IE32 Processor User's Manual

IE32 32-bit RISC-like Processor Reference Manual

Last modified: 2026-07-07

Table of Contents

1. Architecture Overview

The IE32 is a 32-bit RISC-like CPU with fixed 8-byte instructions, 16 general-purpose registers, a flat 32-bit address model, and 32-bit little-endian memory operations.

- **Word size**: 32-bit registers and 32-bit arithmetic.
- **Instruction width**: Fixed 8 bytes per instruction.
- **Byte order**: Little-endian instruction operands and 32-bit memory accesses.
- **Register file**: 16 general-purpose 32-bit registers.
- **Address space**: 32-bit effective addresses for instruction fetch, operand resolution, stack operations, and interrupt-vector reads. The IE32 CPU defines address formation and access width; it does not define a fixed installed-memory size.

- **Integer condition model**: Conditional branches test one register against zero. There is no flags register.
- **Memory access width**: CPU load, store, stack, and ALU memory operands use 32-bit reads and writes.
- **Stack**: Full-descending 32-bit stack. `SP` is an internal CPU register, not one of the 16 general-purpose registers.
- **Programme counter**: `PC` is a 32-bit internal CPU register, not directly addressable as a general-purpose register.

---

2. Register File

IE32 has 16 general-purpose 32-bit registers. Register operands are encoded in the low 4 bits of a register field or operand field.

Index	Register	Description
0	A	Accumulator and general-purpose register
1	X	General-purpose register
2	Y	General-purpose register
3	Z	General-purpose register
4	B	General-purpose register
5	C	General-purpose register
6	D	General-purpose register
7	E	General-purpose register
8	F	General-purpose register
9	G	General-purpose register
10	H	General-purpose register
11	S	General-purpose register
12	T	General-purpose register
13	U	General-purpose register
14	V	General-purpose register
15	W	General-purpose register

Special internal registers:

Register	Width	Initial value	Description
PC	32-bit	`0x1000`	Programme counter
SP	32-bit	`0x9F000`	Stack pointer

Reset initialises `PC` to `PROG_START` (`0x1000`) and `SP` to `STACK_START` (`0x9F000`). General-purpose registers are cleared to zero.

---

3. Instruction Encoding

Every IE32 instruction is exactly 8 bytes.

3.1 Byte-Level Format

```
Byte:    0        1        2        3        4    5    6    7
       +--------+--------+--------+--------+------+------+------+------+
       | opcode | reg    | mode   | zero   |       operand32 (LE)      |
       +--------+--------+--------+--------+------+------+------+------+
Bits:   [7:0]    [7:0]    [7:0]    [7:0]          [31:0]
```

3.2 Field Definitions

Field	Byte	Width	Description
opcode	0	8 bits	Instruction opcode
reg	1	8 bits	Register field. The CPU uses the low 4 bits.
mode	2	8 bits	Addressing mode for instructions that resolve an operand.
zero	3	8 bits	Reserved. Software should encode zero; the CPU ignores this byte.
operand32	4-7	32 bits	Immediate, address, target, register index, or encoded register plus offset.

3.3 Field Extraction

```
opcode    = instr[0]
reg       = instr[1]
addrMode  = instr[2]
operand32 = instr[4] | (instr[5] << 8) | (instr[6] << 16) | (instr[7] << 24)
```

The CPU masks register indexes with `0x0F`, so encodings with high bits set in a register field alias to one of the 16 architectural registers.

3.4 Encoding Helper

```
instr[0] = opcode
instr[1] = register index, or 0 when unused
instr[2] = addressing mode, or 0 when unused
instr[3] = 0
instr[4..7] = operand32, little-endian
```

3.5 Addressing Mode Codes

Code	Name	Meaning
`0x00`	Immediate	Operand field is the value.
`0x01`	Register	Low 4 bits of operand field select a register.
`0x02`	Register indirect	Low 4 bits select a base register; remaining bits are an unsigned 32-bit offset contribution.
`0x03`	Memory indirect	For normal operand reads, `operand32` is read as memory. For stores and memory `INC`/`DEC`, `operand32` points to a 32-bit pointer; that pointer value is the final address.
`0x04`	Direct	Operand field is a direct memory address.

Code	Name	Meaning
`0x05`-`0xFF`	Reserved	Read-style operand resolution returns zero. Store-style resolution treats `operand32` as a direct destination memory address.

Assembly-language forms are defined for immediate, register, register-indirect, and direct operands. Memory-indirect mode `0x03` is an encodable CPU mode; no source form is defined for it in this manual. Addressing-mode bytes `0x05` through `0xFF` are reserved encodings with the deterministic behaviour shown in the table.

---

4. Complete Instruction Reference

4.0 Instruction Entry Schema

Each instruction entry uses the following field schema.

Field	Meaning
Operation	Canonical mnemonic and brief operation class.
Assembler Syntax	Assembly-language form for the instruction encoding.
Attributes	Addressing modes, memory access class, and fixed 32-bit operand constraints.
Description	Instruction-specific behaviour after the addressing mode resolves the operand.
Condition Codes	Integer condition-code effect. IE32 has no flags register; conditional branches test a register value directly.
Instruction Format	Opcode byte and fixed fields within the 8-byte instruction.
Instruction Fields	Meaning of the register field, addressing-mode byte, and `operand32` for the entry.
Exceptions	Stopped-processor conditions architecturally caused by the instruction.
Notes	Architectural aliases, operand restrictions, or compatibility details.

`resolve(operand)` means applying the addressing-mode byte and operand field from section 3.5 for instructions that read an operand: immediates use the 32-bit operand literally, direct and memory-indirect forms read memory, register mode reads the named register, and register-indirect mode reads from the base register plus offset.

`store(value, operand)` writes `value` to a destination memory address. Store instructions do not write architectural registers. For immediate, direct, and register addressing modes, `operand32` is the destination address. For register-indirect store encodings, the low register bits select the base register and the remaining bits form the byte offset described in section 5.5. For memory-indirect store encodings, the CPU first reads a 32-bit pointer from `operand32`, then writes to the address contained in that pointer.

4.1 Data Movement

1. LOAD - LOAD R, operand

Operation: `R = resolve(operand)`.

Assembler Syntax: `LOAD R, operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved using the selected IE32 addressing mode, and the resulting 32-bit value is loaded into `R`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x01`.

Instruction Fields: Byte 0 holds opcode `0x01`. Byte 1 selects destination register `R`; the CPU uses the low 4 bits. Byte 2 selects the addressing mode used to resolve the source operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; `operand32` is interpreted according to byte 2.

Exceptions: None.

Notes: None.

2. LDA - LDA operand

Operation: `A = resolve(operand)`.

Assembler Syntax: `LDA operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved using the selected IE32 addressing mode, and the resulting 32-bit value is loaded into `A`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x20`.

Instruction Fields: Byte 0 holds opcode `0x20`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the source operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the resolved value is written to `A`.

Exceptions: None.

Notes: None.

3. LDX - LDX operand

Operation: `X = resolve(operand)`.

