

OCTERA THROWS A JAVALON

Java-like Synthesizable Processor Targets Deeply Embedded Systems By Tom R. Halfhill {3/17/03-02}

Only a few years ago, engineers despaired of using Java for real-time embedded applications because of its obesity and nondeterministic behavior. Java seemed more suitable for desktop PCs and servers that could satisfy its appetite for megabytes and megahertz. Yet,

from the very beginning, Sun had conceived Java as a programming language and virtual platform for an embedded application—interactive TV set-top boxes. Perhaps it is inevitable that Java is returning to its roots.

Today, Java is showing up in all kinds of embedded systems, from PDAs to next-generation mobile phones. Of course, it's also popular on servers. In fact, Java is successful just about everywhere except the platform that Sun wanted to conquer most: the desktop PC. The irresistible force of Sun is still having trouble overcoming the immovable object in Redmond—but that's another story.

Now some companies are pushing Java into a very different frontier: deeply embedded applications that demand hard real-time performance. Examples might include motor controllers, industrial machinery, smartcards, automotive telematics, and other mundane systems that aren't as sexy as Web-browsing, game-playing cell phones but nevertheless keep the world turning. These applications require high reliability, frugal memory consumption, low cost, low power, and fast response to real-time interrupts—qualities not usually associated with Java.

One of the first pioneers to explore this territory was aJile Systems, which makes a standard-part Java microcontroller. (See *MPR 8/7/00-02*, "Embedded Java Chips Get Real.") Another contender is the Imsys Cjip, a rewritablemicrocode chip that has instruction sets for Java, C, and Forth. (See *MPR 8/14/00-04*, "Imsys Hedges Bets On Java.") Now, San Diego–based Octera is introducing Javalon-1, a synthesizable microprocessor core that natively executes Java bytecode instructions. Javalon-1 is the first member of a small family of cores that will have minor variations on the same basic design. Chip designers can use Javalon-1 as the basis for a self-sufficient microcontroller or as a slave to another microprocessor core on an SoC or ASIC.

Javalon-1's memory requirements are unusually low (about 5K of firmware, not counting application memory). It doesn't need a Java virtual machine (JVM), a Java bytecode interpreter, or a just-in-time (JIT) bytecode compiler, because it natively executes Java bytecode instructions in hardware or software. It doesn't use a garbage collector to manage memory, so it's deterministic and has relatively low interrupt latencies (about 100 clock cycles, worst-case). And it's suitable for small embedded systems because it's compact (about 25,000 logic gates). Octera is wrapping up the design work now and plans to begin licensing Javalon-1 in 3Q03.

What's the catch? Javalon-1 isn't officially a Java processor, because it doesn't fully comply with Sun's Java specifications. In general, it follows the Java 2 Micro Edition (J2ME) and Connected Limited Device Configuration (CLDC) guidelines, but it makes several compromises to achieve its fast interrupt response, efficient memory usage, and low gate count. Javalon-1 is more accurately described as a Java-like processor or as a processor that executes Java bytecodes without fully supporting a Sun-standard Java platform. Some of Octera's trade-offs are sure to be controversial.

