X and Hex

David May: May 6, 2014

The X Language

X is a simple sequential programming language. It is easy to compile and an X compiler written in X is available to simplify porting between architectures. It is relatively easy to modify the compiler to target new architectures or to and extend the language.

Notation

The following examples illustrate the notation used in the definition of X.

The meaning of

$$assignment \quad = \quad variable \texttt{ := } expression$$

is "An $assignment$ is a $variable$ followed by `:=` followed by an $expression$"

The meaning of

$$literal \quad = \quad integer \mid byte \mid string$$

is "An $literal$ is an $integer$ or a $byte$ or a $string$". This may also be written

$$literal \quad = \quad integer$$

$$literal \quad = \quad byte$$

$$literal \quad = \quad string$$

The notation $\{\ process\ \}$ means "a list of zero or more $processes$".

The notation $\{_0\ \texttt{,}\ expression\}$ means "a list of zero or more expressions separated from each other by `,`", and $\{_1\ \texttt{,}\ expression\}$ means "a list of one or more expressions separated from each other by `,`".

The format of an X program is specified by the syntax. Space, tab and line breaks are ignored and can be inserted in text strings using the escape character $\star$.

Comment

$comment \quad = \quad \texttt{|}\ text\ \texttt{|}$

$text \quad = \quad \{_0\ character\}$

A comment is used to describe the operation of the program.

$process \quad = \quad comment\ process$

Let C be a comment and P be a process. Then $C\ P$ behaves like P.

Statement

$$\begin{array}{lcl} process & = & \texttt{skip} \\ & | & \texttt{stop} \\ & | & assignment \\ & | & sequence \\ & | & conditional \\ & | & loop \\ & | & call \end{array}$$

`skip` starts, performs no action, and terminates.

`stop` starts but never proceeds and never terminates.

$assignment \quad = \quad variable\ \texttt{:=}\ expression$

An assignment evaluates the expression, assigns the result to the variable, and then terminates. All other variables are unchanged in value.

$sequence \quad = \quad \texttt{\{}\ \{_0\ \texttt{;}\ process\}\ \texttt{\}}$

A sequence starts with the start of the first process. Each subsequent process starts if and when its predecessor terminates and the sequence terminates when the last process terminates. A sequence with no component processes behaves like `skip`.

Conditional

$conditional \quad = \quad \texttt{if}\ expression\ \texttt{then}\ process\ \texttt{else}\ process$

Let e be an expression and let P and Q be processes. Then

`if` e `then` P `else` Q

behaves like P if the initial value of e is $true$. Otherwise it behaves like Q.

Loop

$loop \quad = \quad \texttt{while}\ expression\ \texttt{do}\ process$

A loop is defined by

`while` e `do` P = `if` e `then { ` P`; while` e `do` P `} else skip`

Scope

$$\begin{array}{lcl} process & = & specification \texttt{ ; } process \\ \\ specification & = & declaration \\ & | & abbreviation \\ & | & definition \end{array}$$

A block N `:` S behaves like its scope S; the specification N specifies a name which may be used with this specification only within S.

Let x and y be names and let $S(x)$ and $S(y)$ be scopes which are similar except that $S(x)$ contains x wherever $S(y)$ contains y, and vice versa. Let $N(x)$ and $N(y)$ be specifications which are similar except that $N(x)$ is a specification of x and $N(y)$ is a specification of y. Then

$N(x)$ `;` $S(x) = N(y)$ `;` $S(y)$

Using this rule it is possible to express a process in a canonical form in which no name is specified more than once.

Declaration

$$\begin{array}{lcl} declaration & = & \texttt{var } name \\ & | & \texttt{array } name \texttt{ [} expression \texttt{]} \end{array}$$

A declaration declares a name as the name of a variable or of an array.

