Record Format
- all records have the same format:
  Record = (Label) + ([Instruction I Directive]) + (Operand) + (; Comment)
  Label = Alphabetic + 0 { Alphanumeric } 32
  Alphabetic = [A..Z I a..z I _]
  Alphanumeric = [A..Z I a..z I 0..9 I _]
- instructions and directives are case insensitive
- label is case sensitive

Labels
- text string of up to 32 characters, each must begin with an alphabetic character
- valid labels are stored in the symbol table w/ either the value of the location counter or the value associated with the equate directive
- duplicate labels are not permitted

Directives
ALIGN
- increments location counter to next even-byte address if location counter is odd

AORG Operand {absolute origin}
- functions as the ORG directive does, changing the value of the location counter to the address specified in the operand
- requires the linker to resolve the address

ASCII Operand
- operand is a character string enclosed in double quotes
- the characters in the string are converted into their binary equivalent & stored in contiguous locations
- directive only accepts 1 string, escaped characters such as tab & nul are supported

BSS Operand {block started by symbol}
- reserves a block of memory of operand bytes
- if label associated it is stored in the symbol table w/ the value of the location counter
- location counter is increased by specified number of bytes, label can be omitted

BYTE Operand
- 1 byte is stored in the memory location associated w/ location counter
- operand larger than 8-bits in length is an error
- if label, it is stored in the symbol table along w/ location counter
- location counter increased by 1
- any byte value that exceeds its range is truncated to the least significant byte (2 nibbles)

CODE
- output from assembler is written to S1 records
- ASCII, BSS, BYTE, and WORD directives ignore CODE directive
- AORG and ORG change the location counter to refer to code memory

DATA
- output from assembler is written to S2 records
- all instructions change the assemblers output back to S1 records
- AORG and ORG change location counter to refer to data memory

END (Operand)
- denotes the end of the program, any records that follow the END record are ignored
- if operand is supplied, it must refer to a label in the symbol table or an actual address, this is the starting address used by the loader

EQU Operand {equate}
- EQU records label is equated with the operand, label and value of operand are stored in the symbol table
- label is required, location counter is not incremented

ORG Operand {origin}
- changes current location counter value to the address specified in operand

WORD Operand
- 2 bytes are stored in consecutive memory locations, starting at the location specified by the current value of the location counter
- if there is a label it is stored in the symbol table along with the location counter
- location counter increased by 2 bytes
- any word value that exceeds its range is truncated to the least significant 2 bytes (4 nibbles)
- Note: 16-bit quantities should fall on even-byte boundaries, ALIGN can ensure this

Operands
- operand contains up to 3 values, separated by commas (no leading or following spaces)
  Operand = Value + 0 { "," + Operand } 3
- value is either a numeric value or a label
  Value = [Numeric I Label I String]
- numeric and label values are distinguished using a prefix ($, #, ', denote a numeric value)
  Numeric = [" $ " + [Unsigned I Signed] I " ' " + Char I " # " + Hex]
  Unsigned = [0 .. 65535]
  Signed = [-32768 .. +0 .. +65535]
  Char = [Alphanumeric I Escaped] + " ' "
  Hex = 1 { 0..9 I A..F I a..f } *Hex values range from #0 to #FFFF*
  Escaped = "\" + Alphanumeric
  String = 1 { Char } 128
- the escaped alphanumeric value is limited to the ASCII-escape sequences below

Character	Converted Valued	Meaning
'\b'	#08	BS - Backspace
'\t\	#09	TAB
'\n'	#0a	Linefeed, Newline
'\r'	#0d	Carriage return
'\0'	#00	NUL
'\\'	#5c	Backslash
'\''	#27	Single quote
'\"'	#22	Double quote
'\Unknown'	#3f('?')	Invalid/unacceptable character

Notes
- location counter is incremented by the number of bytes associated with the ALIGN, BSS, BYTE, ORG, or WORD directive
- directives & instructions are reserved words & cannot be used as labels
- characters begin & end w/ the single quote character
- unsigned values can be signed with the "+" sign (that is, -32768 to +65535)

S-Records
S0: header record containing the name of the .asm file
S1: contains bytes to be written to IMEM
S2: contains bytes to be written to DMEM
S9: starting address of the IMEM
Record = 'S' + type + length + address + data + checksum
  type = [0 I 1 I 2 I 9]
  length = hexadecimal int : remaining data pairs in the record
  address = low-byte + high-byte
  data = byte pairs to be stored in mem
  checksum: length + address + data + checksum = -1 to be a valid record