Assembler Syntax: `LDX operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved using the selected IE32 addressing mode, and the resulting 32-bit value is loaded into `X`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x21`.

Instruction Fields: Byte 0 holds opcode `0x21`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the source operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the resolved value is written to `X`.

Exceptions: None.

Notes: None.

4. LDY - LDY operand

Operation: `Y = resolve(operand)`.

Assembler Syntax: `LDY operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved using the selected IE32 addressing mode, and the resulting 32-bit value is loaded into `Y`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x22`.

Instruction Fields: Byte 0 holds opcode `0x22`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the source operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the resolved value is written to `Y`.

Exceptions: None.

Notes: None.

5. LDZ - LDZ operand

Operation: `Z = resolve(operand)`.

Assembler Syntax: `LDZ operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved using the selected IE32 addressing mode, and the resulting 32-bit value is loaded into `Z`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x23`.

Instruction Fields: Byte 0 holds opcode `0x23`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the source operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the resolved value is written to `Z`.

Exceptions: None.

Notes: None.

6. LDB - LDB operand

Operation: `B = resolve(operand)`.

Assembler Syntax: `LDB operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved using the selected IE32 addressing mode, and the resulting 32-bit value is loaded into `B`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x3A`.

Instruction Fields: Byte 0 holds opcode `0x3A`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the source operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the resolved value is written to `B`.

Exceptions: None.

Notes: None.

7. LDC - LDC operand

Operation: `C = resolve(operand)`.

Assembler Syntax: `LDC operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved using the selected IE32 addressing mode, and the resulting 32-bit value is loaded into `C`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x3B`.

Instruction Fields: Byte 0 holds opcode `0x3B`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the source operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the resolved value is written to `C`.

Exceptions: None.

Notes: None.

8. LDD - LDD operand

Operation: `D = resolve(operand)`.

Assembler Syntax: `LDD operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved using the selected IE32 addressing mode, and the resulting 32-bit value is loaded into `D`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x3C`.

Instruction Fields: Byte 0 holds opcode `0x3C`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the source operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the resolved value is written to `D`.

Exceptions: None.

Notes: None.

9. LDE - LDE operand

Operation: `E = resolve(operand)`.

Assembler Syntax: `LDE operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved using the selected IE32 addressing mode, and the resulting 32-bit value is loaded into `E`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x3D`.

Instruction Fields: Byte 0 holds opcode `0x3D`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the source operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the resolved value is written to `E`.

Exceptions: None.

Notes: None.

10. LDF - LDF operand

Operation: `F = resolve(operand)`.

Assembler Syntax: `LDF operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved using the selected IE32 addressing mode, and the resulting 32-bit value is loaded into `F`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x3E`.

Instruction Fields: Byte 0 holds opcode `0x3E`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the source operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the resolved value is written to `F`.

Exceptions: None.

Notes: None.

11. LDG - LDG operand

Operation: `G = resolve(operand)`.

Assembler Syntax: `LDG operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved using the selected IE32 addressing mode, and the resulting 32-bit value is loaded into `G`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x3F`.

Instruction Fields: Byte 0 holds opcode `0x3F`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the source operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the resolved value is written to `G`.

Exceptions: None.

Notes: None.

12. LDU - LDU operand

Operation: `U = resolve(operand)`.

Assembler Syntax: `LDU operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved using the selected IE32 addressing mode, and the resulting 32-bit value is loaded into `U`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x40`.

Instruction Fields: Byte 0 holds opcode `0x40`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the source operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the resolved value is written to `U`.

Exceptions: None.

Notes: None.

13. LDV - LDV operand

Operation: `V = resolve(operand)`.

Assembler Syntax: `LDV operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved using the selected IE32 addressing mode, and the resulting 32-bit value is loaded into `V`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x41`.

Instruction Fields: Byte 0 holds opcode `0x41`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the source operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the resolved value is written to `V`.

Exceptions: None.

Notes: None.

14. LDW - LDW operand

Operation: `W = resolve(operand)`.

Assembler Syntax: `LDW operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved using the selected IE32 addressing mode, and the resulting 32-bit value is loaded into `W`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x42`.

Instruction Fields: Byte 0 holds opcode `0x42`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the source operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the resolved value is written to `W`.

Exceptions: None.

Notes: None.

15. LDH - LDH operand

Operation: `H = resolve(operand)`.

Assembler Syntax: `LDH operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved using the selected IE32 addressing mode, and the resulting 32-bit value is loaded into `H`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x4C`.

Instruction Fields: Byte 0 holds opcode `0x4C`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the source operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the resolved value is written to `H`.

Exceptions: None.

Notes: None.

16. LDS - LDS operand

Operation: `S = resolve(operand)`.

Assembler Syntax: `LDS operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved using the selected IE32 addressing mode, and the resulting 32-bit value is loaded into `S`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x4D`.

Instruction Fields: Byte 0 holds opcode `0x4D`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the source operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the resolved value is written to `S`.

Exceptions: None.

Notes: None.

17. LDT - LDT operand

Operation: `T = resolve(operand)`.

Assembler Syntax: `LDT operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved using the selected IE32 addressing mode, and the resulting 32-bit value is loaded into `T`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x4E`.

Instruction Fields: Byte 0 holds opcode `0x4E`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the source operand. Byte 3 is reserved by this instruction and ignored by the

processor. Bytes 4-7 hold `operand32` in little-endian order; the resolved value is written to `T`.

Exceptions: None.

Notes: None.

4.2 Load/Store

18. STORE - STORE R, operand

Operation: `store(R, operand)`.

Assembler Syntax: `STORE R, operand`.

Attributes: Memory: Write.

Description: The operand selects a destination memory location, and the 32-bit contents of `R` are written there. Immediate, direct, and register addressing modes use `operand32` as the destination address; register-indirect and memory-indirect modes compute the destination address as described in section 4.0.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x02`.

Instruction Fields: Byte 0 holds opcode `0x02`. Byte 1 selects source register `R`; the CPU uses the low 4 bits. Byte 2 selects the addressing mode used to resolve the destination. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; `operand32` is interpreted according to byte 2.

Exceptions: None.

Notes: None.

19. STA - STA operand

Operation: `store(A, operand)`.

Assembler Syntax: `STA operand`.

Attributes: Memory: Write.

Description: The operand selects a destination memory location, and the 32-bit contents of `A` are written there. Immediate, direct, and register addressing modes use `operand32` as the destination address; register-indirect and memory-indirect modes compute the destination address as described in section 4.0.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x24`.

Instruction Fields: Byte 0 holds opcode `0x24`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the destination. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the value stored is read from `A`.

Exceptions: None.

Notes: None.

20. STX - STX operand

Operation: `store(X, operand)`.

Assembler Syntax: `STX operand`.

Attributes: Memory: Write.