aload Load feat from array fload cms Lead float from local var Izd Convert long to double acont, null Push null freg Negate float Izi Convert long to float aload c. Load ref from local var frem Remainder float Iadd Add long alead, cn. Load ref from local var frem Return float Iadd Bolan AND long areturn Return float fotor Store float ho local var Iadd Bolan AND long astore Store ref into local var getistid Get staft field from class Icont < Push long astore Store ref into local var getistid Get staft field from class Icont < Push long blaad Load byte/sboolean rary Izi Convert into to har Icid Push long onth always Icid onth always Icid Convert into to har Icid cont pool Icid Icid Convert into ho	Instruction	Description	Instruction	Description	Instruction	Description
astore Store into ref array, acorst, null fmull Multiply float 12f Convert long to float aload aload enc. Load ref from local var frem Return float from method aload var ladd Add long, Add long, from array, anewarray arraylength Create new array of ref Store Store float holo cal var arraylength ladd Boolean AND long, arraylength astore Store ref Store sens Store float holo cal var array effetid ladd Boolean AND long, astore astore Store ref into local var astore getidel Get field from dasy gets leenst Push long, compare long, letter baload Load byte/boolean from array, goto geto Branch always (wde index) loca ldc Push long, constrait to byte baload Load dor from array, castore Store into byte/boolean array, load l2b Convert into baload ldc2.cwo- push long robole from run-time const pool. baload Load dorp from array, ladd l2f Convert into baload ldad Load long from local var load castore Store into ind able to into aload load dorb from array, ladd ldad Add into	aaload	Load ref from array	fload_ <n></n>	Load float from local var	l2d	Convert long to double
aconst. null Push null freg Negate float 12i Convert long to init aload Load ref from local var frem Remander float ladd Add long. aload Create new array of ref fistore Store float into local var ladd Boolean AND long. areturm Return for method fistore Store float into local var lastore Store float into local var lastore Store float into local var getistatic Convert into long array. Convert into long array. into do	aastore	Store into ref array	fmul	Multiply float	l2f	Convert long to float
aload Load ref from local var anewaray frem (Conserved output) frem (Ferum (Ferum anewaray) Reum for from method (Store enc) ladd (Store (Inc) (Store) Ladd (Store) Add long (Store) arrayength Create new array of ref (Store enc) fstore enc) Store float into local var (Store) land Boolean AND long arrayength Get med from object Ieron Compare long Compare long astore Store ref into local var (Store) getfield Get field from object Ieron Push horn baload Load byte/boolean raray gbto Branch always (wde index) Ide Push horn runtime baload Load byte/boolean raray 12 Convert into double Ide Push horn outfure const pool castore Store into str/bote/boena raray 12 Convert into long Ide Push horn outfure const pool daid Load double to finat laid Add inf Iload Load long from local var daid Convert double to finat laid Add inf Iload Load long from local var daid Convert double to long	aconst_null	Push null	fneg	Negate float	l2i	Convert long to int
aload -m> Load ref from local var freturn Return float from method land Load long from aray areverary Create new aray of ref force store float into local var lastore construction of aray getter from method for aray getter fload fload local var lastore construction of a store into local var getter construction of the store into local var petited Get tield from dass astore. Store ref into local var getter fload Get static field from dass attrace	aload	Load ref from local var	frem	Remainder float	ladd	Add long
anewaray Create new aray of ref fstore Store float into local var Iand Boolean AND long arraylength Get length of aray fsub Store float into local var Immo blocal ConstI> Push long Push long ConstI> Push long	aload_ <n></n>	Load ref from local var	freturn	Return float from method	laload	Load long from array
areture Return ref from method fstore Store into local var fstore Store into local var fstore Store ref into local var getfield Get kingt from object Compare long Compare long astore Store ref into local var getfield Get kingt from object Iconst> Push from runtime baload Load byte/boolean array geto Branch always (wide index) Ide. <w> Push from runtime const baload Load byte/boolean from array gzb Convert into bodue Ide. <w> Push long or double from runtime const castore Store into byte/boolean array Izb Convert into double Ide. <w> Push long or double from runtime const castore Store into double to font Iadd Convert into bodue Idad Idad</w></w></w>	anewarray	Create new array of ref	fstore	Store float into local var	land	Boolean AND long
array/enth Get length of array fsub Subtract float Imp Compare long. astore Store refinto local var getfeld Get field from object Ioons L<> Push long baload Load byte/boolean from array. goto Branch always (wide index). Ide Oonst L<> Push from number const Load baload Load byte/boolean array. goto Branch always (wide index). Ide. Push from number const Load baload Load brie/boolean array. goto Convert int to double. Push from number const Load Push from load Push from load Push from number const Load Push from load	areturn	Return ref from method	fstore_ <n></n>	Store float into local var	lastore	Store into long array
astore Store ref into local var store scho gettatic Get field from object Ionst -cb Push from nutime const pool astore Throw exception or error paload Load byte/boolean from array its Ide Push from nutime const pool, wide index bipush Push byte Ide Convert int to byte Push from nutime const pool, wide index bipush Push byte Ide Convert int to double caload Ide Push from nutime const pool, wide index caload Load byte/boolean from array Id Convert int to double caload Ide Push from nutime const pool, wide index caload Convert double to int Iadd Convert int to fond Ide Push from nutime const pool, wide index caload Convert double to int Iadd Load int from array Ilde Cod long from local var dalad Load double from array Iso Integ Negate long dalaad Load double from array Id Integ Negate long dalaad Load double from neat var If acmp(cond) Branch fir dc ompare true Integ Negate long	arraylength	Get length of array	fsub	Subtract float	lcmp	Compare long
astore Store ref into local var athrow gets tatic got Get static field from daws baload Idc Push from runtime const pool baload Load byte/boolean from array bastore Store into byte/boolean array breakpoint* Izc Convert into borla Idc Push from runtime const pool, wide index bipush Push byte Izc Convert int to double Idc2_ <rw> Push from runtime const pool, wide index caload Load char from array Izd Convert int to double Idc2_<rw> Push long or double for run-three const pool, wide index castore Store into char array Izd Convert int to fond tailadd Load long from local var dad Add double to int aladad Load double for iastore Store into double array Idv Divide long from method from nethod dadad Load double from array idw Divide int Ior Boolean AND int Ierum Reaminder long from method dadad Load double from array idw Divide int Ior Boolean AND int Ierum Boolean AND int Ierum Boolean AND int Ierum Boolean AND int Ierum for Acras y</rw></rw>	astore	Store ref into local var	getfield	Get field from object	lconst_ <l></l>	Push long
athrow Throw exception or error goto Branch always const pool balada Load byte/boolean from array 12b Convert int to byte Use Push from runner const pool, wide index brashore Store into byte/boolean array 12b Convert int to byte Use Push from runner const pool, wide index caload Load char from array 12f Convert int to short Use Valid index castore Store into char array 12f Convert into short Used index Valid index dalad Convert double to int Iadd Add int Iload Load long from local var dalad Convert double to int Iadd Load int from array Imul Multiply long dalad Load double from array Idiv Divide int Ineg Negate long dalad Load double from array Idiv Divide int Ineg Negate long dalad Load double from array Idiv Divide int Ineg Negate long dalad Load double from array If amp(cond)	astore_ <n></n>	Store ref into local var	getstatic	Get static field from class	ldc	Push from runtime
baload Lead byte/boolean from array bastore Event byte Store into byte/boolean array izb Event byte Convert int to byte Convert int to char Ide Push from runtime const pool, wide index bipush Push byte ize Convert int to othar Ide2_ <wp>Push from runtime const pool, wide index caload Load char from array izd Convert int to fond (action in the array) iulu Nullby long dzi Convert double to int (action in the array) iaload Load inform form local var Imal Multby long dadd Add ouble to int (action in the array) iaload Load long from local var Imal Multby long datad Load double from array idow Divide int Imal Multby long datad Load double from array idw Divide int Imal Multby long datad Load double from array idw Divide int Imal Imal Nultby long datad Load double from array idw</wp>	athrow	Throw exception or error	goto	Branch always		const pool
bastore Store into byte/boolean array 12b Convert int to byte peakpoint peakpoint Push long or double form breakpoint Breakpoint <th>baload</th> <th>Load byte/boolean from array</th> <th>goto_w</th> <th>Branch always (wide index)</th> <th>ldc_<w></w></th> <th>Push from runtime const</th>	baload	Load byte/boolean from array	goto_w	Branch always (wide index)	ldc_ <w></w>	Push from runtime const
bipush Push byte 12c Convert int to data (add ld2_cxw> Push long or double from nu-time const pool, wide index. castore Store into char array 12f Convert int to fond muthine const pool, wide index. d2f Convert int to short Ilaid Divide long Divide long d2f Convert double to float iadd Add int Ilaad Load long from local var d21 Convert double to long, iand Boolean AND int Imeg Negate long, dadd Add double from array ioonst idiv Divide int Inu Multiply long, dadd Load double form array idiv Divide int Ior Boolean OR long, dampcop Compare double if acmg(cond) Branch if ref compare true Irem Remainder long, double if inconcil Branch if ref cont anu Ish Ashtifiet long, Store long into local var double if cond Branch if ref cont anu Ish Ashtifiet long, Store long into local var dodad Load double from local var	bastore	Store into byte/boolean array	i2b	Convert int to byte		pool, wide index
breakpoint*	bipush	Push byte	i2c	Convert int to char	ldc2_ <w></w>	Push long or double from
caload Load char from array 12f Convert int to float wide index castore Store into char array 12 Convert int to short Iload Load long from local var d2f Convert double to float iadd Add int Iload Load long from local var d2i Convert double to long iand Boolean AND int Ileag Nutliply long dadd Add double iastore Store into int array lookupswitch Access jump table dastore Store into double array idiv Divide long Branch if ref compare true Irem Remainder long dadad Load double from nocal var iff comp(cond) Branch if ref compare true Iretum Return long from method dadad Load double from local var iff comp(cond) Branch if ref not null Ishr Arithmetic shift right long dioad, col Load double from local var ifmull Branch if ref not null Ishr Store into local var diad Load double from local var implep1* Implementation-specific istore Store long into local var diad Load double from method iindex implep2* Implementation-specific molitorewit Exit monitor for object ding Ngate	breakpoint*	Breakpoint for debugger	i2d	Convert int to double		run-time const pool,
