

Chapter 25 - IE64

IE64 is the main processor of the Intuition Engine. The other CPUs in Part IV are compatibility processors with their own maps and calling rules. IE64 is where BASIC itself lives and where system routines use the widest register set.

It is a clean 64-bit RISC load-store architecture with fixed 8-byte instructions, 32 general-purpose registers, and no flags. Comparisons fold into the branches that consume them.

25.1 Register file

IE64 has 32 general-purpose registers, each 64 bits wide:

Register	Role
R0	Hardwired zero. Writes are silently ignored; reads always return 0.
R1-R7	Caller-saved scratch. Free for any subroutine to clobber.
R8-R15	Argument and return registers (by convention). R8 is the first argument and the primary return value.
R16-R30	Callee-saved (by convention). IE64 BASIC reserves R16, R17, R26, R27 for BASIC-wide use.
R31	Stack pointer (SP). Decrements before push, increments after pop.

R0 = zero is a hardware fact; the rest are conventions. Nothing in silicon stops a program from using R31 for general data and keeping the stack elsewhere, but every interrupt handler and every library routine assumes the convention.

The FPU adds 16 separate registers, F0-F15, each holding one IEEE 754 single-precision (FP32) value in its low 32 bits. Double-precision instructions operate on pairs of F registers.

Three architecturally visible state words sit beside the GPRs:

- PC - the program counter, 64 bits.
- FPSR / FPCR - floating-point status and control. See section 24.5.
- A bank of 16 control registers CR0-CR15. See section 24.7.

25.2 Instruction encoding

Every instruction is exactly 8 bytes long, little-endian, with this layout:

```
byte 0 : opcode           (8 bits)
byte 1 : Rd[4:0] in bits 7-3, size[1:0] in bits 2-1, X in bit 0
byte 2 : Rs[4:0] in bits 7-3, reserved in bits 2-0
byte 3 : Rt[4:0] in bits 7-3, reserved in bits 2-0
bytes 4-7 : imm32          (32 bits)
```

To encode a byte by hand: $\text{byte1} = (\text{Rd} \ll 3) \mid (\text{size} \ll 1) \mid X$, $\text{byte2} = \text{Rs} \ll 3$, and $\text{byte3} = \text{Rt} \ll 3$.

Size codes are B = 0 (8-bit), W = 1 (16-bit), L = 2 (32-bit), Q = 3 (64-bit). An instruction that does not need a size code ignores those bits.

Fixed length makes the decoder simple and gives the disassembler a predictable cadence: each address \$1000, \$1008, \$1010, ... holds one full instruction.

Instructions are always aligned to 8 bytes. A branch target that is not so aligned faults.

25.3 Addressing modes

The load-store architecture means only LOAD and STORE (and their floating-point and atomic siblings) touch memory; every other operation works register-to-register.

LOAD/STORE accept:

- **Register indirect:** `LOAD.x Rd, (Rs)` reads x bytes from the address in Rs .
- **Register-plus-displacement:** `LOAD.x Rd, imm(Rs)` reads x bytes from $Rs + imm32$. The displacement is signed.
- **Absolute:** `LOAD.x Rd, (imm)` reads x bytes from the absolute address $imm32$.

The size code `.B`, `.W`, `.L`, `.Q` selects the access width. `LOAD.B` zero-extends a byte into a 64-bit register; sign-extend explicitly with the `SXT` instruction if you want a negative byte to sign-extend.

LEA (Load Effective Address) computes the address that a LOAD/STORE would have used, without actually touching memory: `LEA Rd, imm(Rs)` puts $Rs + imm32$ in Rd .

For byte-entered programs, use `MOVE` for a low 32-bit immediate and add `MOVT` when you need to place non-zero bits in the upper half of a 64-bit address.