Description: The operand selects a destination memory location, and the 32-bit contents of `X` are written there. Immediate, direct, and register addressing modes use `operand32` as the destination address; register-indirect and memory-indirect modes compute the destination address as described in section 4.0.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x25`.

Instruction Fields: Byte 0 holds opcode `0x25`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the destination. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the value stored is read from `X`.

Exceptions: None.

Notes: None.

21. STY - STY operand

Operation: `store(Y, operand)`.

Assembler Syntax: `STY operand`.

Attributes: Memory: Write.

Description: The operand selects a destination memory location, and the 32-bit contents of `Y` are written there. Immediate, direct, and register addressing modes use `operand32` as the destination address; register-indirect and memory-indirect modes compute the destination address as described in section 4.0.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x26`.

Instruction Fields: Byte 0 holds opcode `0x26`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the destination. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the value stored is read from `Y`.

Exceptions: None.

Notes: None.

22. STZ - STZ operand

Operation: `store(Z, operand)`.

Assembler Syntax: `STZ operand`.

Attributes: Memory: Write.

Description: The operand selects a destination memory location, and the 32-bit contents of `Z` are written there. Immediate, direct, and register addressing modes use `operand32` as the destination address; register-indirect and memory-indirect modes compute the destination address as described in section 4.0.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x27`.

Instruction Fields: Byte 0 holds opcode `0x27`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the destination. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the value stored is read from `Z`.

Exceptions: None.

Notes: None.

23. STB - STB operand

Operation: `store(B, operand)`.

Assembler Syntax: `STB operand`.

Attributes: Memory: Write.

Description: The operand selects a destination memory location, and the 32-bit contents of `B` are written there. Immediate, direct, and register addressing modes use `operand32` as the destination address; register-indirect and memory-indirect modes compute the destination address as described in section 4.0.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x43`.

Instruction Fields: Byte 0 holds opcode `0x43`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the destination. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the value stored is read from `B`.

Exceptions: None.

Notes: None.

24. STC - STC operand

Operation: `store(C, operand)`.

Assembler Syntax: `STC operand`.

Attributes: Memory: Write.

Description: The operand selects a destination memory location, and the 32-bit contents of `C` are written there. Immediate, direct, and register addressing modes use `operand32` as the destination address; register-indirect and memory-indirect modes compute the destination address as described in section 4.0.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x44`.

Instruction Fields: Byte 0 holds opcode `0x44`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the destination. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the value stored is read from `C`.

Exceptions: None.

Notes: None.

25. STD - STD operand

Operation: `store(D, operand)`.

Assembler Syntax: `STD operand`.

Attributes: Memory: Write.

Description: The operand selects a destination memory location, and the 32-bit contents of `D` are written there. Immediate, direct, and register addressing modes use `operand32` as the destination address; register-indirect and memory-indirect

modes compute the destination address as described in section 4.0.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x45`.

Instruction Fields: Byte 0 holds opcode `0x45`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the destination. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the value stored is read from `D`.

Exceptions: None.

Notes: None.

26. STE - STE operand

Operation: `store(E, operand)`.

Assembler Syntax: `STE operand`.

Attributes: Memory: Write.

Description: The operand selects a destination memory location, and the 32-bit contents of `E` are written there. Immediate, direct, and register addressing modes use `operand32` as the destination address; register-indirect and memory-indirect modes compute the destination address as described in section 4.0.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x46`.

Instruction Fields: Byte 0 holds opcode `0x46`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the destination. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the value stored is read from `E`.

Exceptions: None.

Notes: None.

27. STF - STF operand

Operation: `store(F, operand)`.

Assembler Syntax: `STF operand`.

Attributes: Memory: Write.

Description: The operand selects a destination memory location, and the 32-bit contents of `F` are written there. Immediate, direct, and register addressing modes use `operand32` as the destination address; register-indirect and memory-indirect modes compute the destination address as described in section 4.0.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x47`.

Instruction Fields: Byte 0 holds opcode `0x47`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the destination. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the value stored is read from `F`.

Exceptions: None.

Notes: None.

28. STG - STG operand

Operation: `store(G, operand)`.

Assembler Syntax: `STG operand`.

Attributes: Memory: Write.

Description: The operand selects a destination memory location, and the 32-bit contents of `G` are written there. Immediate, direct, and register addressing modes use `operand32` as the destination address; register-indirect and memory-indirect modes compute the destination address as described in section 4.0.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x48`.

Instruction Fields: Byte 0 holds opcode `0x48`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the destination. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the value stored is read from `G`.

Exceptions: None.

Notes: None.

29. STU - STU operand

Operation: `store(U, operand)`.

Assembler Syntax: `STU operand`.

Attributes: Memory: Write.

Description: The operand selects a destination memory location, and the 32-bit contents of `U` are written there. Immediate, direct, and register addressing modes use `operand32` as the destination address; register-indirect and memory-indirect modes compute the destination address as described in section 4.0.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x49`.

Instruction Fields: Byte 0 holds opcode `0x49`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the destination. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the value stored is read from `U`.

Exceptions: None.

Notes: None.

30. STV - STV operand

Operation: `store(V, operand)`.

Assembler Syntax: `STV operand`.

Attributes: Memory: Write.

Description: The operand selects a destination memory location, and the 32-bit contents of `V` are written there. Immediate, direct, and register addressing modes use `operand32` as the destination address; register-indirect and memory-indirect modes compute the destination address as described in section 4.0.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x4A`.

Instruction Fields: Byte 0 holds opcode `0x4A`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the destination. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the value stored is read from `V`.

Exceptions: None.

Notes: None.

31. STW - STW operand

Operation: `store(W, operand)`.

Assembler Syntax: `STW operand`.

Attributes: Memory: Write.

Description: The operand selects a destination memory location, and the 32-bit contents of `W` are written there. Immediate, direct, and register addressing modes use `operand32` as the destination address; register-indirect and memory-indirect modes compute the destination address as described in section 4.0.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x4B`.

Instruction Fields: Byte 0 holds opcode `0x4B`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the destination. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the value stored is read from `W`.

Exceptions: None.

Notes: None.

32. STH - STH operand

Operation: `store(H, operand)`.

Assembler Syntax: `STH operand`.

Attributes: Memory: Write.

Description: The operand selects a destination memory location, and the 32-bit contents of `H` are written there. Immediate, direct, and register addressing modes use `operand32` as the destination address; register-indirect and memory-indirect modes compute the destination address as described in section 4.0.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x4F`.

Instruction Fields: Byte 0 holds opcode `0x4F`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the destination. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the value stored is read from `H`.

Exceptions: None.

Notes: None.

33. STS - STS operand

Operation: `store(S, operand)`.

Assembler Syntax: `STS operand`.

Attributes: Memory: Write.

Description: The operand selects a destination memory location, and the 32-bit contents of `S` are written there. Immediate, direct, and register addressing modes use `operand32` as the destination address; register-indirect and memory-indirect modes compute the destination address as described in section 4.0.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x50`.

Instruction Fields: Byte 0 holds opcode `0x50`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the destination. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the value stored is read from `S`.

