X and Arm

David May: April 25, 2013

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

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

The definition

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

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

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

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 a bit-pattern consisting entirely of 1 bits and the value of `false` represents a bit pattern consisting entirely of zero bits. Numerically, `true` = (-1) and `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, difference, and product 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 operators `and`, `or`, `xor` produce the bitwise and, or and exclusive or of their operands respectively. Both operands must be integer values and the result is an integer value. Each bit of the result is produced from the corresponding bits of the operands according to the following rules:

b `xor` $0 = b$ $\quad 0$ `xor` $1 = 1$ $\quad 1$ `xor` $1 = 0$
b `and` $0 = 0$ $\quad b$ `and` $1 = b$
b `or` $0 = b$ $\quad b$ `or` $1 = 1$

where b is 0 or 1. The effect of this is that, for the values `true` and `false`:

b `xor false` $=$ `false` $\quad$ `false xor true` $=$ `true` $\quad$ `true xor true` $=$ `false`
b `and false` $=$ `false` $\quad b$ `and true` $= b$
b `or false` $= b$ $\quad b$ `or true` $=$ `true`

where b is `true` or `false`

The logical operator `not` produces the bitwise not of its operand which must be an integer value. Each bit of the result is produced from the operand as follows:

`not` $0 = 1$ $\quad$ `not` $1 = 0$

The effect of this is that:

`not false` $=$ `true` $\quad$ `not true` $=$ `false`

Let n be the number of bits in an integer value. Let x_i be bit i of value x. The bits produced by the shift operators >> and << are defined by:

$(a >> s)_i = a_{s+i}$ $\quad$ if $(s + i) < n$
$(a >> s)_i = 0$ $\quad$ if $(s + i) \geq n$

$(a << s)_i = a_{s-i}$ $\quad$ if $(s - i) \geq 0$
$(a << s)_i = 0$ $\quad$ if $(s - i) < 0$

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

$(x <> y) = \texttt{not}\ \ (x = y) \qquad (x >= y) = \texttt{not}\ \ (x < y)$
$(x > y) = (y < x) \qquad\qquad (x <= y) = \texttt{not}\ \ (x > y)$

where x and y are any values.

$$expression \quad = \quad \texttt{valof}\ process$$

$$process \quad = \quad \texttt{return}\ expression$$

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

$$definition \quad = \quad \texttt{func}\ name\ \texttt{(}\ \{_0\ \texttt{,}\ formal\}\ \texttt{)}\ \texttt{is}\ body$$

The definition

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

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

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

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

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

then within the scope of F

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

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

A 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 ARM X compiler are as follows.

- The instructions are a subset of ARM Thumb. The short (16-bit) instructions allow efficient access to the stack and other data regions allocated by compilers; these also provide efficient branching and subroutine calling.
- The memory is byte addressed; however all accesses must be aligned on natural boundaries so that, for example, the addresses used in 32-bit loads and stores have the two least significant bits zero.
- Input and output is performed using memory mapped registers accessed in the normal by load and store instructions. A *supervisor call* instruction is provided.

Some instructions contain immediate operands which are used to access locations relative to the program counter pc or the stack pointer sp. As the pc is used to access two byte (16-bit) locations, the operands of instructions that access locations relative to the pc are multiplied by 2. Similarly, As the sp is used to access four byte (32-bit) locations, the operands of instructions that access locations relative to the sp are multiplied by 4.

The normal state of a program is represented by 8 operand registers and some special purpose registers.

The eight operand registers $r0$ - $r7$ are used by instructions which perform arithmetic and logical operations and access data structures.

