Abstract

Summary of “Minimal ARM32” machine architecture, assembler language subset, and runtime library.

1 Architecture

This section summarizes the general architecture of the MinARM32 subset.

1.1 Registers

The MinARM32 has the same registers as the user ARM32 architecture with the 32-bit registers described in Table 1 along with their role in the calling convention. Note that PC is special: when used as a value it has the address of the current instruction plus 8, however, when stored into, it should be set to the address of the next instruction to execute.

<table>
<thead>
<tr>
<th>Register</th>
<th>Use</th>
<th>On Call Entry</th>
<th>On Call Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0–1</td>
<td>general purpose</td>
<td>parameter or undefined</td>
<td>return value or undefined</td>
</tr>
<tr>
<td>R2–3</td>
<td>general purpose</td>
<td>parameter or undefined</td>
<td>–</td>
</tr>
<tr>
<td>R4–11</td>
<td>general purpose</td>
<td>–</td>
<td>same as on entry</td>
</tr>
<tr>
<td>R12</td>
<td>frame pointer</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>SP</td>
<td>stack pointer</td>
<td>address of lowest used stack entry</td>
<td>same as on entry</td>
</tr>
<tr>
<td>LR</td>
<td>link register</td>
<td>return address</td>
<td>–</td>
</tr>
<tr>
<td>PC</td>
<td>instruction address + 8</td>
<td>start of callee</td>
<td>continuation in caller</td>
</tr>
</tbody>
</table>

Table 1: Register use and calling conventions.

1.2 Memory

The MinARM32 memory consists of 8-bit bytes, organized into words of four bytes (thus 32 bits). A word must occupy four consecutive bytes starting from an address divisible by four, and is stored in “little endian” form with the least significant byte in the lowest address.
The MinARM32 memory has the following structure:

```
Static          Code&Data
                  lowest address

Dynamic

Heap
↓
Free Memory
↑
Stack
highest address
```

By convention, the SP register points to the most recently pushed word on the stack, which is also the lowest memory address used by the stack, which grows downwards.

The code and data parts of the memory are populated by *directives* and *instructions*, explained in the following section. A code and data word can hold one instruction or one integer value. An integer value is interpreted either as a 32-bit bit pattern or a 2-complement signed integer with values between \(-2^{31}\) and \(2^{31} - 1\).

### 1.3 Calling Convention

Finally, we summarize the calling convention used by MinARM32 when one function calls another. The code making the call is called the *caller*, and the function which is being called is the *callee*. The register conventions were summarized in Table 1 above, with the following additional rules.

1. Before the call, the caller must ensure that
   
   (a) The words with addresses less that the value of SP are unused and may be overwritten by the callee.
   (b) The (at most) first four parameter words are stored in R0–R3 (further parameters can be stored on the stack).
   (c) If the caller is using the frame pointer (R12) then it should be pushed on the stack at this point.

2. The call itself is a BL instruction that branches to the first instruction of the callee.

3. Before executing any other code, the callee must make sure that the *entry values* of R4–R11, SP, and LR, are recorded, such that they can be retrieved later. One way to achieve this is to execute the instruction

   \[
   \text{STMFD SP}, \{R4-R11,LR\}
   \]

   and in addition make sure that SP is used in a balanced way.

4. When the callee is finished, it should make sure any result words are in R0 and R1 and then restore the entry values of the required registers and branch to the entry value of the LR register (which is the return address). One way to do this if the STMFD instruction above was used is to make sure (by other means) that the SP register has the same value as after that instruction and then execute

   \[
   \text{LDMFD SP}, \{R4-R11,PC}\]
that restores the registers except LR, which is instead loaded into the PC, effectively jumping back to the caller.

5. The caller receives control back at the instruction immediately following the BL instruction, and has access to two result words in R0 and R1 and the same values of R4–R11 and SP as just before the call. The values of R2, R3, R12, and LR, are not defined. If the caller is using the frame pointer (R12) then it can be restored from the stack here (as discussed in the first point).

2 Assembly

The “assembly” language of MinARM32 is simply a mechanism of keeping track of labels and filling in the static area of memory with data and code. It is the programmer’s responsibility to follow the runtime conventions described in the following section.

2.1 Directives

MinARM32 supports the directives in Table 2. Note that all labels must point to a word-aligned address, i.e., an address divisible by four. In addition, MinARM32 allows C style comments (from // to the end of the line, and within /* . . . */.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEF ℓ = n</td>
<td>the label ℓ is defined as the absolute address n</td>
</tr>
<tr>
<td>ℓ :</td>
<td>the label ℓ is set to the next address in memory</td>
</tr>
<tr>
<td>DCS &quot;string&quot;</td>
<td>stores the individual characters of string as consecutive bytes</td>
</tr>
<tr>
<td>DCI n</td>
<td>store the integer n into the next consecutive word</td>
</tr>
<tr>
<td>op</td>
<td>insert the word encoding of the instruction op into the next word</td>
</tr>
</tbody>
</table>

Table 2: Directives.