25.4 Integer instructions

25.4.1 Data movement

Mnemonic	Effect
<code>MOVE Rd, Rs</code>	Copy Rs into Rd
<code>MOVE.Q Rd, #imm</code>	Move sign-extended 32-bit immediate into Rd
<code>MOVT Rd, #imm</code>	Move into upper 32 bits of Rd (paired with <code>MOVE</code> for full 64-bit constants)
<code>MOVEQ Rd, Rs</code>	Sign-extend low 32 bits of Rs into 64-bit Rd
<code>LEA Rd, imm(Rs)</code>	Address calculation, no memory access

25.4.2 Memory

Mnemonic	Effect
<code>LOAD.x Rd, ...</code>	Load x bytes from memory into Rd (zero-extended)
<code>STORE.x Rd, ...</code>	Store low x bytes of Rd to memory

25.4.3 Arithmetic

`ADD`, `SUB`, `MULU`, `MULS`, `DIVU`, `DIVS`, `MOD`, `MODS`, `NEG`, `MULHU`, `MULHS`. All three-operand register-register, all 64-bit. The two `MULH` variants return the high 64 bits of a 64×64 product.

Integer division or modulo by zero writes 0 to the destination register and does not trap. Programs that anticipate untrusted divisors should still check the divisor against zero first, because 0 is a result value as well as the divide-by-zero fallback.

25.4.4 Logic and shifts

AND, OR, EOR (XOR), NOT. Shifts: LSL, LSR, ASR, ROL, ROR. Bit utilities: CLZ (count leading zeros), CTZ (count trailing zeros), POPCNT, BSWAP (32-bit byte swap).

SEXT sign-extends a sub-64-bit value to the full register width; size code selects the source width.

25.4.5 Branches

IE64 has no flags register. Each branch instruction is a compare-and-branch in one step:

Mnemonic	Branches if (signed unless noted)
BRA	unconditional
BEQ Rs, Rt, label	Rs == Rt
BNE Rs, Rt, label	Rs != Rt
BLT Rs, Rt, label	Rs < Rt
BGE Rs, Rt, label	Rs >= Rt
BGT Rs, Rt, label	Rs > Rt
BLE Rs, Rt, label	Rs <= Rt
BHI Rs, Rt, label	Rs > Rt (unsigned)
BLS Rs, Rt, label	Rs <= Rt (unsigned)
JMP Rs	unconditional register-indirect jump

To branch on zero, compare against R0: BEQ Rs, R0, target. The same pattern works for the other conditional branches.

25.4.6 Subroutines

Mnemonic	Effect
JSR label	Push return PC to stack (R31), jump to label
JSR Rs	Push return PC, jump to Rs (register-indirect)
RTS	Pop return PC from stack, jump there
PUSH Rs	R31 -= 8; [R31] = Rs
POP Rd	Rd = [R31]; R31 += 8

The stack grows downward. Standalone IE64 assembly programs commonly initialise R31 from STACK_TOP; IE64 BASIC initialises its own stack dynamically from active visible RAM.

25.5 Floating-point unit

The FPU implements IEEE 754 single-precision (FP32) on F registers, plus a double-precision (FP64) extension that operates on register pairs. Operations are register-to-register. Memory operands transit through FLOAD/FSTORE for FP32 and DLOAD/DSTORE for FP64.

Mnemonic	Operation
FMOV	Copy F register

Mnemonic	Operation
FLOAD	Memory -> F register (32 bits)
FSTORE	F register -> memory (32 bits)
FADD/FSUB/FMUL/FDIV/FMOD	Basic arithmetic
FABS/FNEG/FSQRT	Unary
FINT	Round to integer (mode set in FPCR)
FCMP	Compare two FP values, write integer - 1/0/1 to a GPR
FCVTIF	Integer -> float
FCVTFI	Float -> integer (truncate towards zero)
FMOVI	Bitwise GPR -> F (no conversion)
FMOV0	Bitwise F -> GPR
FSIN/FCOS/FTAN/FATAN	Trig
FLOG/FEXP/FPOW	Log / exp / power
FMOVECR	Load a ROM constant by index
FMOVSR/FMOVCR	Read FPSR / FPCR
FMOVSC/FMOVCC	Write FPSR / FPCR
DMOV	Copy an FP64 register pair
DLOAD	Memory -> FP64 register pair (64 bits)
DSTORE	FP64 register pair -> memory (64 bits)
DADD/DSUB/DMUL/DDIV/DMOD/DPOW	FP64 arithmetic
DABS/DNEG/DSQRT/DINT	FP64 unary
DSIN/DCOS/DTAN/DATAN	FP64 trigonometry
DLOG/DEXP	FP64 natural log and exponent
DCMP	Compare two FP64 values
DCVTIF	Integer -> FP64
DCVTFI	FP64 -> integer (truncate towards zero)

Double-precision (DMOV, DLOAD, DSTORE, DADD, ... DCMP, DCVTIF, DCVTFI, DSIN, DCOS, DTAN, DATAN, DLOG, DEXP, and DPOW) operate on even-numbered FP register pairs that hold a single 64-bit value across two F slots. FCVTSF and FCVTFD convert between the two formats.