castoreStore into char array[2]Convert into longIdivDivide longcheckcastCheck object type[2sConvert into shortIloadLoad long from local vard21Convert double to intialadLoad Int from arrayIload_Load long from local vardadadConvert double from arrayialadBoolean AND IntInegNegate longdadadLoad double from arrayiastoreStore into double marrayIdivDivide intAccess jump tabledatastoreStore into double arrayidivDivide intIrom Return long from methodBranch if ref compare trueIremReturn long from methodddivDivide doubleIf comfoullBranch if ref not nullIstoreStore long into local varIffulldloadLoad double from local varIfforullBranch if ref-nullIstore _store long into local varIstore _store long into local vardnadMultiply doubleincIncrement local var load int from local varIstore _store long into local varIstore _store long into local vardnagNegate longNegate longIsod int from local varIstore _store long into local varIstore _store long into local vardnagNegate longImpelpe1*Implementation-specificIstore _store long into local varIstore _store long into local vardnagNegate longNegate longNegate longNegate longNegate longIstore _store long into local vardnagNegate long local varimpdep1*Implementation-	caload	Load char from array	i2f	Convert int to float		wide index
Check object type I2s Convert int to short Iload Load long from local var d21 Convert double to int iaload Load int from array Imul Multiply long d21 Convert double to long iand Boolean AND int Imeg Negate long dadad Add double lastore Store into int array lookupswitch Access jump table dastore Store into double array idiv Divide double Boolean OR Intervention Access jump table dconst_do- Push int Ior Boolean OR Iong Multiply double Boolean OR Intervention Access jump table dconst_do- Push double if comp(cond) Branch if ref compare true Ireturn Return long from method dload Load double from local var ifmonull Branch if ref-not null Istore Store long into local var diadad Load double from local var ifmonull Branch if ref-null Istore Store long into local var diadad Load double from local var ifmonull Branch if ref-null Istore Store long into local var dmail Multiply double inf conto	castore	Store into char array	i2l	Convert int to long	ldiv	Divide long
d21 Convert double to float iadd Add int Iload_ <n> Load long from local var d21 Convert double to long iand Boolean AND int Ineg Negate long dadd Add double iastore Store into int array lookupswitch Acget and a long daload Load double from array iconst.cl> Push int lookupswitch Acget and a long damp const.cd> Push double if acmp(cond) Branch if ref nor null Istr Remainder long dioad Load double from local var Ifmonnull Branch if ref nor null Istr Anthretic shift right long dioad Load double from local var Ifmonnull Branch if ref nor null Istr Anthretic shift right long diad Multiply double iinc Increment local var Istr Store long into local var dmai Multiply double iinc Increment local var Istr Store long into local var dmai Return double from method impdep1* Implementation-specific Istr Arithmetic shift right long <t< th=""><th>checkcast</th><th>Check object type</th><th>i2s</th><th>Convert int to short</th><th>lload</th><th>Load long from local var</th></t<></n>	checkcast	Check object type	i2s	Convert int to short	lload	Load long from local var
d21 Convert double to int dal iaload Load int from array Boolean AND int Inul Multiply long dad Convert double to long iastore Store into int array dadad lookupswitch Negate long dadad Load double from maray dcmp-cop. Convert double array if urmp(cond) Branch if ref compare true Branch if ref compare true Irem Irem Return long from method solean ORI ddu/ Divide double from local var if nonull Branch if ref ron null Branch if ref ron null Istr Arithmetic shift right long Istore Store local var indical var indical var if nonull Branch if ref ron null Istore Store local var istore Store local var istore long into local var indical var indical var indead <n> Istore Store long into local var istore long into local var istore local var istore double into local var imdep1* Implementation-specific trap Istore Store long into local var indical var istore double into local var imdep2* Istore monitorenter Enter monitor for object. dup Duplicate top stack operand, insert 2 or 3 values down Imul Multiply into insert 2 or 3 values down Invoke instance method invoke instance method pop Pop top stack operand invoke instance method pop Pop top 1 or 2 stack values, insert 2 or 3 values down Invoke</n>	d2f	Convert double to float	iadd	Add int	lload_ <n></n>	Load long from local var
dad Convert double to long iand Boolean AND int Ineg Negate long daload Load double from array iconst_ <i>Store into int array lookupswitch Access jump table darmscops Compare double array idiv Divide int lor Boolean OR long dcmp-cops Compare double if acmp(cond) Branch if ref compare true Irem Return long from method dload Load double from local var ifnonnull Branch if ref not null Ishr Arithmetic shift right long dload Load double from local var ifnonnull Branch if ref not null Isbr Arithmetic shift right long dload Load double from local var ifnull Branch if ref not null Isbr Store long into local var dmem Remainder double iinc Increment local var by const Isbr Store long into local var drefurn Remainder double iindd <-n> Load int from local var Isbr Arithmetic shift right long drefurn Remainder local var impdep1* Implementation-specific Isor Roolean X</i>	d2i	Convert double to int	iaload	Load int from array	Imul	Multiply long
dadad Add double iastore Store into int array lookupswitch Access jump table dabad Load double from array idiv Divide int Ior Boolean OR long dcmp-cop- Compare double if acmp(cond) Branch if ref compare true Irem Remainder long dconst_cd> Push double if icmp(cond) Branch if ref compare true Irem Remainder long dload Load double from local var ifnull Branch if ref not null Ishr Arithmetic shift right long dload_n Load double from local var ifnull Branch if ref=null Istre Store long into local var dmail Multiply double iincrement local var y const Istore <n> Store long into local var drem Remainder double iload <n> Load int from local var Isub Subre double into local var dretum Return double from method impdep1* Implementation-specific monitorenter monitorenter monitor or object dstore Store double into local var impdep2* Implementation-specific monitorent</n></n>	d2l	Convert double to long	iand	Boolean AND int	Ineg	Negate long
dalaad Load double from array iconst_ <i>> Push int by key match and jump dastore Store into double array idiv Divide int lor Boolean OR long dconst_ Push double if cemp(cond) Branch if int compare true Irem Remainder long dconst_ A Push double if cond) Branch if int compare true Irem Remainder long dload Load double from local var iffnonull Branch if ref not null Ishr Arithmetic shift right long dload Load double from local var iffnull Branch if ref not null Istore Store long into local var dmaid Multiply double iinc Increment local var Isub Store long into local var dneg Negate double iload Load int from local var Isub Subtract long drem Remainder double iload_ Load int from local var Isub Subtract long dstore Store double into local var impdep1* Implementation-specific monitorexit Exit monitor object dup_x1 Duplicate top stack operand, inset 2 or 3 values down inset 2 or 3 values down invoke instance method</i>	dadd	Add double	iastore	Store into int array	lookupswitch	Access jump table
dastore Store into double array idiv Divide int Ior Boolean OR long. dcmp <op> Compare double if armp(cond) Branch if int compare true Irem Remainder long dconst_ Push double if icon(0) Branch if int compare true Iretum Return long from method dload Load double from local var ifnonnull Branch if int=O Ish Arithmetic shift right long. dload_ Load double from local var ifnonnull Branch if ref=null Istore Store long into local var dmeg Negate double iload Load int from local var Isub Store long into local var dretur Return double into local var impdep1* Implementation-specific monitorenter Enter monitor for object dstore Store double into local var impdep2* Implementation-specific multidm array multidm array dup_x1 Duplicate top stack operand, insert 2 values down insuceof Compare double into local var insuceof nop No operation invoke instance method inset 2 or 3 values down <t< th=""><th>daload</th><th>Load double from array</th><th>iconst_<i></i></th><th>Push int</th><th></th><th>by key match and jump</th></t<></op>	daload	Load double from array	iconst_ <i></i>	Push int		by key match and jump
dcmp.sop> Compare double if. acmp(cond) Branch if ref compare true Irem Remainder long dconst <d>> Push double if.comp(cond) Branch if int compare true Ish1 Nift left long dload Load double from local var ifmonnull Branch if ref-null Ish1 Shift left long dload, cn> Load double from local var ifmull Branch if ref-null Istore Store long into local var dmeg Negate double iload Load inf from local var Istore <n> Store long into local var drem Remainder double impdep1* Load int from local var Istore Store long into local var dstore Store double into local var impdep2* Implementation-specific Ixor Boolean XOR long dup_x1 Duplicate top stack operand, insert 2 values down imstanceof Compare object type multianewarray Create new array dup_x2 Duplicate top 1 or 2 stack values invoke interface method pop Pop top stack operand, invoke interface method pop Pop top 1 or 2 dup_x1 Duplicate top 1 or 2 s</n></d>	dastore	Store into double array	idiv	Divide int	lor	Boolean OR long
dconst. ddiv Push double if, icmp(cond) Branch if int compare true Interum Return	dcmp <op></op>	Compare double	if_acmp(cond)	Branch if ref compare true	Irem	Remainder long
ddiv Divide double if(cond) Branch if ref not null Ishl Shift left long dload Load double from local var ifnonull Branch if ref not null Ishr Arithmetic shift right long dmul Multiply double iinc Increment local var by const Istore Store long into local var dmeg Negate double iload Load int from local var Isub Subtract long dretur Return double from method impdep1* Implementation-specific Isubr Arithmetic shift right long dstore Store double into local var impdep2* Implementation-specific monitorexit Exit monitor for object dup Duplicate top stack operand, inset 2 values down imul Multiply int multianewarray Create new object dup_x1 Duplicate top 1 or 2 stack values, invokeinterface Invoke instance method pop pop top top 2 stack operand, invokeinterface Invoke instance method pop Pop top top 1 or 2 stack operand, invokeinterface Invoke instance method pop Pop top top 2 stack operand, invokeistatic Invoke instance method pop Pop top 1 or 2 stack values, invokeistatic Invoke instance method	dconst_ <d></d>	Push double	if_icmp(cond)	Branch if int compare true	Ireturn	Return long from method
dload Load double from local var ifmonnull Branch if refnot not null Ishr Arithmetic shift right long. dload <n> Load double from local var ifmull Branch if ref=null Istore Store long into local var dmul Multiply double iinc Increment local var y const Istore <n> Store long into local var drem Remainder double iioad Load int from local var Isub Store long into local var dreturn Return double from method impdep1* Implementation-specific Isor Moritenetr Enter monitorenter Enter monitor for object dstore subtract double impdep2* Implementation-specific monitorenter Enter monitor for object dup A Duplicate top stack operand, instanceof instanceof Compare object type newaray Create new object dup_x2 Duplicate top 1 or 2 stack values, instanceof Invoke instance method pop Pop top stack operand, witokespecial Invoke instance method pop Pop top stack operand, instance of 1 or Boolean OR int putietatic Set field in object dup_x2 Duplicate top 1 or 2 stack values, instance of 1 or<!--</th--><th>ddiv</th><th>Divide double</th><th>if(cond)</th><th>Branch if int=0</th><th>lshl</th><th>Shift left long</th></n></n>	ddiv	Divide double	if(cond)	Branch if int=0	lshl	Shift left long