The special purpose registers are:

register	**use**
pc	the program counter
lr	the link register
sp	the stack pointer
rs	the result register

Instruction set Notation and Definitions

In the following description

mem	represents the memory
pc	represents the program counter
sp	represents the stack pointer
lr	represents the link register
rs	represents the result register
$r0 \ldots r7$	represent specific operand registers
x	(a single small letter) represents one of $r0 \ldots r7$
$u5$	is a 5-bit unsigned source operand in the range $[0 : 31]$
$u7$	is a 7-bit unsigned source operand in the range $[0 : 127]$
$u8$	is an 8-bit unsigned source operand in the range $[0 : 255]$
$s8$	is an 8-bit signed source operand in the range $[-128 : 127]$
$s11$	is an 11-bit signed source operand in the range $[-1024 : 1023]$
$s22$	is a 22-bit signed source operand in the range $[-1048576 : 1048575]$

Data access

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

LDRSPI	$d \leftarrow mem[sp + u8_{\times 4}]$	load word from stack
STRSPI	$mem[sp + u8_{\times 4}] \leftarrow s$	store word to stack
ADDSPI	$d \leftarrow sp + u8_{\times 4}$	load address of word in stack

Another is similar, but provides access via the data pointer.

Access to constants and program addresses is provided by instructions which either load values directly or load them from a constant pool.

MOVI	$d \leftarrow u8$	load constant
LDRPCI	$d \leftarrow mem[pc + u8_{\times 4}]$	load word from constant pool
ADDPCI	$d \leftarrow pc + u8_{\times 4}$	load address in program forward

Access to data structures is provided by instructions which use any of the operand registers as a base address, and combine this with a scaled offset. In the case of word accesses, the operand may be a small constant or another operand register, and the instructions are as follows:

LDRI	$d \leftarrow mem[b + u5_{\times 4}]$	load word
STRI	$mem[b + u5_{\times 4}] \leftarrow s$	store word

LDR	$d \leftarrow mem[b + i]$	load word
STR	$mem[b + i] \leftarrow s$	store word

Expression evaluation

Expressions are evaluated by instructions which operate on values in the general purpose registers $r0$ to $r7$. Some instructions have a constant operand, together with one or two register operands.

ADDI	$d \leftarrow d + u8$	add immediate
ADDR	$d \leftarrow l + r$	add
SUBI	$d \leftarrow d - u8$	subtract immediate
SUBR	$d \leftarrow l - r$	subtract
NEGR	$d \leftarrow -s$	negate
ANDR	$d \leftarrow d \wedge r$	and
ORR	$d \leftarrow d \vee r$	or
XORR	$d \leftarrow d \oplus r$	exclusive or
MVNR	$d \leftarrow -1 \oplus s$	not
SHLI	$d \leftarrow l << u5$	logical shift left immediate
SHL	$d \leftarrow d << r$	logical shift left
SHRI	$d \leftarrow l >> u5$	logical shift right immediate
SHR	$d \leftarrow d >> r$	logical shift right
ASHRI	$d \leftarrow l >>_{sgn} u5$	arithmetic shift right immediate
MUL	$d \leftarrow d \times r$	multiply

Branching, jumping and calling

The branch instructions include conditional and unconditional relative branches. These test the result register rs which holds the same value as the destination register of the last arithmetic or logical instruction.

BEQ	if $rs = 0$ then $pc \leftarrow pc + s8_{\times 2}$	branch relative equal
BNE	if $rs \neq 0$ then $pc \leftarrow pc + s8_{\times 2}$	branch relative not equal
BLT	if $rs < 0$ then $pc \leftarrow pc + s8_{\times 2}$	branch relative less than
BGE	if $rs \geq 0$ then $pc \leftarrow pc + s8_{\times 2}$	branch relative greater or equal
BU	$pc \leftarrow pc + s11_{\times 2}$	branch relative unconditional

In some cases, the calling instructions described below can be used to optimise branches; as they overwrite the link register they are not suitable for use in leaf procedures which do not save the link register.

The procedure call instructions include a relative call and a call using an address in a register. The relative call is encoded as two instructions and can therefore support most calls within a single program module.

BL	$lr \leftarrow pc$; $pc \leftarrow pc + s22_{\times 2}$	branch and link relative forward
BLR	$lr \leftarrow pc$; $pc \leftarrow s$	branch and link via register

Calling normally requires saving and restoring the link register lr and may require modification of the stack. Typically, the link is saved and the stack is extended on procedure entry; the stack is contracted and the pc is restored from the stack on exit. The instructions to support this are shown below.