Terms / Symbols
Bit - binary value
Byte = 8 bits
Word = 2 bytes = 16 bits
Unit - byte or word
  - bits in a unit are numbered from right-to-left, starting at 0
LSB - right-most bit, if the value is 0 the unit is even
MSB - left-most bit
- hexadecimal number (cannot be signed)
$ - signed or unsigned integer
arrow - assignment
= - equality
PSW - program status word
subtraction: result = minuend - subtrahend

Abbreviations

instr - instruction	mem - memory	reg - register
addrs - address	lctn - location	dst - destination

Extra Notes
- get one's complement by inverting the bits, denoted by ~
- two's complement is one's complement + 1
- binary values: 0 = 0000, 1 = 0001, 2 = 0010, 3 = 0011, 4 = 0100, 5 = 0101, 6 = 0110, 7 = 0111, 8 = 1000, 9 = 1001, 10 = 1010, 11 = 1011, 12 = 1100, 13 = 1101, 14 = 1110, 15 = 1111
- hexadecimal values: 10 = A, 11 = B, 12 = C, 13 = D, 14 = E, 15 = F
- ASCII values: space = 32, ! = 33, " = 34, # = 35, $ = 36, % = 37, & = 38, ' = 39, (= 40,) = 41, * = 42, + = 43, , = 44, - = 45, . = 46, / = 47, 0-9 = 48-57, : = 58, ; = 59, < = 60, = = 61, > = 62, ? = 63, @ = 64, A-Z = 65-90, [= 91, \ = 92,] = 93, ^ = 94, _ = 95, ` = 96, a-z = 97-122, { = 123, I = 124, } = 125, ~ = 126, delete = 127

Central Processing Unit (CPU)
Components
3 memory access registers (all use the system bus connecting the CPU to mem)
- a bi-directional, 16-bit mem data reg (MDR) for reading instr & data from, & writing data to, mem
- a unidirectional 16-bit mem addrs reg (MAR) that carries an addrs specifying a mem lctn to be read or written
- a control reg indicating whether the action is a read or write

16-bit CPU Data bus
- carrying instr from MDR to instr reg; data from MDR to a reg in the reg file or to the MDR from a reg in the reg file; or the output of the ALU to a reg in the reg file

16-bit CPU Address bus
- carrying an addrs from a reg in the reg file or output from the ALU to the MAR

Instruction Register (IR)
- takes a 16-bit value, assumed to be an instr, fetched from memory, for decoding by the instr decoder
- IR is inaccessible to the programmer

Instruction Decoder
- responsible for taking the value store in the IR & decoding it into its operand & any operands associated with it
- if CPU is in the conditional-execution state, instr can be fetched but not decoded

Register File (containing R0-R7)
- R0-R4: general purpose, R5: Link register (LR), R6: Stack pointer (SP), R7: Program counter (PC)
- each reg is 16-bit wide & can be accessed as a 16-bit word or 8-bit high or 8-bit low

Arithmetic & Logic Unit (ALU)
- performs arithmetic & logic operations using reg contents

Control Unit
- orchestrates the entire operation of the CPU, signalling each component when it is to perform its designated task as part of each instruction cycle
- 3 phases: fetch, decode, execution

Memory
Byte Organization
- 8 bits long (a signed or unsigned char)
- when accessing data as bytes, the addrs range is #0000 through #FFFF
- bytes can fall on odd or even addrs

Word Organization
- 16 bit quantity (equivalent to a signed or unsigned short), spanning 2 bytes
- words must start on even byte boundaries
- a word with address #0001 refers to bytes #0002 & #0003 {the starting byte of any word is simply the word addrs shifted left by 1
- instr & 16-bit integers are stored as words

Byte Ordering
- word is stored as 2 bytes, MSB (high order, bits 15 through 8) & LSB (low order, bits 7 through 0)
- little-endian: LSB-then-MSB
- big-endian: MSB-LSB

Figure 5: Storing #1234 in two different endian structures

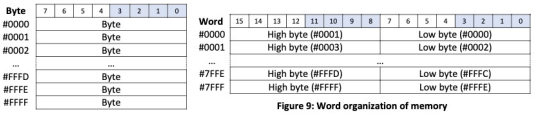

Figure 8: Byte organization of memory

Figure 9: Word organization of memory

Device-Register Memory
- supports 8 devices, each associated w/ a 1 byte control/status reg (low byte) & a 1 byte data reg (high byte) in the first 8 words of memory