Exceptions: None.

Notes: None.

34. STT - STT operand

Operation: `store(T, operand)`.

Assembler Syntax: `STT operand`.

Attributes: Memory: Write.

Description: The operand selects a destination memory location, and the 32-bit contents of `T` are written there. Immediate, direct, and register addressing modes use `operand32` as the destination address; register-indirect and memory-indirect modes compute the destination address as described in section 4.0.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x51`.

Instruction Fields: Byte 0 holds opcode `0x51`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the destination. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; the value stored is read from `T`.

Exceptions: None.

Notes: None.

Store address calculation:

Addressing mode	Store target
Register indirect	`memory32[base register + encoded offset]`
Memory indirect	`memory32[memory32[operand32]]`
Immediate	`memory32[operand32]`
Register	`memory32[register index]`, not the contents of that register
Direct	`memory32[operand32]`

This means that `STORE A, 0x5000`, `STORE A, #0x5000`, and `STORE A, @0x5000` all write `A` to address `0x5000`. `STORE A, X` writes to address `1` because `X` is register index 1. Use direct or register-indirect operands for stores.

All CPU load and store memory operations are 32-bit little-endian reads or writes. There are no byte, halfword, or 8-byte CPU load/store opcodes.

4.3 Arithmetic

35. ADD - ADD R, operand

Operation: `R = R + resolve(operand)`.

Assembler Syntax: `ADD R, operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved as a 32-bit value. The processor adds it with `R` and stores the 32-bit sum back in `R`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x03`.

Instruction Fields: Byte 0 holds opcode `0x03`. Byte 1 selects the destination/source register `R`; the CPU uses the low 4 bits. Byte 2 selects the addressing mode used to resolve the second operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; `operand32` is interpreted according to byte 2.

Exceptions: None.

Notes: None.

36. SUB - SUB R, operand

Operation: `R = R - resolve(operand)`.

Assembler Syntax: `SUB R, operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved as a 32-bit value. The processor subtracts it with `R` and stores the 32-bit difference back in `R`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x04`.

Instruction Fields: Byte 0 holds opcode `0x04`. Byte 1 selects the destination/source register `R`; the CPU uses the low 4 bits. Byte 2 selects the addressing mode used to resolve the second operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; `operand32` is interpreted according to byte 2.

Exceptions: None.

Notes: None.

37. MUL - MUL R, operand

Operation: `R = R * resolve(operand)`.

Assembler Syntax: `MUL R, operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved as a 32-bit value. The processor multiplies it with `R` and stores the 32-bit product back in `R`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x14`.

Instruction Fields: Byte 0 holds opcode `0x14`. Byte 1 selects the destination/source register `R`; the CPU uses the low 4 bits. Byte 2 selects the addressing mode used to resolve the second operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; `operand32` is interpreted according to byte 2.

Exceptions: None.

Notes: None.

38. DIV - DIV R, operand

Operation: `R = R / resolve(operand)`.

Assembler Syntax: `DIV R, operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved as an unsigned 32-bit divisor. If it is non-zero, the unsigned quotient replaces `R`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x15`.

Instruction Fields: Byte 0 holds opcode `0x15`. Byte 1 selects the destination/dividend register `R`. Byte 2 selects the addressing mode used to resolve the divisor operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; `operand32` is interpreted according to byte 2.

Exceptions: If the resolved divisor is zero, the CPU enters the stopped processor state.

Notes: None.

39. MOD - MOD R, operand

Operation: `R = R % resolve(operand)`.

Assembler Syntax: `MOD R, operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved as an unsigned 32-bit divisor. If it is non-zero, the unsigned remainder replaces `R`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x16`.

Instruction Fields: Byte 0 holds opcode `0x16`. Byte 1 selects the destination/dividend register `R`. Byte 2 selects the addressing mode used to resolve the divisor operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; `operand32` is interpreted according to byte 2.

Exceptions: If the resolved divisor is zero, the CPU enters the stopped processor state.

Notes: None.

40. INC - INC operand

Operation: Increment register or memory target selected by operand.

Assembler Syntax: `INC operand`.

Attributes: Memory: R/W for memory targets, N for register target.

Description: The operand selects a register or 32-bit memory location. The selected value is incremented by one and written back.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x28`.

Instruction Fields: Byte 0 holds opcode `0x28`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the read/write target. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; `operand32` is interpreted according to byte 2.

Exceptions: None.

Notes: None.

41. DEC - DEC operand

Operation: Decrement register or memory target selected by operand.

Assembler Syntax: `DEC operand`.

Attributes: Memory: R/W for memory targets, N for register target.

Description: The operand selects a register or 32-bit memory location. The selected value is decremented by one and written back.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x29`.

Instruction Fields: Byte 0 holds opcode `0x29`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the read/write target. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; `operand32` is interpreted according to byte 2.

Exceptions: None.

Notes: None.

Arithmetic is unsigned 32-bit arithmetic with wraparound modulo 2^32.

Division or modulo by zero enters the stopped processor state.

`INC` and `DEC` use the operand mode as the destination selector. Register mode mutates the selected register. Register-indirect, memory-indirect, direct, and immediate encodings mutate the resolved memory target as described in section 5.

Examples:

```
ADD A, #1
SUB X, Y
MUL A, @FACTOR
DIV A, [B+16]
MOD A, 10
```

4.4 Logical

42. AND - AND R, operand

Operation: `R = R & resolve(operand)`.

Assembler Syntax: `AND R, operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved as a 32-bit value. The processor performs a bitwise AND with `R` and stores the result back in `R`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x05`.

Instruction Fields: Byte 0 holds opcode `0x05`. Byte 1 selects the destination/source register `R`; the CPU uses the low 4 bits. Byte 2 selects the addressing mode used to resolve the second operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; `operand32` is interpreted according to byte 2.

Exceptions: None.

Notes: None.

43. OR - OR R, operand

Operation: `R = R \| resolve(operand)`.

Assembler Syntax: `OR R, operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved as a 32-bit value. The processor performs a bitwise OR with `R` and stores the result back in `R`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x09`.

Instruction Fields: Byte 0 holds opcode `0x09`. Byte 1 selects the destination/source register `R`; the CPU uses the low 4 bits. Byte 2 selects the addressing mode used to resolve the second operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; `operand32` is interpreted according to byte 2.

Exceptions: None.

Notes: None.

44. XOR - XOR R, operand

Operation: `R = R ^ resolve(operand)`.

Assembler Syntax: `XOR R, operand`.

Attributes: Memory: Depends on operand.

Description: The operand is resolved as a 32-bit value. The processor performs a bitwise exclusive-OR with `R` and stores the result back in `R`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x0A`.

Instruction Fields: Byte 0 holds opcode `0x0A`. Byte 1 selects the destination/source register `R`; the CPU uses the low 4 bits. Byte 2 selects the addressing mode used to resolve the second operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; `operand32` is interpreted according to byte 2.

Exceptions: None.

Notes: None.

