 31, encoding `TOTAL = TOTAL + I`):

```
00026CB0  INT  IADD  (2 ops)
  op1: SYT(4)=TOTAL
  op2: SYT(2)=I
```

Binary evidence (test_nested.hal, addr 42, encoding `K = K + 1`):

```
00026CB0  INT  IADD  (2 ops)
  op1: SYT(4)=K
  op2: LIT(7)=1.0
```

ISUB (`0x6CC`)

Integer subtraction. Result available as VAC for subsequent operations.

Field	Value
NUMOP	2
op1	Minuend: SYT(n) or LIT(n)
op2	Subtrahend: SYT(n) or LIT(n)

Binary evidence (test_while.hal, addr 60, encoding `CNT = CNT - 1`):

```
00026CC0  INT  ISUB  (2 ops)
  op1: SYT(3)=CNT
  op2: LIT(7)=1.0
```

IIPR (`0x6CD`)

Integer-integer product. Result available as VAC for subsequent operations.

Field	Value
NUMOP	2
op1	First factor: SYT(n), LIT(n), or VAC(n)
op2	Second factor: SYT(n), LIT(n), or VAC(n)

No compiled examples in current test suite. Opcode value computed at runtime from type and operation.

INEG (`0x6D0`)

Integer negation

No references found in compiler source. This opcode may only be used via computed values or array indexing.

IPEX (`0x6D2`)

Integer power expression

No references found in compiler source. This opcode may only be used via computed values or array indexing.

Class 7: Conditional and Comparison

Class 7 handles all comparisons and boolean connectives. Unlike Classes 1-6 which are organised by data type, Class 7 contains comparison operators for every type, grouped by type prefix: B (bit), C (character), M (matrix), V (vector), S (scalar), I (integer). Each type has the same six comparison operators: NEQ, EQU, NGT, GT, NLT, LT.

All comparison operators produce a boolean result available as VAC for use by FBRA (branch on false) or CTST (loop condition). NUMOP is always 2.

Field	Value
NUMOP	2
op1	Left operand: SYT(n), LIT(n), or VAC(n)
op2	Right operand: SYT(n), LIT(n), or VAC(n)

BTRU (`0x720`)

Bit true test

Emission (PASS1):

```
SYNTHESI.xpl:2610
    2609  DO;
>>> 2610  CALL HALMAT_TUPLE(XBTRU,0,MPP1,0,0);
    2611  IF PSEUDO_LENGTH(PTR(MPP1))>1 THEN CALL ERROR(CLASS_GB,1,'IF');
SYNTHESI.xpl:2786
    2785  IF PSEUDO_LENGTH(PTR(SP))>1 THEN CALL ERROR(CLASS_GB,1,'WHILE/UNTIL');
>>> 2786  CALL HALMAT_TUPLE(XBTRU,0,SP,0,0);
    2787  CALL SETUP_VAC(SP,BIT_TYPE);
```

Other References:

```
? GENCLAS1.xpl:140
```

BNEQ (`0x725`)

Bit not equal

Optimisation (OPT):

```
COMPARET.xpl:35
      34  PTR = OPR(PTR) & "FFF1";
>>>   35  RETURN PTR >= XBNEQ AND PTR <= XILT;
      36  END COMPARE_TYPE;
PREPAREH.xpl:166
     165  OPRTR = OPR(TMP) & "FFF1";
>>>  166  IF OPRTR >= XBNEQ THEN
     167  IF OPRTR <= XILT THEN
```

BEQU (0x726)

Bit equal

Emission (PASS1):

```
SYNTHESI.xpl:2305
    2304  DO ;
>>> 2305  TEMP=XBEQU(REL_OP);
    2306  VAR(MP)='BIT';
```

CNEQ (0x745)

Character not equal

No references found in compiler source. This opcode may only be used via computed values or array indexing.

CEQU (0x746)

Character equal

Emission (PASS1):

```
SYNTHESI.xpl:2299
    2298  DO ;
>>> 2299  TEMP=XCEQU(REL_OP);
    2300  VAR(MP)='';
```

CNGT (0x747)

Character not greater than

No references found in compiler source. This opcode may only be used via computed values or array indexing.

CGT (`0x748`)

Character greater than

No references found in compiler source. This opcode may only be used via computed values or array indexing.

CNLT (`0x749`)

Character not less than

No references found in compiler source. This opcode may only be used via computed values or array indexing.

CLT (`0x74A`)

Character less than

No references found in compiler source. This opcode may only be used via computed values or array indexing.

MNEQ (`0x765`)

Matrix not equal

No references found in compiler source. This opcode may only be used via computed values or array indexing.

MEQU (`0x766`)

Matrix equal

Emission (PASS1):

```
SYNTHESI.xpl:2278
    2277  DO;
>>> 2278  TEMP=XMEQU(REL_OP);
    2279  VAR(MP)='MATRIX';
```

VNEQ (`0x785`)

Vector not equal

No references found in compiler source. This opcode may only be used via computed values or array indexing.

VEQU (`0x786`)

Vector equal

Emission (PASS1):

```
SYNTHESI.xpl:2282
    2281  DO;
>>> 2282  TEMP=XVEQU(REL_OP);
    2283  VAR(MP)='VECTOR';
```

SNEQ (`0x7A5`)

Scalar not equal

No references found in compiler source. This opcode may only be used via computed values or array indexing.

SEQU (`0x7A6`)

Scalar equal

Emission (PASS1):

```
SYNTHESI.xpl:2286
    2285  DO;
>>> 2286  TEMP=XSEQU(REL_OP);
    2287  VAR(MP)='';
```

SNGT (`0x7A7`)

Scalar not greater than

No references found in compiler source. This opcode may only be used via computed values or array indexing.

SGT (`0x7A8`)

Scalar greater than

No references found in compiler source. This opcode may only be used via computed values or array indexing.

SNLT (`0x7A9`)

Scalar not less than

No references found in compiler source. This opcode may only be used via computed values or array indexing.

SLT (`0x7AA`)

Scalar less than

No references found in compiler source. This opcode may only be used via computed values or array indexing.

INEQ (`0x7C5`)

Integer not equal

No references found in compiler source. This opcode may only be used via computed values or array indexing.

IEQU (`0x7C6`)

Integer equal.

Binary evidence (test_while.hal, addr 53, encoding `DO UNTIL CNT = 0`):

```
00027C62  COND  IEQU  (2 ops)
  op1: SYT(3)=CNT
  op2: LIT(6)=0.0
```

Result at VAC(53) consumed by CTST at addr 56.

Emission (PASS1):

```
SYNTHESI.xpl:2290
    2289  DO;
>>> 2290  TEMP=XIEQU(REL_OP);
    2291  VAR(MP)='';
```

INGT (`0x7C7`)

Integer not greater than

No references found in compiler source. This opcode may only be used via computed values or array indexing.

