THE INSIDERS' GUIDE TO MICROPROCESSOR HARDWARE

68HC11 Grows Up to 16 Bits Motorola's 68HC12 Line Boosts Performance up to 10 Times

by Jim Turley

Motorola's popular 8-bit microcontroller architecture, the 68HC11, now has a big brother. The new 68HC12 maintains all of its sibling's distinguishing features but doubles both clock rate and bus width, extends the address space, improves code density, and adds dozens of new instructions, including some for fuzzy logic.

Strategically, the 16-bit HC12 family—which initially includes two members—fills a small gap between Motorola's 8-bit 68HC11 family and the 16-bit 68HC16 line. The HC12 provides a source-level software upgrade for 8-bit designers who are outgrowing the HC11 but don't want to develop entirely new code and new hardware for the HC16.

Motorola believes the HC12 and HC16 can peacefully coexist in the company's 16-bit microcontroller lineup because the HC16 offers better performance and better signal-processing capabilities, while the HC12 offers compatibility with the HC11 and lower power dissipation. The first two chips in the new family are both priced below \$25 and will begin sampling in June and October of this year.

Programming Model Identical to HC11

The HC12's register set and programming model are identical to those of the HC11, as Figure 1 shows. The two 8-bit accumulators, A and B, can be concatenated into a single 16bit register, D. Index registers X and Y are used to reference memory-resident operands, while the stack pointer (SP) and program counter (PC) fulfill the obvious functions.

The HC12's exception stack and fault model are also identical to those of the HC11 chips, so users with existing code that examines or manipulates the stack will find that it works without modification. Source-code compatibility was a primary concern for the HC12's designers; given that Motorola already has a 16-bit product family, the HC12 had to offer something the HC16 didn't.

The familiar programming model belies major changes lurking beneath the surface. Motorola's customer surveys indicated that HC11 users were frustrated by the irregular and nonorthogonal treatment of the X and Y index registers. For example, most HC11 instructions can reference memory through either the X or Y index registers, but using the Y register adds a prefix byte to the object code, which requires an additional clock cycle to fetch over the HC11's 8-bit bus.

These concerns were addressed with the HC12, which encodes X- and Y-indexed instructions equally. At the same time, addressing modes were modified to support both SP and PC as index registers. Thus, compiler writers can more easily reference operands passed on the stack, and positionindependent code can access data relative to the program counter.

Instruction Set Gets 65 New Mnemonics

The design goals for the HC12 stipulated total source-level (but not binary) compatibility with the HC11 so users could transfer existing assembly source without modification. This it does, duplicating every HC11 mnemonic and addressing mode, right down to some unintended quirks that users have learned to accept.

7	А	0	7		В				0
	8-bit accumulators A & B or 16-bit double accumulator D							D	
15	IX							0	
	Index register X								
15	IY							0	
	Index register Y								
15		SP)						0
Stack pointer									
15		РС	:						0
Program counter									
		[S X	н	T	Ν	Z	v	С
	Condition codes register								

Figure 1. The register set of the 68HC12 is identical to that of its predecessor, the 8-bit 68HC11.

The HC12 truly implements its predecessor's instruction set one for one; unlike Philips' 8051XA tools (*see* **081304.PDF**), the HC12 assembler does not replace unimplemented legacy instructions with equivalent constructs. In addition to the HC11 instruction set, the HC12 has more than 65 new operations that should cheer compiler writers and relieve assembly programmers.

Table 1 lists the complete set of HC12 instructions with enhancements over the HC11 indicated. Some examples include the TFR (transfer) and EXG (exchange) instructions, which now handle mismatched register sizes. Exchanging an 8-bit with a 16-bit register zero-extends the 8-bit value; copying mismatched registers sign-extends the smaller register. This latter operation can be used by C compilers to cast a *char* to an *int*.

In cases where HC11 instructions sometimes produce unwanted side effects, as when the TAB and TBA (transfer A/B) instructions set condition codes even though most transfer instructions do not, the HC12 includes compatible instructions (complete with side effects) in addition to the preferred (and more orthogonal) versions. The generalized TFR instruction is now preferred over TAB and TBA, while ANDCC and ORCC replace SEI, CLI, SEC, and CLC.