45. NOT - NOT R

Operation: `R = ^R`.

Assembler Syntax: `NOT R`.

Attributes: Memory: None.

Description: The processor inverts every bit of `R` and stores the 32-bit result back in `R`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x0D`.

Instruction Fields: Byte 0 holds opcode `0x0D`. Byte 1 selects register `R`; the CPU uses the low 4 bits. Bytes 2-7 are reserved by this instruction and ignored by the processor.

Exceptions: None.

Notes: None.

`NOT` takes one register operand. It does not resolve the instruction's operand32 field.

4.5 Shifts

46. SHL - SHL R, operand

Operation: `R = R << resolve(operand)`.

Assembler Syntax: `SHL R, operand`.

Attributes: Memory: Depends on operand.

Description: The operand supplies the shift count. The processor shifts `R` left and writes the 32-bit result back in `R`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x0B`.

Instruction Fields: Byte 0 holds opcode `0x0B`. Byte 1 selects the destination/source register `R`; the CPU uses the low 4 bits. Byte 2 selects the addressing mode used to resolve the second operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; `operand32` is interpreted according to byte 2.

Exceptions: None.

Notes: None.

47. SHR - SHR R, operand

Operation: `R = R >> resolve(operand)`.

Assembler Syntax: `SHR R, operand`.

Attributes: Memory: Depends on operand.

Description: The operand supplies the shift count. The processor shifts `R` right logically and writes the 32-bit result back in `R`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x0C`.

Instruction Fields: Byte 0 holds opcode `0x0C`. Byte 1 selects the destination/source register `R`; the CPU uses the low 4 bits. Byte 2 selects the addressing mode used to resolve the second operand. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; `operand32` is interpreted according to byte 2.

Exceptions: None.

Notes: None.

Shifts are logical shifts on 32-bit unsigned values. The resolved operand is used directly as the shift count.

4.6 Branches

48. JMP - JMP target

Operation: `PC = target`.

Assembler Syntax: `JMP target`.

Attributes: Memory: None.

Description: The processor loads the branch target into `PC`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x06`.

Instruction Fields: Byte 0 holds opcode `0x06`. Bytes 1-3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold the absolute 32-bit target address in little-endian order.

Exceptions: None.

Notes: None.

49. JNZ - JNZ R, target

Operation: If `R != 0`, `PC = target`; else `PC += 8`.

Assembler Syntax: `JNZ R, target`.

Attributes: Memory: None.

Description: If `R` is non-zero, the processor loads `target` into `PC`; otherwise, execution continues at the next instruction.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x07`.

Instruction Fields: Byte 0 holds opcode `0x07`. Byte 1 selects the condition register `R`; the CPU uses the low 4 bits. Bytes 2-3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold the absolute 32-bit target address in little-endian order.

Exceptions: None.

Notes: None.

50. JZ - JZ R, target

Operation: If `R == 0`, `PC = target`; else `PC += 8`.

Assembler Syntax: `JZ R, target`.

Attributes: Memory: None.

Description: If `R` is zero, the processor loads `target` into `PC`; otherwise, execution continues at the next instruction.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x08`.

Instruction Fields: Byte 0 holds opcode `0x08`. Byte 1 selects the condition register `R`; the CPU uses the low 4 bits. Bytes 2-3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold the absolute 32-bit target address in little-endian order.

Exceptions: None.

Notes: None.

51. JGT - JGT R, target

Operation: If `int32(R) > 0`, `PC = target`; else `PC += 8`.

Assembler Syntax: `JGT R, target`.

Attributes: Memory: None.

Description: If `R`, interpreted as a signed 32-bit integer, is greater than zero, the processor loads `target` into `PC`; otherwise, execution continues at the next instruction.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x0E`.

Instruction Fields: Byte 0 holds opcode `0x0E`. Byte 1 selects the condition register `R`; the CPU uses the low 4 bits. Bytes 2-3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold the absolute 32-bit target address in little-endian order.

Exceptions: None.

Notes: None.

52. JGE - JGE R, target

Operation: If `int32(R) >= 0`, `PC = target`; else `PC += 8`.

Assembler Syntax: `JGE R, target`.

Attributes: Memory: None.

Description: If `R`, interpreted as a signed 32-bit integer, is greater than or equal to zero, the processor loads `target` into `PC`; otherwise, execution continues at the next instruction.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x0F`.

Instruction Fields: Byte 0 holds opcode `0x0F`. Byte 1 selects the condition register `R`; the CPU uses the low 4 bits. Bytes 2-3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold the absolute 32-bit target address in little-endian order.

Exceptions: None.

Notes: None.

53. JLT - JLT R, target

Operation: If `int32(R) < 0`, `PC = target`; else `PC += 8`.

Assembler Syntax: `JLT R, target`.

Attributes: Memory: None.

Description: If `R`, interpreted as a signed 32-bit integer, is less than zero, the processor loads `target` into `PC`; otherwise, execution continues at the next instruction.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x10`.

Instruction Fields: Byte 0 holds opcode `0x10`. Byte 1 selects the condition register `R`; the CPU uses the low 4 bits. Bytes 2-3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold the absolute 32-bit target address in little-endian order.

Exceptions: None.

Notes: None.

54. JLE - JLE R, target

Operation: If `int32(R) <= 0`, `PC = target`; else `PC += 8`.

Assembler Syntax: `JLE R, target`.

Attributes: Memory: None.

Description: If `R`, interpreted as a signed 32-bit integer, is less than or equal to zero, the processor loads `target` into `PC`; otherwise, execution continues at the next instruction.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x11`.

Instruction Fields: Byte 0 holds opcode `0x11`. Byte 1 selects the condition register `R`; the CPU uses the low 4 bits. Bytes 2-3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold the absolute 32-bit target address in little-endian order.

Exceptions: None.

Notes: None.

Branch, jump, and subroutine-call targets are absolute 32-bit addresses encoded in the operand field. The CPU loads that field directly into `PC` when the transfer is taken.

Conditional branches compare only the selected register against zero. They do not compare two registers and do not use a flags register.

4.7 Subroutine / Stack

55. PUSH - PUSH R

Operation: `SP -= 4; memory32[SP] = R`.

Assembler Syntax: `PUSH R`.

Attributes: Memory: Write.

Description: The processor decrements `SP` by four bytes and stores `R` at the new stack top.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x12`.

Instruction Fields: Byte 0 holds opcode `0x12`. Byte 1 selects register `R`; the CPU uses the low 4 bits. Bytes 2-7 are reserved by this instruction and ignored by the processor.

Exceptions: Stack overflow enters the stopped processor state.

Notes: None.

56. POP - POP R

Operation: `R = memory32[SP]; SP += 4`.

Assembler Syntax: `POP R`.

Attributes: Memory: Read.

Description: The processor loads a 32-bit value from the stack top into `R` and then increments `SP` by four bytes.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x13`.

Instruction Fields: Byte 0 holds opcode `0x13`. Byte 1 selects register `R`; the CPU uses the low 4 bits. Bytes 2-7 are reserved by this instruction and ignored by the processor.

