Minimal ARM32 Assembler

Kristoffer Rose
krisrose@cs.nyu.edu

Wednesday 10th December, 2014

This document summarizes the “Minimal ARM32” assembler language subset.

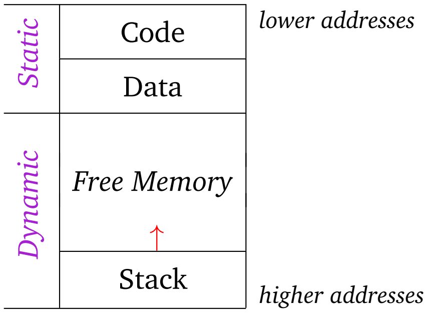
1 Registers

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.

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:



By convention, the stack starts at the address contained in the SP register, and grows downwards.

The code and data parts of the memory are populated by *directives* and *instructions*, explained in the following sections. 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 $2^{31} - 1$.

3 Directives

MinARM32 supports the directives in Table 2. In addition, MinARM32 allows C style comments (from // to the end of the line, and within /*...*/.

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

Table 1: Register use and calling conventions.

Directive	*Meaning*
ℓ	the label ℓ is set to the next address in memory
$\ell = n$	the label ℓ is set to the value of the integer n
DCI $n_1, \ldots, n_k$	store the integers $n_1, \ldots, n_k$ ($k \geq 1$) into consecutive words
op	insert the word encoding of the instruction *op* into the next word

Table 2: Directives.

4 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.
- *arg* refers to one of these value forms:
  - $\#n$ – the "immediate" value n ($0 \leq n \leq 255$).
  - $\&\ell$ – the value of the defined label ℓ.
  - r – the value in the indicated register.
  - r, LSL $\#n$ – the value in the indicated register shifted left by n bits ($0 \leq n \leq 31$).
  - r, LSR $\#n$ – the value in the indicated register shifted right by n bits ($0 \leq n < 31$).
- ℓ denotes a *label*.

4.1 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.

4.2 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[. . .]" denotes the array of all memory words 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 $-4096 \leq n \leq 4095$.

Instruction	*Effect*	*Notes*
MOV r_d, *arg*	$r_d := arg$	
MVN r_d, *arg*	$r_d := \sim arg$	bitwise not
ADD r_d, r_1, *arg*	$r_d := r_1 + arg$	
SUB r_d, r_1, *arg*	$r_d := r_1 - arg$	
AND r_d, r_1, *arg*	$r_d := r_1 \,\&\, arg$	bitwise and
ORR r_d, r_1, *arg*	$r_d := r_1 \mid arg$	bitwise or
EOR r_d, r_1, *arg*	$r_d := r_1 \hat{}\, arg$	bitwise exclusive or
MUL r_d, r_1, r_2	$r_d := r_1 \times r_2$	

Table 3: Data Processing Instructions.

Instruction	*Effect*	*Notes*
LDR r, *mem*	$r :=$ m[*mem*]	
STR r, *mem*	m[*mem*] $:= r$	data moves from *left to right*

Table 4: Load/Store instructions.

Instruction	*Effect*
LDMFD $r!$, {*mreg*}	pop all registers in *mreg* from stack with r as stack pointer
STMFD $r!$, {*mreg*}	push all registers in *mreg* onto stack with r as stack pointer

Table 5: Load/Store Multiple Instructions.

Instruction	*Effect*	*Notes*
CMP r_1, *arg*	*cond* $:= r_1 \,?\, arg$	

Table 6: Compare Instructions.

Instruction	*Effect*	*Notes*
B ℓ	PC $:= \ell$	
BEQ ℓ	**if** *cond*$(=)$ **then** PC $:= \ell$	Tests last CMP with $=$ for $?$
BNE ℓ	**if** *cond*$(\neq)$ **then** PC $:= \ell$	Tests last CMP with $\neq$ for $?$
BGT ℓ	**if** *cond*$(>)$ **then** PC $:= \ell$	Tests last CMP with $>$ for $?$
BLT ℓ	**if** *cond*$(<)$ **then** PC $:= \ell$	Tests last CMP with $<$ for $?$
BGE ℓ	**if** *cond*$(\geq)$ **then** PC $:= \ell$	Tests last CMP with $\geq$ for $?$
BLE ℓ	**if** *cond*$(\leq)$ **then** PC $:= \ell$	Tests last CMP with $\leq$ for $?$
BL ℓ	LR $:=$ PC; PC $:= \ell$	

Table 7: Branch instructions.

- $[r, \&\ell]$ – address is $r + \ell$ for ℓ defined between -4096 and 4095.
- $[r, \pm r']$ – address is $r \pm r'$ with ± meaning + or −.
- $[r, \pm r', \text{LSL}\ \#n]$ – 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 $[r, \mathit{arg}]$.

4.3 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.

4.4 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.

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.

6 Calling Convention

Finally, we summarize the calling convention used by MiniARM32 when one function calls another. The code making the call is called the *caller*, and the function which is being called is the *callee*. In addition to the register conventions summarized in Table 1, the basic rules are these:

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).

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

   STMFD SP!, {R4,R5,R6.R7,R8,R9,R10,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 not changed since then and then execute

   LDMFD SP!, {R4,R5,R6.R7,R8,R9,R10,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.

7 Example

The small C function

```
int addbig(int one, int two) {
    return one*1000 + two;
}
```

can, for example, be implemented with the schematic MinARM32 code

```
// R0=one, R1=two
addbig    STMFD SP!, {R4,R5,R6.R7,R8,R9,R10,R11,LR}
          MOV   R4, #0               // R4 := 0
          LDR   R4, [R4,&thousand]   // R4 := 1000
          MUL   R5, R0, R4           // R5 := one*1000
          ADD   R0, R5, R1           // R0 := one*1000 + two
          LDMFD SP!, {R4,R5,R6.R7,R8,R9,R10,R11,PC}

thousand  DCI 1000
```

Since the function uses very few registers, it can be simplified to

```
// R0=one, R1=two
addbig    MOV   R3, #0               // R3 := 0
          LDR   R3, [R3,&thousand]   // R3 := 1000
          MUL   R3, R0, R3           // R3 := one*1000
          ADD   R0, R3, R1           // R0 := one*1000 + two
          MOV   PC, LR

thousand  DCI 1000
```

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