No Binary Compatibility with HC11

The HC12's opcode map was completely rewritten, so binary

MOVB Move byte BCS/BCC Branch if carry set/clear ANDA/ANDE Logical AND A/B MOVW Move word BCG/BLE Branch if greater/less or equal ANDCC Logical AND A/B LDD Load dark pointer BC/BLE Branch if greater/less or equal ANDCC Logical AND A/B LDXLDY Load dark pointer BC/BLE Branch if greater/less or equal ANDCC Anthmetic shift left A/B LEAS Load effective address, stack BH/SBLS Branch if higher/lower or same ARA/A/SB Arithmetic shift left A/B LEAXLEXA Load effective address, stack BRCL/RRSET Branch if bits cleared/set COM Compare A/B STM STS Store dauble D BYSRVS Branch to subroutine COM Compare A/B STMSTS Store dauble D BYSRVS Branch to subroutine CPD Compare stack pointer STMSTS Store A/B Store dauble D Descrement, branch if roz CPM COMACOMB Complement A/B PSHXPBHP Puth condition codes totack DBEN Decrement, branch if razry set/clear CRX.ASR Logical shift left A/B PULC Pull condition codes totack DBEN Decrement, branch if razry set/clear CRX.ASR Logical shift left A/B PULX/PULY Pull	Data Transfer Flow Control				Shift and Logical			
MOW Move word BEC/BINE Branch if equal/not equal ANDCC Logical AND condition codes LDAALDAB Load double D BGE/BILE Branch if greater/Hso requal ASUASR Arithmetic shift left/right LDD Load double D BGE/BILE Branch if greater/Hso requal ASUASR Arithmetic shift left A/8 LDXLDY Load defective address, stat. BH/BILO Branch if higher/lower or same ARUASR Arithmetic shift left A/8 LEAS Load effective address, stat. BPUEM Branch if higher/lower or same Compare A/8 Compare A/8 STD Store duble D BYLBN Branch if higher/lower or same CPD Compare A/8 Compare A/8 STD Store duble D BYLBN Branch to subroutine CPZ CPC Compare A/8 PSHAPSHB Push condition codes to stack DBSC Decrement, branch if zero DPXCP Compare A/8 PULX/PULB Puil condition codes to stack DBN Decrement, branch if arrow CBLUSPUL Compare A/8 PULX/PULB Puil Condition codes to stack DBN Dec	MOVB	Move byte	BCS/BCC	Branch if carry set/clear	ANDA/ANDB	Logical AND A/B		
LDAALDAB Load A/B LDD LDD LDD LDD LDD LDD LDD LD	MOVW		BEQ/BNE					
LDDLoad double DBGT/BLTBranch if greater than/less thanSLAASLBArithmetic shift left A/8LDSLoad stack pointerBH/BLOBranch if higher/lower or sameASLAASLBArithmetic shift left A/8LDX/LDYLoad effective address, stackBPL/BLSBranch if higher/lower or sameASLAASLBArithmetic shift left A/8LEAXLoad effective address, stackBPL/BLMBranch if bits/fniumsCMPA/CMMBComplement A/8STAASTABStore A/8BRCLR/BRSETBranch if bits/fniumsCMMA/CMMBComplement A/8STASTABStore stack pointerBSRBranch to subroutineCPDCOM/CMBComplement A/8STNSTYStore X/YStore X/YRTCReturn from CALLCPDCompare stack pointerSTNSTYStore X/YStackDEFCDecrement, branch if rot zeroLSULSRLogical shift left A/8PSHX/PSHYPuish A/8 to stackDEFCDecrement, branch if rot zeroLSULSRLogical shift left A/8PULZPuil condition codes to stackJMPJump to subroutineLSULSRLogical shift left A/8PULXPULBPuil A/8 from stackJMPJump to subroutineLSUALSLBLogical shift left double DPULXPULPPuil A/16 to AASEA/ASERTSReturn from subroutineLSRALSRBLogical shift left double DPULXPULPPuil A/17 to stackJSRJump to subroutineLSRALSRBLogical shift left double DPULXPULPPuil A/17 to stackJSRJump to subroutineLSRALS								
LDS LDXLDYLoad stack pointer LDXLDYBH/BLO Barch if higher/lower BH/SBLSSLDArithmetic shift left double D ASRA-ASRAAtthmetic shift left double D Compare A/BAsra-ASRA Atthmetic shift left A/B COMPACMPBArithmetic shift left double D Compare A/BCompare A/B Compare A/BCompare A/B 								
LDXLDY Load Load HSrbLS Branch if higher/Jower or same SRAASRB Arithmetic shift right A/B LEAX Lead effective address, stack BPU/BM BRACHRN Branch if blus/minus CMPACMPB Complement memory STD Store double D BYUBNCK BRACHRN Branch if blus/minus CMACOMB Complement A/B STD Store double D BSVS/BVCK Branch to subroutine CPA COMACOMB Complement A/B STXSTY Store double D BSR Branch to subroutine CPA Complement A/B PSHACPSHP Push condition codes to stack DBRC Decrement, branch if not zero LSLALSLB Logical shift left A/B PULX/PULD Pull condition codes to stack DBRE Decrement, branch if not zero LSLALSLB Logical shift left A/B PULX/PULY Pull A/B from stack Transfer A to condition codes to A BRUBNE LSCLSEC Long branch if zero/not zero PULX/PULY Pull X/PULY Pull X/PULY Pull X/PULY BGTLBL LBGTLBL LB								
LEAX Lad efective address, stack BPL/BMI Branch if plis/minus CMPA/CMPB Compare A/B LEAXLEAY Lad efective address, stack BPL/BMI Branch always/never CMPA/CMPB Complement memory STD Store stack pointer BS/BV/C Branch always/never CPS Compare stack pointer STX/STY Store stack pointer BS/BV/C Branch always/never CPS Compare stack pointer STX/STY Store stack pointer CALL Call subroutine CPS Compare stack pointer STX/STY Store stack pointer DBR/E Decrement, branch if not zero LSL/LSLB Logical shift eft A/B PSHAP/SHP Push Avbit for stack IBEN Increment, branch if not zero LSR/LSLB Logical shift eft A/B PULX/PUL Pull A/B for stack IBEN Increment, branch if ero zero LSR/LSLB Logical shift eft A/B PULX/PUL Pull A/B for stack IBEN Increment, branch if not zero LSR/LSLB Logical shift eft A/B PULX/PUL Pull X/PU for stack IBES/LBNE Logical shift eft A/B RCR RCR RCR RCR Tansfer A to B //B to A </td <td></td> <td></td> <td></td> <td></td> <td>-</td> <td></td>					-			
LEAXUEAY Load effective address X/Y STAASTAB Store A/B STD Store double D STD Store double D STD Store double D STS Store stack pointer STXSTY Store X/Y PSHAPSHB Push A/B to stack PSHX/PSHP Push A/B to stack PSHX/PSHP Push X/Y to stack IBRC Increment, branch if zero PULX/PULP Pull A/B from stack PULX/PULP PULP PULP PULP PULP PULP PULP PULP				0		0		
STAARSTABStore A/BBRA/BRNBranch always/neverCOMACOMBCompare double DSTDStore double DBSRBRA/BRNBranch overflow set/clearCPDCompare double DSTX/STYStore K/YCALLCall subroutineCPDCompare A/BPSHA/PSHPPush A/B to stackDBEQDecrement, branch if rot zeroLogical shift left A/BPSHA/PSHYPush A/Dble D to stackDBEQDecrement, branch if rot zeroLogical shift left A/BPULA/PULAPull condition codes from stackIBNEIncrement, branch if rot zeroLSLALSLBLogical shift left alght A/BPULA/PULAPull double D from stackIBNEIncrement, branch if rot zeroLSRALSRBLogical shift left alght A/BPULA/PULAPull double D from stackIBNEIncrement, branch if rot zeroLSRALSRBLogical shift left alght A/BPULA/PULAPull A/P from stackIBNEIncrement, branch if rot zeroLSRALSRBLogical shift left alght A/BPULA/PULAPull A/P from stackIBNEIncrement, branch if rot zeroLSRALSRBLogical Shift left alght A/BTARTBATransfer A to B/B to AIBSC/ILBNELong branch if reary set/clearNEGA/NEGBNegate A/BTARTBATransfer to condition codesLBG2/LBNELong branch if fight/owNRCARDRBNotate left A/BTARTBATransfer A/BLBG2/LBNELong branch if rotro/rimeNCACARDLBNotate left A/BTARTBATransfer Add b to AAdd B to AAdd B to ANANegate A/B <td>-</td> <td></td> <td></td> <td></td> <td></td> <td></td>	-							