FPCR (FP control) holds the rounding mode (0=nearest, 1= towards zero, 2=down, 3=up). IE64 BASIC sets it to 1 at startup because INT(x) is defined to truncate towards zero.

FPSR (FP status) latches exception flags: inexact, overflow, underflow, invalid, divide-by-zero. Read with FMOVSR. The FPU does not trap on these flags; software polls FPSR if it cares.

25.6 Atomic memory operations

Six atomic read-modify-write instructions, each operating on a naturally-aligned 64-bit word:

Mnemonic	Operation
CAS Rd, Rt, (Rs)	If [Rs] == Rd then [Rs] = Rt; Rd = old
XCHG Rd, Rt, (Rs)	old = [Rs]; [Rs] = Rt; Rd = old
FAA Rd, Rt, (Rs)	old = [Rs]; [Rs] = old + Rt; Rd = old
FAND Rd, Rt, (Rs)	old = [Rs]; [Rs] = old & Rt; Rd = old
FOR Rd, Rt, (Rs)	old = [Rs]; [Rs] = old \ Rt; Rd = old
FXOR Rd, Rt, (Rs)	old = [Rs]; [Rs] = old ^ Rt; Rd = old

A misaligned address (not divisible by 8) raises FAULT_MISALIGNED. All six are sequentially consistent on the memory subsystem.

25.7 Privilege and the MMU

IE64 has two privilege levels: supervisor and user. A freshly loaded IE64 program starts in supervisor mode. IE64 BASIC also runs in supervisor mode and uses the full address space directly.

A program that wants user-mode isolation enables the MMU through the CR_MMU_CTRL control register.

25.7.1 Control registers

MTCR (Move To Control Register) and MFCR (Move From) access the bank:

CR index	Name	Purpose
0	PTBR	Page-table base register (physical)
1	FAULT_ADDR	VA that caused the last fault
2	FAULT_CAUSE	Fault cause code
3	FAULT_PC	PC saved at trap entry
4	TRAP_VEC	Trap handler vector
5	MMU_CTRL	MMU enable + protection flags
6	TP	Thread pointer (user-readable)
7	INTR_VEC	Timer-interrupt vector
8	KSP	Supervisor stack pointer
9	TIMER_PERIOD	Timer reload value, in timer steps
10	TIMER_COUNT	Current timer-step countdown
11	TIMER_CTRL	b0 enable, b1 IRQ enable
12	USP	Saved user stack pointer
13	PREV_MODE	Previous privilege mode

CR index	Name	Purpose
14	SAVED_SUA	Saved SUA latch
15	RAM_SIZE_BYTES	Live read of active visible RAM

CR_RAM_SIZE_BYTES is read-only; writing it raises FAULT_ILLEGAL_INSTRUCTION.

The timer counts decoded IE64 instruction steps, not wall-clock time and not platform CPU cycles. TIMER_PERIOD is the reload value for that step counter. Use WAIT when you need a short microsecond delay.

25.7.2 The MMU

The MMU walks a sparse 6-level radix page table. The top level indexes 7 bits of the virtual page number (128 entries); the five lower levels index 9 bits each (512 entries each), so the full VPN is 52 bits. Page size is 4 kilobytes.

Each page-table entry is 64 bits: the low 7 bits hold flags (P, R, W, X, U, A, D); bits 12-63 hold the physical page number. A missing PTE raises FAULT_NOT_PRESENT. A read of a non-readable page raises FAULT_READ_DENIED, and so on.