Exceptions: Stack underflow enters the stopped processor state.

Notes: None.

57. JSR - JSR target

Operation: Push return address, then `PC = target`.

Assembler Syntax: `JSR target`.

Attributes: Memory: Write.

Description: The processor pushes the return address and then loads `target` into `PC`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x18`.

Instruction Fields: Byte 0 holds opcode `0x18`. Bytes 1-3 are reserved by this instruction and ignored by the processor. Bytes 4-7 hold the absolute 32-bit subroutine target address in little-endian order.

Exceptions: Stack overflow enters the stopped processor state.

Notes: None.

58. RTS - RTS

Operation: Pop return address into `PC`.

Assembler Syntax: `RTS`.

Attributes: Memory: Read.

Description: The processor pops a return address from the stack and loads it into `PC`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x19`.

Instruction Fields: Byte 0 holds opcode `0x19`. Bytes 1-7 are reserved by this instruction and ignored by the processor.

Exceptions: Stack underflow enters the stopped processor state.

Notes: None.

`JSR` pushes `PC + 8`, then jumps to the absolute target address in operand32. `RTS` pops a 32-bit return address from the stack.

Stack overflow on push or stack underflow on pop enters the stopped processor state.

4.8 Interrupt / Timer Control

59. SEI - SEI

Operation: Enable interrupt delivery.

Assembler Syntax: `SEI`.

Attributes: Memory: None.

Description: The processor enables maskable interrupt delivery.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x1A`.

Instruction Fields: Byte 0 holds opcode `0x1A`. Bytes 1-7 are reserved by this instruction and ignored by the processor.

Exceptions: None.

Notes: None.

60. CLI - CLI

Operation: Disable interrupt delivery.

Assembler Syntax: `CLI`.

Attributes: Memory: None.

Description: The processor disables maskable interrupt delivery.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x1B`.

Instruction Fields: Byte 0 holds opcode `0x1B`. Bytes 1-7 are reserved by this instruction and ignored by the processor.

Exceptions: None.

Notes: None.

61. RTI - RTI

Operation: Pop return address into `PC`; clear in-interrupt flag.

Assembler Syntax: `RTI`.

Attributes: Memory: Read.

Description: The processor pops the interrupted `PC` from the stack and clears the in-interrupt state.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x1C`.

Instruction Fields: Byte 0 holds opcode `0x1C`. Bytes 1-7 are reserved by this instruction and ignored by the processor.

Exceptions: Stack underflow enters the stopped processor state.

Notes: None.

62. WAIT - WAIT operand

Operation: Sleep for `resolve(operand)` microseconds.

Assembler Syntax: `WAIT operand`.

Attributes: Memory: Depends on operand.

Description: The processor waits for the number of microseconds selected by `resolve(operand)`.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0x17`.

Instruction Fields: Byte 0 holds opcode `0x17`. Byte 1 is reserved by this instruction and ignored by the processor. Byte 2 selects the addressing mode used to resolve the wait interval. Byte 3 is reserved by this instruction and ignored by the processor. Bytes 4-7 hold `operand32` in little-endian order; `operand32` is interpreted according to byte 2.

Exceptions: None.

Notes: None.

`SEI` and `CLI` manipulate the CPU's interrupt-enable latch. `RTI` pops the saved return address and clears the interrupt-active latch.

`WAIT` resolves its operand using the standard addressing modes. Non-zero values request a delay of that many microseconds. A zero value does not delay.

4.9 System

63. NOP - NOP

Operation: `PC += 8`.

Assembler Syntax: `NOP`.

Attributes: Memory: None.

Description: No registers or memory are modified; execution continues at the next 8-byte instruction.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0xEE`.

Instruction Fields: Byte 0 holds opcode `0xEE`. Bytes 1-7 are reserved by this instruction and ignored by the processor.

Exceptions: None.

Notes: None.

64. HALT - HALT

Operation: Stop execution.

Assembler Syntax: `HALT`.

Attributes: Memory: None.

Description: `HALT` enters the stopped processor state.

Condition Codes: No integer condition-code register is modified.

Instruction Format: Fixed 8-byte instruction; opcode = `0xFF`.

Instruction Fields: Byte 0 holds opcode `0xFF`. Bytes 1-7 are reserved by this instruction and ignored by the processor.

Exceptions: None.

Notes: `HALT` does not advance `PC`.

---

5. Addressing Modes

5.1 Immediate

Syntax:

```
#expr
```

Encoding:

Field	Value
mode	`0x00`
operand32	expression value

For loads and ALU instructions, immediate mode supplies the value directly. For stores and memory `INC`/`DEC`, immediate mode supplies the direct target address because those instructions write to `operand32` for all non-register-indirect and non-memory-indirect modes.

5.2 Bare Expression

Syntax:

```
expr
```

Bare-expression encodings use immediate mode. The encoded `operand32` field is therefore a value for loads and ALU instructions, but an address for stores and memory `INC`/`DEC`.

Example:

```
LOAD A, 0x5000   ; A = 0x5000
STORE A, 0x5000  ; memory32[0x5000] = A
```

5.3 Register

Syntax:

```
A
X
Y
Z
B
C
D
E
F
G
H
S
T
U
V
W
```

Encoding:

Field	Value
mode	`0x01`
operand32	register index

For loads and ALU instructions, register mode resolves to the selected register's current value. For `INC` and `DEC`, register mode increments or decrements the selected register.

Register mode is not a useful store destination: store instructions write to the numeric register index as a memory address.

5.4 Direct Memory

Syntax:

```
@expr
```

Encoding:

Field	Value
mode	`0x04`
operand32	expression value

For loads and ALU instructions, direct mode resolves to `memory32[operand32]`. For stores, direct mode writes to `memory32[operand32]`.

5.5 Register Indirect

Syntax:

```
[R]
[R+expr]
[R-expr]
```

Encoding:

Field	Value
mode	`0x02`
operand32 bits 0-3	base register index
operand32 bits 4-31	offset bits

The effective address is:

```
address = register[operand32 & 0x0F] + (operand32 & 0xFFFFFFF0)
```

The offset contribution must have its low 4 bits clear so those bits remain available for the base-register index. Use offsets that are multiples of 16. Negative offsets are encoded as two's-complement 32-bit values and preserve the low-nibble rule when they are multiples of 16.

Examples:

```
LDA [B]       ; memory32[B]
LDA [B+16]    ; memory32[B + 16]
LDA [B-16]    ; memory32[B - 16], modulo 32-bit address arithmetic
```

5.6 Memory Indirect

Encoding:

Field	Value
mode	`0x03`
operand32	address used by the memory-indirect mode

For load, ALU, branch-helper operand resolution, and `WAIT`, memory-indirect mode resolves exactly like direct mode:

```
value = memory32[operand32]
```

For stores and memory `INC`/`DEC`, memory-indirect mode uses `operand32` as the address of a 32-bit pointer to the final target address:

```
target = memory32[operand32]
memory32[target] = new value
```

No assembly-language source form is defined for this mode.