STDStore double DBVS/BVCBranch overflow set/clearCPDCompare double DSTSStore stack pointerBSRBranch to subroutineCPSCompare stack pointerSTX/STYStore X/YCALLCall subroutineCPX/CPYCompare stack pointerPSHAP/BSHBPush A/B to stackRTCReturn from CALLEDRAEORBExclusive-OR A/BPSHAPPush condition codes to stackDBNEDecrement, branch if zeroLSL/LSRLogical shift left/right memoryPSHAP/BSHPPulk A/B from stackIBNEIDECIncrement, branch if not zeroLSR/LSRLogical shift left/right memoryPULCPull condition codes from stackJSRJumpLSR/LSRLogical shift right A/BPULDPull double D from stackJSRJump to subroutineNEGNegate memoryPULTAPULYPull XY from stackERS/LBCCLong branch if zero/not zeroNRACORANegate memoryTAB/TBATransfer A to condition codes to ALBSC/LBCCLong branch if sers/greater, eq.ROL RAAORABRotate left A/BTSXTSYTransfer stack pointer to XYLBH/LBLDLong branch if ifgh/lowROARAORABRotate left A/BSKD/SYYTransfer double D with X/YLBH/LBLDLong branch if ifgh/lowROARAORABRotate left A/BSKD/SYYTransfer double D with X/YEXAlarge double D with X/YLBH/LBLDLBH/LBLDRota right/lowSKD/SYYStack apointer to X/YExcharge double D with X/YLBH/LBLDLBC/LBLELONg branch if ifg								
STS Store stack pointer BSR Branch to subroutine CPS Compare stack pointer STV/STY Store X/Y Call subroutine Call subroutine CPS Compare stack pointer PSHA/PSHB Push A/B to stack DBEQ Decrement, branch if zero EQALACS Lagical shift left/right A/B PSHA/PSHY Push double D to stack DBEQ Decrement, branch if zero Lagical shift left A/B PULAPULC Pull condition codes from stack DME Decrement, branch if zero LSRALSR Logical shift left A/B PULAPULP Pull A/P to stack IBNE Increment, branch if carry set/clar Logical shift right A/B PULAPULY Pull A/P to condition codes from stack JMP Jump LBRA LBRA </td <td></td> <td></td> <td></td> <td></td> <td></td> <td></td>								
STX/STYStore X/YCALLCall subroutineCPX/CPYCompare X/YPSHA/PSHBPush A/B to stackRTCReturn from CALLDBFQDecrement, branch if zeroDBFQDegreat the the the the the the the the the th	-				-			
PSHAPSHBPush A/B to stackFTCReturn from CALLCALLCALLCALLCALLCall SV-COR A/BPSHCPush double D to stackDBRDecrement, branch if not zeroLS/LSRLogical shift left/right memoryPSHAPSHYPush X/Y to stackIBEQIncrement, branch if not zeroLS/LSRLogical shift left A/BPULAPULDPull aduble D from stackIBNEIncrement, branch if not zeroLS/LSRLogical shift left A/BPULAPULPPull double D from stackJMPJump to subroutineLSRALSRBLogical shift left A/BPULXPULPPull double D from stackJSRJump to subroutineLSRALSRBLogical shift left A/BPULXPULPPull Avit from stackRTSReturn from subroutineLSRALSRBLogical -OR A/BTAB/TBATransfer A to S/B to ALBGC/LBLELong branch if carry set/clearORACORBNegate memoryTSXTSYTransfer stack pointer to X/YLBH/LBLDLong branch if less/greater, eq.ROLAROLBROLAROLBROLAROLBXGDXXGDYExchange registersLBH/LBLDLong branch if high/lowSCALEROR Rotate right A/BXGDXXGDYExchange registersLBH/LBLDLong branch if high/low, samtSEC/CLCSet/clear carryAbd Ad b to AAdd to A/BINCIncrement A/BSEC/CLCSet/clear carryADA/ADDBAdd to A/BINCIncrement A/BSEC/CLCSet/clear carrySBASubract double DINXINYIncrement A/BSEC/CLCSet/clear carry<			-					
PSHC Push condition codes to stack DBEQ Decrement, branch if zero LSL/LSR Logical shift left/right memory PSHZ PSHZ Push X/Y to stack DBNE Decrement, branch if zero LSLALSL Logical shift left A/B PULAPULB Pull A/B from stack IBNE Increment, branch if not zero LSRALSRB Logical shift left A/B PULC Pull double D from stack JSR Jump LSRALSRB Logical shift left double D PULT Pull X/Y from stack JSR Jump to subroutine NEG Negate memory TAB/TBA Transfer A to B/B to A LBCS/LBC Long branch if carry set/clar ORAA/ORAB Logical shift left double D TAB/TBA Transfer rodition codes to A LBC/LBNE Long branch if ferst/right/low NEG Negate A/B TSX/TSY Transfer stack pointer to X/Y LBGT/LBLE Long branch if ferst/right/low ROLA/ROLB Rotate left A/B XGDX/ROPY Exchange registers LBH/LBLO Long branch if high/low RORA/RORB Rotate left A/B XGDX/ADDE Add b to A Add b to A RTI Return from interrupt Rotate left A/B ADCA/ADDE Add to A/B INS Increment A/B Increment A/B SEV/CLC Set/clear oref10ow <		Push A/B to stack	RTC	Return from CALL	EORA/EORB			
PSH0 Push double D to stack DBNE Decrement, branch if not zero PSHX/PSHY Push X/Y to stack IBEQ Increment, branch if not zero PULAPULB Pull A/B from stack IBNE Increment, branch if not zero PULC Pull condition codes from stack JMP Jump to subroutine PULXPULY Pull X/Y from stack RTS Return from subroutine PULXPULY Pull X/Y from stack RTS Return from subroutine TAP Transfer A to B/B to A LBG2/LBNE Long branch if carry set/clear TAF Transfer condition codes to A LBG2/LBNE Long branch if gergreater, eq. TAF Transfer At to condition codes to A LBG7/LBLE Long branch if high/low, samt SXSTXY Transfer At pointer to X/Y LBH/LBU Long branch if high/low, samt SK Sign-extend 8 → 16 LBG7/LBLE Long branch if high/low, samt SK Sign-extend 8 → 16 LBRA/LBNE Long branch if high/low, samt ADCA/ADDB Add B to A Add B to A ABA Add B to A Add B to A ADCA/ADDB Add to A/B ADDA Nuth (arry to A/B ADDA/ADDB Subract accumulators SBA Subract accumulators SBA			-					
$\begin{array}{c c c c c c c c c c c c c c c c c c c $	PSHD		DBNE					
PULAPULB PULCPull A/B from stack PULCIBNEIncrement, branch if not zero JMPLSRALSRBLogical shift right A/B Logical shift right double DPULDPull double D from stack PULXPULYPull X/Y from stackIBNEJumpLSRDLogical shift right A/B Logical shift right A/BPULXPULYPull X/Y from stack TAB/TBATransfer A to B/B to A LBEG/LBNELSRLSRDNEGA/NEGBNegate memoryTAB/TBATransfer A to condition codes TFRTransfer condition codes to A LBGT/LBLTLog branch if arro for zero Log branch if fsetor/not zero Log branch if fsetor/less than LBGT/LBLTORA/ORABLogical-OR A/B ORA/ORABNegate A/BTSXTSYTransfer stack pointer to X/Y TXS/TXYTransfer stack pointer to X/Y LBHS/LBLSLog branch if fight/low, setorah if plus/minusRORRotate left A/B RORRotate left A/BXGDXXGDYExchange registersLBH/LBLOLong branch if fuels/minusTSTTEst memoryXGDXXGDYExchange registersLBV/LBNLong branch if oranch if zero/not zero RTIRORRotate left A/BABAAdd B to A ABZ/ADDBAdd B to A Add to A/BAdd B to A Add to A/BNicellaneousSET/BCL Set/clear carry SEC/CLCSet/clear carry SEC/CLCSBCANSDCBSubract A/BINCA/INCBIncrement A/B INCA/INCBSEV/CLYSet/clear overflow SEC/CLSBCANSDCBSubract A/BDESDecrement A/B INCA/INCBSITSEC/CLBSUBDSubract A/BDEXDESDecrement A/B INCA/I								