dload <cr> Load double from local var dmul innull Branch if ref=null Istore Store long into local var store long into local var dmul Multiply double iinc Increment local var by const iload Load int from local var Istore Store long into local var drem Remainder double iload Load int from local var Istore Store long into local var dreturm Return double from method impdep1* Implementation-specific Iushr Arithmetic shift right long dstore Store double into local var impdep2* Implementation-specific Iushr Arithmetic shift right long dup_x1 Duplicate top stack operand, insert 2 values down img Negate int new Create new array dup2 Duplicate top stack operand, insert 2 or 3 values down invoke instance of Compare object type new array Create new array dup2 Duplicate top 1 or 2 stack values, insert 2 or 3 values down invoke instance method nop No operation dup2_x2 Duplicate top 1 or 2 stack values, insert 2, 3, or 4 values down invoke instance method putfield Set static field in class dup2_x2 Duplicate top 1 or 2 stack values, ins</cr>	dload	Load double from local var	ifnonnull	Branch if ref not null	lshr	Arithmetic shift right long
dmulMultiply doubleiincIncrement local var by constIstore_ <n>Store long into local vardnegNegate doubleiloadLoad int from local varIsubrSubtract longdreturnReturn double from methodimpdep1*Implementation-specificIsubrArithmetic shift right longdstore_Store double into local varimpdep1*Implementation-specificIxorBoolean XOR longdstore_Store double into local varimpdep2*Implementation-specificIxorBoolean XOR longdup_Duplicate top stack operand, insert 2 values downimulMultiply intmultianewarrayCreate new objectdup_x2Duplicate top stack operand, insert 2 or 3 values downinvokeinterface invokeinterfaceInvoke interface methodnopNo operationdup2_x1Duplicate top 1 or 2 stack values, insert 2 or 3 values downinvokeinterface invokeistaticInvoke instance methodpop2Pop top 1 or 2dup2_x2Duplicate top 1 or 2 stack values, insert 2, 3, or 4 values downishrArithmetic shift right int ishlShift right intSaloadLoad int from subroutinef2dConvert float to longistore <n>Store into ilocal varStore into short arrayf21Convert float to longistore <n>Store into ilocal varsatoref21Convert float to longistore <n>Store into ilocal varsatoref21Convert float to longistore <n>Store int into local varsatoreStore into short a</n></n></n></n></n>	dload_ <n></n>	Load double from local var	ifnull	Branch if ret=null	Istore	Store long into local var
dneg Negate double Iload Load int from local var Isub Subtract long drem Remainder double iload_ <n> Load int from local var lushr Arithmetic shift right long dreturn Return double from method impdep1* Implementation-specific lsor Boolean XOR long dstore Store double into local var impdep2* Implementation-specific monitorexit Exit monitor for object dup Duplicate top stack operand, insert 2 values down imul Multiply int multianewarray Create new object dup_x1 Duplicate top stack operand, insert 2 values down invoke interface Invoke interface method nop No operation dup2 Duplicate top 1 or 2 stack values invokestatic Invoke instance method pop Pop top 1 or 2 dup2_x1 Duplicate top 1 or 2 stack values, insert 2 or 3 values down invoke instance method pop Pop top 1 or 2 dup2_x1 Duplicate top 1 or 2 stack values, insert 2 or 3 values down invoke instance method putstatic Set stack operands dup2_x2 Duplicate top 1 or 2 stack values, insert 2 or</n>	dmul	Multiply double	linc	Increment local var by const	_lstore_ <n></n>	Store long into local var
drem Remainder double lioad_<>> Load int from local var lushr Anthmetic shift right long. dretum Return double into local var impdep1* Implementation-specific Ixor Boolean XOR long dstore Store double into local var impdep2* Implementation-specific monitorenter Enter monitor for object dub Subtract double impdep2* Implementation-specific multianewarray Create new dup.x1 Duplicate top stack operand, insert 2 values down instanceof Compare object type new Create new object dup.x2 Duplicate top stack operand, insert 2 or 3 values down invokeinterface Invoke instance method nop No operation dup2 x1 Duplicate top 1 or invokestatic Invoke instance method pop Pop top stack operand dup2 x2 Duplicate top 1 or 2 stack values, invokestatic Invoke static method putfield Set static field in object dup2 x2 Duplicate top 1 or 2 stack values, insert 2 or 3 values down invokestatic Invoke static method putstatic Set static field in class dup2 x2	dneg	Negate double	iload	Load int from local var	lsub	Subtract long
dreturn Return double from method impdep1* Implementation-specific Ixor Boolean XOR long dstore Store double into local var impdep1* Implementation-specific monitorenter Enter monitor for object dsub Subtract double impdep2* Implementation-specific monitorenter Enter monitor for object dup_x1 Duplicate top stack operand, insert 2 values down img Negate int new Create new object dup_x2 Duplicate top stack operand, insert 2 values down invokeinterface Invoke interface method nop No operation dup2 Duplicate top 1 or 2 invokestatic Invoke instance method pop Pop top stack operand dup2_x1 Duplicate top 1 or 2 stack values, insert 2 or 3 values down invokestatic Invoke instance method putfield Set field in object dup2_x2 Duplicate top 1 or 2 stack values, insert 2, 3, or 4 values down invokestatic Invoke instance method putfield Set field in class dup2_x2 Duplicate top 1 or 2 irem Return int from method return Return int from method return	drem	Remainder double	iload_ <n></n>	Load int from local var	lushr	Arithmetic shift right long
dstore dstore <n> Store double into local var dsub Trap Trap monitorenter Enter monitor for object dsub Subtract double impdep2* Implementation-specific trap monitorenter Exit monitor for object dup Duplicate top stack operand, insert 2 values down imul Multiply int multianewarray Create new object dup_x2 Duplicate top stack operand, insert 2 or 3 values down instanceof Compare object type new Create new object dup2_x1 Duplicate top 1 or 2 stack values invokeinterface Invoke instance method pop No operation dup2_x1 Duplicate top 1 or 2 stack values, insert 2 or 3 values down invokestatic Invoke instance method pop Pop top 1 or 2 2 stack values, insert 2 or 3 values down invokevirtual Invoke instance method putfield Set field in object dup2_x2 Duplicate top 1 or 2 stack values, insert 2, 3, or 4 values down irem Remainder int ret Return from subroutine dup2_x2 Duplicate top 1 or 2 stack values, ishr Arithmetic shift right int saload Load short from array f2d Convert float to double ishr Arithmetic shift right int saload Load short from array f21 Convert float to long is</n>	dreturn	Return double from method	impdep1*	Implementation-specific	Ixor	Boolean XOR long
dsub Store double into local Var impdep2* implementation-specific trap monitorexit Exit monitor for object dup Duplicate top stack operand, insert 2 values down imul Multiply int multianewarray Create new object dup_x2 Duplicate top stack operand, insert 2 values down instanceof Compare object type new array Create new array dup_x2 Duplicate top 1 or 2 stack values invokeinterface Invoke instance method nop No operation dup2_x1 Duplicate top 1 or 2 stack values invokestatic Invoke instance method pop2 Pop top stack operand, with special handling pop2 Pop top 1 or 2 Stack operands dup2_x1 Duplicate top 1 or 2 stack values, insert 2 or 3 values down invokestatic Invoke instance method putfield Set field in object dup2_x2 Duplicate top 1 or 2 insert 2 or 3 values down ireturn Return int from method stack values, ireturn Return int from method issert 2, 3, or 4 values down ish Shift left int saload Load short from array f2d Convert float to double	dstore	Store double into local var		trap	monitorenter	Enter monitor for object
dup dupDuplicate top stack operand, insert 2 values downimulMultiply intmultidinewarrayCreate new multidim arraydup_x1Duplicate top stack operand, insert 2 values downinegNegate intnewCreate new objectdup_x2Duplicate top stack operand, insert 2 or 3 values downinstanceofCompare object typenewarrayCreate new arraydup_x1Duplicate top stack operand, insert 2 or 3 values downinvokeinterfaceInvoke interface methodnopNo operationdup2Duplicate top 1 or 2 stack valuesinvokestaticInvoke instance methodpopPop top stack operand, stack operanddup2_x1Duplicate top 1 or 2 stack values, insert 2 or 3 values downinvokestaticInvoke instance methodputfieldSet field in objectdup2_x2Duplicate top 1 or 2 stack values, insert 2, 3, or 4 values downinvokevirtualInvoke instance methodputstaticSet static field in classdup2_x2Duplicate top 1 or 2 stack values, insert 2, 3, or 4 values downishReturn int from methodreturnReturn void from arrayf2dConvert float to doubleishrArithmetic shift right intsaloadLoad short from arrayf2iConvert float to longistore <n>Store int into local varswapSwap top two stackfaddAdd floatisubSubtract intoperandsoperandsfaloadLoad float from array isubisorStore int into local varswapSwap top two stackfal</n>	dstore_ <n></n>	Store double into local var	impdep2*	Implementation-specific	monitorexit	Exit monitor for object
dupDuplicate top stack operand, insert 2 values downimuMultiply intInduction analydup_x1Duplicate top stack operand, insert 2 values downinegNegate intnewCreate new objectdup_x2Duplicate top stack operand, insert 2 or 3 values downinstanceofCompare object typenewarrayCreate new objectdup2Duplicate top stack operand, insert 2 or 3 values downinvokeinterfaceInvoke instance method with special handlingnopNo operationdup2_x1Duplicate top 1 or 2 stack values, insert 2 or 3 values downinvokestaticInvoke instance method with special handlingpopPop top stack operanddup2_x2Duplicate top 1 or 2 stack values, insert 2, 3, or 4 values downinvokevirtualInvoke instance method ishlputfieldSet field in objectf2dConvert float to doubleishrArithmetic shift right int istore <n>saloadLoad short from arrayf2lConvert float to intistore <n>Store int into local var isubswapSwap top two stack operandsfaloadLoad floatisubSubtract intoperandsfaloadLoad float from arrayisorBoolean XOR intwideExtend local var index operandsfaloadLoad floatjsrJump to subroutinewideExtend local var index operandsfaloadLoad floatjsrJump to subroutinewideExtend local var index operandsfaloadLoad floatjsrJump to subroutine</n></n>	dsub	Subtract double	l		multianewarray	Create new
dup_x1Duplicate top stack operand, insert 2 values downinegNegate intnew arrayCreate new objectdup_x2Duplicate top stack operand, insert 2 or 3 values downinvokeinterface invokeinterfaceInvoke interface method with special handlingnopNo operationdup2Duplicate top 1 or 2 stack valuesinvokeinterface invokestaticInvoke instance method with special handlingpopPop top stack operanddup2_x1Duplicate top 1 or 2 stack values, insert 2 or 3 values downinvokestatic invokestaticInvoke instance method Boolean OR intputfieldSet field in objectdup2_x2Duplicate top 1 or 2 stack values, insert 2, 3, or 4 values downiremReturn int from method ishlShift left intSaloadLoad short from arrayf2dConvert float to doubleishrArithmetic shift right int ishlSaloadLoad short from arraySwap top two stack operandsf2iConvert float to longistore <n>Store int into local var isubSubtract intswapSwap top two stack operandsfaloadLoad float from arrayiwshrLogical shift right int faloadtableswitchAccess jump table by index and jumpfaloadLoad float from arrayixorBoolean XOR int itywideExtend local var index by index and jumpf2iConvert float to longistore <n>Store int into local var isubSubtract int Boolean XOR intAccess jump table by index and jumpfaloadLoad float from arrayixor</n></n>	dup v1	Duplicate top stack operand	inog	Multiply Int	2014	Create now object