---

6. Branch Architecture

IE32 branch instructions use absolute 32-bit targets. There is no PC-relative branch encoding.

`JMP` and `JSR` unconditionally load `PC` from operand32. Conditional branch instructions use byte 1 to select a register and operand32 as the absolute target address.

Signed branch tests reinterpret the register value as `int32`:

Branch	Test
JNZ	`R != 0`
JZ	`R == 0`
JGT	`int32(R) > 0`
JGE	`int32(R) >= 0`
JLT	`int32(R) < 0`
JLE	`int32(R) <= 0`

The target operand for `JMP`, `JSR`, and conditional branches encodes an absolute 32-bit address.

---

7. Address Space and Reset Vectors

IE32 defines a flat 32-bit byte-addressed CPU address space. The ISA assigns only the reset, stack, and interrupt-vector conventions needed by the CPU itself.

Address / range	ISA role
`0x00000`	Interrupt vector table base. Timer interrupt entry reads a 32-bit handler address from this address.
`0x01000`	`PROG_START`. Reset value loaded into `PC`.
`0x02000`	`STACK_BOTTOM`. Normal stack overflow checks use this lower stack boundary.
`0x9F000`	`STACK_START`. Initial `SP`.

CPU load, store, stack, and interrupt-vector accesses transfer 32-bit little-endian words. Addresses outside the reset, stack, and interrupt-vector conventions above have no additional meaning in the IE32 CPU ISA.

---

8. Stack

The stack is full-descending and stores 32-bit little-endian words.

Initial state:

```
SP = 0x9F000
```

Push:

```
SP = SP - 4
memory32[SP] = value
```

Pop:

```
value = memory32[SP]
SP = SP + 4
```

Normal execution checks:

Operation	Failure condition
Inlined `PUSH` and `JSR` instruction execution	`SP < STACK_BOTTOM + 4` before decrement
Interrupt entry through the CPU push helper	`SP <= STACK_BOTTOM` before decrement
Pop / RTS / RTI	`SP >= STACK_START` before read

CPU stack operations maintain 32-bit word alignment for `SP`; with an aligned `SP`, the inlined instruction and interrupt-entry predicates reject the same lower boundary. `STACK_BOTTOM` is `0x2000`. Stack overflow or underflow enters the stopped processor state.

---

9. Timer and Interrupt Model

The IE32 timer is CPU-integrated. It is not controlled through an ISA-level externally addressable timer register block. Architecturally, the timer has the following state:

State element	Meaning
Timer enable latch	Enables or disables timer countdown.
Timer countdown	Current countdown value.
Timer reload period	Value reloaded after expiry when the timer remains enabled.
Timer state	Stopped, running, or expired.
Interrupt-enable latch	Enables or disables timer interrupt delivery.
Interrupt-active latch	Suppresses nested timer interrupt delivery while an interrupt handler is active.

Architectural timer constant:

Constant	Value	Meaning
`IE32_TIMER_DIVIDER`	44,100	Decoded-instruction-step divider for one timer countdown decrement.

When the timer is enabled, the CPU increments a timer prescaler once per decoded instruction step, after the instruction word and operand fields have been fetched and before the decoded instruction body is executed. When the prescaler reaches `IE32_TIMER_DIVIDER`, the prescaler is cleared and the timer countdown is decremented if it is non-zero. When the count reaches zero:

1. Timer state becomes expired.
2. If interrupts are enabled and the CPU is not already in an interrupt, interrupt handling is entered.
3. If the timer remains enabled, the countdown is reloaded from the timer reload period.

Interrupt entry pushes the current `PC` and then reads a 32-bit handler address from `VECTOR_TABLE` (`0x0000`) into `PC`.

`SEI` enables interrupt delivery. `CLI` disables interrupt delivery. `RTI` pops the saved `PC` and clears the in-interrupt flag.

There is no ISA-level externally addressable timer-control register. Timer state is controlled by the CPU-integrated timer state and the instruction-level interrupt controls above.

9.1 Timer State Transitions

Event	State change
Timer disabled	Timer countdown state is not advanced.
Enabled timer completes operand resolution	The timer prescaler increments by one before the decoded instruction body executes.
Timer prescaler below `IE32_TIMER_DIVIDER`	No timer countdown change occurs.
Timer prescaler reaches `IE32_TIMER_DIVIDER` and countdown is non-zero	The prescaler clears and the timer countdown decrements.
Decrement reaches zero	Timer state becomes expired; interrupt entry occurs only when interrupt delivery is enabled and the interrupt-active latch is clear.
Timer remains enabled after expiry	The countdown reloads from the timer reload period.
`RTI` executes	The saved `PC` is popped and the interrupt-active latch is cleared.

---

10. Architectural Caveats

10.1 Encodable Mode Coverage

The CPU decodes the mode byte directly. Encoded instruction streams can use memory-indirect mode `0x03`; no assembly-language source form is defined for that mode.

10.2 Store Register Operand Hazard

Register operands for store instructions name register indexes, not indirect addresses. Stores do not treat register mode as "store to address contained in that register".

Example:

```
STORE A, X
```

This writes `A` to address `1`, because `X` is register index 1. To store through a register, use:

```
STORE A, [X]
```

10.3 Register-Indirect Offset Granularity

Register-indirect offset encoding stores the register index in the low 4 bits of operand32. The CPU masks those bits out to recover the offset. Encodable register-indirect offsets must therefore be multiples of 16.

Appendix A: Opcode Map

A.1 Instruction Set Summary

Opcode	Operation	Syntax
LOAD	Data movement	`LOAD R, operand`

Opcode	Operation	Syntax
STORE	Load/store	`STORE R, operand`
ADD	Arithmetic	`ADD R, operand`
SUB	Arithmetic	`SUB R, operand`
AND	Logical	`AND R, operand`
JMP	Branch	`JMP target`
JNZ	Branch	`JNZ R, target`
JZ	Branch	`JZ R, target`
OR	Logical	`OR R, operand`
XOR	Logical	`XOR R, operand`
SHL	Shift	`SHL R, operand`
SHR	Shift	`SHR R, operand`
NOT	Logical	`NOT R`
JGT	Branch	`JGT R, target`
JGE	Branch	`JGE R, target`
JLT	Branch	`JLT R, target`
JLE	Branch	`JLE R, target`
PUSH	Stack	`PUSH R`
POP	Stack	`POP R`
MUL	Arithmetic	`MUL R, operand`
DIV	Arithmetic	`DIV R, operand`
MOD	Arithmetic	`MOD R, operand`
WAIT	Timer	`WAIT operand`
JSR	Stack / branch	`JSR target`
RTS	Stack / branch	`RTS`
SEI	Interrupt	`SEI`
CLI	Interrupt	`CLI`
RTI	Interrupt	`RTI`
LDA	Data movement	`LDA operand`
LDX	Data movement	`LDX operand`
LDY	Data movement	`LDY operand`
LDZ	Data movement	`LDZ operand`
STA	Load/store	`STA operand`
STX	Load/store	`STX operand`
STY	Load/store	`STY operand`