2.2 Instructions

Instructions fall in a couple of groups, described below. Common to the groups are the following notations:

- r refers to any of the registers in the previous section.
- ℓ denotes a label.
- arg refers to one of these value forms:
  - #n : the “immediate” value n (0 ≤ n ≤ 255).
  - &ℓ : the memory address denoted by the label ℓ.
  - r : the value in the indicated register.
  - r, LSL #n : the value in the indicated register shifted left by n bits (0 ≤ n ≤ 31).
  - r, LSR #n : the value in the indicated register shifted right by n bits (0 ≤ n < 31).

2.2.1 Data Processing

Table 3 summarizes the MinARM32 data processing instructions: Notice the two exceptions: MOV and MVN take only two registers, and MUL takes only registers arguments.

http://cs.nyu.edu/courses/fall15/CSCI-GA.2130-001/MinARM32.pdf
### Table 3: Data Processing Instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV ( r_d, \text{arg} )</td>
<td>( r_d := \text{arg} )</td>
<td></td>
</tr>
<tr>
<td>MVN ( r_d, \text{arg} )</td>
<td>( r_d := \sim \text{arg} )</td>
<td>bitwise not</td>
</tr>
<tr>
<td>ADD ( r_d, r_1, \text{arg} )</td>
<td>( r_d := r_1 + \text{arg} )</td>
<td></td>
</tr>
<tr>
<td>SUB ( r_d, r_1, \text{arg} )</td>
<td>( r_d := r_1 - \text{arg} )</td>
<td></td>
</tr>
<tr>
<td>RSB ( r_d, r_1, \text{arg} )</td>
<td>( r_d := \text{arg} - r_1 )</td>
<td></td>
</tr>
<tr>
<td>AND ( r_d, r_1, \text{arg} )</td>
<td>( r_d := r_1 \land \text{arg} )</td>
<td>bitwise and</td>
</tr>
<tr>
<td>ORR ( r_d, r_1, \text{arg} )</td>
<td>( r_d := r_1 \lor \text{arg} )</td>
<td>bitwise or</td>
</tr>
<tr>
<td>EOR ( r_d, r_1, \text{arg} )</td>
<td>( r_d := r_1 \oplus \text{arg} )</td>
<td>bitwise exclusive or</td>
</tr>
<tr>
<td>MUL ( r_d, r_1, r_2 )</td>
<td>( r_d := r_1 \times r_2 )</td>
<td></td>
</tr>
</tbody>
</table>

### Table 4: Load/Store instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR ( r, \text{mem} )</td>
<td>( r := m_4[\text{mem}] )</td>
<td>load 4 bytes in little endian order</td>
</tr>
<tr>
<td>STR ( r, \text{mem} )</td>
<td>( m_4[\text{mem}] := r )</td>
<td>store 4 bytes in little endian order</td>
</tr>
<tr>
<td>LDRB ( r, \text{mem} )</td>
<td>( r := m_1[\text{mem}] )</td>
<td>load one byte from memory into least significant byte of register</td>
</tr>
<tr>
<td>STRB ( r, \text{mem} )</td>
<td>( m_1[\text{mem}] := r )</td>
<td>store least significant byte of register into memory</td>
</tr>
</tbody>
</table>

### Table 5: Load/Store Multiple Instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDMFD ( !, {mreg} )</td>
<td>pop all registers in ( mreg ) from stack with ( r ) as stack pointer</td>
</tr>
<tr>
<td>STMFD ( !, {mreg} )</td>
<td>push all registers in ( mreg ) onto stack with ( r ) as stack pointer</td>
</tr>
</tbody>
</table>

### Table 6: Compare Instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP ( r_1, \text{arg} )</td>
<td>cond := r_1 ? \text{arg}</td>
<td></td>
</tr>
</tbody>
</table>

### Table 7: Branch instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>B ( \ell )</td>
<td>( \text{PC} := \ell )</td>
<td>Tests last CMP with = for ?</td>
</tr>
<tr>
<td>BEQ ( \ell )</td>
<td>if cond(=) then ( \text{PC} := \ell )</td>
<td>Tests last CMP with = ?</td>
</tr>
<tr>
<td>BNE ( \ell )</td>
<td>if cond(#) then ( \text{PC} := \ell )</td>
<td>Tests last CMP with # #</td>
</tr>
<tr>
<td>BGT ( \ell )</td>
<td>if cond(&gt;#) then ( \text{PC} := \ell )</td>
<td>Tests last CMP with &gt; #</td>
</tr>
<tr>
<td>BLT ( \ell )</td>
<td>if cond(&lt;#) then ( \text{PC} := \ell )</td>
<td>Tests last CMP with &lt; #</td>
</tr>
<tr>
<td>BGE ( \ell )</td>
<td>if cond(##) then ( \text{PC} := \ell )</td>
<td>Tests last CMP with # #</td>
</tr>
<tr>
<td>BLE ( \ell )</td>
<td>if cond(##) then ( \text{PC} := \ell )</td>
<td>Tests last CMP with # #</td>
</tr>
<tr>
<td>BL ( \ell )</td>
<td>( \text{LR} := \text{PC}; \text{PC} := \ell )</td>
<td>Tests last CMP with # #</td>
</tr>
</tbody>
</table>