Abbreviation

$$\begin{array}{lcl} abbreviation & = & \texttt{val } name = expression \\ & | & \texttt{array } name = name \\ & | & \texttt{proc } name = name \\ & | & \texttt{func } name = name \end{array}$$

An abbreviation `val` $n = e$ specifies n as an abbreviation for expression e. Let e be an expression and $P(e)$ be a process. Then

`val` $n = e$ `;` $P(n) = P(e)$

Let T be `array`, `proc` or `func`. Then

$T\ n = m$ `;` $P(n) = P(m)$

Procedure

$$\begin{array}{lcl} definition & = & \texttt{proc}\ name\ \texttt{(}\ \{_0\ \texttt{,}\ formal\}\ \texttt{)}\ \texttt{is}\ body \\ \\ formal & = & \texttt{val}\ name \\ & | & \texttt{array}\ name \\ & | & \texttt{proc}\ name \\ & | & \texttt{func}\ name \\ \\ body & = & process \end{array}$$

The definition

$\texttt{proc}\ n\ \texttt{(}\ \{_0\ \texttt{,}\ formal\}\ \texttt{)}\ \texttt{is}\ B$

defines n as the name of a procedure.

$$\begin{array}{lcl} instance & = & name\ \texttt{(}\ \{_0\ \texttt{,}\ actual\}\ \texttt{)} \\ \\ actual & = & expression \\ & | & name \end{array}$$

Let X be a program expressed in the canonical form in which no name is specified more than once. If X contains a procedure definition

$P\texttt{(}F_0\texttt{,}\ F_1\texttt{,}\ ...\texttt{,}\ F_n\texttt{)}\ \texttt{is}\ B$

then within the scope of P

$P\texttt{(}A_0\texttt{,}\ A_1\texttt{,}\ ...\texttt{,}\ A_n\texttt{)} = F_0 = A_0\ \texttt{;}\ F_1 = A_1\ \texttt{;}\ ...\ F_n = A_n\ \texttt{;}\ B$

provided that each abbreviation $F_i = A_i$ is valid.

A procedure can always be compiled either by substitution of its body as described above or as a closed subroutine.

Element

Elements enable variables or arrays be selected from arrays.

$$\begin{array}{lcl} element & = & element\texttt{[}\ subscript\ \texttt{]} \\ & | & name \\ \\ subscript & = & expression \end{array}$$

Let a be an array with n components and e an expression of value s. Then $v\texttt{[}e\texttt{]}$ is valid only if $0 \leq s$ and $s < n$; it is the component of v selected by s.

Variable

variable = *element*

Every variable has a value that can be changed by assignment or input. The value of a variable is the value most recently assigned to it, or is arbitrary if no value has been assigned to it.

Let a be an array with n components, e be an expression of value s, and x be an expression. If $0 \leq s$ and $s < n$, then v`[`e`]` `:=` x assigns to v a new value in which the component of v selected by s is replaced by the value of x and all other components are unchanged. Otherwise the assignment is invalid.

Literal

literal = *integer* | *byte* | *string* | `true` | `false`

integer = *digits* | `#`*digits*

byte = `'`*character*`'`

An integer literal is a decimal number, or `#` followed by a hexadecimal number. A byte literal is an ASCII character enclosed in single quotation marks: `'`.

A string literal is represented by a sequence of ASCII characters enclosed by double quotation marks: `"`. Let s be a string of n characters, where $n < 256$. The value of s is an array containing the value n, followed by ASCII values of the characters in the string. The string is packed into the array.

The literal `true` represents the logical value *true*; numerically `true` = 1. The literal `false` represents the logical value *false*; numerically `false` = 0.

Expression

An expression has a data type and a value. Expressions are constructed from operands, operators and parentheses.

operand = *element* | *literal*
| `(` *expression* `)`

The value of an operand is that of an element, literal or expression.

expression = *monadic.operator operand*
| *operand diadic.operator operand*
| *operand*

The arithmetic operators + and − produce the arithmetic sum and difference of

their operands respectively. Both operands must be integer values and the result is an integer value. The arithmetic operators treat their operands as signed integer values and produce signed integer results. If n is an operand, then $-n$ = (0−n).

The logical operator `and` produces the logical and of its operands, both of which must have value `true` or `false`. If the value of the first operand is `false`, the result is `false`; otherwise the result is the value of the second operand.

The logical operator `or` produces the logical or of its operands, both of which must have value `true` or `false`. If the value of the first operand is `true`, the result is `true`; otherwise the result is the value of the second operand.

The logical operator `not` produces the logical not of its operand which must have value `true` or `false`:

`not false` = `true` `not true` = `false`

Let **O** be one of the associative operators +, `and`, `or`. Then

e_1 **O** e_2 **O** ... **O** e_n = ((e_1 **O** ((e_2 **O** (... **O** e_n) ...))