The same page-table shape is used whenever an Intuition Engine service honours an IE64 user pointer. CPU loads and stores may update accessed and dirty bits as part of normal translation. Service bridges use the same physical result, then apply their own read, write, and user-page checks before touching memory.

MMU_CTRL controls translation:

Bit	Name	Effect
0	ENABLE	Translation active
1	SUPER	Read-only: current mode is supervisor
2	SKEF	Fault on supervisor instruction fetch from a user page
3	SKAC	Fault on supervisor data access to a user page (outside SUA region)
4	SUA	Supervisor-user-access latch (set by SUAEN, cleared by SUADIS)

SKEF/SKAC together give the equivalent of SMEP/SMAP: a supervisor that has set them cannot accidentally execute or read user-controlled memory.

25.7.3 Traps

The SYSCALL instruction is the user-to-supervisor doorway. It saves PC in FAULT_PC, sets FAULT_CAUSE to FAULT_SYSCALL, swaps to the supervisor stack (KSP), and jumps to TRAP_VEC. The imm32 field of the instruction is the syscall number, available to the handler.

ERET (Exception Return) restores PC from FAULT_PC, restores the previous mode from PREV_MODE, and swaps the user stack back in. Chapter 31 details the full trap sequence. Nested trap state is held by the CPU's trap-frame stack, so an ordinary handler does not need to save FAULT_PC or SAVED_SUA merely to survive a nested trap. A handler may still write SAVED_SUA deliberately before ERET if it wants to stage a different latch value.

TLBFLUSH invalidates the whole address-translation cache. TLBINVAL Rs treats Rs as a virtual address and invalidates the single cached entry for that address's virtual page.

SMODE Rd reports the current mode in Rd (0 user, 1 supervisor). Unlike most privilege-related instructions, SMODE is available to user code.

25.8 Calling convention

The convention used by IE64 BASIC and system routines is:

- Up to 8 integer arguments in R8-R15.
- Return value in R8.
- R1-R7 caller-saved.
- R16-R30 callee-saved.
- Stack R31, grows downward, aligned to 8 bytes.
- Floating-point arguments in F0-F7, return in F0.

Subroutines either preserve R16-R30 or push them on entry and pop on return.

25.9 Reset and boot

On reset the CPU starts at \$0000, the VECTOR_RESET slot. The ordinary convention is for the reset vector to do nothing more than jump to PROG_START = \$1000, the usual start address for monitor-entered IE64 programs and loaded IE64 images.

VECTOR_IRQ = \$0004 is the interrupt vector when the MMU is disabled. With the MMU enabled, INTR_VEC takes over.

The initial stack pointer for a plain IE64 assembly program is commonly loaded from STACK_TOP. IE64 BASIC does not use that fixed value; it reserves a dynamic stack near the top of active visible RAM.

25.9.1 Flat IE64 images

A flat .ie64 image is copied to PROG_START, and execution begins there. BASIC's COMPILE "name" command writes this same image form: the saved file is ordinary IE64 machine code and can later be started with RUN "name.ie64".

The loader checks the whole image before it copies any byte. If the image cannot fit in active RAM at PROG_START, the load is refused; the old RAM contents and the IE64 program counter are left unchanged. This makes a failed large image load a clean error rather than a half-loaded program.

25.9.2 Source made inside the machine

BASIC can also make IE64 source and assemble it from the prompt. The command TRANSPILE "name" writes the stored BASIC program as self-contained IE64 assembly text. The command ASSEMBLE "name" reads the matching assembly source, assembles it inside the machine at PROG_START, and writes name.ie64.

This is still an Intuition Engine workflow. The source may use IE64 instructions, labels, dc.b, dc.w, dc.l, dc.q, align, and the standard symbolic constants known to the in-machine assembler. A conventional constants include line is accepted as a no-op, because those constants are already known.

The source may also use MOVT for the upper half of a 64-bit constant and the zero-test branch forms BEQZ, BNEZ, BLTZ, BGEZ, BGTZ, and BLEZ. These are assembler forms. They assemble to the ordinary IE64 compare-and-branch instructions with the second register set to R0.

For generated BASIC source, TRANSPILE "name" followed by ASSEMBLE "name" produces the same kind of flat image as COMPILE "name". Chapter 35 gives the file command details and error rules.

