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 \( d \leftarrow mem[sp + u8_{\times4}] \) load word from stack
STRSPI \( mem[sp + u8_{\times4}] \leftarrow s \) store word to stack
ADDSPSI \( d \leftarrow sp + u8_{\times4} \) 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  \quad d \leftarrow u8 \quad \text{load constant}

LDRPCI \quad d \leftarrow \text{mem}[pc + u8_{x4}] \quad \text{load word from constant pool}

ADDPCI \quad d \leftarrow pc + u8_{x4} \quad \text{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 \quad d \leftarrow \text{mem}[b + u5_{x4}] \quad \text{load word}

STRI \quad \text{mem}[b + u5_{x4}] \leftarrow s \quad \text{store word}

LDR \quad d \leftarrow \text{mem}[b + i] \quad \text{load word}

STR \quad \text{mem}[b + i] \leftarrow s \quad \text{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 \quad d \leftarrow d + u8 \quad \text{add immediate}

ADDR \quad d \leftarrow l + r \quad \text{add}

SUBI \quad d \leftarrow d - u8 \quad \text{subtract immediate}

SUBR \quad d \leftarrow l - r \quad \text{subtract}

NEGR \quad d \leftarrow -s \quad \text{negate}

ANDR \quad d \leftarrow d \land r \quad \text{and}

ORR \quad d \leftarrow d \lor r \quad \text{or}

XORR \quad d \leftarrow d \oplus r \quad \text{exclusive or}

MVNR \quad d \leftarrow -1 \oplus s \quad \text{not}

SHLI \quad d \leftarrow l \ll u5 \quad \text{logical shift left immediate}

SHL \quad d \leftarrow d \ll r \quad \text{logical shift left}

SHRI \quad d \leftarrow l \gg u5 \quad \text{logical shift right immediate}

SHR \quad d \leftarrow d \gg r \quad \text{logical shift right}

ASHRI \quad d \leftarrow l \gg_{\text{sgn}} u5 \quad \text{arithmetic shift right immediate}

MUL \quad d \leftarrow d \times r \quad \text{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 \( pc \leftarrow pc + s8x2 \) branch relative equal

**BNE** if \( rs \neq 0 \) then \( pc \leftarrow pc + s8x2 \) branch relative not equal

**BLT** if \( rs < 0 \) then \( pc \leftarrow pc + s8x2 \) branch relative less than

**BGE** if \( rs \geq 0 \) then \( pc \leftarrow pc + s8x2 \) branch relative greater or equal

**BU** \( pc \leftarrow pc + s11x2 \) 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** \( lr \leftarrow pc; \) branch and link relative forward
\( pc \leftarrow pc + s22x2 \)

**BLR** \( lr \leftarrow pc; \) branch and link via register
\( pc \leftarrow s \)

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** \( sp \leftarrow sp - 1 \times 4; \) push link to stack
\( \text{mem}[sp] \leftarrow lr \)

**POP** \( pc \leftarrow \text{mem}[sp]; \) pop link from stack to pc
\( sp \leftarrow sp + 1 \times 4 \)

**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
Supporting a Kernel

The instructions above were selected to support sequential programs with no memory protection; they do not support an operating system - or an operating system *kernel* running application programs and operating system components.

An interesting question is: what needs to be added to support a kernel?

A key issue is to be able to contain errors in application programs, preventing them giving rise to further errors in other programs or in the kernel itself.

The instruction set architecture has to:

- protect the kernel from an error in an application program
- protect an application program from an error in another application program
- enable the kernel to remove failed application programs
- protect the external environment from errors in applications programs (by performing input and output via the kernel)
- enable the kernel to allocate resources such as memory and input-output to applications programs

A starting point is to provide some registers to define the region of memory used by a currently executing application program, along with a (boolean) register to record whether the processor is executing kernel software or application software:

```plaintext
register use

ab  the base address of the application memory
as  the size of the application memory

ink  the executing in kernel flag
```

Some instructions are needed to set these registers. They must only be executed by the kernel; otherwise an application program could change its own memory region.

- **KSETAB** if *ink* then *ab ← s* else *error*  set application base
- **KSETAS** if *ink* then *as ← s* else *error*  set application size
This makes it possible to re-define some of the instructions so as to prevent an application program from corrupting (or branching into) the kernel; for example:

**STR**

if \(ink\)
then \(mem[b + i] \leftarrow s\)
else
if \((b + i) < as\)
then \(mem[ab + b + i] \leftarrow s\)
else \textit{error}

**LDR**

if \(ink\)
then \(d \leftarrow mem[b + i]\)
else
if \((b + i) < as\)
then \(d \leftarrow mem[ab + b + i]\)
else \textit{error}

**BU**

if \(ink\)
then \(pc \leftarrow pc + s11\)
else
if \((pc + s11) < as\)
then \(pc \leftarrow pc + s11\)
else \textit{error}

All of the instructions that access memory must be modified in this way, including the stack access and \textit{PUSH} and \textit{POP} instructions; also all of the branch instructions must be modified.

Notice that this has resulted in application programs being \textit{relocatable}; they can be moved around in memory by the kernel because all of the addresses they use are offsets relative to the \textit{ab} register.