The relational operators =, <>, <, <=, >, >= produce a result of `true` or `false`. The operands must both be integer values. The result of x = y is `true` if the value of x is equal to that of y. The result of x < y is `true` if the integer value of x is strictly less than that of y. The other operators obey the following rules:

(x <> y) = `not` (x = y) (x >= y) = `not` (x < y)
(x > y) = (y < x) (x <= y) = `not` (x > y)

where x and y are any values.

expression = `valof` *process*

process = `return` *expression*

A valof expression executes a process to produce a value. The final process executed in a valof must be a return. The return evaluates its expression and the resulting value is the value of the valof.

Function

definition = `func` *name* `(` {$_0$ `,` *formal*} `)` `is` *body*

The definition

`func` n `(` {$_0$ `,` *formal*} `)` `is` B

defines n as the name of a function with a body B that computes a value.

$expression \quad = \quad name$ `(` $\{_0$ `,` $actual\}$ `)`

Let X be a program expressed in the canonical form in which no name is specified more than once. If X contains a function definition

`func` F `(`F_0`,` F_1`, ...,` F_n`)` `is` B

then within the scope of F

F `(`A_0`,` A_1`, ...,` A_n`)` $=$ `valof` $F_0 = A_0$ `;` $F_1 = A_1$ `;` ... $F_n = A_n$ `;` B

provided that each abbreviation $F_i = A_i$ is valid.

A function can always be compiled either by substitution of its body as described above or as a closed subroutine.

Character set

The characters used in X are as follows.

Alphabetic characters

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

```
abcdefghijklmnopqrstuvwxyz
```

Digits

```
0123456789
```

Special characters

```
!"#&'()*+,-_./:;<=>?[]{}
```

Strings and character constants may contain any X character except `*`, `'` and `"`. Certain characters are represented as follows:

`*c`	carriage return
`*n`	newline
`*t`	horizontal tabulate
`*s`	space
`*'`	quotation mark
`*"`	double quotation mark
`**`	asterisk

If a string contains the character pair `*l` immediately following the opening `"`, then the value of byte 0 of the string is the subscript of the last character in the string.

Any character can be represented by `*#` followed by two hexadecimal digits.

A name consists of a sequence of alphabetic characters, decimal digits and underscores (_), the first of which must be an alphabetic character. Two names are the same only if they consist of the same sequence of characters and corresponding characters have the same case.

The Instructions

The main features of the instruction set used by the X HEX compiler are:

- Short instructions are provided to allow efficient access to the stack and other data regions allocated by compilers; these also provide efficient branching and subroutine calling.
- The memory is word addressed; however the instructions are all single byte so instruction addresses refer to a specific byte position within a word.
- The same instruction set can be used for processors with different wordlengths; the only requirement is that the wordlength is a number of bytes.
- The processor has a small number of registers. Some registers are used for specific purposes such as accessing the program or building large constants.
- Instructions are easy to decode.

All instructions are 8-bit; each instruction contains 4 bits representing an operation and 4 bits of immediate data. A special instruction, OPR causes its operand to be interpreted as an inter-register operation. Instruction prefixes are used to extend the range of immediate operands and to provide more inter-register operations:

- PFIX concatenates its 4-bit immediate with the 4-bit immediate of the next 8-bit instruction.
- NFIX complements its its 4-bit immediate and then concatenates the result with the 4-bit immediate of the next 8-bit instruction.

The prefixes are inserted automatically by the compiler.

The normal state of a processor is represented by 4 registers. Two of the registers are used to hold the sources and destination of arithmetic and logic operations. Another (the operand register) is used to accumulate the operands of the prefixes.

register	**use**
pc	the program counter
$oreg$	the operand register
$areg$	left-hand operand and result of arithmetic
$breg$	right-hand operand of arithmetic

Instruction set Notation and Definitions

In the following description

mem	represents the memory
pc	represents the program counter
$oreg$	represents the operand register
$areg$	represents the left-hand operand register
$breg$	represents the right-hand operand register
$u4$	is a 4-bit unsigned source operand in the range $[0:15]$

Data access

The data access instructions fall into several groups. One of these provides access via the stack pointer.