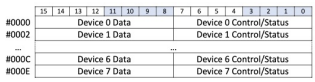

Figure 10: XM-23's device-register memory (#0000-#000F)

Interrupt Vectors
- mem lctns #FFC0 through #FFFA are interrupt vectors
- vectors hold the addrs of exception handlers responsible for dealing w/ exceptions: device interrupts, system-faults, & traps
- final 2 words of high mem (#FFFC & #FFFE) hold the systems restart PSW & addrs of restart code

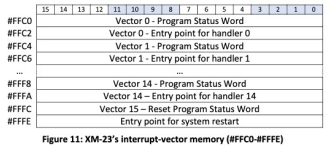

Figure 11: XM-23's interrupt-vector memory (#FFC0-#FFFE)

CPU Registers
General Purpose Registers - R0, R1, R2, R3
- used for addressing or arithmetic & logic instructions
- can hold signed or unsigned quantities
- sign bit is context driven, bit 7 in 8-bit arithmetic, bit 15 in 16-bit arithmetic

Base Pointer (BP) - R4
- general purpose register that can be used as a subroutines base pointer
- base pointer is intended to hold the addrs of the current stack frame during a subroutine call

Link Register (LR) - R5
Can hold one of:
- subroutine calls are made w/ the BL instr, the return addrs is stored in link reg, LSB always clear

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0

- when an interrupt occurs the CPU stores an invalid addrs (#FFFF) in the link reg to indicate that an exception handler is active, used by CPU when returning from the interrupt

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

- if necessary, can be used as a general purpose reg

Stack Pointer (SP) - R6
- points to the value on the current top of stack, a pull reads this value & increments the SP while a push decrements the SP & then writes the value making it the new top of stack
- SP should always refer to an even addrs instr (LSB clear)
- using the SP as a general purpose reg or setting LSB can lead to unpredictable results

Program Counter (PC) - R7
- contains the addrs of the next instr to be executed, instr must fall on even byte boundaries
- moving a value to the PC is equivalent to a JUMP instr, control will pass to specified addrs

Machine State (PSW)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Previous priority				-	-	-	-	FLT	Current priority			V	SLP	N	Z	C

Figure 12: Program Status Word layout

C: carry
- indicates a carry has occurred in either addition, subtraction, or compare
- carry has no meaning in signed arithmetic

Z: zero
- indicates whether the last operation resulted in a zero value (#0000 word, #00 byte)
- Z = 1: result is zero, Z = 0: result is non-zero

N: negative
- sign bit is MSB used to indicate whether the structure is positive (0) or negative (1)

V: overflow
- indicates that the result of an addition or 2 positive or 2 negative numbers produce a change in sign
- functionally equivalent to a carry, condition not met (0), condition met (1)

SLP: sleep state
- indicates whether the CPU is in the sleep (SLP = 1) or execution state (SLP = 0)
- when entering or leaving an interrupt service routine, SLP cleared (0)
- when CPU priority is 7, SLP cannot be set

Current Priority: priority of the currently executing application

FLT: fault state
- indicates whether the CPU has experienced a system fault (FLT = 1) or not (FLT = 0)

Previous Priority: priority of the previously executing application, previous always less than current

Data Dictionary Symbols

Symbol	Name	Meaning
=	Equals	Is composed of, or is defined as
+	Plus	And
( )	Parenthesis	Optional
{ }	Braces	Iteration
[ ]	Brackets	Selection
I	Vertical bar	Choice (used with selection)
-	Hyphen	Through (used with selection)
**	Asterisk	Comment

Instructions

Register Initialization Instructions

Table 2: Register initialization instructions
(LSB – least-significant byte; MSB – most-significant byte)[5]

Instruction	Operation	Description
MOVL Value,DST	*DST.LSB← Value* *DST.MSB unchanged*	Value is assigned to the low-byte of the destination register. The high-byte is unchanged.
MOVLZ Value,DST	*DST.LSB ← Value* *DST.MSB ←#00*	Value is assigned to the low-byte (LSB) of the destination register. The high-byte (MSB) is zeroed.
MOVLS Value,DST	*DST.LSB ← Value* *DST.MSB ← #FF*	Value is assigned to the low-byte (LSB) of the destination register. The high-byte (MSB) is assigned #FF.
MOVH Value,DST	*DST.LSB unchanged* *DST.MSB ← Value*	Value is assigned to the high-byte of the destination register. The low-byte is unchanged.