The BASIC AOT path uses direct IE64 for its supported integer subset. Scalar numeric variables are predeclared in the normal BASIC variable table, but native integer reads load their payload slots directly and do not call the BASIC tag conversion helpers in hot arithmetic, MMIO, IF, or simple FOR . . . NEXT paths. Decimal and &H hexadecimal integer literals in that subset may use 64-bit values, so explicit wrap calculations can stay native. RUN AOT, TRANSPILE, COMPILE,

and ASSEMBLE size generated text and native-code buffers from the active top-of-RAM AOT arena, so generated BASIC programmes fail cleanly only when they exceed available active guest RAM.

25.10 A small example

This IE64 monitor-entered program turns on the main SoundChip and starts a three-voice C major chord. It uses the shared bus directly: IE64 writes the same SoundChip registers that BASIC's SOUND command writes in Chapter 11.

IE64 has one convenience the other CPU chapters do not have: IE Mon can assemble one IE64 instruction at a time with A addr. That is still a native monitor workflow. It writes RAM immediately, prints the bytes it emitted, and advances by one fixed 8-byte instruction. The byte listing below remains the proof copy, and d remains the check before you run the code.

Here is the first block using A mode:

```
(ie64)> A 1000
IE64 assemble at $0000000000001000; empty line exits
asm $0000000000001000> move.q r2,$F0800
$0000000000001000: 01 17 00 00 00 08 0F 00  move.q r2, #$F0800
asm $0000000000001008> move.q r1,#1
$0000000000001008: 01 0F 00 00 01 00 00 00  move.q r1, #1
asm $0000000000001010> store.b r1,(r2)
$0000000000001010: 11 08 10 00 00 00 00 00  store.b r1, (r2)
asm $0000000000001018> move.q r2,$F0900
$0000000000001018: 01 17 00 00 00 09 0F 00  move.q r2, #$F0900
asm $0000000000001020> move.q r1,$10600
$0000000000001020: 01 0F 00 00 00 06 01 00  move.q r1, #$10600
asm $0000000000001028> store.l r1,(r2)
$0000000000001028: 11 0C 10 00 00 00 00 00  store.l r1, (r2)
asm $0000000000001030>
Exited IE64 assemble mode
```

The monitor has written the same first six instructions shown in the byte listing. Continue in the same way from the disassembly column below, or type the complete byte form directly.