dup_x2Duplicate top stack operand, insert 2 or 3 values downinvokeinterface invokeinterfaceInvoke interface method nvoke instance methodnopNo operationdup2Duplicate top 1 or 2 stack valuesinvokeinterface invokespecialInvoke instance method with special handlingpopPop top stack operand stack operanddup2_x1Duplicate top 1 or 2 stack values, insert 2 or 3 values downinvokeinterface invokestaticInvoke instance method invoke instance methodpop2Pop top 1 or 2 stack operandsdup2_x2Duplicate top 1 or 2 stack values, insert 2, 3, or 4 values downinvoke instance method ishlputfieldSet field in objectf2dConvert float to doubleishrArithmetic shift left int storesaloadLoad short from arrayf2iConvert float to long faddistore <n>Store int into local var isubswapSwap top two stack operandsfaloadLoad float faloadLoad float from array issriushrLogical shift left int subtract inttableswitchAccess jump table by index and jumpfcmp<op>Convert float tojsr_wJump to subroutine index)wideExtend local var index</op></n>	aup_x1	insert 2 values down	instanceof	Compare object type	new	Create new object
dup_x2Duplicate top stack operand, insert 2 or 3 values downinvoke interfaceinvoke interfacepopPop top stack operanddup2_x1Duplicate top 1 or 2 stack values, insert 2 or 3 values downinvoke instance methodinvoke instance methodputfieldSet field in objectdup2_x2Duplicate top 1 or 2 stack values, insert 2, 3, or 4 values downinvoke instance methodputfieldSet static field in classf2dConvert float to doubleishrArithmetic shift left intsaloadLoad short from arrayf2iConvert float to intistoreStore int into local varsipushPush shortf2iConvert float to longistore <n>Store int into local varswapSwap top two stackfaloadLoad float from arrayiushrLogical shift right inttableswitchAccess jump tablefaloadLoad float from arrayixorBoolean XOR intby index and jumpfatoreStore into float arrayixorBoolean XOR intby index and jumpfaloadLoad float from arrayixorBoolean XOR intby index and jumpform<t< th=""><th>dun v2</th><th>Duplicate top stack operand</th><th>involvointorfaco</th><th>Lougho interface method</th><th>non</th><th>No operation</th></t<></n>	dun v2	Duplicate top stack operand	involvointorfaco	Lougho interface method	non	No operation
dup2Duplicate top 1 or 2 stack valuesInvoke special with special handlingpopTop top stack operanddup2_x1Duplicate top 1 or 2 stack values, insert 2 or 3 values downinvokestatic invokestaticInvoke instance methodpop2Pop top 1 or 2 stack operandsdup2_x2Duplicate top 1 or 2 stack values, insert 2, 3, or 4 values downinvokestatic ireturnInvoke instance methodputfieldSet static field in objectf2dConvert float to double ishlishrReturn int from method ishlreturnReturn void from methodf2iConvert float to longistore <n>Store int into local var isubStore int into local varsubpath operandsf2lConvert float to longistore <n>Store int into local var isubSubtract intswapSwap top two stack operandsfaloadLoad float from array isoriswrLogical shift right int iswrbolean XOR int by index and jumptableswitchAccess jump table by index and jumpfatoreStore into float array iswrjsr_wJump to subroutine index)wideExtend local var index by bytes</n></n>	dup_xz	insert 2 or 3 values down	invokespecial	Invoke instance method	nop	Pop top stack operand
dup22 stack valuesinvokestaticInvoke static methodpufp21 op top 1 of 2dup2_x1Duplicate top 1 or 2 stack values, insert 2 or 3 values downinvokestaticInvoke instance methodputfieldSet field in objectdup2_x2Duplicate top 1 or 2 stack values, insert 2, 3, or 4 values downiremRemainder intretReturn from subroutinef2dConvert float to doubleishrArithmetic shift right intsaloadLoad short from arrayf2iConvert float to intistoreStore int into local varsipushPush shortf2lConvert float to longistore <n>Store int into local varswapSwap top two stackfaddAdd floatisubSubtract intoperandsoperandsfaloadLoad float from arrayiushrLogical shift right inttableswitchAccess jump table by index and jumpfcms<op>Compare floatjsrJump to subroutinewideExtend local var index by index and jumpfcms<op>Dompare floatjsr_wJump to subroutine index)wideExtend local var index by bytes</op></op></n>	dun2	Duplicate top 1 or	linvokespeciai	with special handling	pop2	Pop top 1 or 2
dup2_x1Duplicate top 1 or 2 stack values, insert 2 or 3 values downinvokestattinvoke instance methodputfieldSet field in objectdup2_x2Duplicate top 1 or 2 stack values, insert 2, 3, or 4 values downiorBoolean OR intputstaticSet static field in classdup2_x2Duplicate top 1 or 2 stack values, insert 2, 3, or 4 values downiremRemainder intretReturn from subroutinef2dConvert float to doubleishrArithmetic shift left intsaloadLoad short from arrayf2iConvert float to intistoreStore int into local varsipushPush shortf2lConvert float to longistore <n>Store int into local varswapSwap top two stack operandsfaddAdd floatisubSubtract intoperandsoperandsfaloadLoad float from arrayiushrLogical shift right inttableswitchAccess jump table by index and jumpfcmp<op>Compare floatjsrJump to subroutinewideExtend local var index by bytes</op></n>	land	2 stack values	invokestatic	Invoke static method	popz	stack operands
dup2_x1Duplicate top 1 or 2 statist values, insert 2 or 3 values downinforce instance instance methodpatientDescriptiondup2_x2Duplicate top 1 or 2 stack values, insert 2, 3, or 4 values downiorBoolean OR intputstaticSet static field in classf2dConvert float to doubleish1Return int from methodreturnReturn void from methodf2iConvert float to doubleishrArithmetic shift right intsaloadLoad short from arrayf2iConvert float to longistore Store int into local varsipushPush shortf2iConvert float to longistore <n> Store int into local varswapSwap top two stack operandsfaddAdd floatisubSubtract intoperandsfaloadLoad float from arrayiushrLogical shift right inttableswitchfastoreStore into float arrayisvorBoolean XOR intwidefcmp<op>Compare floatjsrJump to subroutinewidefdivDivide floatjsr_wJump to subroutine (wide index)by index and jump</op></n>	dun2 x1	Duplicate top 1 or 2 stack values	invokevirtual	Invoke instance method	nutfield	Set field in object
dup2_x2Duplicate top 1 or 2 stack values, insert 2, 3, or 4 values downiremRemainder intretReturn from subroutinef2dConvert float to doubleishlShift left intsaloadLoad short from arrayf2iConvert float to doubleishrArithmetic shift right intsastoreStore into short arrayf2iConvert float to intistoreStore int into local varsipushPush shortf2iConvert float to longistore <n>Store int into local varsipushPush shortfaddAdd floatisubSubtract intoperandsoperandsfaloadLoad float from arrayiushrLogical shift right inttableswitchAccess jump tablefastoreStore into float arrayisvorBoolean XOR intby index and jumpfcmp<op>Compare floatjsrJump to subroutinewideExtend local var indexfdivDivide floatisotindex)index)by bytes</op></n>		insert 2 or 3 values down	ior	Boolean OR int	nutstatic	Set static field in class
InternationInternationInternationInternationInternationstack values, insert 2, 3, or 4 values downistReturnReturn int from methodreturnReturn void from methodf2dConvert float to doubleishShift left intsaloadLoad short from arrayf2iConvert float to intistoreStore int into local varsipushPush shortf2iConvert float to longistore <n>Store int into local varsipushPush shortf2iConvert float to longistore <n>Store int into local varswapSwap top two stackfaddAdd floatisubSubtract intoperandsfaloadLoad float from arrayiushrLogical shift right inttableswitchAccess jump tablefastoreStore into float arrayixorBoolean XOR intwideExtend local var indexfconst <f>Push floatjsr_wJump to subroutinewideExtend local var indexfdivDivide floatisotindex)index)index)index</f></n></n>	dun2 x2	Duplicate top 1 or 2	irem	Remainder int	ret	Return from subroutine
insert 2, 3, or 4 values downinstantinstantinstantishShiftShift left intsaloadLoad short from arrayishishrArithmetic shift right intsastoreStore into short arrayiziConvert float to intistoreStore int into local varsipushPush shortiziConvert float to longistore <n>Store int into local varsipushPush shortiziConvert float to longistore <n>Store int into local varswapSwap top two stackfaddAdd floatisubSubtract intoperandsfaloadLoad float from arrayiushrLogical shift right inttableswitchAccess jump tablefastoreStore into float arrayixorBoolean XOR intby index and jumpfcmp<op>Compare floatjsrJump to subroutinewideExtend local var index by by tesfdivDivide floatisr_wJump to subroutine (wide index)by by tesby by tes</op></n></n>	aupz_xz	stack values.	ireturn	Return int from method	return	Return void from method
f2dConvert float to doubleishrArithmetic shift right intsastoreStore into short arrayf2iConvert float to intistoreStore int into local varsipushPush shortf2lConvert float to longistore <n>Store int into local varsipushPush shortf2lConvert float to longistore <n>Store int into local varswapSwap top two stackfaddAdd floatisubSubtract intoperandsfaloadLoad float from arrayiushrLogical shift right inttableswitchAccess jump tablefastoreStore into float arrayixorBoolean XOR intby index and jumpfcmp<op>Compare floatjsrJump to subroutinewideExtend local var index by by tesfdivDivide floatisotindex)index)index</op></n></n>		insert 2, 3, or 4 values down	ishl	Shift left int	saload	Load short from array
f2i Convert float to int istore Store int into local var sipush Push short f2l Convert float to long istore <n> Store int into local var swap Swap top two stack operands fadd Add float isub Subtract int swap Swap top two stack operands faload Load float from array iushr Logical shift right int tableswitch Access jump table fastore Store into float array ixor Boolean XOR int wide Extend local var index operands fcmp<op> Compare float jsr Jump to subroutine (wide index) wide Extend local var index operands</op></n>	f2d	Convert float to double	ishr	Arithmetic shift right int	sastore	Store into short array
f2l Convert float to long istore <n> istore <n> Store int into local var swap Swap top two stack operands fadd Add float isub Subtract int swap Swap top two stack operands faload Load float from array iushr Logical shift right int tableswitch Access jump table by index and jump fastore Store into float array ixor Boolean XOR int wide Extend local var index by by tes fcmp<op> Compare float jsr Jump to subroutine (wide index) wide Extend local var index by by tes</op></n></n>	f2i	Convert float to int	istore	Store int into local var	sipush	Push short
fadd Add float isub Subtract int operands faload Load float from array iushr Logical shift right int tableswitch Access jump table fastore Store into float array ixor Boolean XOR int by index and jump fcmp <op> Compare float jsr Jump to subroutine wide Extend local var index by by tes fdiv Divide float jsr_w Jump to subroutine (wide index) by by tes</op>	f2l	Convert float to long	istore_ <n></n>	Store int into local var	swap	Swap top two stack
faloadLoad float from arrayiushrLogical shift right inttableswitchAccess jump table by index and jumpfastoreStore into float arrayixorBoolean XOR intby index and jumpfcmp <op>Compare floatjsrJump to subroutinewideExtend local var index by by tesfcup<Push floatjsr_wJump to subroutine (wide index)wideExtend local var index by by tes</op>	fadd	Add float	isub	Subtract int	· ·	operands
fastore Store into float array ixor Boolean XOR int by index and jump fcmp <op> Compare float jsr Jump to subroutine wide Extend local var index fconst <f> Push float jsr_w Jump to subroutine (wide index) wide Extend local var index</f></op>	faload	Load float from array	iushr	Logical shift right int	tableswitch	Access jump table
fcmp <op> Compare float jsr Jump to subroutine wide Extend local var index fconst <f> Push float jsr_w Jump to subroutine (wide index) wide Extend local var index by bytes</f></op>	fastore	Store into float array	ixor	Boolean XOR int		by index and jump
fconst <f> Push float jsr_w Jump to subroutine (wide index) by bytes</f>	fcmp <op></op>	Compare float	jsr	Jump to subroutine	wide	Extend local var index
fdiv Divide float index)	fconst_ <f></f>	Push float	jsr_w	Jump to subroutine (wide		by bytes
	fdiv	Divide float		index)		
fload Load float from local var	fload	Load float from local var				