- dst reg can be any reg (R0 - R7) or an equated value
- value can be any 8-bit quantity recognized by the assembler:
- decimal values: signed values from $-128 through $0 to $127 & unsigned values from $0 to $225
- hexadecimal values: any hexadecimal value between #0 (or #00) to #FF
- characters: any ASCII character enclosed in single quotations

Memory Access

- allow a program to access data mem for loading or storing contents

Direct Addressing

W/B: indicated whether a word (0) or a byte (1) is to be accessed
SRC: source reg, indicates where the data is coming from either a mem lctn (load) or a reg (store)
DST: dst reg, indicates where the data is going either a mem lctn (store) or reg (load)

Table 3: Direct memory addressing

Instruction	Operation	Description
LD(.B or .W) SRC,DST	*EA ← SRC* *DST ← memory [EA]*	Load a register (DST) from memory location specified by the effective address (EA), the value in the SRC register. Reading a byte stores the value in the low-byte of the DST register; the high-byte is unchanged.
ST(.B or .W) SRC,DST	*EA ← DST* *memory [EA] ← SRC*	Store a register (SRC) in memory location specified by the effective address. Writing a byte to the low-byte of a word does not change the word's high-byte.

Indexed Addressing

- effective addrs (EA) obtained from a reg using as an index reg

PRPO: pre- or post- increment or decrement of the reg specifying the mem lctn to access. 0 indicates a post-increment or post-decrement or no action, 1 indicates a pre-increment or pre-decrement action
DEC: decrement the reg (before or after based on PRPO). 0 indicates no decrementing while 1 indicates decrementing
INC: increment the reg (before or after based on PRPO). 0 indicates no incrementing, 1 indicates incrementing
SRC: source reg, indicates where data comes from
DST: dst reg, indicates where the data goes

Table 4: Valid PRPO, DEC, and INC combinations and their meanings
(PRPO, DEC, and INC combinations 011, 100, and 111 are undefined)

Register format	Definition	Effective address (EA) and register value	PRPO	DEC	INC
Rn	Unmodified register (This is direct addressing.)	*EA = Rn* *Access memory [EA]*	0	0	0
+Rn	Pre-increment register	*EA = Rn + 1 (byte) or + 2 (word)* *Access memory [EA]* *Rn = EA*	1	0	1
Rn+	Post-increment register	*EA = Rn* *Access memory [EA]* *Rn = Rn + 1 (byte) or + 2 (word)*	0	0	1
-Rn	Pre-decrement the register	*EA = Rn - 1 (byte) or - 2 (word)* *Access memory [EA]* *Rn = EA*	1	1	0
Rn-	Post-decrement the register	*EA = Rn* *Access memory [EA]* *Rn = Rn - 1 (byte) or - 2 (word)*	0	1	0

Table 5: Load and store register-direct and
register-direct with pre- or post-auto-increment or auto-decrement

Instruction	Operation	Description
LD(.B or .W) +SRC,DST LD(.B or .W) -SRC,DST LD(.B or .W) SRC+,DST LD(.B or .W) SRC-,DST	*if Pre-Incr or Pre-Decr then* *SRC = SRC + address modifiers* *EA = SRC* *DST ← memory[EA]* *if Post-Incr or Post-Decr then* *SRC = SRC + address modifiers*	Load a register (DST) from memory location specified by the effective address (EA), the value in the SRC register and the address-modifier bits. Loading a byte stores the value in the low-byte of the DST register; the high-byte is unchanged.
ST(.B or .W) SRC,+DST ST(.B or .W) SRC,-DST ST(.B or .W) SRC,DST+ ST(.B or .W) SRC,DST-	*if Pre-Incr or Pre-Decr then* *DST = DST + address modifiers* *EA = DST* *memory [EA] ← SRC* *if Post-Incr or Post-Decr then* *DST = DST + address modifiers*	Store a register (SRC) in memory location specified by the effective address. The effective address is obtained from the DST value and the address modifier bits. Writing a byte to the low-byte of a word does not change the word's high-byte.

