8

Appendix A: ARM Reference

8.1 Register Mnemonics for A32 Calling Convention

Name	Register	Purpose
a1	0	argument, return value, or scratch
a2	1	argument, return value, or scratch
a3	2	argument, return value, or scratch
a4	3	argument, return value, or scratch
v1	4	local variable
v2	5	local variable
v3	6	local variable
v4	7	local variable
v5	8	local variable
sb	9	static base
sl	10	stack limit
fp	11	frame pointer
ip	12	intra-procedure-call scratch register
sp	13	stack pointer
lr	14	link register (i.e., return address)
pc	15	program counter

All registers can also be referred to generically using `r0`–`r15`. More information can be found at the Wikipedia page on calling conventions or the ARM developer reference on predeclared register names.

8.2 *Status Flags*

ARM instructions sometimes set status codes, in particular arithmetic instructions. Status codes are:

Name	Meaning	Purpose
`n`	negative	Set when the result is negative.
`z`	zero	Set when the result is zero.
`c`	carry	Set when the result of an unsigned operation overflows the 32-bit result register.
`v`	overflow	Same as the `c` flag, but for signed operations.

8.3 A32 Calling Convention

A32 tries to pass function arguments using registers, for speed. The first five local variables are also stored in registers. Whenever there are more arguments or more local variables, allocation *spills* to the stack. The caller is responsible for setting up stack allocation.

If the type of value returned is too large to fit in `a1 to a4`, or whose size cannot be determined statically at compile time, then the caller must allocate space for that value at run time, and pass a pointer to that space in `a1`.

A32 is mostly *callee save*, meaning that the called subroutine (the "callee") is responsible for preserving `v1`–`v5`, `sb`, `sl`, `fp`, and `sp` (i.e., `r4`–`r11` and `r13`). However, the function doing the call (the "caller") is responsible for saving the return address in `lr` (i.e., `r14`) to the stack. In other words, any subroutine that intends to call another subroutine must save the return address found in the link register to the stack before the call is made; `lr` is *caller saved*.

A32 is *full-descending*, meaning that:

- the "bottom" of the stack is allocated at a high address and grows toward lower addresses, and
- the stack pointer, `sp`, points to the location in which the last item was stored; `push` *decrements* `sp` *and then* stores the value.

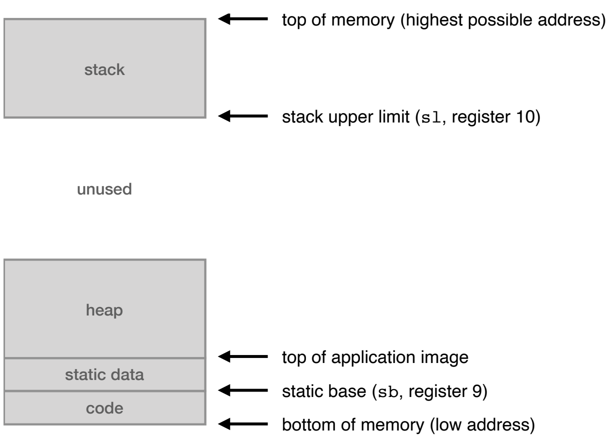

Figure 8.1: Layout of a program's memory.

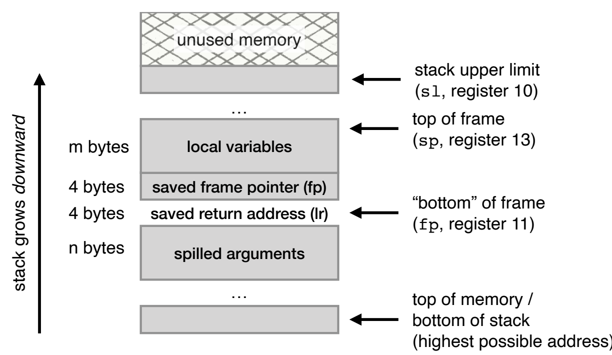

Figure 8.2: Layout of a program's stack.

Figure 8.1 shows a loaded program's virtual memory layout. Figure 8.2 shows a loaded program's stack layout. Note that Figure 8.2 is displayed upside-down for readability; stacks grow *downward*, toward *lower* memory addresses.

Whenever there are too many arguments to fit in registers `a1`–`a4` (i.e., `r0`–`r3`), values are *spilled* (n bytes $= k$ spilled arguments $\times$ 4 bytes) and stored below the `fp`. Local variables and other temporary values are stored above the saved frame pointer and return address. Instructions that access stack memory are usually `fp`-relative.