Table 1. The Java bytecode instruction set has 204 instructions, including three reserved instruction slots. This table shows only 151 mnemonics, because some instructions (such as aload_<n>) have multiple numbered mnemonics (aload_1, aload_2, etc.) for different operand types. Operand types are *byte* (8 bits), *short* (16-bit integer), *int* (32-bit integer), *long* (64-bit integer), *float* (32-bit floating-point), and *double* (64-bit floating-point). Gray cells in this table indicate floating-point instructions not supported by Javalon-1. *These opcodes are reserved for debuggers and implementation-specific instructions; they never appear in standard Java class files.

×.

2

Genesis of a Core

If you've never heard of Octera, it's probably because Javalon-1 is the company's first venture into intellectual-property (IP) licensing. Octera is primarily an embedded-systems design center with experience in ASICs, SoCs, and board-level design. Some of Octera's founders and managers have been working with stack-based computers since the days of Burroughs and Unisys, so they feel at home with Java's retro stack architecture. They have also noticed that universities are turning out thousands of young programmers who are more comfortable with Java than with assembly language and C, the longtime staples of embedded-system development. Octera thinks the time is ripe for a microprocessor core that can execute Java bytecodes in hard real-time applications.

Javalon-1 is the product of a four-year design effort. The processor has a 32-bit stack-based architecture created from scratch to run Java bytecodes and nothing else. Most microprocessors, of course, have a native machine language unrelated to any particular high-level programming language. In contrast, Javalon-1's machine language is derived from the Java bytecode instruction set—the processor has no other machine language.

Bytecodes are the virtual machine instructions in Java class files that an interpreter or JIT compiler normally translates into native executable code at run time. As their name implies, bytecodes are 8 bits long, and they manipulate operands ranging in size from 8 to 64 bits. Table 1 lists all the instructions in the Java instruction-set architecture (ISA), some of which aren't supported by Javalon-1.

The Java ISA is unusual when compared with other microprocessor ISAs, mainly because Sun designed it primarily as a software virtual machine, not for implementation in logic as a conventional microprocessor. Sun optimized the Java ISA for high code density (hence the byte-sized instructions) and for easy portability to any CPU architecture. The ISA also has several unusual instructions that support the object-oriented features of the Java programming language.