IGT (`0x7C8`)

Integer greater than.

Binary evidence (test_nested.hal, addr 34, encoding `IF I > 5`):

```
00027C82  COND  IGT  (2 ops)
  op1: SYT(2)=I
  op2: LIT(6)=5.0
```

Result at VAC(34) consumed by FBRA at addr 37.

INLT (0x7C9)

Integer not less than

No references found in compiler source. This opcode may only be used via computed values or array indexing.

ILT (0x7CA)

Integer less than.

Binary evidence (test_nested.hal, addr 19, encoding `DO WHILE J < 100`):

```
00027CA2  COND  ILT  (2 ops)
  op1: SYT(3)=J
  op2: LIT(3)=100.0
```

Result at VAC(19) consumed by CTST at addr 22.

Optimisation (OPT):

```
COMPARET.xpl:35
      34  PTR = OPR(PTR) & "FFF1";
>>>   35  RETURN PTR >= XBNEQ AND PTR <= XILT;
      36  END COMPARE_TYPE;
PREPAREH.xpl:167
     166  IF OPRTR >= XBNEQ THEN
>>>  167  IF OPRTR <= XILT THEN
     168  OPR(TMP) = OPR(TMP) | "1000000";  /* SETS TAG = 1  */
```

CAND (`0x7E2`)

Conditional AND

Emission (PASS1):

```
SYNTHESI.xpl:2329
    2328  DO ;
>>> 2329  CALL HALMAT_TUPLE(XCAND,XCO_N,MP,SP,0);
    2330  CALL SETUP_VAC(MP,0);
```

Optimisation (OPT):

```
ANDORTYP.xpl:36
      35  PTR = OPR(PTR) & "FFF1";
>>>   36  RETURN PTR = XCAND   OR    PTR = XCOR;
      37  END ANDOR_TYPE;
```

COR (`0x7E3`)

Conditional OR

Emission (PASS1):

```
SYNTHESI.xpl:2337
    2336  DO ;
>>> 2337  CALL HALMAT_TUPLE(XCOR,XCO_N,MP,SP,0);
    2338  CALL SETUP_VAC(MP,0);
```

Optimisation (OPT):

```
ANDORTYP.xpl:36
      35  PTR = OPR(PTR) & "FFF1";
>>>   36  RETURN PTR = XCAND   OR    PTR = XCOR;
      37  END ANDOR_TYPE;
```

CNOT (`0x7E4`)

Conditional NOT

Emission (PASS1):

```
SYNTHESI.xpl:2345
    2344  DO ;
>>> 2345  CALL HALMAT_TUPLE(XCNOT,XCO_N,MP+2,0,0);
    2346  PTR(MP)=PTR(MP+2);
```

Optimisation (OPT):

```
NOTTYPE.xpl:32
      31  DECLARE PTR BIT(16);
>>>   32  RETURN (OPR(PTR) & "FFF1") = XCNOT;
      33  END NOT_TYPE;
```

Class 8: Initialisation

STRI (`0x801`)

Start of repeated initial list

Emission (PASS1):

```
ICQOUTPU.xpl:116
     115  END;
>>>  116  CALL HALMAT_POP(XSTRI,1,0,0);
     117  CALL HALMAT_PIP(CT,K,0,0);
```

Notes (Burkey, HALMAT.md):

TBD

SLRI (`0x802`)

Start of literal repeated initial list

Emission (PASS1):

```
ICQOUTPU.xpl:152
     151  IF IC_FORM(K)=1 THEN DO;  /*  SLRI  */
>>>  152  CALL HALMAT_POP(XSLRI,2,0,IC_VAL(K));
     153  CALL HALMAT_PIP(IC_LOC(K),XIMD,0,0);
```

Notes (Burkey, HALMAT.md):

TBD

ELRI (0x803)

End of literal repeated initial list

Emission (PASS1):

```
ICQOUTPU.xpl:156
     155  END;
>>>  156  ELSE CALL HALMAT_POP(XELRI,0,0,IC_VAL(K));  /* ELRI  */
     157  END;
```

Notes (Burkey, HALMAT.md):

TBD

ETRI (0x804)

End of repeated initial list

Emission (PASS1):

```
ICQOUTPU.xpl:160
     159  IF CT_LIT>0 THEN CALL HALMAT_FIX_PIPTAGS(NEXT_ATOM#-1,CT_LIT,0);
>>>  160  CALL HALMAT_POP(XETRI,0,0,0);
     161  END ICQ_OUTPUT;
```

Notes (Burkey, HALMAT.md):

TBD

BINT (`0x821`)

Bit initialisation value

Emission (PASS1):

```
ICQCHECK.xpl:76
      75  CALL ERROR(CLASS_DI,6,VAR(MP));
>>>   76  RETURN XBINT(CHAR_TYPE-BIT_TYPE);
      77  END;
ICQCHECK.xpl:82
      81  CALL ERROR(CLASS_DI,7,VAR(MP));
>>>   82  RETURN XBINT;
      83  END;
ICQCHECK.xpl:88
      87  CALL ERROR(CLASS_DI,7,VAR(MP));
>>>   88  RETURN XBINT;
      89  END;
ICQCHECK.xpl:93
      92  CALL ERROR(CLASS_DI,8,VAR(MP));
>>>   93  IF K THEN RETURN XBINT(SYT_TYPE(SYT)-BIT_TYPE);
      94  RETURN XBINT(SCALAR_TYPE-BIT_TYPE);
ICQCHECK.xpl:94
      93  IF K THEN RETURN XBINT(SYT_TYPE(SYT)-BIT_TYPE);
>>>   94  RETURN XBINT(SCALAR_TYPE-BIT_TYPE);
      95
```

Notes (Burkey, HALMAT.md):

TBD

CINT (`0x841`)

Character initialisation value

Notes (Burkey, HALMAT.md):

TBD

MINT (0x861)

Matrix initialisation value

Notes (Burkey, HALMAT.md):

TBD

VINT (0x881)

Vector initialisation value

Notes (Burkey, HALMAT.md):

TBD

SINT (0x8A1)

Scalar initialisation value

Notes (Burkey, HALMAT.md):

TBD

IINT (0x8C1)

Integer initialisation value

Notes (Burkey, HALMAT.md):

TBD

NINT (`0x8E1`)

Name initialisation value

Emission (PASS1):

```
ICQCHECK.xpl:70
      69  I=IC_TYPE(J)&"7F";
>>>   70  IF NAME_IMPLIED THEN RETURN XNINT;                /*DR109044*/
      71  IF SYT_TYPE(ID_LOC)=MAJ_STRUC THEN RETURN XTINT;  /*DR109044*/
```

Notes (Burkey, HALMAT.md):

TBD

TINT (`0x8E2`)