Relative Addressing

- effective addrs is obtained by adding a base addrs to an offset
- relative addressing is useful when accessing C-like structs and stack frames
- the signed value in the internal reg is added to the SRC or DST reg (src for load, dst for store) to become the effective addrs

word offsets: refer to words w/ a word addrs range relative to the base addrs of -32 through 0 to +31, equivalent to the base addrs + offset ANDed w/ 0xFFFE
byte offsets: refers to bytes on odd or even byte boundaries & have a byte addrs range of -64 through 0 to +63 relative to the base addrs

Table 6: Relative addressing using word or byte offsets
Range of addressable memory: Base address - 64 through Base address + 63

Offset values (bits 13..7)	Sign-extended value (original 7-bits in red)	Offset	Effective address	
			Byte	Word
100.0000	1111.1111.1100.0000	-64	Base_addr - 64	(Base_addr - 64) & 0xFFFE
100.0001	1111.1111.1100.0001	-63	Base_addr - 63	(Base_addr - 63) & 0xFFFE
	. . .			
111.1110	1111.1111.1111.1110	-2	Base_addr - 2	(Base_addr - 2) & 0xFFFE
111.1111	1111.1111.1111.1111	-1	Base_addr - 1	(Base_addr - 1) & 0xFFFE
000.0000	0000.0000.0000.0000	+0	Base_addr + 0	(Base_addr + 0) & 0xFFFE
000.0001	0000.0000.0000.0001	+1	Base_addr + 1	(Base_addr + 1) & 0xFFFE
000.0000	0000.0000.0000.0010	+2	Base_addr + 2	(Base_addr + 2) & 0xFFFE
	. . .			
011.1110	0000.0000.0011.1110	+62	Base_addr + 62	(Base_addr + 62) & 0xFFFE
011.1111	0000.0000.0011.1111	+63	Base_addr + 63	(Base_addr + 63) & 0xFFFE

Table 7: Load and store register-relative instructions[8]

Instruction	Operation	Description
LDR(.B or .W) SRC,OFF,DST	*EA = SRC + sign-extended offset* *DST ← memory [EA]*	Load a register (DST) from memory location specified by the effective address (EA).
STR(.B or .W) SRC,DST,OFF	*EA = DST + sign-extended offset* *memory [EA] ← SRC*	Store a register (SRC) in memory location specified by the effective address.

Table 22: Bit value definitions for XM-23 Instruction Set (Table 25)

0	1	
		Instruction opcode bit values (0 or 1).
PRPO		Pre- or post-increment or pre- or post-decrement (Load and Store).
DEC		Decrement the register (before or after the instruction is executed).
INC		Increment the register (before or after the instruction is executed).
W/B		Word (16-bits) or byte (8-bits) addressing or register size.
R/C		Register (0) or Constant (1).
S		Source register bit (one of 3).
D		Destination register bit (one of 3).
B		Bit (one of 8) in MOVL, MOVLZ, MOVLS, and MOVH instructions.
OFF		A bit used in an offset (in LDR, STR, and branching instructions).
S/C		Source register or constant value (see Table 23)
SA		SVC (Service Call) vector address (#0 through #F).
C		Conditional execution code (#0 to #E)
T		THEN (True) count (#0 to #7)
F		ELSE (False) count (#0 to #7)
V, SLP, N, Z, C		Condition code values (oVerflow, Sleep, Negative, Zero, and Carry).

Two-Operand (Register-Register and Constant-Register) Instructions

Opcode: indicates instruction W/B: word = 0, byte = 1 DST: dst reg
R/C: indicates whether the src field is a reg or an encoded constant

Table 8: Interpretation of R/C bit and SRC/CON bits

R/C		SRC/CON
0	1	
Register	Constant value	Value (bits 3-5)
R0	0	000
R1	1	001
R2	2	010
R3	4	011
R4	8	100
R5/LR	16	101
R6/SP	32	110
R7/PC	-1	111

0 and -1 often indicate the end of a quantity
1, 2, 4, & 8 used to increment through a byte
- short (2-bytes)
- integer (4-bytes)
- double (8-bytes)

32 represents ASCII space & is the difference between lower- & upper-case ASCII characters

Table 9: Two-operand instructions
(SRC can be a register or a constant [see Table 8])