Because Sun's Java definition largely determined the processor's architecture, Javalon-1 is a rudimentary stackbased CISC machine. Octera also kept the microarchitecture simple. There is no instruction pipeline in the usual sense, unless you consider the fetch and execution units to be a two-stage pipe. For the sake of determinism, there are no conventional instruction or data caches, because their behavior is unpredictable. Nor are there any programmervisible registers, because Java bytecode instructions manipulate all their operands on the stack.

Although it's possible for programmers to directly access the stack with low-level bytecode instructions—Java bytecode assemblers are rarely used but available—the stack is completely transparent to high-level Java-language programmers. In effect, Javalon-1 is a black box from the Java programmer's point of view. Figure 1 shows what's inside the box.

The only registers in Javalon-1 are a small top-ofstack cache and a handful of datapath registers for internal bookkeeping. There is also a small instruction-fetch buffer. Different members of the Javalon family will have stack caches and fetch buffers of different sizes. Javalon-1 has a mere three-entry stack cache, 32 bits wide, and an eight-byte fetch buffer, enough for eight instructions. According to Octera's code profiling, those are the minimum sizes for a stack cache and fetch buffer that are large enough to significantly boost performance while scarcely affecting the size of the core; they add only about 1,000 gates. Octera found that a stack cache with fewer than three entries caused excessive memory thrashing, canceling any advantage from the cache.

For customers that want higher performance at the expense of, perhaps, a couple of thousand more gates, Octera will offer an eight-entry stack cache and a slightly larger instruction-fetch buffer, doubling the processor's performance. That may seem like a surprising improvement for such minor enhancements, but Octera says the larger cache and fetch buffer dramatically reduce memory contentions between the fetch and execution units. Nevertheless, the enhancements aren't a standard feature in Javalon-1, because Octera believes the processor's performance already exceeds requirements for the intended market. The deeper cache and fetch buffer are luxuries for those who want them.

Javalon-1's internal datapath registers keep track of the stack pointer, class-file index, local-variable index, and other Java housekeeping details. Java programmers will never see them. The fetch unit, shown in Figure 1, prefetches instructions into the fetch buffer and hands them off to the execution unit for decoding and execution. The most common instructions execute in a single clock cycle. If Octera



Figure 1. Javalon-1 block diagram. There are no programmer-visible register files or instructions other than what's available in the Java byte-code definition. Internal datapaths are 32 bits wide. Note the AMBA High-speed Bus (AHB) interface, which allows chip designers to use Javalon-1 with other microprocessor cores and peripheral IP.

3



Octera's Javalon-1 design team, from left: Laury Flora, David Castle, Dale Rigtrup, Bob Noble, Randy Rhodes, Lee Burke, Howard Keller, Jill Kiggens, Joe Mizerak, Gary Whitlock, Carlos Guerra, and Paul Nunnally.

perceives demand for a more powerful processor, a future Javalon might have superscalar pipelines and a fetch unit that predecodes instructions for the execution units. For now, Octera is keeping the core as small as possible.

Thanks to the core's fast datapaths and simple microarchitecture, Octera estimates that Javalon-1 could run at 300MHz (worst case) in a 0.13-micron CMOS process. However, the company believes anything close to 300MHz is overkill for the vast majority of deeply embedded applications, which probably use 8- or 16-bit processors today. Therefore, Octera expects most customers to run Javalon-1 at a more leisurely pace of 1MHz to 10MHz, which would drop power consumption to extremely low levels.

Octera's engineers have early implementations of the processor running at 50MHz in an FPGA and expect to reach 100MHz soon. That's fast enough for many embedded controllers. Indeed, a low-volume embedded system could forgo the high nonrecurring engineering costs of spinning a silicon chip and deploy the Javalon-1 in a PLD instead. With about 25,000 ASIC gates, the core fits into a 150,000-programmable-gate FPGA costing about \$20.

Trashing the Garbage Collector

Because Javalon-1 natively executes Java bytecodes, it doesn't need a bytecode interpreter or JIT compiler. That alone can save a megabyte or more of memory, especially since JIT compilers create a duplicate memory image of each program they translate (the bytecode version and the recompiled native version). In fact, Javalon-1 doesn't need a JVM at all, because it either renders the functions of a JVM redundant or omits the features that would require a JVM.

For example, to eliminate Java's prime source of nondeterministic behavior, Javalon-1 dispenses with the JVM's garbage collector. Normally, Java relieves programmers of the burden of memory management by automatically allocating and deallocating memory while a program executes. The JVM does this by running a garbage-collection routine in a background thread to periodically free up memory no longer referenced by objects. Unfortunately, garbage collection is an unpredictable and fairly heavyweight task that can seriously impair a system's ability to respond to real-time events.

Octera's solution is simple: never deallocate memory. Any memory allocated for an object remains encumbered for as long as the program runs. This eliminates the need for a garbage collector, but it also shifts the burden of memory management back to the programmer. Programs written for Javalon-1 must create all their objects at startup and ensure there's enough memory to accommodate them at runtime.

This is sacrilege to Java purists, who will protest that Octera's solution negates one of the biggest advantages of Java and reintroduces a probable source of bugs that plagues C/C++. However, Javalon-1 programs should never suffer from memory leaks—a common memory-management problem in C programs—because they're required to initialize all objects at startup and never deallocate the memory. In effect, a Javalon-1 program "leaks" all its memory during initialization and never needs more, because it doesn't create any additional objects.

Contrary to popular belief, garbage collection isn't required by Sun's JVM specification, which Javalon-1 doesn't religiously follow anyway. The official specification states that the JVM "assumes no particular type of automatic storage management system, and the storage management technique may be chosen according to the implementor's system requirements." (Sun's JVM specification is available online at *http://java.sun.com/docs/books/vmspec/.*)

Nevertheless, Octera isn't deaf to the cries of Sun worshippers. The company says future members of the Javalon family may restore automatic memory management, at least as an option. There are simpler alternatives to a garbage collector, such as object-reference counters.

5

Different Ways of Going Native

Another Java feature that Javalon-1 makes redundant—in the usual sense, at least—is native method calls. To Java programmers, a native method call is the ability to run program code written in another language (such as C/C++ or assembly language) from a Java program. Normally, when a JVM encounters a native method call, it hands off execution to the system's host processor, which executes the native code and returns any results to the Java program. Native methods are generally the last resort for subroutines that simply won't run fast enough in 100%-pure Java. They also provide a gateway to legacy code so programmers don't have to rewrite a whole project in Java.

In a Javalon-1 system, however, there may not be another host processor, so Java bytecodes *are* the native machine language. In that sense, Javalon-1 doesn't support native methods compiled to executable code for other processors, because it can't run anything but Java bytecodes. This limitation sets Javalon-1 apart from some other embedded processors with bolt-on Java accelerators, such as the ARM1026EJ-S, ARM1136JF-S, and ARCtangent-A5. (See *MPR* 10/21/02-02, "MPF Hosts Premiere of ARM1136.")

Nevertheless, Javalon-1 does support native methods just not in the way most Java programmers would think. Sun's JVM specification opens a little-known door for defining implementation-specific bytecode instructions that aren't part of the standard Java ISA. Indeed, the ISA reserves two opcodes for this purpose, with the mnemonics impdep1 and impdep2 (seen in Table 1). Although implementationspecific instructions are verboten in standard Java class files, programmers can use them in subroutines called as native methods. And because subroutines with implementationspecific Java bytecodes deviate from the standard Java ISA, technically speaking, they are native methods.

In that sense, then, Javalon-1 supports native methods. Octera has created about a half-dozen proprietary instructions that it prefers not to disclose publicly at this time. Octera has also created a bytecode assembler that recognizes those instructions as well as standard instructions. The proprietary instructions appear in native methods stored in the processor's supporting firmware. Among other things, they help execute some standard instructions that Javalon-1 doesn't implement in logic. (More on this later.)

Developers who port existing Java code to Javalon-1 will have to rewrite their C/C++ or assembly-language native methods in Java. One of the tools Octera supplies will scan Java programs for native method calls so they won't be overlooked.

No Downward Class Mobility

To conserve memory, Javalon-1 also doesn't use the Sunstandard Java class-file format. Class files are the product of a Java compiler that converts high-level Java source code into bytecodes. Java class files contain a great deal of symbolic information about variables, objects, and other structures, because, usually, the class files aren't directly executable: a Java interpreter or JIT compiler must translate them into the native machine language of the underlying hardware. But because Javalon-1 *does* execute bytecodes without further translation, it can dispense with some redundant information in the class files and save some application memory.

One downside of abandoning the standard class-file format is that Javalon-1 isn't suitable for embedded systems that must be compatible with the universe of Java software. For example, next-generation mobile phones and PDAs that can download and execute Java applets must be able to handle standard Java class files. Ditto for embedded Web browsers that need to run Java applets.