**Switching to the Kernel**

There are three potential reasons for switching to the kernel:

- An error has been detected in an application program
- An application program has made a request to the kernel (a \textit{system call})
- An input-output device has made a request to the processor

All of these perform a similar operation on kernel entry, selecting different addresses within the kernel corresponding to the different reasons for entry. The
same kernel return instruction can be used to return to an application program (which may be different from the one on kernel entry) regardless of the reason for switching to the kernel.

It is not possible to use the link register $lr$ to hold the return address when switching to the kernel because it may already be in use having been loaded by a BL instruction but not yet saved by a PUSH instruction. Consequently, a new register $spc$ is needed to hold the saved $pc$.

- **error**  
  \[
  \text{error} \quad spc \leftarrow pc; \\
  \text{kernel entry from error} \\
  pc \leftarrow \text{mem}[kepe + e] \\
  ink \leftarrow \text{true}
  \]

- **KCALL**  
  \[
  \text{KCALL} \quad spc \leftarrow pc; \\
  \text{kernel entry from system call} \\
  pc \leftarrow \text{mem}[kepec + u8] \\
  ink \leftarrow \text{true}
  \]

- **intreq**  
  \[
  \text{intreq} \quad spc \leftarrow pc; \\
  \text{kernel entry from input-output request} \\
  pc \leftarrow \text{mem}[kepi + i] \\
  ink \leftarrow \text{true}
  \]

- **KRET**  
  \[
  \text{KRET} \quad pc \leftarrow spc \\
  \text{kernel exit} \\
  ink \leftarrow \text{false}
  \]

Finally, it should be possible to write the kernel itself in a high level language using the stack in the normal way. This requires a register to define the top location of the kernel stack:

- **register**  
  \[
  \text{use} \\
  \text{the top location of the kernel stack}
  \]

It also requires some instructions to switch to and from the kernel stack:

- **KSETSP**  
  \[
  \text{KSETSP} \quad \text{if} \ ink \text{ then } \{ \text{mem}[ksp] \leftarrow sp; sp \leftarrow ksp \} \text{ else } \text{error};
  \]

- **KRESTSP**  
  \[
  \text{KRESTSP} \quad \text{if} \ ink \text{ then } sp \leftarrow \text{mem}[sp] \text{ else } \text{error};
  \]

and to save and restore the $spc$, $lr$ and $rs$ registers:
A typical kernel procedure would look like:

KSETSP switch to kernel stack
PUSHSPC save the application spc
PUSH save the application lr
PUSHRS save the application rs
... procedure body here
POPRS restore the application rs
POPLR restore the application lr
POPSPC restore the application spc
KRESTSP switch to application stack
KRET return to application

Interrupts

An interrupt can, in principle, occur between any two instructions. However, there are places where this would cause difficulties:

- during entry to the kernel as a result of a system call or an error, but before the spc (for example) has been saved
- during a kernel instruction sequence which is modifying a data structure (such as a buffer) used to communicate with the interrupting input-output device.

One way to avoid these problems is not to permit interrupts when executing the kernel. An interrupt request from an input-output device is accepted by the processor only when ink is false. This means that, for responsive input-output, the
kernel procedures must be kept short; some of the longer procedures can be treated as application programs.
Instruction set summary

LDRSPI \( d \leftarrow \text{mem}[sp + u8 \times 4] \) load word from stack
STRSPI \( \text{mem}[sp + u8 \times 4] \leftarrow s \) store word to stack
ADDSPI \( d \leftarrow sp + u8 \times 4 \) load address of word in stack
MOVI \( d \leftarrow u8 \) load constant
LDRPCI \( d \leftarrow \text{mem}[pc + u8 \times 4] \) load word from constant pool
ADDPCI \( d \leftarrow pc + u8 \times 4 \) load address in constant pool
LDRI \( d \leftarrow \text{mem}[b + u5 \times 4] \) load word immediate offset
STRI \( \text{mem}[b + u5 \times 4] \leftarrow s \) store word immediate offset
LDR \( d \leftarrow \text{mem}[b + i] \) load word
STR \( \text{mem}[b + i] \leftarrow s \) store word

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
MULR \( d \leftarrow d \times r \) multiply
NEGR \( d \leftarrow -s \) negate
ANDR \( d \leftarrow d \land r \) and
ORR \( d \leftarrow d \lor r \) 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 \( d \leftarrow l >>_{\text{sgn}} u5 \) arithmetic shift right immediate
BEQ  if $rs = 0$ then $pc ← pc + s8_{x2}$  branch relative equal
BNE  if $rs \neq 0$ then $pc ← pc + s8_{x2}$  branch relative not equal
BLT  if $rs < 0$ then $pc ← pc + s8_{x2}$  branch relative less than
BGE  if $rs \geq 0$ then $pc ← pc + s8_{x2}$  branch relative greater or equal
BU   $pc ← pc + s11_{x2}$  branch relative unconditional
BL   $lr ← pc; pc ← pc + s22_{x2}$  branch and link relative
BLR  $lr ← pc; pc ← s$  branch and link via register

PUSH  $sp ← sp - 1_{x4}$;  push link to stack
      $mem[sp] ← lr$
POP   $pc ← mem[sp]$;  pop link from stack to pc
      $sp ← sp + 1_{x4}$
DECSP $sp ← sp + u7_{x4}$  extend stack
INCSP $sp ← sp - u7_{x4}$  contract stack
SETSP $sp ← s$  set stack pointer