PUSH	$sp \leftarrow sp - 1_{\times 4}$; $mem[sp] \leftarrow lr$	push link to stack
POP	$pc \leftarrow mem[sp]$; $sp \leftarrow sp + 1_{\times 4}$	pop link from stack to pc
DECSP	$sp \leftarrow sp + u7_{\times 4}$	extend stack
INCSP	$sp \leftarrow sp - u7_{\times 4}$	contract stack

At the start of a program, and in some other situations, it is necessary to set the stack pointer to a new value.

SETSP	$sp \leftarrow s$	set stack pointer

Instruction set by format

In the following description

rd is a destination register r0 - r7
rs is a source register r0 - r7
rsd is a source and destination register r0 - r7
u5 is a 5-bit unsigned source operand in the range $[0:31]$
u7 is a 7-bit unsigned source operand in the range $[0:127]$
u8 is an 8-bit unsigned source operand in the range $[0:255]$
s8 is an 8-bit signed source operand in the range $[-128:127]$
s11 is an 11-bit signed source operand in the range $[-1024:1023]$
s22 is a 22-bit signed source operand in the range $[-1048576:1048575]$

Three register

op	dest	src1	src2	Description
ADD	rd	rs	rs	Add
SUB	rd	rs	rs	Subtract
LDW	rd	rs	rs	Load word
STW	rs	rs	rs	Load word
SHL	rd	rs	rs	Shift left
SHR	rd	rs	rs	Shift right

Two register and immediate

op	dest	src1	src2	Description
SHLI	rd	rs	u5	Shift left immediate
SHRI	rd	rs	u5	Shift right immediate
ASHRI	rd	rs	u5	Shift right immediate
STWI	gs	rs	u5	Store word
LDWI	rd	rs	u5	Load word

Two register

op	src/dest	imm	Description
AND	rsd	rs	And
OR	rsd	rs	Or
XOR	rsd	rs	And
MUL	rsd	rs	And
MVN	rd	rs	Or
NEG	rd	rs	Or

Register and immediate

op	src/dest	imm	Description
ADDI	rsd	u8	Add immediate
SUBI	rsd	u8	Subtract immediate
LDWSP	rd	u8	Load word from the stack
STWSP	rs	u8	Store word to the stack
ADDSP	rd	u8	Store word to the stack
LDWPC	rd	u8	Load word from the stack
ADDPC	rd	u8	Store word to the stack
MOV	rd	u8	Load constant

Register

op	imm	Description
SETSP	rs	Branch link relative forward
BLR	rs	Branch link relative back

Immediate 7-bit

op	imm	Description
INCSP	u7	Branch link relative forward
DECSP	u7	Branch link relative back

Immediate 8-bit

op	imm	Description
BEQ	s8	Branch relative forward true
BNE	s8	Branch relative forward false
BLT	s8	Branch relative back true
BGE	s8	Branch relative back false
SVC	u8	Branch relative back false

Immediate 11-bit

op	imm	Description
BL1	s11	Branch link relative forward
BL2	u11	Branch link relative back
BU	s11	Branch link absolute via pool

None

op	Description
PUSH	Branch link relative forward
POP	Branch link relative back

The Compiler

The compiler compiles X into a subset of the ARM Thumb instructions. 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 location of the stack top is predefined. The executable form of the compiler occupies about 17,000 bytes (8,500 instructions) 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.

Use of Registers

The registers are used as follows:

- the pc and lr are used in the normal way. lr is only loaded by a call and is only stored by a push.
- the sp is used in the normal way. it is changed on initialisation, by increment and decrement instructions and by push and pop instructions.
- register $r7$ is used to access the global variable region.
- register $r6$ is used as a temporary location.
- registers $r0$ to $r5$ are used for parameter passing.
- register $r0$ is used for function results.

To evaluate an expression leaving a result in register r_n, registers r_{n+1}, r_{n+2} ... r_6 are used. If more registers are needed, temporary locations are allocated on the stack.