PULCPull condition codes from stackJMPJumpJumpPULXPull double D from stackJMRJJRJJRPULX/PULYPull double D from stackJSRJJRJump to subroutineTAB/TBATransfer A to B/B to ALBSC/LBCLong branch if carry set/clearORA/ORABLogical-OR A/BTAPTransfer register to registerLBSC/LBNELong branch if fess/greater, eq.NEGNEGA/NEGBORCCLogical-OR A/BTXS/TSYTransfer register to registerLBGF/LBLELong branch if fess/greater, eq.ROLRotate left A/BORCASGDX/SOYExchange double D with X/YEscAluserLBR/LBLSLong branch if high/low, sameRORRotate right memoryArithmeticExchange registersBRA/LBRNLong branch if overflw set/clearTSTTest memoryABX/ABYAdd B to AAdd b to X/YIncrement A/BTicrement A/BNiscellaneousADDA/ADDBAdd to A/BINCIncrement A/BSEt/CLCSet/clear carrySBCA/SBC8Subtract A/BDECDecrement A/BTBL/ETBLTable lookup and interpolateSUBDSubract A/BDESDecrement A/BTBL/ETBLTable lookup and interpolateSUBDSubract A/BMAXAMMaximum to ATRAPTable lookup and interpolateSUBDMultiply 16 × 16MAXAMaximum to ATRAPTest holesSUBDMultiply 16 × 16MAXAMaximum to ATRAPUnimplemented opcode trapMVLMult			IBNE		-			
PULD PULD PULL PULX/PULY PUIX/Y from stack PUX.PULY PUIX/Y from stack PUX.PULY PUIX/Y from stack PUX.PULY PUIX/Y from stack TAB/TBA Transfer A to B/B to A TAB/TBA Transfer A to B/B to A TAB/TBA Transfer a to B/B to A TRASTER To condition codes to A TSX/TSY Transfer stack pointer to X/Y TXS/TXY Transfer stack pointer to X/Y TXS/TXY TRASTER stack pointer to X/Y TXS/TXY TRASTER stack pointer to X/Y AGB to A ABX/ABY ADDA/ADDB Add to A/B ADDDA ADDD ADDD ADDD Add double D SUBA/SUBB Subract A/B SUBA/SUBB Subract A/B 								
PULX/PULY TAB/TBAPull X/Y from stack Transfer A to B/B to A Tarnsfer A to condition codesRTS A LBC/LBKReturn from suboutine Long branch if carry set/clear LBC/LBKNEGA/NEGBNEgate A/B LORA/ORABTAP Transfer A to condition codesLBCS/LBCCLong branch if carry set/clear LBC/LBKORA/ORABLOgical-OR A/B ORA/ORABTFR Transfer register to register TPA Transfer stack pointer to X/YTransfer atack pointer to X/Y LBH/LENILong branch if igreater/less than por anch if high/low LBH/LENIROR Rotate left A/BRotate left memory ROLA/ROLBSKDSVXGDY Exchange double D DDD ADDA/ADDBAdd b to A A Add obuble DNetwork LBPL/LBMILong branch if plus/minus Test and branch if zero/not zero RTINEGA/NOLBNotate left A/B RORA/RORBABA ADD/AADDBAdd b to A/Y Add double DArithmetic (contt)Network LBR/LRMINetwork Long branch if zero/not zero RTINEGA/RORBNetate left A/B RORA/RORBADD/AADDB Add double DAdd double DNC Increment A/B INS Increment X/YNetwork Increment A/B Increment X/YSet/Clear overflow SEV/CLVSUBD Subract A/B SUBA/SUBB Subract A/BSubract A/B Subract A/BNetwork A/B DEC Decrement A/B DECDecrement A/B Decrement X/YStop processing TA/BMUL MUL MUL Multiply 16 × 16 EMULSMultiply 16 × 16, signedMAXM Maximum to A MAXMMaximum to A Maximum to D, 16-bitStop processing TRAPFWULS DIVIde 32 + 16, signedMiNM EMAXMMaximum to memory, 1	PULD	Pull double D from stack	JSR					
TAB/TBA TAPTransfer A to B/B to A TAPLogSLECC Transfer A to condition codesLog Scal-OR A/B Logical-OR A/B ORCCLogical-OR A/B Logical-OR A/B ORCCLogical-OR A/B Logical-OR A/B ORCCTFR Transfer condition codes to A TSX/TSYTransfer condition codes to A TsX/TSYLBGE/LBLE Transfer X/Y to stack pointer to X/Y Transfer X/Y to stack pointer to X/Y LBHX/LBLLog branch if greater/less than LBGF/LBLT LDG branch if high/low, same por and if high/low, same Decompt and if plas/InvROR A/ORAB ROLAROLBRotate left A/B ROR Rotate left A/BXGDX/XGDYExchange double D with X/Y EXGEBF/LBLN LBRA/LBRN LBRA/LBRN LBRA/LBRN LDG branch if plas/InvNORRotate left A/B RORARORB RORARORB Rotate left A/BABA ADCA/ADCBAdd B to A ABX/ABY ADCA/ADCBAdd B to A Add b to A/A BDDA Add to A/BNOCIncrement A/B Increment A/BMiscellaneousADDA/ADDB Add double DINS Subract with carry fom A/B SUBA/SUBBINCIncrement A/B INC/INCBSEV/CLVSet/clear otering SEV/CLVSBA SUBA SUBA SUBASSubract A/B Subract A/BDEC Decrement A/B DEC/DECPDecrement A/B Decrement A/BClear A/B SUB/DECSUBD SUBA SUBASSubract A/B Subract A/BDEC/DECP MAXA MAXimum to A MAXA MAximum to AStoP Adaimum to A MAXimum to A MAXimum to A MAXimum to A MAXimum to D, 16-bitORCA DECFUV DIvide 32 + 16Divide 32 + 16 EMAXDMiNM Maximum to D, 16-bitFuzzy LogicFUNC DIvide 32 + 16EMAXD <td>PULX/PULY</td> <td>Pull X/Y from stack</td> <td></td> <td></td> <td>NEGA/NEGB</td> <td></td>	PULX/PULY	Pull X/Y from stack			NEGA/NEGB			
TAPTransfer A to condition codesLBEQ/LBNELong branch if zero/not zeroORCCLogical-OR condition codes toTFRTransfer register to registerLBGE/LBLELong branch if ges/greater, eq.ROLRotate left memoryTXX/TSYTransfer X/Y to stack pointer to X/YLBGT/LBLTLong branch if high/low, sameRORRotate left A/BSEXSign-extend 8 > 16LBH/LBLOLong branch if high/low, sameRORARORBRotate right A/BSEXSign-extend 8 > 16LBRA/LBRNLong branch if pign/low, sameRORARORBRotate right A/BSEXSign-extend 8 > 16LBRA/LBRNLong branch if pign/low, sameRORARORBRotate right A/BABX/ABYAdd B to ATest and branch if zero/not zeroRTIReturn from interruptSET/BCLRSet/clear atryADDA/ADDBAdd with carry to A/BIncrement A/BIncrement A/BSEV/CLVSet/clear atryADDA/ADDBAdd to A/BINCIncrement A/BSEV/CLVSet/clear atrySBASubract double DINSIncrement A/BSEV/CLVSet/clear overflowBLA/SUBBSubract double DDECDecrement A/BSUVSUVSoftware interruptBAADecimal adjust ADEC/DECBDecrement A/BSUVSUP Stop processingBUBLSMultiply 16 × 16, signedMAXAMaximum to ASTOPStop processingFMULMultiply 16 × 16, signedMAXAMaximum to ANOPNO eperationFMUSDivide 32 + 16SMXMMa	ТАВ/ТВА	Transfer A to B/B to A	LBCS/LBCC	Long branch if carry set/clear	ORAA/ORAB			
TFR TPATransfer register to register TPALBGE/LBLE LGGT/LBLTLong branch if less/greater, eq. LGGT/LBLTROLRotate left memory Rotate left A/BTXS/TSYTransfer condition codes to A TXS/TXYTransfer stack pointer to X/Y Transfer X/Y to stack pointer SEXLBGE/LBLE LGGT/LBLTLong branch if high/low, same LBHS/LBLSRORRotate right A/B Rotate right A/BSEXSign-extend 8 → 16 EXGDX/XGDYExchange double D with X/Y EXGDX/XGDYLBHS/LBLS EXchange registersLOng branch if plus/minus LBP/LBNI LBRA/LBRN LBRA/LBRN LBRA/LBRN LBV/LBNCLong branch if overflw set/clear Test and branch if zero/not zero RTIRORRotate right A/B RORA/ROBArithmeticTBEQ/TBNE TBC/TSTTest and branch if zero/not zero RTIBSET/BCLRSet/clear carryABA ADDA/ADDB Add to A/BAdd to A/B ADDA/ADDBNCA/INCB NCAINCB SUBAC subract with carry for A/B SUBAC/SUBBIncrement A/B INCAINCBSEV/CLVSet/clear overflow SEC/LCSUBA SUBACSubract A/B Subract A/BDEC DEC/DECBDecrement A/B DEC/DECBStoftware interrupt DEcrement X/YStoftware interrupt WAX MAximum to A MAXMMaximum to A Minimum to A MINAStop processing TRAPMUL MUL MULiply-16 × 16 EMAXDMiNA Minimum to D, 16-bitMEMDetermine fuzzy membershipEMULS DIVide 32 + 16EMAXDMaximum to memory MINATBLFuzzy Logic	TAP	Transfer A to condition codes						