[http://cs.nyu.edu/courses/fall15/CSCI-GA.2130-001/MinARM32.pdf](http://cs.nyu.edu/courses/fall15/CSCI-GA.2130-001/MinARM32.pdf)
2.2.2 Load and Store

The load and store instructions take care of the communication with main memory. Each instruction is parameterized by the register(s) to load and store, and the address in memory where this should happen. Table 4 gives the details, where “$m_n[...]$” denotes the array of all memory units on $n$ bytes indexed by the address of their lowest byte, and $mem$ denotes an address in memory in one of the following forms:

- $[r, #n]$ – address is $r + n$ for $-4096 \leq n \leq 4095$.
- $[r, &\ell]$ – address is $r + \ell$.
- $[r, \pm r']$ – address is $r \pm r'$ with $\pm$ meaning + or $-$.
- $[r, \pm r', LSL \ #n]$ – address is $r \pm (r' \times 2^n)$ for $1 \leq n \leq 31$.
- $[r, \pm r', LSR \ #n]$ – address is $r \pm (r' \times 2^n)$ for $1 \leq n \leq 31$.

These are similar to but in fact not quite the same as what we could write as $[r, \text{arg}]$.

2.2.3 Load and Store Multiple

The load and store multiple instructions provides a simple way to load and store any subset of the registers from or onto a stack. Table 5 gives the form of the instruction, where

- $mreg$ stands for a set of registers separated by commas.

See the Calling Conventions below for the main use of these instructions.

2.2.4 Compare

MinARM32 is equipped with a subset of the ARM32 condition bits in the form of a condition state. The condition status is set by the comparison instruction in Table 6. The condition state is used by the conditional branch instructions described next, where the conditional instructions can insert a specific test for the $? in the effect description.

2.2.5 Branching

MinARM32 supports the branching instructions in Table 7. Note that it is also possible to use most of the other instructions as a branching instruction by designating PC as the target register. See the calling conventions below for the main use of the BL instruction.

2.3 Example

The small C function

```c
int addbig(int one, int two) {
    return one*1000 + (two > 0 ? two : 1000);
}
```

can, for example, be implemented with the schematic MinARM32 code
// R0=one, R1=two
addbig:  MOV R12,SP       // set frame pointer
        STMFD SP!, {R4-R11,LR} // save caller context
        MOV R4, #0       // R4 := 0
        LDR R4, [R4,&thousand] // R4 := 1000
        MUL R5, R0, R4 // R5 := one*1000
        CMP R1, #0
        BLZ else       // goto 'else' if two \leq 0
        ADD R0, R5, R1 // R0 := one*1000 + two
        B end
else:   ADD R0, R5, R4 // R0 := one*1000 + 1000
end:    LDMFD SP!, {R4-R11,PC} // restore caller context

thousand DCI 1000

Since the function uses very few registers and does not access it's frame, it can be simplified to

// R0=one, R1=two
addbig:  MOV R2, #0       // R2 := 0
        LDR R2, [R2,&thousand] // R2 := 1000
        MUL R3, R0, R2 // R3 := one*1000
        CMP R1, #0
        BLZ else       // goto 'else' if two \leq 0
        ADD R0, R3, R1 // R0 := one*1000 + two
        B end
else:   ADD R0, R3, R2 // R0 := one*1000 + 1000
end:    MOV PC, LR

thousand DCI 1000

where we exploit that the code does not need to change R4–R11 and SP.

3 Runtime

The MinARM32 runtime is specified as follows.

3.1 Initial code

The system will jump to the first address in the static area and start executing there, so put code first in the assembly input. The code should otherwise follow the calling convention.

3.2 Library

The system has access to the following library functions:

int div(int numerator, int denominator);  // integer division
int mod(int numerator, int denominator);  // integer remainder
int length(char* string) // length of zero-terminated string
void* malloc(int size) // newly allocated area on heap
char* substr(char* string, int start, int length) // allocate substring on heap
char* itoa(int number) // allocate string representation of number on heap
int atoi(char* string) // numeric value of as much of string as possible
void free(void* p) // free previously allocated heap area