However, a nonstandard class-file format is an acceptable compromise for a deeply embedded system that doesn't need to dynamically load and run new classes. Indeed, a creative product manager could even pitch it as a security feature that prevents malicious hackers from altering the software of mission-critical systems.

Instead of using standard Java class files, Javalon-1 uses a slightly modified, proprietary version of the Java Executable File Format (JEFF). JEFF was created by the independent J Consortium and was adopted as an ISO standard (ISO/IEC DIS 20970) in 2001. JEFF compresses Java class files to save memory and improves upon the Java archive (JAR) format, an older compressed class-file format created by Sun. Among other things, JEFF allows Java classes to execute from ROM, a valuable feature for embedded systems. (For more information about JEFF, visit the J Consortium's Web site at *www.jconsortium.org/*.)

Octera, a member of the J Consortium, modified JEFF to make the class files even more compact and easier for Javalon-1 to decode. Dynamic class loading is sacrificed, but this also eliminates the need for a conventional class loader and verifier, two other runtime components that occupy memory in a JVM. Octera will provide a design-time utility that converts standard JEFF class files into the modified JEFF files that Javalon-1 requires. Another tool in the package is Octera's own JEFF generator, which converts standard class files into JEFF files, although other JEFF generators may also work.

A Few More Missing Pieces

Octera deleted a few more features from the standard Java platform that were deemed irrelevant for embedded controllers, such as support for a computer keyboard and mouse. If a more sophisticated embedded system (such as a point-ofsale terminal) requires those devices, Javalon-1 could function as a slave to another processor via the AMBA bus.

Two more-serious omissions are multithreading and floating-point math. Multithreading is a stellar feature of Java—built-in classes make it easy to implement—but Octera believes that thread management would use too much memory and impair the processor's real-time response. Likewise, Javalon-1 doesn't support floating-point math at all, either in hardware or software. Octera says a software floating-point package could occupy as little as 1K of memory, but it would be slow and isn't written yet. Until then, the workaround is to use scaled integers—or a host processor.

In a more familiar compromise, Javalon-1 executes about 80 of the most-complex or seldom-used Java bytecode instructions in software instead of in logic. It's difficult to justify the extra gates required to implement some bytecode instructions in logic; every Java processor we've seen traps at least a few instructions for software execution. For instance, the instruction that creates a new multidimensional array (multianewarray) occurs infrequently and requires an indeterminate number of cycles to execute, depending on the data type and dimensions of the array it must construct. The most thoroughly hard-wired Java processor is aJile's aJ-100, which executes all but two bytecode instructions in hardware: multianewarray and athrow.

Deciding which instructions to implement in logic and which in software depended not only on how often the instruction occurs and its complexity but also on its effect on real-time performance. The reason that Javalon-1 requires as many as 100 clock cycles to service an interrupt—which is somewhat high, by hard real-time standards—is that it can stop executing a complex instruction in software, save the partial state, service the interrupt, restore the partial state, and resume execution where it left off. Javalon-1 can do this even while executing in software an instruction for an interrupt-service routine that is interrupted by a higherpriority interrupt.

Octera says future Javalon processors will slash the worst-case response time to five to ten clock cycles by implementing more instructions in logic. With Javalon-1, the most complex hardware instruction executes in only eight clock cycles (or perhaps a few more, depending on memory contention and the state of the stack cache). Therefore, when Javalon-1 is executing the most-common instructions likely to occur in embedded-controller code, its interrupt-response time is much better than 100 clocks.

Javalon-1 executes a few instructions in software that could significantly affect performance in other ways, depending on the application. The most important of these are integer multiplication and division operations. Octera has designed hardware implementations of those instructions, but they require about 5,000 gates, so Javalon-1 executes them in software instead. The company says a future, higherperformance member of the Javalon family will probably use the hardware multiply/divide implementations.

Thanks to all these economies, Javalon-1 requires only 5K of supporting firmware, exclusive of application memory. The firmware includes the software routines that trap and execute instructions not supported in logic, plus some miscellaneous native methods. All are written in Javalon-1's native machine language: Java bytecodes augmented by Octera's proprietary bytecodes.

In silicon, a Javalon-based chip should compare favorably with other 32-bit embedded processors, whether they can natively execute Java bytecodes or not. At 25,000 gates, Javalon-1 is about the same size as a small ARM or ARC core, especially after the addition of a Java accelerator, such as ARM's Jazelle or DCT's Bigfoot. (See *MPR 2/12/01-01*, "Java to Go: Part 1," and *MPR 1/28/02-04*, "DCT Marches Into Java Processors.") Javalon-1's firmware is small enough to fit inside on-chip flash memory or mask ROM, probably occupying about as much room as the instruction and data caches of a conventional embedded processor.

By Java standards, 5K of firmware is positively microscopic. Normally, a standard JVM requires hundreds of kilobytes or megabytes, especially if it includes such luxuries as a JIT compiler. Even Sun's stripped-down K virtual machine (KVM) for J2ME typically occupies 128K of memory and requires still more memory to operate. Of course, J2ME is a Sun-standard Java platform that has many features Javalon-1 lacks.

Is It Coffee or Postum?

Octera makes no claims that Javalon-1 provides a Sunstandard Java platform. Obviously, Octera's designers made several compromises in their quest to bring some flavor of Java to deeply embedded real-time controllers. Squeezing everything down to 25,000 gates and 5K of firmware simply isn't possible without discarding some baggage. The first question for developers is whether they need anything in that discarded baggage to reach their destination.

The list of what's missing appears lengthy: automatic memory management, native method calls (in the usual sense), universal class-file compatibility, dynamic class loading, keyboard/mouse support, multithreading, and floatingpoint math. Keep in mind, however, that those features are uncommon in embedded controllers of any flavor. They seem like sacrifices only to developers accustomed to using a richer Java platform on larger systems.

The next question is whether there's enough Java left in Javalon-1 to make it a better alternative than any other embedded processor. The answer is a qualified yes. Octera's diluted version of Java still has some advantages over programming in C and assembly language, including easier code portability to other embedded processors in the future, whether those processors are Java-native or not.

Still, some Java features will be sorely missed. Automatic memory management improves programmer productivity and eliminates a common source of bugs. The ability to invoke native methods written in C or assembly language preserves investments in legacy code. Standard class-file compatibility and dynamic class loading are imperative for Javaenabled Web browsers and applications that must download periodic software updates in the field. Multithreading is invaluable for managing multiple tasks. Floating-point math may be required for some controllers. If a project depends on any of those features, Javalon-1 isn't the best solution.

Finally, there's the question of whether Javalon-1 can satisfy the performance requirements of an application. Is it

fast enough to run the software at the specified power level, and will it respond quickly enough to real-time interrupts?

Benchmark data for Java processors is hard to come by; the often-quoted CaffeineMark is a poor substitute for a rigorous embedded test suite. EEMBC has a new suite of embedded Java benchmarks, but because Javalon-1 isn't 100% Java compatible, the benchmark code probably won't run without extensive modifications. For now, the best way to evaluate Javalon-1 is to extrapolate from its performance in a simulator or FPGA.

Our conclusion is that Javalon-1 will be exactly what some developers are looking for. Others will find Javalon-1's compromises too bitter to swallow. And many developers will remain skeptical that Java is appropriate for hard real-time applications at all. Selling the concept is a bigger challenge than selling the processor. Previous embedded Java processors have found a mixed welcome in the marketplace, with at least one company—Vulcan Machines—recently folding its tent. (See *MPR* 7/1/02-03, "Vulcan Moon Shines Again.")

On the other hand, ARM is enjoying some success with its Jazelle accelerator for the ARM10 and ARM11

Price & Availability

Octera plans to begin licensing the Javalon-1 processor core in 3Q03. It will be supplied in Verilog or VHDL with some special development tools, including a JEFF classfile generator and a JEFF-to-Javalon converter. Future Javalon-family cores will be announced later. Licensing fees have not yet been determined.

cores, and ARC is tempting similar customers with DCT's Java extensions for ARCtangent-A5. What Octera has in common with ARM and ARC is that embedded Java processors are not central to its business. Octera is still primarily an independent design house; Javalon-1 isn't a bet-the-company proposition.

For embedded-system developers, more choices are always better than fewer choices. Octera's Javalon family is a different and creative new option. \diamond

To subscribe to Microprocessor Report, phone 480.609.4551 or visit www.MDRonline.com