Opcode	Operation	Syntax
STZ	Load/store	`STZ operand`
INC	Arithmetic	`INC operand`
DEC	Arithmetic	`DEC operand`
LDB	Data movement	`LDB operand`
LDC	Data movement	`LDC operand`
LDD	Data movement	`LDD operand`
LDE	Data movement	`LDE operand`
LDF	Data movement	`LDF operand`
LDG	Data movement	`LDG operand`
LDU	Data movement	`LDU operand`
LDV	Data movement	`LDV operand`
LDW	Data movement	`LDW operand`
STB	Load/store	`STB operand`
STC	Load/store	`STC operand`
STD	Load/store	`STD operand`
STE	Load/store	`STE operand`
STF	Load/store	`STF operand`
STG	Load/store	`STG operand`
STU	Load/store	`STU operand`
STV	Load/store	`STV operand`
STW	Load/store	`STW operand`
LDH	Data movement	`LDH operand`
LDS	Data movement	`LDS operand`
LDT	Data movement	`LDT operand`
STH	Load/store	`STH operand`
STS	Load/store	`STS operand`
STT	Load/store	`STT operand`
NOP	System	`NOP`
HALT	System	`HALT`

A.2 Machine Opcode Encoding Map

Opcode Byte	Mnemonic	Category	Operands
`0x01`	LOAD	Data movement	R, operand
`0x02`	STORE	Load/store	R, operand

Opcode Byte	Mnemonic	Category	Operands
0x03	ADD	Arithmetic	R, operand
0x04	SUB	Arithmetic	R, operand
0x05	AND	Logical	R, operand
0x06	JMP	Branch	target
0x07	JNZ	Branch	R, target
0x08	JZ	Branch	R, target
0x09	OR	Logical	R, operand
0x0A	XOR	Logical	R, operand
0x0B	SHL	Shift	R, operand
0x0C	SHR	Shift	R, operand
0x0D	NOT	Logical	R
0x0E	JGT	Branch	R, target
0x0F	JGE	Branch	R, target
0x10	JLT	Branch	R, target
0x11	JLE	Branch	R, target
0x12	PUSH	Stack	R
0x13	POP	Stack	R
0x14	MUL	Arithmetic	R, operand
0x15	DIV	Arithmetic	R, operand
0x16	MOD	Arithmetic	R, operand
0x17	WAIT	Timer	operand
0x18	JSR	Stack / branch	target
0x19	RTS	Stack / branch	none
0x1A	SEI	Interrupt	none
0x1B	CLI	Interrupt	none
0x1C	RTI	Interrupt	none
0x20	LDA	Data movement	operand
0x21	LDX	Data movement	operand
0x22	LDY	Data movement	operand
0x23	LDZ	Data movement	operand
0x24	STA	Load/store	operand
0x25	STX	Load/store	operand
0x26	STY	Load/store	operand
0x27	STZ	Load/store	operand

Opcode Byte	Mnemonic	Category	Operands
0x28	INC	Arithmetic	operand
0x29	DEC	Arithmetic	operand
0x3A	LDB	Data movement	operand
0x3B	LDC	Data movement	operand
0x3C	LDD	Data movement	operand
0x3D	LDE	Data movement	operand
0x3E	LDF	Data movement	operand
0x3F	LDG	Data movement	operand
0x40	LDU	Data movement	operand
0x41	LDV	Data movement	operand
0x42	LDW	Data movement	operand
0x43	STB	Load/store	operand
0x44	STC	Load/store	operand
0x45	STD	Load/store	operand
0x46	STE	Load/store	operand
0x47	STF	Load/store	operand
0x48	STG	Load/store	operand
0x49	STU	Load/store	operand
0x4A	STV	Load/store	operand
0x4B	STW	Load/store	operand
0x4C	LDH	Data movement	operand
0x4D	LDS	Data movement	operand
0x4E	LDT	Data movement	operand
0x4F	STH	Load/store	operand
0x50	STS	Load/store	operand
0x51	STT	Load/store	operand
0xEE	NOP	System	none
0xFF	HALT	System	none

A.3 Opcode Ranges

Range	Category
$01-$05	Core data movement, load/store, arithmetic, and logical operations
$06-$11	Branches
$12-$19	Stack, subroutine, timer, and return operations

Range	Category
`$1A-$1C`	Interrupt control
`$20-$29`	Primary register load/store and increment/decrement operations
`$3A-$51`	Extended register load/store operations
`$EE`	No operation
`$FF`	Halt

Opcodes not listed above are reserved. Executing a reserved opcode enters the stopped processor state without architecturally advancing PC.

A.4 Addressing Mode Summary

Mode	Name	Assembler syntax	Architectural resolution
`0x00`	Immediate	`#expr` or bare `expr`	`operand32`
`0x01`	Register	`R`	`register[operand32 & 0x0F]`
`0x02`	Register indirect	`[R]`, `[R+expr]`, `[R-expr]`	`memory32[register[operand32 & 0x0F] + (operand32 & 0xFFFFFFF0)]`
`0x03`	Memory indirect	none	Read operands: `memory32[operand32]`. Store and memory `INC`/`DEC` targets: `memory32[memory32[operand32]]`.
`0x04`	Direct	`@expr`	`memory32[operand32]`

Appendix B: Encoding Examples

B.1 `LDA #$12345678`

Fields:

Field	Value
opcode	`0x20`
reg	`0x00`
mode	`0x00`
zero	`0x00`
operand32	`0x12345678`

Bytes:

```
20 00 00 00 78 56 34 12
```

B.2 `LOAD X, A`

Fields:

Field	Value
opcode	`0x01`
reg	`0x01`
mode	`0x01`
zero	`0x00`
operand32	`0x00000000`

Bytes:

```
01 01 01 00 00 00 00 00
```

B.3 `STA @0x5000`

Fields:

Field	Value
opcode	`0x24`
reg	`0x00`
mode	`0x04`
zero	`0x00`
operand32	`0x00005000`

Bytes:

```
24 00 04 00 00 50 00 00
```

B.4 `LDA [B+16]`

`B` is register index 4. The encoded operand is `0x10 | 0x04 = 0x14`.

Fields:

Field	Value
opcode	`0x20`
reg	`0x00`
mode	`0x02`
zero	`0x00`
operand32	`0x00000014`

Bytes:

```
20 00 02 00 14 00 00 00
```

B.5 `JNZ A, loop`

If `loop` resolves to `0x00001020`:

Fields:

Field	Value
opcode	`0x07`
reg	`0x00`
mode	`0x00`
zero	`0x00`
operand32	`0x00001020`

Bytes:

```
07 00 00 00 20 10 00 00
```

B.6 `PUSH W`

`W` is register index 15.

Fields:

Field	Value
opcode	`0x12`
reg	`0x0F`
mode	`0x00`
zero	`0x00`
operand32	`0x00000000`

Bytes:

```
12 0F 00 00 00 00 00 00
```