It may be hard to appreciate by looking at the above diagrams, but a stack containing a sequence of stack frames is a linked list, where the saved frame pointer points to the next (previous) frame.

8.4 Instruction Mnemonics

This manual was adapted from the ARM KEIL developer documentation page. Most modifications omit detail that is not relevant to this class. However, formal syntax has been changed substantially to make the documentation easier to use and more consistent with `gcc`'s assembler output. An extensive set of examples have also been added.

Since this manual glosses over some details for the sake of readability, it has some minor inaccuracies. It also does not include every single instruction. For all the gory details, refer to ARM's official Assembler User Guide.

Typographical conventions.

The first thing to note about ARM assembly is that, when using `gcc`, the syntax is neither "Intel syntax" nor "AT&T syntax." You probably learned AT&T syntax in CSCI 237. Treat ARM assembly as if it were a new programming language, and if you don't understand something, ask about it or look it up. That said, assembly is simple—some would even say simplistic—and ARM assembly is *much* simpler than x86 assembly. You'll likely find that most of what you know translates to ARM with only minor changes in syntax.

The first element in any instruction is the name of the instruction; names are also sometimes referred to as *instruction mnemonics* because the computer itself never sees the name. Mnemonics are translated into *opcodes*, literally numbers, by the assembler.

Argument names are *italicized*. Refer to the definition below an instruction's formal syntax for an explanation of its use.

Optional syntax is <u>underlined</u>. Unusually, not only does ARM assembly have optional *arguments*, it also has optional *instruction suffixes*. Many instruction names have optional suffixes. Putting a suffix on an instruction name changes the meaning of the operation. For example, the `add` instruction can take a suffix, like the `addscs` variant that only adds two numbers when the `cs` flag is set. We describe the *condition code suffixes* below. You will never type an underline in your assembly; this is simply a typographical convention (i.e., abstract syntax) to help you understand which parts of an instruction are optional.

Because ARM instructions can have many variants, it can be hard to tell where spaces should go. Therefore, this guide always puts a visible space character ␣ in the formal syntax definition whenever you should put a space. If there is no ␣, don't put a space there or the assembler won't understand you.

Any assembly starting with a period (`.`) is an *assembler directive*. Assembler directives supply data to the assembler to control the assembly process. They are *not* ARM instructions, and the processor will never see them.

`@` is the start of a comment. Yes, assembly can have comments. Good assembly programmers actually use them!

{Curly braces} denote a list of values. Curly braces are not abstract syntax—you actually have to type them.

Comma characters (`,`) are used in instructions that take multiple mandatory arguments. Commas are not abstract syntax—you actually have to type them.

The hash sign (`#`) denotes that the value succeeding it is an *immediate value*. Immediate values are constants. In most ordinary programming languages, we call these values *literal values*. Hashes are not abstract syntax—you actually have to type them.

Indirect address expressions are enclosed in `[`square brackets`]`. This syntax is used to load an address into a register. Because *all* ARM instructions are 32 bits wide, and the opcode and target take some space, there is no way to directly load a 32-bit address—there just isn't enough space in an instruction. Instead, ARM assembly lets you use an indirect address expression that computes an offset from a known base. The format is `[`*base*, *offset*`]`. For example, `[fp, #-12]` returns the value obtained by subtracting 12 from the address stored in the `fp` register. Square brackets are not abstract syntax—you actually have to type them.

cond denotes a *condition code suffix*. The meaning of the instruction with a condition code depends on the operation. Valid condition code suffixes are:

Code	Meaning
`eq`	equal
`ne`	not equal
`cs`	carry set (same as `hs`)
`hs`	unsigned higher or same (same as `cs`)
`cc`	carry clear (same as `lo`)
`lo`	unsigned lower (same as `cc`)
`mi`	minus or negative result
`pl`	positive or zero result
`vs`	overflow
`vc`	no overflow
`hi`	unsigned higher
`ls`	unsigned lower or same
`ge`	signed greater than or equal
`lt`	signed less than
`gt`	signed greater than
`le`	signed less than or equal
`al`	always (this is the default)

8.4.1 *add*

Add without carry.