LDAM	$areg \leftarrow mem[oreg]$	load from memory
LDBM	$breg \leftarrow mem[oreg]$	load from memory
STAM	$mem[oreg] \leftarrow areg$	store to memory

Access to constants and program addresses is provided by instructions which either load values directly or enable them to be loaded from a location in the program:

LDAC	$areg \leftarrow oreg$	load constant
LDBC	$breg \leftarrow oreg$	load constant
LDAP	$areg \leftarrow pc + oreg$	load address in program

Access to data structures is provided by instructions which combine an address with an offset:

LDAI	$areg \leftarrow mem[areg + oreg]$	load from memory
LDBI	$breg \leftarrow mem[breg + oreg]$	load from memory
STAI	$mem[breg + oreg] \leftarrow areg$	store to memory

Branching, jumping and calling

The branch instructions include conditional and unconditional relative branches. A branch using an offset in the stack is provided to support jump tables.

BR	$pc \leftarrow pc + oreg$	branch relative unconditional
BRZ	if $areg = 0$ then $pc \leftarrow pc + oreg$	branch relative zero
BRN	if $areg < 0$ then $pc \leftarrow pc + oreg$	branch relative negative
BRB	$pc \leftarrow breg$	branch absolute
SVC		system call

To call a procedure, the return address can be loaded using the LDAP instruction and the BR instruction can be used to branch to the procedure entrypoint. The procedure entry will store the return address; the exit will load this return address into $breg$ and use a BRB instruction to branch back to the calling procedure.

Expression evaluation

ADD	$areg \leftarrow areg + breg$	add
SUB	$areg \leftarrow areg - breg$	subtract

Instruction summary

LDAM	$areg \leftarrow mem[oreg]$	load from memory
LDBM	$breg \leftarrow mem[oreg]$	load from memory
STAM	$mem[oreg] \leftarrow areg$	store to memory
LDAC	$areg \leftarrow oreg$	load constant
LDBC	$breg \leftarrow oreg$	load constant
LDAP	$areg \leftarrow pc + oreg$	load address in program
LDAI	$areg \leftarrow mem[areg + oreg]$	load from memory
LDBI	$breg \leftarrow mem[breg + oreg]$	load from memory
STAI	$mem[breg + oreg] \leftarrow areg$	store to memory
BR	$pc \leftarrow pc + oreg$	branch relative unconditional
BRZ	if $areg = 0$ then $pc \leftarrow pc + oreg$	branch relative zero
BRN	if $areg < 0$ then $pc \leftarrow pc + oreg$	branch relative negative
BRB	$pc \leftarrow breg$	branch absolute
ADD	$areg \leftarrow areg + breg$	add
SUB	$areg \leftarrow areg - breg$	subtract
SVC		system call

The Compiler

The compiler compiles X into a the HEX instruction set. It is written in X and can be enhanced by bootstrapping. It has been written so as to be fairly easy to understand and modify. It performs a only a few simple optimisations which makes the object code and its relationship to the source program is easy to follow.

The compiler generates executable binary. The object code is position independent and can be placed anywhere in memory; only the highest used address in memory is predefined. The executable form of the compiler occupies about 17,000 bytes and it requires about 150,000 bytes in order to compile itself.

Structure

The compiler operates by first translating the source text into an internal data structure; this is a tree built from nodes. The first word in each node contains a symbol; the number of words and their meaning is defined by this symbol.

The compiler has the following main components

- The *lexical analyser* that translates the source text into internal symbols. The lexical analyser includes a nametable which is used to look up both pre-defined names and program defined names.
- The *syntax analyser* that calls the lexical analyser each time it needs a new symbol and builds the tree representing the incoming symbol stream. It operates by *recursive descent* and many of its component functions follow the BNF representation of the syntactic structure they read.
- The *translator* that converts the tree into a sequence of instructions. It calls the codebuffer procedures so as to build up an internal representation of the instruction sequence. The translator maintains a stack of program defined names so as to implement the scope rules of X. It contains an optimiser which performs simple optimisations, such as replacing $x + 0$ with x, by modifying the tree.
- The *codebuffer* that stores a representation of the compiled program which it converts into an executable binary and outputs. The codebuffer calculates program offsets used for branches and access to constant data.