TPATransfer condition codes to ALBGT/LBLTLong branch if greater/less thanROLA/ROLBRotate left A/BTSX/TSYTransfer stack pointer to X/YLBHS/LBLSLong branch if high/lowRORRotate right memoryXS/TXYTransfer stack pointerSign-extend 8 → 16LBH/LBLOLong branch if high/low, sameRORRotate right A/BXGDX/XGDYExchange double D with X/YExchange registersLBH/LBLMLong branch if plus/minusROR/ARORBRotate right A/BArithmeticTBEO/TBNETest and branch if zero/not zeroTest and branch if zero/not zeroRTISEC/ALCSet/clear bits in memoryABAAdd B to AABAAdd B to X/YAithmetic (con't)Set/clear carrySEC/CLSet/clear carryADDA/ADDBAdd to A/BINCIncrement memoryINCA/INCBIncrement A/BADDA/ADDBAdd double DINSIncrement A/BCLRClear emorySUBA/SUBBSubract A/BDECDecrement A/KCLRClear emorySUBA/SUBBSubract A/BDECA/DECBDecrement A/KSUAMULMultiply 16 × 16, signedMAXAMaximum to memorySTOPStop processingEMULSMultiply 16 × 16, signedMINAMinimum to memory, 16-bitFuzzy LogicEDIVDivide 32 + 16EMAXMMaximum to memory, 16-bitMEM	TFR	Transfer register to register	LBGE/LBLE		ROL			
TXS/TXY SEX Sign-extend $8 \rightarrow 16$ Sign-extend $8 \rightarrow 16$ LBHS/LBLS LGRLBHS/LBLS LOng branch if plus/minus Long branch if plus/minus LBRA/LBNI LOng branch if plus/minus LBRA/LBNI LBNI LBRA/LBNI LBNI LBRA/LBNI LBRA/LBNI LBRA/LBNI LBRA/LBNI LBRA/LBNI LBRA/LBNI LBRA/LBNI LBRA/LBNI LBRA/LBNI <b< td=""><td>TPA</td><td></td><td>LBGT/LBLT</td><td></td><td>ROLA/ROLB</td><td>,</td></b<>	TPA		LBGT/LBLT		ROLA/ROLB	,		
TXS/TXY SEX XGDX/XGDYTransfer X/Y to stack pointer Sign-extend $8 \rightarrow 16$ XGDX/XGDYLBHS/LBLS Exchange double D with X/Y EXGLBHS/LBLS LBHS/LBN LBRA/LBNN LBNN LBRA/LBNN LBNN LBNN LBNN LBNN LBRA/LBNN LBNN	TSX/TSY	Transfer stack pointer to X/Y	LBHI/LBLO	Long branch if high/low	ROR	Rotate right memory		
SEXSign-extend $8 \rightarrow 16$ Exchange double D with X/Y EXGLBPL/LBMI ERALABRNLong branch if plus/minus Long branch if poerflw set/clear Test and branch if zero/not zero RTITSTTest memory Test A/BArithmeticTSTTest A/BABAAdd B to A ABX/ABYAdd B to X/YAdd B to X/YArithmetic (con't)BSET/BCLRSet/clear carry SEC/CLCSet/clear carry overflow SEC/CLCADDA/ADDBAdd to A/B Add double DINCIncrement memory INCA/INCBIncrement A/B DECBit test A/BSUBA/SUBBSubract accumulators SUBA/SUBBSubract A/B DECA/DECBDecrement A/B DECA/DECBTEL/TBLTable lookup and interpolate SUBA/SUBBSUBDSubract A/B DEX/DEYDecrement X/Y DECA/DECBDecrement X/Y Decrement X/YMaximum to A Maximum to A MAXAMaximum to A Maximum to A MAXAMaximum to A Maximum to A MAXAMaximum to A Maximum to A Minimum to A MINAMinimum to memory Minimum to A Maximum to D, 16-bitStudy Curve EMAXMMULDivide $32 + 16$ EMAXMMaximum to memory, 16-bitMeMDetermine fuzzy membership	TXS/TXY	Transfer X/Y to stack pointer	LBHS/LBLS	Long branch if high/low, same	RORA/RORB			
XGDX/XGDY EXchange double D with X/Y EXGLBRA/LBRN EXG/TERC/TBEQ/TBNELong branch always/never LBV/S/LBVC TBEQ/TBNETSTA/TSTBTest A/BArithmeticABA Add B to A ABX/ABYAdd B to A Add b to X/YLBRA/LBRN LBV/S/LBVC TBEQ/TBNELong branch always/never Long branch if overflw set/clear TBE and branch if overflw set/clear Return from interruptTSTA/TSTBTest A/BABA ABX/ABY Add B to X/YAdd B to A ABX/ADDB Add to A/B ADDA/ADDB Add to A/BArithmetic (con't)BSET/BCLR Set/clear bits in memory SEC/CLCSet/clear arry SEC/CLVADDA/ADDB ADDA/ADDB Add double DAdd double D INSIncrement memory INX/INYIncrement A/B Increment X/YBITA/BITB Bit test A/BSUBA/SUBB SUBA/SUBB SUBAA DAA Decimal adjust A MUL MULL MULL MULL MULL MULL MULL MULL MULL MULL MULL MULL MULL MULL MULL MULL MULL MULtiply 16 × 16 EMACS MULL MULtiply-add 16 × 16 · 322 EDIVSDecomal adjust A MAXM MAXM MAXM MAXM Maximum to memory MAXM Maximum to p, 16-bitTRAP Maximum to memory TRAPTable lookup and interpolate SVMI Stop processing TRAPMINA Minimum to memory EMACS EDIVSDivide $32 + 16$, signedMAXM EMAXM Maximum to memory, 16-bitTest A/BMINA Maximum to memory, 16-bitDetermine fuzzy membership	SEX	Sign-extend $8 \rightarrow 16$	LBPL/LBMI		TST			
EXGExchange registersLBVS/LBVCLong branch if overflw set/clear Test and branch if zero/not zero Return from interruptMiscellaneousABAAdd B to A ABX/ABYAdd B to X/YReturn from interruptBSET/BCLRSet/clear bits in memory SEC/CLCADCA/ADCBAdd with carry to A/B ADDA/ADDBAdd to A/BIncrement memory INCA/INCBIncrement A/B Increment X/YSEV/CLVSet/clear overflow SEV/CLVBBASubract accumulatorsINSIncrement A/B INSBITA/BITBBit test A/B CLRSUBA/SUBBSubract A/B SUBA/SUBBDECDecrement A/B DECCLRClear A/B SUBA/SUBBSUBDSubract A/B DECDECDecrement A/B DECA/DECBSUISoftware interruptMULMultiply 8 × 8 \rightarrow 16 EMULSMAXMMaximum to A Minimum to A Minimum to A MINAMaximum to memory MINATRAPUnimplemented opcode trap NOPFUUSDivide 32 \div 16 EDIVSDivide 32 \div 16, signedMAXMMaximum to memory, 16-bit EMAXMMeXMeXMeX	XGDX/XGDY	Exchange double D with X/Y	LBRA/LBRN	Long branch always/never	TSTA/TSTB	Test A/B		
ArithmeticTBEQ/TBNE RTITest and branch if zero/not zero Return from interruptMiscentareousABAAdd B to A ABX/ABYAdd B to X/YRtilReturn from interruptBSET/BCLRSet/clear bits in memory SEC/CLCADCA/ADCBAdd with carry to A/B ADDA/ADDBAdd to A/BINCIncrement memory INCA/INCBSet/CLVSet/clear overflow SEV/CLVADDAAdd double DINCIncrement A/B INSIncrement X/YSet/CLRCLRClear memory CLRA/CLRBSBCA/SBCBSubract accumulatorsINSIncrement X/YBITA/BITBBit test A/BSUBA/SUBBSubract A/B DECA/DECBDECDecrement A/B DECA/DECBSWISoftware interruptDAADecimal adjust A MULMultiply $8 \times 8 \rightarrow 16$ EMULMAXAMaximum to A Minimum to A MAXAStop processing MAXMTRAPEMULMultiply 16×16 , signedMINA EMAXDMinimum to D, 16 -bit EMAXDMAXMMaximum to memory Maximum to memoryTRAPEDIVSDivide $32 \div 16$, signedEMAXMMaximum to memory, 16 -bitMEMDetermine fuzzy membership	EXG	Exchange registers	LBVS/LBVC	Long branch if overflw set/clear	Miscellaneous			