Instruction	Operation	Description
ADD(.B or .W) SRC,DST	*DST ← DST + SRC*	Add SRC to DST
ADDC(.B or .W) SRC,DST	*DST ← DST + SRC + C*	Add SRC and carry to DST
SUB(.B or .W) SRC,DST	*DST ← DST + ~SRC + 1*	Subtract SRC from DST
SUBC(.B or .W) SRC,DST	*DST ← DST + ~SRC + C*	Subtract SRC from DST plus carry
DADD(.B or .W) SRC,DST	*DST ← DST + SRC + C*	Decimal-add SRC and carry to DST
CMP(.B or .W) SRC,DST	*DST + ~SRC + 1*	Compare DST with SRC (subtraction)
XOR(.B or .W) SRC,DST	*DST ← DST ⊕ SRC*	XOR SRC with DST
AND(.B or .W) SRC,DST	*DST ← DST & SRC*	AND SRC with DST
OR(.B or .W) SRC,DST	*DST ← DST | SRC*	OR SRC with DST
BIT(.B or .W) SRC,DST	*DST & (1 << SRC)*	Test if bit set in SRC is set in DST
BIC(.B or .W) SRC,DST	*DST ← DST & ~(1 << SRC)*	Clear bit in DST specified by SRC
BIS(.B or .W) SRC,DST	*DST ← DST | (1 << SRC)*	Set bit in DST specified by SRC

- 4 arithmetic operators: ADD, ADDC, SUB, & SUBC
- byte operations affect the LSB only, MSB untouched
- when using a constant 1 less instr is require, not necessary to use a temp reg
- an unsigned 16-bit number has values from #0 ($0) to #FFFF ($65535)
- a signed 16-bit number uses the same 16-bits to represent the sign of the quantity (MSB) & the remaining bits constitute the number, values range from #8000 ($-32768) through #FFFF ($-1), #0 ($0), #1 ($1) to #7FFF ($32767)
- subtract & compare instrs are performed using two's complement addition to determine result of DST - SRC: DST is added to the one's complement of the SRC then 1 is added: Result = DST + ~SRC + 1
  - two's complement therefore result can be unsigned or signed
  - if unsigned, carry bit indicates result of addition of MSBs that is to be added to the complement of the LSB
- when subtracting a multiple structure result is obtained by DST + ~SRC + X
  - for SUB, X = 1 to obtain the 2's complement of the low-order structure
  - for SUBC, X is the value of the carry bit
  - sign bit only has meaning in the most-significant word of the structure
- in signed arithmetic the negative bit (N) is the value of the MSB of the result (1 = neg, 0 = non neg)
- overflow bit (V) indicates whether the result of the subtraction has overwritten the sign bit
  - V = 1 when DST & ~SRC have same sign but result has opposite (otherwise V = 0)

DST	~SRC	Result
Positive	Positive	Negative
Negative	Negative	Positive

- CMP & BIT instrs are used for explicit comparisons & can change the PSW bits, both instrs are non-destructive in that dst reg is not changed
- bit instrs (BIT, BIC, BIS) require source value to have a value that specifies bit being accessed (0 to 15)
- BIC sets PSW Z equal to 1 if result is 0, BIS clears the zero-status bit

Register-Exchange Instructions

- do not change the PSW status bits

Table 10: Register-exchange instructions

Instruction	Operation	Description
MOV(.B or .W) SRC,DST	*DST ← SRC*	Move SRC to DST. SRC can be a register or a constant.
SWAP SRC,DST	*TMP ← DST* *DST ← SRC* *SRC ← TMP*	Swap or exchange SRC and DST. TMP is an internal register that cannot be accessed by the programmer. SRC and DST are registers. R/C and W/B are ignored since SWAP exchanges register.

- SWAP: exchanges SRC and DST regs - MOV: moves SRC reg to DST reg

Single-Register Instructions without an Operand

- modify the contents of dst reg
- shift: data is shifted either left or right, equivalent to multiplying or dividing by 2
- left shift: equivalent to adding a number to itself
  - a zero or the carry bit can be fed into the LSB & the MSB can be discarded or assigned to the carry bit, left shifting can be emulated using the ADD or ADDC instr
- right shift: requires a special instr or integer division
  - LSB can be discarded or copied into carry bit, MSB can be fed a zero bit (loss of the sign bit)
  - to maintain sign bit, duplicate the value of the MSB (arithmetic shifting)
- rotate: moves simultaneously left or right
  - in right shift the MSB is copied into the LSB, in left shift the LSB is copied into the MSB
  - carry bit can be used as an intermediary, if carry is used, rotating N+1 times (N = size of structure being rotated) returns the structure to its original value