Globals

The global variables and arrays are allocated space in a region at the top of memory. The space for the arrays is allocated at the top of this region. Beneath this is a set of word locations; each of these is either a variable or holds the address of the lowest location of one of the arrays.

The compiler generates a sequence of instructions to initialise the locations in the global region that hold the addresses of arrays, and to initialise $r7$ and sp to the address of the lowest location in the global region.

The stack is located just beneath the global region and $r7$ can be used to access global variables and arrays. The use of locations to hold the addresses of arrays means that most global variables and arrays can be accessed using only one or two instructions in conjunction with $r7$.

Stack, Parameters and Locals

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

1. the link register is pushed on to the stack
2. the stack pointer sp is decremented by the number of locations needed to store the formal parameters, local variables and any temporary values needed during expression evaluation

3. the formal parameters are stored in their stack locations

On exit from a procedure or function:

1. in a function, register $r0$ is loaded with the value to be returned
2. the stack pointer sp is incremented by the number of locations needed to store the formal parameters, local variables and temporary values needed during expression evaluation
3. the program counter is popped from the stack

A procedure of function is called by loading the actual parameters into registers $r0$ to $r5$ and performing a branch and link instruction. The values returned by functions are in $r0$. One effect of this is that where a function call appears as the first actual parameter of a call, the value returned from the function call is already in the correct location ready for the next call.

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 transfer 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 and logical operators all correspond directly to instructions. The value -1 is used as the representation of true. There is no difference between boolean and bitwise operations and no use is (currently) made of short-circuit evaluation.

Access to local variables is provided by LDRSPI and access to global variables by LDRI using r7 as the base address (or in cases where the offset is greater than

31, by LDR using r7 as the base address having first loaded the offset into another register).

In a few cases a constant can be used as an instruction operand. These include addition and subtraction of an 8-bit unsigned value and shifting by a constant. In all other cases, a constant must first be loaded into a register.

Comparisons and Conditionals

The Thumb instruction set has 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:

$d \leftarrow l = r$ SUBR d, l, r
BEQ 1
MOVI d, 1
SUBI d, d, 1

$d \leftarrow l \neq r$ SUBR d, l, r
BEQ 1
MOVI d, 1
NEG d, d

$d \leftarrow l < r$ SUBR d, l, r
ASHRI d, d, 31

$d \leftarrow l \geq r$ SUBR d, l, r
ASHRI d, d, 31
MVN d, d

In the common case of using a comparison in a conditional branch, there is no need to convert the result to a boolean. However, there is another problem in that the range of a conditional branch is limited because only a signed 8-bit operand is provided in the conditional branch instructions. This is handled in the codebuffer.

Another issue arises where the conditional in a branch is a boolean variable in that the result register rs is not set by load instructions. So, for example, if v is a local variable:

`if` v `then` ...

must be implemented by

LDRSPI r, v; ADDI r, 0; BNE ... .

Whether the range of a branch is sufficient can only be determined when the instructions have reached the codebuffer; if the range is not sufficient the conditional branch must be replaced by a pair of instructions including an unconditional branch:

BEQ L = BNE 1; BU L

BNE L = BEQ 1; BU L

BLT L = BGE 1; BU L

BGE L = BLT 1; BU L

Constants and Strings

Constants can be loaded into a register using a MOVI instruction, provided that they are in the range $0...255$. A negative constant c in the range $-255... -1$ can be loaded by MOVI r, -c; NEGR r, r. Certain larger constants can be loaded by an instruction sequence of the form MOVI, r, c; LSL3I r, r, offset.

All other constants are allocated space in a *constant pool* which is placed in the instruction stream after the LDRPCI instruction that refers to the constant. An item in a constant pool must occur within 256 words of the instruction that refers to it; as a result a program will normally contain a collection of constant pools distributed through the instruction stream. The compiler places these after procedure returns, or after unconditional transfers of control.

Strings are also located in the constant pools. Access to a string is provided by the ADDPCI instruction which forms the address of the first word in the string.