ABAAdd B to AAdd B to AAdd B to AAdd B to X/YBSET/BCLRSet/clear bits in memoryABX/ABYAdd B to X/YAdd B to X/YAdd B to X/YSEC/CLCSet/clear carrySEC/CLCADCA/ADCBAdd with carry to A/BINCIncrement memorySEV/CLVSet/clear overflowADDDAdd double DINCIncrement A/BBITA/BITBBit test A/BADDDAdd double DINSIncrement stack pointerCLRClear memorySBASubract accumulatorsINX/INYIncrement X/YCLRA/CLRBClear A/BSUBA/SUBBSubract with carry from A/BDECDecrement A/BSWISoftware interruptSUBDSubract double DDESDecrement A/BSWISoftware interruptDAADecimal adjust ADEX/DEYDecrement X/YBGNDEnter background debugMULMultiply 8 × 8 \rightarrow 16MAXAMaximum to ASTOPStop processingEMULMultiply 16 \times 16, signedMINAMinimum to memoryTRAPUnimplemented opcode trapEMVLSDivide 32 \div 16EMAXMMaximum to D, 16-bitFuzzy LogicEDIVSDivide 32 \div 16, signedEMAXMMaximum to memory, 16-bitMEMDetermine fuzzy membership	Arithmetic		TBEQ/TBNE		Miscellaneous			
ABX/ABYAdd B to X/YArithmetic (cont)SEI/CLISet/clear interrupt maskADCA/ADCBAdd with carry to A/BIncrement memoryIncrement memorySEI/CLISet/clear interrupt maskADDA/ADDBAdd to A/BIncrement A/BIncrement A/BSEV/CLVSet/clear overflowADDDAdd double DINCA/INCBIncrement stack pointerINCA/INCBBITA/BITBBit test A/BSBASubract accumulatorsINX/INYIncrement X/YCLRA/CLRBClear memorySBA/SUBBSubract A/BDECDecrement A/BDEC//DECBCLRA/CLRBClear A/BSUBDSubract double DDESDecrement A/BSWISoftware interruptDAADecimal adjust ADEX/DEYDecrement X/YBGNDEnter background debugMULMultiply $8 \times 8 \rightarrow 16$ MAXAMaximum to ASTOPStop processingEMULMultiply 16×16 MAXMMaximum to memoryTRAPUnimplemented opcode trapEMULSMultiply $-add 16 \times 16 \rightarrow 32$ EMAXDMaximum to D, 16-bitFuzzy LogicEDIVSDivide $32 + 16$ EMAXMMaximum to memory, 16-bitMEMDetermine fuzzy membership			RTI	Return from interrupt		,		
ABX/ABYAdd b to X/YSEV/Cl1Set/Cl21Set/		Arithmetic (con't)		n't)				
ADDA/ADDBAdd to A/BINCA/INCBIncrement A/BBITA/BITBBit test A/BADDDAdd double DINSIncrement stack pointerCLRClear memorySBASubract accumulatorsINX/INYIncrement X/YCLRA/CLRBClear A/BSBCA/SBCBSubtract with carry from A/BDECDecrement memoryTBL/ETBLTable lookup and interpolateSUBA/SUBBSubract A/BDECDecrement stack pointerSWISoftware interruptDAADecimal adjust ADEX/DEYDecrement X/YBGNDEnter background debugMULMultiply 8 × 8 → 16MAXAMaximum to ASTOPStop processingEMULMultiply 16 × 16MAXAMaximum to ANOPNo operationEMULSMultiply-add 16 × 16 → 32EMAXDMaximum to D, 16-bitFuzzy LogicEDIVSDivide 32 ÷ 16EMAXMMaximum to memory, 16-bitMEMDetermine fuzzy membership				•				
ADDDAdd double DINSIncrement stack pointerCLRClear memorySBASubract accumulatorsINSIncrement stack pointerCLRClear A/BSBCA/SBCBSubtract with carry from A/BDECDecrement memoryTBL/ETBLTable lookup and interpolateSUBA/SUBBSubract double DDECDecrement A/BSWISoftware interruptDAADecimal adjust ADESDecrement X/YBGNDEnter background debugMULMultiply $8 \times 8 \rightarrow 16$ MAXAMaximum to ASTOPStop processingEMULMultiply 16×16 MAXMMaximum to memoryTRAPUnimplemented opcode trapEMVLSMultiply $-add 16 \times 16 \rightarrow 32$ MINMMinimum to memoryNOPNo operationEDIVDivide $32 \div 16$ EMAXMMaximum to memory, 16-bitMEMDetermine fuzzy membership								
SBASubract accumulatorsINX/INYIncrement X/YCLRA/CLRBClear A/BSBCA/SBCBSubract with carry from A/BDECDecrement memoryTBL/ETBLTable lookup and interpolateSUBA/SUBBSubract double DDECDecrement A/BSWISoftware interruptDAADecimal adjust ADEX/DEYDecrement X/YBGNDEnter background debugMULMultiply $8 \times 8 \rightarrow 16$ MAXAMaximum to ASTOPStop processingEMULMultiply 16×16 MAXMMaximum to memoryNOPNo operationEMULSMultiply 16×16 MINMMinimum to memoryNOPNo operationEDIVDivide $32 \div 16$ EMAXMMaximum to memory, 16-bitFuzzy LogicEDIVSDivide $32 \div 16$, signedEMAXMMaximum to memory, 16-bitMEMDetermine fuzzy membership								
$\begin{array}{cccccccccccccccccccccccccccccccccccc$					-			
$ \begin{array}{c ccccccccccccccccccccccccccccccccccc$	-							
SUBDSubract double DDESDecrement stack pointerWAIWait for interruptDAADecimal adjust ADEX/DEYDecrement X/YBGNDEnter background debugMULMultiply $8 \times 8 \rightarrow 16$ MAXAMaximum to ASTOPStop processingEMULMultiply 16×16 MAXMMaximum to memoryTRAPUnimplemented opcode trapEMULSMultiply 16×16 , signedMINAMinimum to memoryNOPNo operationEMULSMultiply-add $16 \times 16 \rightarrow 32$ MINMMinimum to D, 16-bitFuzzy LogicEDIVDivide $32 \div 16$ EMAXMMaximum to memory, 16-bitMEMDetermine fuzzy membership			-					
DAADecimal adjust ADEX/DEYDecrement X/YBGNDEnter background debugMULMultiply 8 × 8 → 16MAXAMaximum to ASTOPStop processingEMULMultiply 16 × 16MAXAMaximum to memoryTRAPUnimplemented opcode trapEMULSMultiply 16 × 16, signedMINAMinimum to ANOPNo operationEMACSMultiply-add 16 × 16 → 32MINMMinimum to memoryFuzzy LogicFuzzy LogicEDIVDivide 32 ÷ 16, signedEMAXMMaximum to memory, 16-bitMEMDetermine fuzzy membership								
MULMultiply $8 \times 8 \rightarrow 16$ MAXAMaximum to ASTOPStop processingEMULMultiply 16×16 MAXMMaximum to memoryTRAPUnimplemented opcode trapEMULSMultiply 16×16 , signedMINAMinimum to ANOPNo operationEMACSMultiply-add $16 \times 16 \rightarrow 32$ MINMMinimum to memoryNOPNo operationEDIVDivide $32 \div 16$ EMAXMMaximum to memory, 16-bitFuzzy LogicEDIVSDivide $32 \div 16$, signedEMAXMMaximum to memory, 16-bitMEM				1				
EMULMultiply 16×16 MAXMMaximum to memoryTRAPUnimplemented opcode trapEMULSMultiply 16×16 , signedMINAMinimum to ANOPNo operationEMACSMultiply-add $16 \times 16 \rightarrow 32$ MINAMinimum to memoryFuzzy LogicFuzzy LogicEDIVDivide $32 \div 16$ EMAXMMaximum to memory, 16-bitMEMDetermine fuzzy membership					-			
EMULSMultiply 16×16 , signedMINAMinimum to ANOPNo operationEMACSMultiply-add $16 \times 16 \rightarrow 32$ MINAMinimum to memoryFuzzy LogicEDIVDivide $32 \div 16$ EMAXDMaximum to D, 16-bitMEMDetermine fuzzy membership	-							
EMACSMultiply-add $16 \times 16 \rightarrow 32$ MINMMinimum to memoryFuzzy LogicEDIVDivide $32 \div 16$ EMAXDMaximum to D, 16-bitMEMDetermine fuzzy membershipEDIVSDivide $32 \div 16$, signedEMAXMMaximum to memory, 16-bitMEMDetermine fuzzy membership	-			,				
EDIVDivide 32 ÷ 16EMAXDMaximum to D, 16-bitFuzzy LogicEDIVSDivide 32 ÷ 16, signedEMAXMMaximum to memory, 16-bitMEMDetermine fuzzy membership					NOP	NO operation		
EDIV Divide 32 ÷ 16 EMAXD Maximum to D, 16-bit DETV EDIVS Divide 32 ÷ 16, signed EMAXM Maximum to memory, 16-bit MEM Determine fuzzy membership					Fuzzy Logic			
					, 0	Determine furner and such		
	-							
	IDIV	Divide 16 ÷ 16	EMIND	Minimum to D, 16-bit	REV	Rule evaluation		
IDIVS Divide 16 ÷ 16, signed EMINM Minimum to memory, 16-bit REVW Rule evaluation, weighted	-		EMINM	ivinimum to memory, 16-bit		, 0		
FDIV Divide 16 ÷ 16, remainder 16 WAV Weighted average		Divide to ÷ to, ternalider 16			VVAV	vveignieu average		