Memory layout

The compiler uses the memory as follows:

- jump to start of program (word 0)
- stack pointer (word 1)
- global variables
- large constants
- strings
- program
- stack
- arrays

Each global variable and array is allocated a word location at the bottom of memory. In the case of a global array, this word location holds the address of the array itself, which is allocated space at the top of memory. The array can then be accessed by first loading its associated word location at the bottom of memory using a LDAM or LDBM instruction.

The compiler initialises the locations in the global region that hold the addresses of arrays, and also initialises the stack pointer to the address of the location just below the arrays at the top of memory.

Most constants can be loaded directly as instruction operands. The compiler uses the minimum number of prefix instructions needed to represent the constant. Large constants are stored in memory locations above the globals and are accessed using LDAM and LDBM instructions. These are followed by strings which are accessed by LDAC and LDBC instructions.

Stack, Parameters and Locals

Local variables and formal parameters of procedures and functions are held on the stack and accessed relative to the stack pointer. On entry to a procedure or function:

1. *areg*, which holds the return address, is stored on to the stack
2. the stack pointer is decremented by the number of locations needed to store the formal parameters, local variables and any temporary values needed during expression evaluation

On exit from a procedure or function:

1. in a function, the value to be returned is stored on the stack
2. the stack pointer is incremented by the number of locations needed to store the formal parameters, local variables and temporary values needed during expression evaluation
3. the return address is loaded from the stack to *breg* and a BRB instruction is executed to transfer control back to the caller

BRB is used to return to the caller after a functions has finished executing

Copies the value of breg to the pc.

A procedure or function is called by storing the actual parameters on the stack, loading the return address into *areg* using the LDAP instruction, and branching to the procedure of function entrypoint. The branch is performed using a BRU instruction unless the entrypoint has been passed as a parameter to the caller in which case the branch is performed by loading the entrypoint and using a BRB instruction.

Control structure

When translating a statement, the translator keeps track of where execution is to continue after the statement has been executed. This is done using parameter *seq* of *genstatement*. If *seq* is true then execution can continue with the next statement in sequence; otherwise the statement is compiled so as to end by branching to a specified label (the value of parameter *clab*).

In addition, a parameter *tail* of *genstatement* is used to identify the statements which must be immediately followed by a return; this parameter is used to eliminate tail recursions where possible by branching to the point just after the stack adjustment in the procedure entry sequence.

A simple optimisation removes code that would otherwise be generated for conditionals with `skip` components.

Arithmetic and Logic

The arithmetic operators (+, −) correspond directly to instructions.

The logical operators (or, and, ∼) are implemented using conditional branches.

Load stack pointer which is stored in locat. 1 in memory

Load offset to access local variable on stack

Access to local variables is performed via the stack pointer using a pair of instructions such as (LDAM 1, LDAI n). Access to global variables is performed by an instruction such as LDAM n.

Global variabales are directly stored in memory, thus can be loaded using LDAM

Constants can normally be loaded directly as instruction operands, but large constants are accessed from memory. again using LDAM

Comparisons and Conditionals

The is no direct method of producing a boolean value as a result of a comparison. This has to be implemented using a subtraction and then converting the result of the subtraction into a boolean:

$areg \leftarrow areg = breg$	SUB BRZ 2 LDAC 0 BRU 1 LDAC 1	Subtract areg−breg and store to areg. If result is 0 go to the last command and load constant 1(true). if not then load constant 0(false) and skip the last command with a BR 1
$areg \leftarrow areg \neq breg$	SUB BRZ 1 LDAC 1	Same logic. If 0, skip command LDAC 1
$areg \leftarrow areg < breg$	SUB BRN 2 LDAC 0 BRU 1 LDAC 1	Subtract. If negative, then areg was less than breg, therefore load 1.
$areg \leftarrow areg \geq breg$	SUB BRN 2 LDAC 1 BRU 1 LDAC 0	Same. If negative, load 0 because areg is less than breg

In the common case of using a comparison in a conditional branch, there is no need to convert the result to a logical value.

Footnote

The absence of any bit-manipulation instructions (such as bitwise logicals and shifts) helps to keep the instruction set and processor small. The compiler includes functions for multiplication, division and remainder which are written using only addition and subtraction; packing and unpacking of data is then expressed in terms of these.