Type it in IE Mon:

```

(ie64)> w 1000 01 17 00 00 00 08 0F 00 01 0F 00 00 01 00 00 00
(ie64)> w 1010 11 08 10 00 00 00 00 00 01 17 00 00 00 09 0F 00
(ie64)> w 1020 01 0F 00 00 00 06 01 00 11 0C 10 00 00 00 00 00
(ie64)> w 1030 01 0F 00 00 BE 00 00 00 11 08 10 00 04 00 00 00
(ie64)> w 1040 01 0F 00 00 02 00 00 00 11 08 10 00 08 00 00 00
(ie64)> w 1050 01 17 00 00 40 09 0F 00 01 0F 00 00 00 4A 01 00
(ie64)> w 1060 11 0C 10 00 00 00 00 00 01 0F 00 00 96 00 00 00
(ie64)> w 1070 11 08 10 00 04 00 00 00 01 0F 00 00 02 00 00 00
(ie64)> w 1080 11 08 10 00 08 00 00 00 01 17 00 00 80 09 0F 00
(ie64)> w 1090 01 0F 00 00 00 88 01 00 11 0C 10 00 00 00 00 00
(ie64)> w 10A0 01 0F 00 00 82 00 00 00 11 08 10 00 04 00 00 00
(ie64)> w 10B0 01 0F 00 00 02 00 00 00 11 08 10 00 08 00 00 00
(ie64)> w 10C0 40 06 00 00 00 00 00 00
(ie64)> d 1000 #25
00001000: 01 17 00 00 00 08 0F 00  move.q r2, #$F0800
00001008: 01 0F 00 00 01 00 00 00  move.q r1, #$1
00001010: 11 08 10 00 00 00 00 00  store.b r1, (r2)
00001018: 01 17 00 00 00 09 0F 00  move.q r2, #$F0900
00001020: 01 0F 00 00 00 06 01 00  move.q r1, #$10600
00001028: 11 0C 10 00 00 00 00 00  store.l r1, (r2)
00001030: 01 0F 00 00 BE 00 00 00  move.q r1, #$BE
00001038: 11 08 10 00 04 00 00 00  store.b r1, 4(r2)
00001040: 01 0F 00 00 02 00 00 00  move.q r1, #$2
00001048: 11 08 10 00 08 00 00 00  store.b r1, 8(r2)
00001050: 01 17 00 00 40 09 0F 00  move.q r2, #$F0940
00001058: 01 0F 00 00 00 4A 01 00  move.q r1, #$14A00
00001060: 11 0C 10 00 00 00 00 00  store.l r1, (r2)
00001068: 01 0F 00 00 96 00 00 00  move.q r1, #$96
00001070: 11 08 10 00 04 00 00 00  store.b r1, 4(r2)
00001078: 01 0F 00 00 02 00 00 00  move.q r1, #$2
00001080: 11 08 10 00 08 00 00 00  store.b r1, 8(r2)
00001088: 01 17 00 00 80 09 0F 00  move.q r2, #$F0980
00001090: 01 0F 00 00 00 88 01 00  move.q r1, #$18800
00001098: 11 0C 10 00 00 00 00 00  store.l r1, (r2)
000010A0: 01 0F 00 00 82 00 00 00  move.q r1, #$82
000010A8: 11 08 10 00 04 00 00 00  store.b r1, 4(r2)
000010B0: 01 0F 00 00 02 00 00 00  move.q r1, #$2
000010B8: 11 08 10 00 08 00 00 00  store.b r1, 8(r2)
T 000010C0: 40 06 00 00 00 00 00 00  bra $0010C0
(ie64)> r pc 1000
(ie64)> b 10C0
(ie64)> g
(ie64)> m F0900 1
000000000000F0900: 00 06 01 00 BE 00 00 00 02 00 00 00 00 00 00 00 .....
(ie64)> m F0940 1
000000000000F0940: 00 4A 01 00 96 00 00 00 02 00 00 00 00 00 00 00 .J.....
(ie64)> m F0980 1
000000000000F0980: 00 88 01 00 82 00 00 00 02 00 00 00 00 00 00 00 .....
(ie64)> bc 10C0

```

You should hear a steady three-voice chord. The memory dumps show the three legacy SoundChip channel blocks after IE64 has written them: square at \$F0900, triangle at \$F0940, and sine at \$F0980.

The byte groups are short enough to audit by hand:

Address	Bytes	Meaning
\$1000	01 17 00 00 00 08 0F 00	OP_MOVE, Rd=2, size Q, immediate present. Loads \$F0800 into R2. Byte 17 is $(2 \ll 3) \mid (3 \ll 1) \mid 1$.
\$1008	01 0F 00 00 01 00 00 00	Loads 1 into R1. Byte 0F is R1, size Q, immediate present.
\$1010	11 08 10 00 00 00 00 00	STORE.B R1, (R2). Byte 08 selects R1 and byte size; byte 10 selects base register R2. This enables AUDIO_CTRL.
\$1018-\$1048	Six instructions	Point R2 at \$F0900, then write square frequency \$00010600 ($262 * 256$), volume \$BE, and control \$02.
\$1050-\$1080	Six instructions	Point R2 at \$F0940, then write triangle frequency \$00014A00 ($330 * 256$), volume \$96, and control \$02.
\$1088-\$10B8	Six instructions	Point R2 at \$F0980, then write sine frequency \$00018800 ($392 * 256$), volume \$82, and control \$02.
\$10C0	40 06 00 00 00 00 00 00	BRA with displacement 0; the branch target is its own address.

The SoundChip frequency registers use 16.8 fixed-point hertz, so the program stores $\text{Hz} * 256$. The volume registers use the low byte only. Control value \$02 sets the gate bit and enables the voice because any non-zero control value marks the channel active.

Try changing one frequency word in the byte listing, then re-enter only that w line and run again. For example, change the bytes at \$1090 from 00 88 01 00 to 00 B8 01 00 and the sine voice moves from 392 Hz to 440 Hz.