Table 1. The 68HC12 includes more than 50 new instructions compared with the 68HC11. New instructions, highlighted in purple, include new stack operations, extended-precision multiply and divide, minimum and maximum calculations, and new intersegment branches.

Price & Availability

The 68HC812A4 is sampling now for \$19 in a 112lead TQFP package; production quantities will be priced in the \$10-\$15 range. Production is scheduled for 1Q97. The 68HC912B32 will begin sampling in 4Q96. Production pricing is expected to be \$20-\$25.

For more information, contact Motorola (Austin, Texas) at 800.765.7795; fax 512.891.4465; or browse to freeware.aus.sps.mot.com/amcu/home.html.

code from the HC11 will not run on the HC12, although about half the HC12's mnemonics produce the same encodings as on the HC11. The reworked encoding means some instructions will be shorter than their HC11 counterparts and some will be longer.

Motorola's tests indicate that simply reassembling HC11 source code for the HC12 has a negligible effect on object size. On the other hand, rewriting assembly code or passing C source through a new compiler can yield code-size reductions of as much as 30%, according to the company.

Users should also see a performance improvement along with tighter code. Several HC12 instructions execute in a single clock cycle, whereas the HC11 always takes at least two. Arithmetic instructions, in particular, have been improved, as Table 2 shows.

Fuzzy Logic Support a First

Four truly new instructions have been added to provide support for fuzzy-logic control applications. These complex instructions take from a few cycles to a few hundred cycles to execute and were made interruptible to avoid compromising worst-case latency.

The MEM instruction computes a *Y* value, given an *X* value as the index. Its operation goes beyond a simple linear table interpolation (which the HC12 can also perform) in that it operates on a trapezoidal membership function. That

		68HC11		68HC12	
Mnemonic	Math Operations	cycles	time	cycles	time
MUL	$8 \times 8 \rightarrow 16$ (signed)	10	2.5 μs	3	0.375 μs
EMUL	$16 \times 16 \rightarrow 32$ (unsigned)†	20	5 µs	3	0.375 µs
EMULS	$16 \times 16 \rightarrow 32$ (signed)†	20	5 µs	3	0.375 μs
IDIV	$16 \div 16 \rightarrow 16$ (unsigned)	41	10.25 μs	12	1.5 μs
IDIVS	$16 \div 16 \rightarrow 16$ (signed)	—	—	12	1.5 μs
FDIV	$16 \div 16 \rightarrow 16$ (fractional)	41	10.25 μs	12	1.5 μs
EDIV	$32 \div 16 \rightarrow 16$ (unsigned)†	33	8.25 μs	11	1.375 μs
EDIVS	32 ÷ 16 \rightarrow 16 (signed)†	37	9.25 μs	12	1.5 μs
EMACS	$16 \times 16 \rightarrow 32$ (signed)†	20*	5 μs*	12*	1.5 μs*

Table 2. Multiply and divide performance varies considerably between the HC11 and HC12 cores, in part because the HC12 is clocked twice as fast as its predecessor. toperations possible only on HC11 with math coprocessor. *per iteration.

is, for values of X that are under the sloping side of a conceptual trapezoid, Y is interpolated from its slope; for values under the flat side of the trapezoid, Y is capped at 0xFF.

The rule-evaluation instructions REV and REVW process a list of pointers to values, alternately performing minimum and maximum operations to implement min-max fuzzy-rule evaluation. The REVW variation allows 8-bit weighting factors to be applied to each rule.

The WAV instruction is basically an 8-bit multiplyaccumulate instruction, useful for low-precision pseudo-DSP operations but also as the basis for a weighted-average defuzzification. (*For background on fuzzy-logic programming, see* 061702.PDF)

The New Bus: Same as the Old Bus

The HC12 copies its predecessor's straightforward synchronous bus interface rather than following the asynchronous example of the HC16, 68300, and early 68000 processors. HC11 customers are used to simple peripheral logic with fixed and predictable (albeit sluggish) response times measured in multiples of a 4-MHz bus clock. No external logic is required (or able) to insert wait cycles or extend access times. Instead, chip-select logic in the HC12 can be programmed with the desired access time for each external device. We expect Motorola to expand the HC12 family in this way.

Internally, the HC12 uses the same intermodule bus (IMB) design philosophy that Motorola uses on its HC16 and most of the 68300-family devices. In fact, many HC16 and 68300 chips can and do use identical peripheral blocks, a fact that allows Motorola to spin application-specific variations of these two families on short notice.