Structure initialisation value

Emission (PASS1):

```
ICQCHECK.xpl:71
      70  IF NAME_IMPLIED THEN RETURN XNINT;                /*DR109044*/
>>>   71  IF SYT_TYPE(ID_LOC)=MAJ_STRUC THEN RETURN XTINT;  /*DR109044*/
      72  ELSE SYT=ID_LOC;                                  /*DR109044*/
```

Notes (Burkey, HALMAT.md):

TBD

EINT (`0x8E3`)

End of initialisation

Emission (PASS1):

```
SYNTHESI.xpl:4642
    4641  CALL ERROR(CLASS_DU, 12, VAR(SP - 1));
>>> 4642  CALL HALMAT_POP(XEINT, 2, 0, PSEUDO_TYPE(TEMP));
    4643  CALL HALMAT_PIP(FIXL(MP + 2), XSYT, 0, 0);
```

Notes (Burkey, HALMAT.md):

TBD

Control Flow Patterns

Individual HALMAT opcodes don't mean anything by themselves. A DFOR without an EFOR is half a thought. An FBRA without its LBL is a jump to nowhere. The compiler emits these opcodes in structured groups - paired start/end markers with bodies between them - and the semantics of each opcode depend entirely on where it sits within the group.

None of this was documented. The only way to figure out how these groups work was to trace the compiler from both ends: PASS1's SYNTHESI.xpl (which emits the patterns during parsing) and PASS2's GENCLAS0.xpl (which tears them apart to generate AP-101 machine code). Then compile actual HAL/S programs, disassemble the binary output, and verify that what the source says matches what the compiler actually produces.

Every pattern documented here has been verified against real HALMAT binary compiled by the HAL/S-FC compiler. The worked examples are not hypothetical - they are annotated disassembly of compiled test programs, with raw hex words and operand encodings traced back to the source.

XPL Boolean Semantics. A trap for anyone reading the compiler source: XPL tests only the **least significant bit** (bit 0) of an expression in boolean context. `IF X THEN` is true when `X & 1 = 1`, not when `X != 0`. This means `IF 2 THEN` is **false** in XPL. This matters throughout the DFOR handler and elsewhere, where multi-valued TAG fields are tested with innocent-looking `IF field THEN` guards that actually only check bit 0. It tripped up multiple analyses before we caught it.

FOR Loop - `DFOR` / `EFOR` / `CFOR` / `AFOR`

The `DO FOR` loop compiles to a DFOR–EFOR pair, with optional CFOR (for WHILE/UNTIL conditions) and AFOR (for discrete value lists).

TAG Encoding

DFOR's TAG field encodes the loop type in a bitfield:

```
TAG bits:  [7:5 unused][4 WHILE_TYPE][3 TEMP][2:0 DOTYPE]
```

Field	Bits	Values
DOTYPE	2:0	0 = discrete (value list), 1 = iterative (implicit BY 1), 2 = iterative (explicit BY)
TEMP	3	1 = loop variable is compiler-allocated TEMPORARY
WHILE_TYPE	4	0 = no condition or WHILE, 1 = UNTIL

Source: PASS1 sets DOTYPE from `TEMP2` in SYNTHESI.xpl. `TEMP2=1` for `TO expr` (implicit step), `TEMP2=2` for `TO expr BY expr` (explicit step), `TEMP2=0` is never set for iterative (it falls to the discrete path). The TEMP bit comes from `SHR(TAG,3)` in GENCLAS0.xpl:3240. The WHILE_TYPE comes from `SHL(INX(TEMP),4)` in SYNTHESI.xpl:2674, where `INX(TEMP)` is 0 for WHILE, 1 for UNTIL. For WHILE loops, bit 4 is clear - the presence of a CFOR operator (not the DFOR TAG) signals that a condition exists. Bit 4 is only set for UNTIL, which needs special first-iteration handling (skip the condition check on entry).

PASS2 extracts these fields (GENCLAS0.xpl:3226–3241):

```
DOTYPE = TAG & "3";          /* bits 2:0 */
IF SHR(TAG, 4) THEN ...      /* bit 4: has WHILE/UNTIL */
IF SHR(TAG, 3) THEN ...      /* bit 3: TEMPORARY */
```

Operand Layout

DOTYPE	NUMOP	Operands
0 (discrete)	2	op1=INL(exit), op2=SYT(loop_var)
1 (iter, step=1)	4	op1=INL(exit), op2=SYT(loop_var), op3=from, op4=to
2 (iter, explicit)	5	op1=INL(exit), op2=SYT(loop_var), op3=from, op4=to, op5=step

Operand qualifiers for op3/op4/op5 are typically LIT (literal constant) or VAC (result of a prior expression).

INL operand detail: The INL value is a flow label number assigned by EMIT_PUSH_DO. Five consecutive flow labels are reserved (FL_NO through FL_NO+4):

Label	Purpose
DOLBL+0	Loop exit point
DOLBL+1	CFOR label / WHILE test (if present)
DOLBL+2	Loop body start
DOLBL+3	UNTIL skip label
DOLBL+4	Iterative entry point (first-iteration comparison)

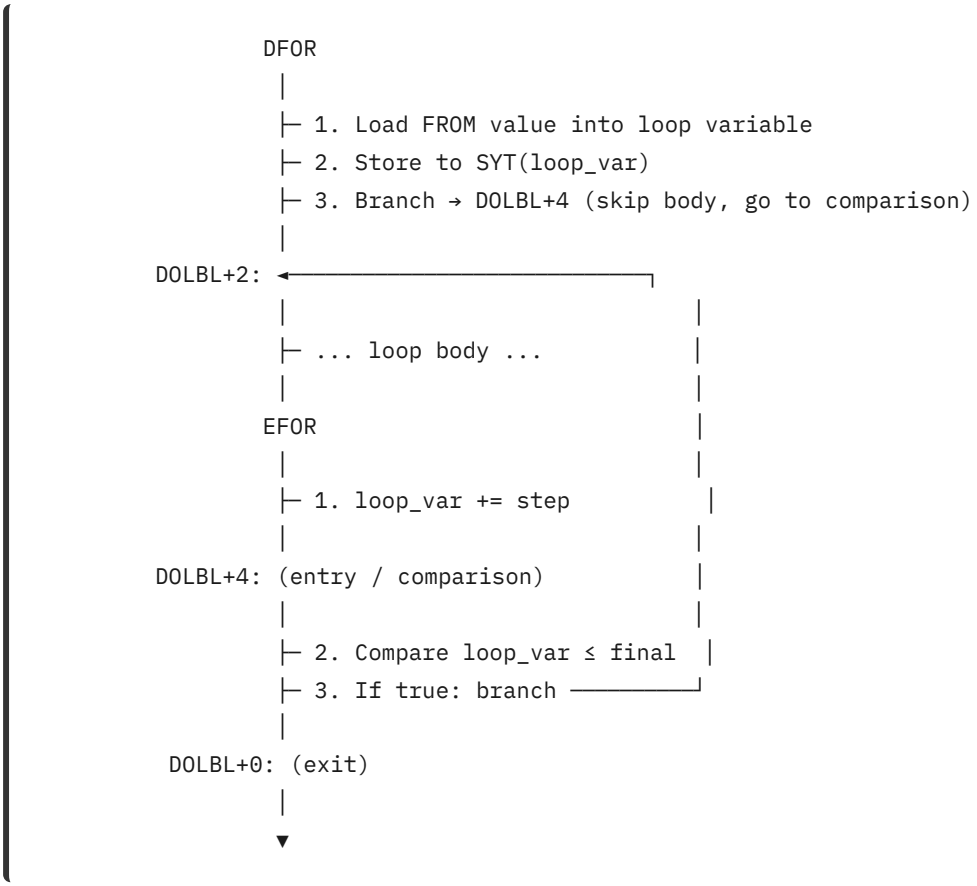
Step Determination (XPL Bit-Test)