25.11 VideoChip Mode 7 example

The audio example proves that IE64 can drive a sound engine directly. This second monitor-entered program proves the same point for graphics. It builds a tiny 2 by 2 RGBA texture at \$1800, then asks the VideoChip blitter's Mode 7 affine mapper to scale that texture into a 16 by 16 block at the start of VRAM.

Mode 7 uses 16.16 fixed-point texture coordinates. A step of \$4000 is one quarter of a texel per output pixel, so each source texel becomes a 4 pixel wide block. With a 2 by 2 texture, the visible result is a chunky red, green, blue, and white pattern in the top-left corner of the VideoChip display.

You may enter this program with A 1100 by typing the mnemonic column shown in the d listing. For a longer graphics example the full byte copy is still printed first, because it is the easiest form to compare exactly and to re-enter a damaged line.

Type this at IE Mon:

```

(ie64)> w 1100 01 17 00 00 00 18 00 00 01 0F 00 00 00 00 FF 00
(ie64)> w 1110 11 0C 10 00 00 00 00 00 01 0F 00 00 00 FF 00 00
(ie64)> w 1120 11 0C 10 00 04 00 00 00 01 0F 00 00 FF 00 00 00
(ie64)> w 1130 11 0C 10 00 08 00 00 00 01 0F 00 00 FF FF FF 00
(ie64)> w 1140 11 0C 10 00 0C 00 00 00 01 17 00 00 00 00 0F 00
(ie64)> w 1150 01 0F 00 00 05 00 00 00 11 0C 10 00 20 00 00 00
(ie64)> w 1160 01 0F 00 00 00 18 00 00 11 0C 10 00 24 00 00 00
(ie64)> w 1170 01 0F 00 00 00 10 00 00 11 0C 10 00 28 00 00 00
(ie64)> w 1180 01 0F 00 00 10 00 00 00 11 0C 10 00 2C 00 00 00
(ie64)> w 1190 11 0C 10 00 30 00 00 00 01 0F 00 00 08 00 00 00
(ie64)> w 11A0 11 0C 10 00 34 00 00 00 01 0F 00 00 00 0F 00 00
(ie64)> w 11B0 11 0C 10 00 38 00 00 00 11 04 10 00 58 00 00 00
(ie64)> w 11C0 11 04 10 00 5C 00 00 00 01 0F 00 00 00 40 00 00
(ie64)> w 11D0 11 0C 10 00 60 00 00 00 11 04 10 00 64 00 00 00
(ie64)> w 11E0 11 04 10 00 68 00 00 00 11 0C 10 00 6C 00 00 00
(ie64)> w 11F0 01 0F 00 00 01 00 00 00 11 0C 10 00 70 00 00 00
(ie64)> w 1200 11 0C 10 00 74 00 00 00 11 0C 10 00 1C 00 00 00
(ie64)> w 1210 40 06 00 00 00 00 00 00
(ie64)> d 1100 #35
00001100: 01 17 00 00 00 18 00 00  move.q r2, #$1800
00001108: 01 0F 00 00 00 00 FF 00  move.q r1, #FF0000
00001110: 11 0C 10 00 00 00 00 00  store.l r1, (r2)
00001118: 01 0F 00 00 00 FF 00 00  move.q r1, #FF00
00001120: 11 0C 10 00 04 00 00 00  store.l r1, 4(r2)
00001128: 01 0F 00 00 FF 00 00 00  move.q r1, #FF
00001130: 11 0C 10 00 08 00 00 00  store.l r1, 8(r2)
00001138: 01 0F 00 00 FF FF FF 00  move.q r1, #FFFFFF
00001140: 11 0C 10 00 0C 00 00 00  store.l r1, 12(r2)
00001148: 01 17 00 00 00 00 0F 00  move.q r2, #F0000
00001150: 01 0F 00 00 05 00 00 00  move.q r1, #5
00001158: 11 0C 10 00 20 00 00 00  store.l r1, 32(r2)
00001160: 01 0F 00 00 00 18 00 00  move.q r1, #$1800
00001168: 11 0C 10 00 24 00 00 00  store.l r1, 36(r2)
00001170: 01 0F 00 00 00 00 10 00  move.q r1, #100000
00001178: 11 0C 10 00 28 00 00 00  store.l r1, 40(r2)
00001180: 01 0F 00 00 10 00 00 00  move.q r1, #10
00001188: 11 0C 10 00 2C 00 00 00  store.l r1, 44(r2)
00001190: 11 0C 10 00 30 00 00 00  store.l r1, 48(r2)
00001198: 01 0F 00 00 08 00 00 00  move.q r1, #8
000011A0: 11 0C 10 00 34 00 00 00  store.l r1, 52(r2)
000011A8: 01 0F 00 00 00 0F 00 00  move.q r1, #F00
000011B0: 11 0C 10 00 38 00 00 00  store.l r1, 56(r2)
000011B8: 11 04 10 00 58 00 00 00  store.l r0, 88(r2)
000011C0: 11 04 10 00 5C 00 00 00  store.l r0, 92(r2)
000011C8: 01 0F 00 00 00 40 00 00  move.q r1, #4000
000011D0: 11 0C 10 00 60 00 00 00  store.l r1, 96(r2)
000011D8: 11 04 10 00 64 00 00 00  store.l r0, 100(r2)
000011E0: 11 04 10 00 68 00 00 00  store.l r0, 104(r2)
000011E8: 11 0C 10 00 6C 00 00 00  store.l r1, 108(r2)
000011F0: 01 0F 00 00 01 00 00 00  move.q r1, #1
000011F8: 11 0C 10 00 70 00 00 00  store.l r1, 112(r2)
00001200: 11 0C 10 00 74 00 00 00  store.l r1, 116(r2)
00001208: 11 0C 10 00 1C 00 00 00  store.l r1, 28(r2)
T 00001210: 40 06 00 00 00 00 00 00  bra $001210
(ie64)> r pc 1100
(ie64)> b 1210
(ie64)> g