Table 11: Single-operand bit-movement instructions

Instruction	Operation	Description
SRA(.B or .W) DST	*DST.MSB →…→ DST.LSB → C*	Arithmetic shift DST right one bit through Carry with sign extension. Shift can operate on a word or a byte. The Most Significant Bit (MSB) remains unchanged
RRC(.B or .W) DST	*C→ DST.MSB →…→ DST.LSB → C*	Rotate DST right one bit through Carry. Rotate can operate on a word or a byte.

SRA: maintains the value of the sign bit, thereby supporting signed division by 2
RRC: stores the carry bit into the MSB while shifting the regs bits to the right by 1, LSB -> carry bit

Table 12: Single-operand byte-exchange and sign extension instructions

Instruction	Operation	Description
SWPB DST	*TMP ← DST.MSB* *DST.MSB ← DST.LSB* *DST.LSB ← TMP*	Swap bytes in DST (word only). W/B bit is ignored.
SXT DST	*bit 7→ bit 8 →…→ bit 15*	Sign-extend LSB byte to word in DST (word only). W/B bit is ignored.

SWPB: exchanges the 2 bytes in a word SXT: duplicates MSB in the first byte in the second byte

Program Status Word and Supervisory Call Instructions

Table 17: CPU state instructions

Instruction	Operation	Description
SETPRI	*IF NewPri < PSW.Current Priority THEN* *PSW.Current Priority ← NewPri* *ELSE* *A CPU priority fault occurs*	Change the applications to a new priority that is lower than its current priority
SVC	*IF NewPri > PSW.Current Priority THEN* *Stack ← PC, LR, PSW, and CEX state* *PSW ← mem [VectBase + SA]* *PC ← mem [VectBase + SA + 2]* *LR ← #FFFF* *Clear CEX STATE information* *ELSE* *A priority fault occurs*	Control passes to supervisory control routine specified in one of XM-23's interrupt vectors. The requested priority must be greater than the current priority, otherwise a priority fault occurs.
SETCC	*IF PSW.Current Priority = 7 THEN* *SLP ← 0* *Set the specified PSW bits*	Set one or more of the PSW condition code bits or the SLP bit, or a combination of all five. Sleep cannot be set at priority 7.
CLRCC	*Clear the specified PSW bits*	Clear one or more of the PSW condition code bits or the SLP bit, or a combination of all five.

Transfer of Control

Table 13: Transfer of control instructions

Instruction	Operation (*label* is the left-shifted, sign-extended value of the offset)	Description	Type
BL label	*LR ← PC* *PC ← PC + label*	Branch with link to subroutine; store return address in LR. A return can be realized by using any instruction that copies LR into the PC, such as MOV or SWAP. See Chapter 11 on exceptions for additional information on LR.	-
BEQ label BZ label	*PC ← PSW.Z = 1 ? PC + label : PC*	Branch to label if equal (Z = 1) Branch to label if zero flag is set	-
BNE label BNZ label	*PC ← PSW.Z = 0 ? PC + label : PC*	Branch to label if not equal (Z = 0) Branch to label if zero flag is cleared	-
BC label BHS label	*PC ← PSW.C = 1 ? PC + label : PC*	Branch to label if carry set (C = 1) Branch to label if higher or same	Unsigned
BNC label BLO label	*PC ← PSW.C = 0 ? PC + label : PC*	Branch to label if carry clear (C = 0) Branch to label if lower	Unsigned
BN label	*PC ← PSW.N = 1 ? PC + label : PC*	Branch to label if negative (N = 1)	-
BGE label	*PC ← (PSW.N ⊕ PSW.V) = 0 ? PC + label : PC*	Branch to label if greater or equal	Signed
BLT label	*PC ← (PSW.N ⊕ PSW.V) = 1 ? PC + label : PC*	Branch to label if less than	Signed
BRA label	*PC ← PC + label*	Branch always (unconditional) to label	-

Calculating the effective address of the new program counter

- same algorithm regardless of instr
- extract offset from instr (13 or 10 bits)
  - range of values for 13-bit offset in BL instr is #0000 through #1FFF w/ bit 12 as sign bit
  - range of values for 10-bit offset in BRA instr is #000 through #3FF w/ bit 9 as sign bit