PASS2 determines whether to read an explicit step from the HALMAT stream or synthesise an implicit step=1 with this test (GENCLAS0.xpl:3253):

```
IF DOTYPE THEN
   DOFORINCR = GET_LIT_ONE(LITTYPE);
ELSE DOFORINCR = GET_OPERAND(5);
```

Because XPL tests only bit 0: DOTYPE=1 (0b01) is true → implicit step; DOTYPE=2 (0b10) is false → `GET_OPERAND(5)` reads the explicit BY value.

Execution Flow (Iterative)



On the first iteration, DFOR branches directly to DOLBL+4, so the initial value is compared against the final value before the body executes. This produces a bottom-tested loop with one conditional branch per iteration.

The comparison direction depends on the sign of the step value (GENCLAS0.xpl:3365–3366):

- Positive step: `LQ` (less than or equal) → continue while `loop_var ≤ final`
- Negative step: `GQ` (greater than or equal) → continue while `loop_var ≥ final`

Execution Flow (Discrete)

For `DO FOR I = val1, val2, val3;` , the compiler emits:

```
DFOR (TAG & 3 = 0, NUMOP=2)
  op1: INL(exit)
  op2: SYT(loop_var)
AFOR                          ← one per value in the list
  op1: value1
AFOR
  op1: value2
AFOR
  op1: value3
... body ...
EFOR
  op1: INL(exit)
```

Each AFOR assigns its value to the loop variable. The loop body executes once per AFOR. EFOR generates a computed branch back through the AFOR chain using a stored return address (GENCLAS0.xpl:3326–3333).

AFOR Sentinel: The last AFOR in the value list always has TAG=1, set by `HALMAT_FIX_POPTAG(FIXV(SP), 1)` in SYNTHESI.xpl:2811. This mirrors the CLBL TAG=1 sentinel pattern in DO CASE. All preceding AFORs have TAG=0. The emulator uses this to detect end-of-list without a separate count.

CFOR - Conditional FOR (WHILE/UNTIL)

When a FOR loop has a WHILE or UNTIL clause, a CFOR operator appears between the body and EFOR:

```
DFOR (TAG bit 4 set for UNTIL only; WHILE leaves bit 4 clear)
  ...
CFOR
  op1: VAC(condition_result)       ← from a BTRU or comparison
... body ...
EFOR
```

CFOR calls `EMIT_WHILE_TEST` (GENCLAS0.xpl:3410) which branches to DOLBL (exit) if the condition fails. For UNTIL, the test is skipped on the first iteration (DOLBL+3 label).

Worked Example - HELLO.hal

```
HAL/S:  DO FOR I = 1 TO 5;
HALMAT:
   27:  01040102  DFOR  T=1, 4 ops
         00010023    INL(1)           ← exit label, flow #1
         00020011    SYT(2)=I         ← loop variable
         00050051    LIT(5)=1.0       ← FROM
         00060051    LIT(6)=5.0       ← TO
   ... body (inner loop, I/O) ...
   70:  00010110  EFOR  1 op
         00010021    INL(1)           ← matches DFOR

HAL/S:  DO FOR J = 2 TO 8 BY 2;
HALMAT:
   45:  02050102  DFOR  T=2, 5 ops
         00060023    INL(6)           ← exit label, flow #6
         00040011    SYT(4)=J         ← loop variable
         00090051    LIT(9)=2.0       ← FROM
         000A0051    LIT(10)=8.0      ← TO
         000B0051    LIT(11)=2.0      ← BY (step=2)
   ... body ...
   66:  00010110  EFOR  1 op
         00060021    INL(6)           ← matches DFOR
```

TAG=1: DOTYPE=1 (iterative, step=1), no TEMP, no WHILE. TAG=2: DOTYPE=2 (iterative, explicit step=2), no TEMP, no WHILE.

IF / ELSE - `IFHD` / `FBRA` / `BRA` / `LBL`

The IF statement compiles to a sequence of IFHD, a condition evaluation, FBRA (false branch), the then body, and a closing LBL. When an ELSE clause is present, a BRA (unconditional branch) and a second LBL are added.

Structure

IF without ELSE:

```
IFHD                        ← marks start of IF construct (C=1)
<condition expression>      ← typically a comparison (IGT, ILT, IEQU, etc.)
FBRA                        ← false branch: skip body if condition false
  op1: INL(exit_label)      ← where to jump if false
  op2: VAC(condition)       ← back-pointer to comparison result
... then body ...
LBL  (TAG=1)                ← exit label
  op1: INL(exit_label)      ← matches FBRA's INL
```

IF with ELSE:

```
IFHD                        ← marks start of IF construct (C=1)
<condition expression>      ← comparison operator
FBRA                        ← false branch: jump to else if false
  op1: INL(else_label)      ← where ELSE starts
  op2: VAC(condition)       ← back-pointer to comparison result
... then body ...
BRA  (TAG=1)                ← unconditional branch over else
  op1: INL(exit_label)      ← jump to end
LBL  (TAG=0)                ← ELSE entry point
  op1: INL(else_label)      ← matches FBRA's INL
... else body ...
LBL  (TAG=1)                ← exit point
  op1: INL(exit_label)      ← matches BRA's INL
```

TAG Encoding

Opcode	TAG	Meaning
IFHD	0	Always 0
FBRA	0	Always 0
BRA	1	Unconditional forward branch
LBL (else entry)	0	False-branch target (not an exit)
LBL (exit)	1	Exit target - final label in the construct

The TAG field on LBL distinguishes between the ELSE entry point (TAG=0) and the exit label (TAG=1). This is critical for the emulator: when scanning for the exit LBL, match TAG=1.

Operand Details

FBRA has exactly 2 operands: - op1: `INL(label)` - flow label for the false-branch target - op2: `VAC(addr)` - back-pointer to the condition result (the comparison operator that produced the boolean)

BRA has exactly 1 operand: - op1: `INL(label)` - flow label for the unconditional jump target

LBL has exactly 1 operand: - op1: `INL(label)` - flow label that this label defines

COPT Field

IFHD, FBRA, and LBL all have COPT=1 (C=1). BRA has COPT=0.