The first two 68HC12 devices, the 68HC812A4 and 68HC912B32, both run at a comparatively speedy 8 MHz, twice as fast as the quickest HC11 part, boosting performance further. Their 16-bit buses also allow them to fetch code and access the stack in half the time.

With variable-length instructions, misalignment can be a problem on a 16-bit bus—something HC11 users never had to worry about. The HC12 design includes three 16-bit instruction buffers with byte-steering logic. The buffers allow the HC12 to fetch a 16-bit word on every cycle, ignoring instruction alignment. With the instruction in the buffer, decoding logic determines the instruction boundaries and aligns the instruction accordingly before execution. Although not as advanced as a cache, the pair of buffers helps keep the HC12's simple pipeline full.

Paging Expands Address Space

Expanding the tiny 64K address space of the HC11 proved to be tricky without compromising software compatibility or enlarging registers. The HC12's designers used a paging approach that stretches the chip's reach to 4M and is transparent to ported code unaware of the HC12's enhancements.

As Figure 2 shows, one-quarter of the HC12's address map, from 0x8000 through 0xBFFF, is a window onto addi-

tional external memory. Any access to this 16K space causes the contents of an 8-bit page register to be driven on the chip's high-order address lines. The page register selects one of 256 possible 16K pages. To software, the page register itself appears as another control register in the chip's memorymapped on-chip peripheral space.

The page register is literally just gated onto the address bus; its contents are not available to the address-generation logic or to the ALU. A pair of new flow-control instructions, CALL and RTC, allow HC12 programs to jump between pages. The CALL instruction specifies a 24-bit target address, the upper eight bits of which are copied into the paging register after pushing the previous page value onto the stack. The RTC instruction restores the page address from the stack.

As simple as it seems, the HC12's method of code and data paging is more elegant than most. Other 8-bit and 16bit controllers—such as the 8051, 68HC05, and 80251 require at least two instructions to change the page and jump to the target address, a construct that necessitates disabling interrupts, lest the chip get interrupted at an inopportune moment and vector to the wrong page.

An atomic page-switching instruction can switch pages while executing out of paged memory. In other systems, the paging code must be located in memory that cannot be paged. Otherwise, incautious programmers can wind up pulling the rug out from under themselves.

Two additional paging windows, located at 0x0000 and 0x7000, expand the data space even further. Like the 16K page at 0x8000, these 1K and 4K data pages each have their own paging register. Because they are not linked to flow-control instructions, they would normally be used only for accessing data.

Motorola Plays Low-Power Card

One distinction between the new HC12 parts and the faster HC16 devices is the former's lower power consumption. Whereas the original HC16 chips were 5-V–only parts, the A4 and B32 are designed to tolerate a wide supply range, from 3 to 5 V. A specified 10% tolerance brings the viable range from a low of only 2.7 V to a high of 5.5 V, covering the sweet spot for most two-cell battery technologies.

Typically, a broad supply tolerance forces significant speed derating at lower voltages, but both the A4 and the B32 run at 8 MHz across their entire supply range. This flat voltage rolloff suggests that these chips may have some substantial clock-speed headroom near 5 V. Although Motorola won't officially sanction operation beyond the published limit, the company hints that users might be able to discover a performance upside without too much effort.

The chips are built in a 0.65-micron two-layer-metal CMOS process, using Motorola's UDR (universal design rules) geometry, a process optimized for low power dissipation rather than for high speed. Still, the HC12's frequency characteristics and the fact that Motorola's PowerPC 602 runs at 66 MHz in a similar process make it appear that the

performance of the HC12 is being artificially limited to avoid competing with the faster 16-bit and 32-bit devices.

The HC12 chips offer the familiar sleep and standby modes. On the A4, entering sleep mode cuts power consumption by 75% to 32 mW (typical) while the chip waits for an interrupt to resume operation. In standby mode, all operation ceases, and power drops to just 25 μ W. Peripherals that aren't active have their clocks gated off automatically, and software can disable unused peripherals entirely.

Debug Functions Reduced to a Single Pin

Motorola continues to hone its background debug mode (BDM) feature, a set of on-chip debugging resources that first appeared with the debut of the 68300 family in 1989. As the name might imply, BDM works by placing the chip in a background mode, similar to low-power sleep or standby operation. While BDM is active, a developer can interrogate and modify system resources like CPU registers, memory-mapped I/O registers, internal EEPROM, or external memory via simple commands from an emulator or development system. On the HC12, the BDM interface has been reduced to a single pin.

Taking advantage of the BDM functions is simply a matter of serially transferring an 8-bit command word over the chip's BDM pin. Data may be clocked in asynchronously at conveniently arbitrary rates, up to a maximum of 500 kHz. The transfer protocol is simple enough that the emulator software can control it by toggling an external I/O line connected to the BDM pin.

The BDM features access memory-mapped resources while the processor is running at full speed. Both on-chip and off-chip memory can be examined and modified without altering the system's execution profile. The HC12 delays BDM memory accesses until the bus is unused, then steals a memory cycle. In the pathological case where the bus is always busy, the chip will wait for up to 128 cycles (16 microseconds) before it seizes the bus to access memory.

In addition to the basic BDM functions, the B32 version includes a set of internal breakpoint registers. Loading 16-bit address or data

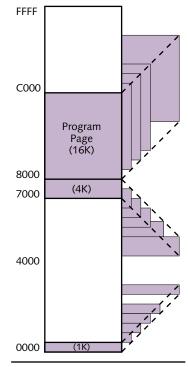


Figure 2. The HC12's memory map includes three segmented windows for data, one of which can also be used to expand the instruction space.

values into these registers enables the hardware breakpoints, which can force an exception on selective patterns of address, data, or cycle type.

New Options for Controller Users

Prices for the HC12 chips will fall below those of many members of the HC16 family. For 8-bit and 16-bit microcontrollers, price is determined largely by a chip's peripherals or on-chip memory, not its CPU core, which accounts for only a small portion of the die. High-end HC11 and low-end HC16 devices already overlap in price; placing the new HC12 parts in the middle will further muddy the price differential between these families.

The HC11 family is currently ranked third in worldwide unit shipments for 8-bit microcontrollers (behind the 68HC05 and 8051), according to published reports. With that kind of volume, it's logical for Motorola to do everything it can to protect is lucrative customer base.

Before the HC12, customers loyal to Motorola but looking for an upgrade for their 8-bit controller were steered toward the HC16, an architecture with a completely different programming model and hardware interface. Although most software tools for the HC16 accept HC11 assembly syntax, the translation is not straightforward, and the hardware differences are impossible to mask. Both Intel and Philips lured their 8051 customers with easy 16-bit upgrades; Intel's is binary compatible, while Philips followed a looser definition and created a chip that maintained only source-level compatibility and overhauled the interior design. Both companies are now successfully introducing 8-bit customers to the joys of 16-bit computing.

In addition to HC11 upgrades, the prospects for the HC12 look good. Judging from the first two chips, Motorola is offering competitive performance for a reasonable price. The company's modular design philosophy, tried and proven with the HC16 and 68300, will allow it to duck and swerve as changes in the market drive new peripheral requirements. By sacrificing binary compatibility, the HC12 is able to achieve good code density even with an entirely new instruction set.

A few years ago, the conventional wisdom held that 8-bit users would skip over the 16-bit generation and move directly to 32-bit CPUs. The rumors of the demise of 16-bit microprocessors have been greatly exaggerated; in any field as costcompetitive as microcontrollers, every penny is worth saving. Thus, today's 16-bit controllers are going strong. When a design like the HC12 can offer users significant and tangible advantages in code compatibility, improved performance, better code density, and reduced power consumption, it should have no trouble making its way in the thick of the steadily growing 16-bit market.