1) extend the sign bit of the offset, if set the MSBs are set otherwise cleared
2) left shift offset, LBS will be clear (0)
3) add the offset to the program counter to obtain the effective addrs, range of possible EAs relative to the addrs of the branching instr = PC + 2 + sign-extended, left shifted offset
  - for BL range is PC - 8190 to PC + 8192 - for BRA range is PC - 1022 to PC + 1024

EA is then assigned to the program counter, if the offset is -2 an infinite loop will occur

CEX Instruction

Table 14: Conditional execution codes and their meanings

Assembler code	Description	PSW bit values inspected	Instruction code
EQ	Equal / equals zero	Z = 1	0000
NE	Not equal	Z = 0	0001
CS / HS	Carry set / unsigned higher or same	C = 1	0010
CC / LO	Carry clear / unsigned lower	C = 0	0011
MI	Minus / negative	N = 1	0100
PL	Plus / positive or zero	N = 0	0101
VS	Overflow	V = 1	0110
VC	No overflow	V = 0	0111
HI	Unsigned higher	C = 1 and Z = 0	1000
LS	Unsigned lower or same	C = 0 or Z = 1	1001
GE	Signed greater than or equal	N == V	1010
LT	Signed less than	N != V	1011
GT	Signed greater than	Z = 0 and (N == V)	1100
LE	Signed less than or equal	Z = 1 or (N != V)	1101
TR	True part is always executed	Ignored	1110
FL	False part is always executed	Ignored	1111

Emulated Instructions

Table 18: Emulated instructions
(`.x` indicates a byte, `.B`, or a word, `.W`, can be specified)

Instruction	Emulation	Description
`ADC.x Rx`	`ADDC.x #0,Rx`	Add carry to Rx
`CALL subr`	`BL subr`	Call subr; Return address put in LR
`CLC`	`CLRCC C`	Clear PSW Carry bit
`CLN`	`CLRCC N`	Clear PSW Negative bit
`CLS`	`CLRCC S`	Clear PSW Sleep bit
`CLV`	`CLRCC V`	Clear PSW oVerflow bit
`CLZ`	`CLRCC Z`	Clear PSW Zero bit
`CLR Rx`	`MOVLZ #0,Rx`	Clear Rx
`COMP.x Rx`	`XOR.x #FFFF,Rx`	One's complement of Rx
`DADC.x Rx`	`DADD.x #0,Rx`	Decimal add carry to Rx
`DEC.x Rx`	`SUB.x #1,Rx`	Decrement Rx
`DECD.x Rx`	`SUB.x #2,Rx`	Double Rx
`INC.x Rx`	`ADD.x #1,Rx`	Increment Rx
`INCD.x Rx`	`ADD.x #2,Rx`	Double increment Rx
`JUMP Rx`	`MOV Rx,PC`	Jump to destination (in Rx)
`NOP`	`MOV R0,R0`	No operation
`PULL Rx`	`LD SP+,Rx`	Stack Pull (POP) Rx
`PUSH Rx`	`ST Rx,-SP`	Stack Push Rx
`RET`	`MOV LR,PC`	Return from subroutine or interrupt Service Routine
`RLC.x Rx`	`ADDC.x Rx,Rx`	Rotate left Rx through carry
`SBC.x Rx`	`SUBC.x #0,Rx`	Subtract carry from Rx
`SEC`	`SETCC C`	Set PSW Carry bit
`SEN`	`SETCC N`	Set PSW Negative bit
`SEV`	`SETCC V`	Set PSW oVerflow bit
`SEZ`	`SETCC Z`	Set PSW Zero bit
`SLA.x Rx`	`ADD.x Rx,Rx`	Shift left arithmetic (shift left 1 bit) Rx; Multiply by 2
`TST.x Rx`	`CMP.x #0,Rx`	Test Rx for zero

Arithmetic

Destination sign	Opr	Source sign	Destination sign (result)
Positive	+	Positive	Negative
Negative	+	Negative	Positive

Destination sign	Opr	Source sign	Destination sign (result)
Positive	–	Negative	Negative
Negative	–	Positive	Positive

- subtraction is performed using the method of complements by taking the one's complement of the subtrahend, adding it to the minuend, & adding 1 to the result to give the difference