Flow Label Allocation

PASS1's EMITPUSH procedure reserves flow labels for IF/ELSE:

- For IF without ELSE: 2 labels allocated. Label N is the exit.
- For IF with ELSE: 2+ labels. Label N is the else-entry, Label N+1 is the exit.

The FBRA's INL always points to the first label after IFHD. If there is an ELSE, this is the else-entry LBL (TAG=0). If there is no ELSE, this is the exit LBL (TAG=1).

Condition Expressions

The condition is evaluated between IFHD and FBRA. Common patterns:

HAL/S	HALMAT
`IF A > B`	`IGT` (Class 7: integer greater-than)
`IF A < B`	`ILT`
`IF A = B`	`IEQU`
`IF A >= B`	`IGE`
`IF A <= B`	`ILE`
`IF A ^= B`	`INEQ`
`IF scalar > scalar`	`SGT` , `SLT` , `SEQU` , etc.

The comparison operator is emitted between IFHD and FBRA, and FBRA's VAC operand points back to it.

Worked Example - test_ifelse.hal

```
HAL/S:  IF A > B THEN C = A; ELSE C = B;

HALMAT:
   19:  IFHD  C=1                    ← IF construct start
   20:  IGT   C=1, 2 ops             ← integer greater-than
         SYT(2)=A, SYT(3)=B
   23:  FBRA  C=1, 2 ops             ← false branch
         INL(1)                       ← jump here if A <= B
         VAC(20)                      ← result of IGT at addr 20
   26:  SMRK                         ← stmt marker (then clause)
   28:  IASN  2 ops                  ← C = A
         SYT(2)=A, SYT(4)=C
   31:  SMRK
   33:  BRA   T=1, 1 op              ← skip over ELSE
         INL(2)                       ← jump to exit
   35:  LBL   T=0, C=1, 1 op         ← ELSE entry
         INL(1)                       ← matches FBRA
   37:  IASN  2 ops                  ← C = B
         SYT(3)=B, SYT(4)=C
   40:  SMRK
   42:  LBL   T=1, C=1, 1 op         ← exit label
         INL(2)                       ← matches BRA
```

IF without ELSE (from same program):

```
HAL/S:  IF C = 5 THEN WRITE(6) 'C IS FIVE';

HALMAT:
   44:  IFHD  C=1
   45:  IEQU  C=1, 2 ops             ← integer equals
         SYT(4)=C, LIT(3)=5
   48:  FBRA  C=1, 2 ops
         INL(3)                       ← exit (no else)
         VAC(45)                      ← result of IEQU
   ... WRITE group ...
   62:  LBL   T=1, C=1, 1 op         ← exit label
         INL(3)                       ← matches FBRA
```

Note: without ELSE, there is no BRA and no TAG=0 LBL. FBRA's INL directly targets the exit LBL (TAG=1).

IASN Operand Order

A notable detail visible in the IF/ELSE pattern: the integer assignment operator `IASN` has operand order `(source, destination)`. In the then clause, `IASN SYT(2)=A, SYT(4)=C` means `C = A` - op1 is the value, op2 is the target.

DO WHILE / DO UNTIL - `DTST / CTST / ETST`

The DO WHILE and DO UNTIL loops compile to a DTST–CTST–ETST triplet. DTST marks the start, CTST evaluates the condition, and ETST closes the loop (branches back to re-evaluate the condition).

Structure

```
DTST                        ← loop start
  op1: INL(exit_label)      ← where to go when done
<condition expression>      ← comparison operator
CTST                        ← condition test: exit if false
  op1: VAC(condition)       ← back-pointer to comparison result
... loop body ...
ETST                        ← loop end: branch back to DTST
  op1: INL(exit_label)      ← matches DTST's INL
```

TAG Encoding

Opcode	TAG	Meaning
DTST (WHILE)	0	Pre-tested loop
DTST (UNTIL)	1	Post-tested loop (condition negated)
CTST (WHILE)	0	Normal sense: exit if condition FALSE
CTST (UNTIL)	1	Inverted: exit if condition TRUE
ETST	0	Always 0

The TAG on DTST distinguishes WHILE from UNTIL. CTST's TAG mirrors it.

Condition Semantics

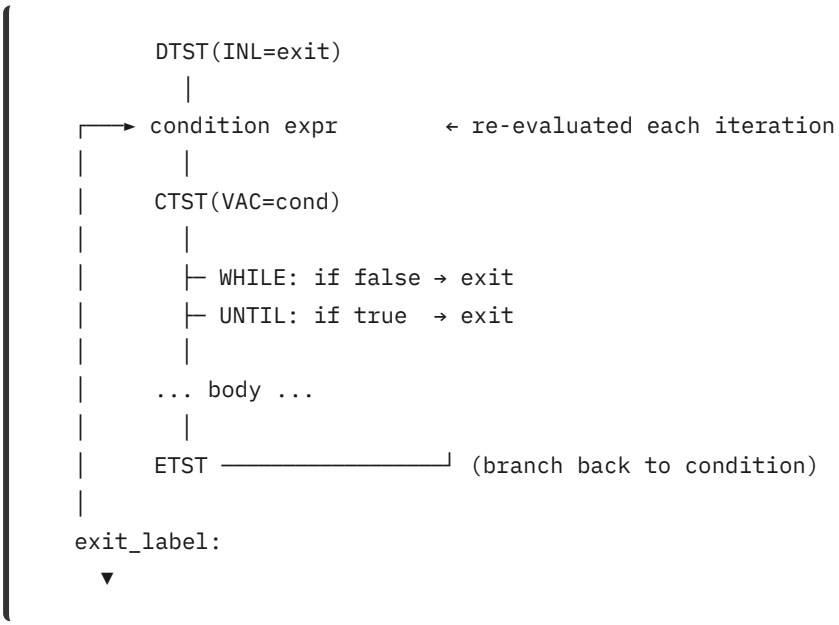
For **DO WHILE**, the condition is tested at the top of each iteration. If the condition is false, execution jumps past ETST to the exit label.

For **DO UNTIL**, the test is logically inverted: the loop exits when the condition becomes true. In HALMAT, this is encoded by setting TAG=1 on both DTST and CTST, which tells PASS2 to invert the branch direction.

Flow Label Allocation

DTST reserves flow labels. The first label (op1) is the exit label. ETST branches unconditionally back to the condition evaluation point (the atom after DTST).

Execution Flow



Note: Unlike iterative FOR loops (which are bottom-tested), WHILE and UNTIL loops are **top-tested**: the condition is always evaluated before the body on every iteration, including the first.