```

```
(ie64)> m 1800 1
0000000000001800: 00 00 FF 00 00 FF 00 00  FF 00 00 00 FF FF FF 00  .....
(ie64)> m 100000 1
00000000000100000: 00 00 FF 00 00 00 FF 00  00 00 FF 00 00 00 FF 00  .....
(ie64)> m 103C00 1
00000000000103C00: FF 00 00 00 FF 00 00 00  FF 00 00 00 FF 00 00 00  .....
(ie64)> bc 1210
```

The byte stream has three parts:

Address range	Purpose
\$1100-\$1140	Build the four-textel source image at \$1800. The words are stored little-endian, so colour \$00FF0000 appears in memory as 00 00 FF 00.
\$1148-\$11B0	Point R2 at VIDEO_REG_BASE = \$F0000, then stage the blitter operation, source address, destination address, width, height, source stride, and destination stride.
\$11B8-\$1208	Stage the Mode 7 origin, per-column step, per-row step, texture masks, and finally write 1 to BLT_CTRL to start the blitter.

The important Mode 7 registers are the six fixed-point values:

Register	Value	Meaning
BLT_MODE7_U0, BLT_MODE7_V0	0, 0	Start sampling from the top-left texel.
BLT_MODE7_DU_COL, BLT_MODE7_DV_COL	\$4000, 0	Move one quarter texel to the right for each output pixel.
BLT_MODE7_DU_ROW, BLT_MODE7_DV_ROW	0, \$4000	Move one quarter texel down for each output row.
BLT_MODE7_TEX_W, BLT_MODE7_TEX_H	1, 1	Wrap with a 2 by 2 texture mask.

The two VRAM dumps prove the scale-up. \$100000 is the first row of VRAM: four red pixels followed by four green pixels. \$103C00 is row 4 in the default 960 pixel mode: four blue pixels followed by four white pixels. On screen this appears as a small chunky checker block in the top-left corner.

Try changing both \$4000 step values at \$11C8 to \$8000. That makes each texel cover only two output pixels, so the pattern repeats more quickly across the same 16 by 16 area.

25.12 What comes next

Chapter 26 covers IE32, the 32-bit predecessor of IE64. IE32 has fewer registers, smaller addresses, and a simpler MMU model, but otherwise looks much the same to a programmer who has already met IE64.