Worked Example - test_while.hal

```
HAL/S:  DO WHILE I < 10;
           TOTAL = TOTAL + I;
           I = I + 1;
        END;

HALMAT:
   22:  DTST  C=1, 1 op              ← DO WHILE start (TAG=0)
         INL(1)                       ← exit label #1
   24:  ILT   C=1, 2 ops             ← integer less-than
         SYT(2)=I, LIT(4)=10
   27:  CTST  1 op                    ← condition check (TAG=0)
         VAC(24)                      ← result of ILT at addr 24
   ... body: IADD + IASN for TOTAL, IADD + IASN for I ...
   47:  ETST  1 op                    ← loop end
         INL(1)                       ← branch back to condition
```

DO UNTIL from same program:

```
HAL/S:  DO UNTIL J = 0;
           J = J - 1;
        END;

HALMAT:
   51:  DTST  T=1, C=1, 1 op         ← DO UNTIL start (TAG=1)
         INL(4)                       ← exit label #4
   53:  IEQU  C=1, 2 ops             ← integer equals
         SYT(3)=J, LIT(6)=0
   56:  CTST  T=1, 1 op              ← condition check (TAG=1, inverted)
         VAC(53)                      ← result of IEQU at addr 53
   ... body: ISUB + IASN for J ...
   68:  ETST  1 op                    ← loop end
         INL(4)                       ← branch back to condition
```

TAG=0 on DTST/CTST → WHILE semantics (exit when false). TAG=1 on DTST/CTST → UNTIL semantics (exit when true).

DO CASE - DCAS / CLBL / ECAS

The DO CASE statement compiles to a DCAS–CLBL*–ECAS sequence. DCAS opens the construct and names the selector variable. Each case body is preceded by a CLBL (case label). ECAS closes the construct.

Structure

```
DCAS                        ← case construct start
  op1: INL(exit_label)      ← exit flow label
  op2: SYT(selector)        ← selector variable (or VAC for expression)
... SMRK ...
CLBL  (TAG=0)               ← case 0 entry
  op1: INL(exit_label)      ← matches DCAS
  op2: INL(case_0_exit)     ← this case's local exit label
... case 0 body ...
CLBL  (TAG=0)               ← case 1 entry
  op1: INL(exit_label)
  op2: INL(case_1_exit)
... case 1 body ...
CLBL  (TAG=0)               ← case 2 entry
  op1: INL(exit_label)
  op2: INL(case_2_exit)
... case 2 body ...
CLBL  (TAG=1)               ← sentinel (final CLBL)
  op1: INL(exit_label)
  op2: INL(sentinel_exit)
ECAS                        ← case construct end
  op1: INL(exit_label)      ← matches DCAS
```

TAG Encoding

Opcode	TAG	Meaning
DCAS	0	Always 0
CLBL (normal)	0	Regular case label
CLBL (last)	1	Sentinel - marks end of case list
ECAS	0	Always 0

The sentinel CLBL (TAG=1) always appears immediately before ECAS, even if the last case has an empty body. It marks the boundary of the case table and serves as the fall-through exit point.

Operand Details

DCAS has 2 operands: - op1: `INL(exit)` - flow label for the exit point - op2: `SYT(n)` or `VAC(n)` - the selector expression

CLBL has 2 operands: - op1: `INL(exit)` - same exit label as DCAS (used for BRA after each case) - op2: `INL(case_exit)` - unique label for this case's exit point

The second INL on each CLBL is used by PASS2 to generate a branch table. Each case's exit label points past that case's body to either the next CLBL or to ECAS.

Case Numbering

Cases are 0-indexed. The selector value determines which CLBL's body executes. If SEL=0, the first CLBL's body runs. If SEL=1, the second. And so on. If the selector is out of range, behavior is undefined (PASS2 may generate a runtime error check).

Execution Flow

```
DCAS(INL=exit, selector)
  │
  ├─ Evaluate selector
  ├─ Generate indexed branch into CLBL table
  │
  ├──► CLBL[0] body → BRA → exit
  ├──► CLBL[1] body → BRA → exit
  ├──► CLBL[2] body → BRA → exit
  │
ECAS(INL=exit)
  │
exit:
  ▼
```

Worked Example - test_case.hal

```
HAL/S:  DO CASE SEL;
           RESULT = 10;
           RESULT = 20;
           RESULT = 30;
        END;

HALMAT:
   14:  DCAS  2 ops                  ← case construct
         INL(1)                       ← exit label #1
         SYT(2)=SEL                   ← selector variable
   17:  SMRK
   19:  CLBL  C=1, 2 ops             ← case 0 (TAG=0)
         INL(1), INL(5)
   22:  IASN  2 ops                  ← RESULT = 10
         LIT(2)=10, SYT(3)=RESULT
   25:  SMRK
   27:  CLBL  C=1, 2 ops             ← case 1 (TAG=0)
         INL(1), INL(7)
   30:  IASN  2 ops                  ← RESULT = 20
         LIT(3)=20, SYT(3)=RESULT
   33:  SMRK
   35:  CLBL  C=1, 2 ops             ← case 2 (TAG=0)
         INL(1), INL(9)
   38:  IASN  2 ops                  ← RESULT = 30
         LIT(4)=30, SYT(3)=RESULT
   41:  SMRK
   43:  CLBL  T=1, C=1, 2 ops        ← sentinel (TAG=1)
         INL(1), INL(11)
   46:  ECAS  1 op                    ← end case
         INL(1)                       ← matches DCAS
```

Note the pattern: 3 real cases + 1 sentinel CLBL = 4 CLBLs total. Each CLBL has the same exit label (INL(1)) in op1 and a unique case-specific label in op2 (INL(5), INL(7), INL(9), INL(11) - spaced by 2, matching the flow label allocation stride).

Simple DO - `DSMP / ESMP`

The simplest control flow pattern. `DO; ... END;` with no loop, no condition, no case selector - just a scope boundary. The compiler emits a DSMP–ESMP pair around the body.

Structure

```
DSMP                        ← block start
  op1: INL(exit_label)      ← flow label (for EXIT targeting)
... body ...
ESMP                        ← block end
  op1: INL(exit_label)      ← matches DSMP
```

Both DSMP and ESMP have exactly 1 operand (INL). TAG is always 0. The INL label exists so that an EXIT statement inside the block can target it.

Worked Example - test_simple_do.hal

```
HAL/S:  DO;
           X = X + Y;
           Y = X - Y;
           X = X - Y;
        END;

HALMAT:
   17:  DSMP  1 op                    ← simple DO start
         INL(1)                       ← exit label #1
   ... IADD + IASN (X = X + Y) ...
   ... ISUB + IASN (Y = X - Y) ...
   ... ISUB + IASN (X = X - Y) ...
   45:  ESMP  1 op                    ← block end
         INL(1)                       ← matches DSMP
```

Verified Discrete FOR - test_discrete_for.hal

The discrete FOR pattern was described in the FOR Loop section above. Here is the verified binary evidence:

```
HAL/S:  DO FOR I = 3, 7, 11, 42;
           RESULT = RESULT + I;
        END;

HALMAT:
   12:  DFOR  T=0, 2 ops            ← discrete FOR (TAG & 3 = 0)
         INL(1)                      ← exit label
         SYT(2)=I                    ← loop variable
   15:  AFOR  T=0, C=1, 1 op        ← value 3
         LIT(2)=3
   17:  AFOR  T=0, C=1, 1 op        ← value 7
         LIT(3)=7
   19:  AFOR  T=0, C=1, 1 op        ← value 11
         LIT(4)=11
   21:  AFOR  T=1, C=1, 1 op        ← value 42 (TAG=1 = last!)
         LIT(5)=42
   ... body ...
   33:  EFOR  1 op                   ← loop end
         INL(1)                      ← matches DFOR
```

TAG=1 on the final AFOR marks end-of-list. The emulator uses this instead of a separate value count.

I/O Groups - `XXST / XXAR / WRIT / READ / XXND`

I/O operations (WRITE and READ) compile to a structured group enclosed by XXST (start) and XXND (end), with XXAR (argument) operators for each value in the I/O list.

Structure

```
XXST                        ← I/O group start
  op1: IMD(n)               ← argument count (number of XXARs)
XXAR  (per argument)        ← I/O argument
  op1: <value>              ← SYT (variable), LIT (string/constant), or VAC
... more XXARs ...
WRIT (or READ/RDAL)         ← I/O operation
  op1: IMD(unit)            ← output unit number (6=stdout)
XXND                        ← I/O group end
```

Opcode Variants

Opcode	HAL/S	Direction
WRIT	`WRITE(n)`	Output
READ	`READ(n)`	Input
RDAL	`READALL(n)`	Input (entire record)

TAG Encoding

For simple WRITE/READ, all TAGs are 0. The XXST and XXAR opcodes have COPT=1 (C=1).

Operand Details

XXST op1: `IMD(n)` - the number `n` is the count of XXAR operators in this group.

XXAR op1: the data item. Qualifier encodes the type: - `SYT(n)` - variable reference (with TAG1 encoding the type: TAG1=6 for integer, TAG1=2 for character) - `LIT(n)` - literal string or constant (TAG1=2 for character) - `VAC(n)` - result of a prior computation

WRIT/READ op1: `IMD(unit)` - the I/O unit number from the HAL/S `WRITE(unit)` syntax. Unit 6 is the standard output device.

XXND has 0 operands.

TAG1 Encoding on XXAR Operands

The TAG1 field on XXAR operands encodes the data type being passed:

TAG1	Type
0	Default/unspecified
2	Character string
6	Integer

Worked Example - HELLO.hal

```
HAL/S:  WRITE(6) I, MY_NAME, ' IS A FRIEND OF MINE';

HALMAT:
   53:  XXST  C=1, 1 op
         IMD(2)                      ← 2 is arg count (not unit!)
   55:  XXAR  C=1, 1 op
         SYT(4)=J  [T1=6]           ← integer variable
   57:  XXAR  C=1, 1 op
         SYT(3)=MY_NAME  [T1=2]     ← character variable
   59:  XXAR  C=1, 1 op
         LIT(13)=CHAR(20) [T1=2]    ← string literal
   61:  WRIT  1 op
         IMD(6)                      ← unit 6 (stdout)
   63:  XXND                        ← end of I/O group
```

Note: the XXST's IMD operand counts the number of XXARs (2 in the HALMAT encoding, though there are actually 3 XXAR operators here - this may indicate the IMD counts output *expressions* differently from individual XXAR atoms). The WRIT's IMD operand is the unit number.

Function / Procedure Calls - `FCAL` / `PCAL` / `XXST` / `XXAR` / `XXND`

Function and procedure calls reuse the same XXST/XXAR/XXND framing as I/O groups, but with FCAL or PCAL as the operation instead of WRIT/READ.

Structure

Function call (returns a value):

```
XXST  (TAG=1)              ← call group start
  op1: SYT(func_name)      ← function being called
XXAR  (TAG=1, per argument) ← argument
  op1: SYT(arg) or VAC/LIT ← argument value
FCAL  (TAG=1)              ← function call
  op1: SYT(func_name)      ← function being called
XXND  (TAG=1)              ← call group end
```

Procedure call (no return value):

```
XXST  (TAG=1)              ← call group start
  op1: SYT(proc_name)
XXAR  (TAG=1, per argument)
  op1: <argument>
PCAL  (TAG=1)              ← procedure call
  op1: SYT(proc_name)
XXND  (TAG=1)              ← call group end
```

Distinguishing Calls from I/O

The key differences between function/procedure calls and I/O groups:

Feature	I/O Group	Call Group
TAG on all opcodes	0	1
XXST op1	IMD(arg_count)	SYT(func/proc)
Operation opcode	WRIT/READ/RDAL	FCAL/PCAL
WRIT/READ op1	IMD(unit)	-
FCAL/PCAL op1	-	SYT(func/proc)

TAG=1 on the entire group is the primary distinguishing feature.

Function Return Values

After an FCAL, the result is available as a VAC back-pointer. Subsequent operators can reference the FCAL's result via `VAC(addr_of_FCAL)`:

```
FCAL at addr 36: SYT(func)
IASN: VAC(36), SYT(Y)          ← Y = result of function call
```

Function / Procedure Definition

Functions and procedures are defined with FDEF/PDEF...CLOS blocks:

```
FDEF                        ← function definition start
  op1: SYT(func_name)       ← symbol table entry
... EDCL ...                 ← end of declarations
... body ...
RTRN                        ← return statement
  op1: VAC(result) [T1=6]   ← return value (T1=type)
... SMRK ...
CLOS                        ← function definition end
  op1: SYT(func_name)       ← matches FDEF
```

PDEF has the same structure but uses PCAL for calls and RTRN may have no operand (procedure returns no value).

RTRN Operand Details

RTRN has 1 operand for function returns: - op1: `VAC(n)` - back-pointer to the expression being returned - TAG1 on the operand encodes the return type (6=integer, etc.)

Worked Example - test_proc.hal

```
HAL/S:  ADD_ONE: FUNCTION(N) INTEGER;
           DECLARE INTEGER, N;
           RETURN N + 1;
        CLOSE ADD_ONE;

HALMAT (definition):
    9:  FDEF  1 op                    ← function definition
         SYT(4)=ADD_ONE
   ... EDCL ...
   16:  IADD  2 ops                  ← N + 1
         SYT(5)=N, LIT(1)=1
   19:  RTRN  1 op                    ← return
         VAC(16) [T1=6]              ← return IADD result (integer)
   23:  CLOS  1 op                    ← definition end
         SYT(4)=ADD_ONE

HAL/S:  Y = ADD_ONE(X);

HALMAT (call):
   32:  XXST  T=1, C=1, 1 op         ← call group (TAG=1!)
         SYT(4)=ADD_ONE
   34:  XXAR  T=1, C=1, 1 op         ← argument
         SYT(2)=X [T1=6]
   36:  FCAL  T=1, 1 op              ← function call
         SYT(4)=ADD_ONE
   38:  XXND  T=1, C=1               ← call group end
   39:  IASN  2 ops                  ← Y = result
         VAC(36)                      ← back-pointer to FCAL
         SYT(3)=Y
```

Note TAG=1 on XXST/XXAR/FCAL/XXND - this marks the entire group as a function call rather than an I/O operation.

Subscript Access - `DSUB / IDLP / TSUB`

Array and matrix element access is handled by subscript operators. DSUB is the primary subscript specifier; IDLP and TSUB serve specialised roles for indexed loops and the `#` (size) operator.

DSUB - Regular Subscript

DSUB creates a reference to a subscripted element. Subsequent operators use the DSUB result via VAC to read from or write to the element.

Structure:

```
DSUB                        ← subscript operation
  op1: SYT(array)           ← the array/matrix being subscripted
  op2: <index1>             ← first subscript (SYT, IMD, LIT, or VAC)
  [op3: <index2>]           ← second subscript (for 2D: matrices)
  [opN: ...]                ← additional subscripts for higher dimensions
```

TAG encoding: The TAG field on DSUB encodes type information. From the binary examples, TAG=5 (`0b00000101`) appears for both integer arrays and scalar matrices. The exact encoding of the TAG is complex and involves NAME_BIT and PSEUDO_TYPE from the compiler's type system. The low bits encode `PSEUDO_TYPE(H1)` (the data type of the result), and bit 5+ may encode `NAME_BIT` (whether the subscripted item is a NAME variable - a pointer type in HAL/S).

NUMOP: Equals 1 + number of subscript dimensions: - 1D array: NUMOP = 2 (array + index) - 2D matrix: NUMOP = 3 (matrix + row + col)

Operand qualifiers for subscripts:

Qualifier	Meaning	Example
SYT	Variable subscript	`ARR(I)` → op2=SYT(I)
IMD	Constant subscript	`M(1,2)` → op2=IMD(1), op3=IMD(2)
LIT	Literal subscript	constant from literal table
VAC	Computed subscript	result of an expression

TAG1 on subscript operands: Encodes the subscript's data type. TAG1=1 for integer constant subscripts (IMD). TAG1=5 for variable subscripts. TAG1=0 for the array variable itself in op1.

TAG2 on operands: TAG2=1 on op1 (the array) marks it as the base variable. TAG2=0 on subscript operands.

Result consumption: The DSUB operator produces a subscripted reference. Subsequent assignments or reads use `VAC(dsub_addr)`：

```
DSUB at addr 18: SYT(ARR), SYT(I)    ← ARR(I)
IASN: SYT(I), VAC(18)                ← ARR(I) = I
```

```
DSUB at addr 20: SYT(M), IMD(2), IMD(3)   ← M(2,3)
SASN: VAC(20), SYT(VAL)                    ← VAL = M(2,3)
```

DSUB Operand Format (from Burkey's HALMAT.md)

The full DSUB operand encoding supports additional complexity:

- Operands with qualifier `CSZ` (8) or `ASZ` (9) may be followed by an additional operand word for dynamic size (`#`) expressions
- When `D` (data field) = 0: size is just `#` alone
- When `D` = 1: size is `# + expr` (additional operand follows)
- When `D` = 2: size is `# - expr` (additional operand follows)

This handles HAL/S's `#` operator for runtime-determined array sizes.

IDLP - Indexed DO Loop Pointer

IDLP appears when an array is subscripted implicitly by a DO FOR loop variable. This is the HAL/S "array DO FOR" construct where a loop iterates over array elements:

```
IF (SYT_FLAGS(ID_LOC) & AUTO_FLAG) ^= 0 THEN I = XADLP;
ELSE I = XIDLP;
```

ADLP (automatic) vs IDLP (explicit) depends on whether the loop variable is compiler-generated or user-specified.

TSUB - Terminal Subscript

TSUB handles the `#` (pound/size) operator, which queries the runtime size of a `*`-dimensioned array or character string:

```
HAL/S:  DECLARE CHAR CHARACTER(*);
        L = LENGTH(CHAR);      ← uses # implicitly
```

TSUB is "a very stripped down DSUB" (Burkey) with only one subscript dimension.

Worked Example - test_array.hal (1D array)

```
HAL/S:  DECLARE ARR ARRAY(5) SCALAR;
        DO FOR I = 1 TO 5;
M          ARR  = I;
S             I
        END;

HALMAT:
   18:  DSUB  T=5, 2 ops             ← subscript ARR(I)
         SYT(2)=ARR [T1=0, T2=1]     ← base array
         SYT(3)=I   [T1=5, T2=0]     ← subscript index
   21:  IASN  2 ops                  ← ARR(I) = I
         SYT(3)=I                     ← source value
         VAC(18)                      ← destination = DSUB result
```

Worked Example - test_matrix.hal (2D matrix)

```
HAL/S:  DECLARE M MATRIX(3, 3);
M       M    = 42.5;
S        1,2

HALMAT:
   11:  DSUB  T=5, 3 ops             ← subscript M(1,2)
         SYT(2)=M   [T1=0, T2=1]     ← base matrix
         IMD(1)      [T1=1, T2=0]     ← row subscript (constant 1)
         IMD(2)      [T1=1, T2=0]     ← column subscript (constant 2)
   15:  IASN  2 ops                  ← M(1,2) = 42.5
         LIT(5)=42.5                  ← source value
         VAC(11)                      ← destination = DSUB result
```

```
HAL/S:  VAL = M   ;
S              2,3

HALMAT:
   20:  DSUB  T=5, 3 ops             ← subscript M(2,3)
         SYT(2)=M                     ← base matrix
         IMD(2)      [T1=1, T2=0]     ← row
         IMD(3)      [T1=1, T2=0]     ← col
   24:  SASN  2 ops                  ← VAL = M(2,3)
         VAC(20)                      ← source = DSUB result
         SYT(3)=VAL                   ← destination
```

Note: IASN is used for the assignment `M(1,2) = 42.5` because the literal 42.5 is stored as an integer-typed value in the literal table (the HALMAT compiler coerces it). SASN (scalar assign) is used for `VAL = M(2,3)` because the destination is SCALAR.

Summary

Class	Name	Opcodes	References
0	Control and Subscripting	74	169
1	Bit Operations	13	7
2	Character Operations	6	5
3	Matrix Arithmetic	14	21
4	Vector Arithmetic	9	4
5	Scalar Arithmetic	13	14
6	Integer Arithmetic	10	6
7	Conditional and Comparison	28	19
8	Initialisation	13	12

Total: 180 opcodes, 257 references across 630 files