Syntax.

    add <u>s</u> <u>*cond*</u> ␣ *rdst* , ␣ *rnum1* , ␣ *num2*
or add <u>*cond*</u> ␣ *rdst* , ␣ *rnum1* , ␣ #*imm12*

where:

`s`
: is an optional suffix. If `s` is appended, condition flags are updated on the result of the operation.

cond
: is an optional condition code.

rdst
: is the destination register.

rnum1
: is the register holding the first operand.

num2
: is either a constant or a register with optional shift.

imm12
: is any value in the range 0-4095.

The `add` instruction adds the values in `rnum1` with `num2` or `imm12`. In certain circumstances, the assembler may substitute one instruction for another. Be aware of this when reading disassembly listings.

Example.

```
add fp, sp, #4
```

adds 4 to the contents of the `sp` register and stores the result in the `fp` register.

8.4.2 *b*

Branch to an address.

Syntax.

b *cond* ␣ *addr*

where:
cond
is an optional condition code.
addr
is a PC-relative expression, like a label.

The b instruction causes a branch to addr. In other words, bl simply "jumps" to another location in the code.

Example.

```
b .L14
```

branches to the instruction given by the assembly label .L14.

8.4.3 *bl*

Branch with link.

Syntax.

`bl` *cond* ␣ *addr*

where:

cond
  is an optional condition code.

addr
  is a PC-relative expression, like a label.

The `bl` instruction copies the address of the next instruction (`pc`+4) into `lr` (`r14`, the link register), and then branches to the given `label`. `bl` is typically used to call a function.

Example.

```
bl time
```

branches to the instruction given by the assembly label `time` and copies the address of the instruction appearing after the `bl` into the `lr` register. In other words, the example calls the `time` function.

8.4.4 *bx*

Branch and exchange instruction set.

Syntax.

`bx` *cond* ␣ *addr*

where:

cond
: is an optional condition code.

addr
: is a PC-relative expression, like a label.

The `bx` instruction branches to the given `addr`. If the least significant bit of the given address is 1, then switch into *Thumb mode*, otherwise stay in *ARM mode*. `bx` is typically used to return from a function.

Example.

```
bx lr
```

branches to the instruction stored in the `lr` register. In other words, the example returns from the current function.

8.4.5 *cmp*

Compares two values.

Syntax.

`cmp` *cond* ␣ *rnum1* , *num2*

where:

cond
: is an optional condition code.

rnum1
: is a register containing the first value.

num2
: is either a constant or a register with optional shift.

`cmp` compares the value in a register with *num2*. It updates the condition flags on the result, but does not place the result in any register. The `cmp` instruction subtracts the value of *num2* from the value in *rnum1*. This is the same as a `subs` instruction, except that the result is discarded. The `n`, `z`, `c` and `v` flags are updated according to the result.

Example.

```
cmp r3, #0
```

compares the value in the register `r3` with 0. If the two are equal,

8.4.6 *eor*

Bitwise exclusive or.

Syntax.

`eor` <u>`s`</u> <u>*cond*</u> ␣ *rdst* `,` ␣ *rnum1* `,` ␣ *num2*
where:
`s`
is an optional suffix. If `s` is appended, condition flags are updated on the result of the operation.
cond
is an optional condition code.
rdst
is the destination register.
rnum1
is the register holding the first operand.
num2
is either a constant or a register with optional shift.
imm12
is any value in the range 0-4095.

The `eor` instruction performs a bitwise exclusive OR operation on the values in *rnum1* and *num2*, storing it in *rdst*.

Example.

```
b .L14
```

branches to the instruction given by the assembly label `.L14`.

8.4.7 *ldr*

Copies a value into a register. Unlike `mov`, the `ldr` instruction loads values indirectly. This instruction is useful for loading values that must be 32 bits wide, like addresses. Values are loaded *from* a target address. To fit this instruction into 32 bits, the assembler computes a target address relative to the program counter (`pc`).

Syntax.

`ldr` <u>*cond*</u> ␣ *rdst* `,` ␣ *addr*

where:
cond
 is an optional condition code.
rdst
 is the register to be loaded.
addr
 is a label or a numeric value.

When using `pc`-relative address, the "true value" of the `pc` is two instructions ahead of the address of the executing instruction (4 bytes per instruction $\times$ 2 instructions $=$ 8 bytes). The reason for this inconsistency is because `pc`-relative addressing occurs after an instruction has progressed through the ARM processor's instruction pipeline.

Example 1.

```
ldr r0, .L16
```

loads the *the address of the label* `.L16` into the `r0` register.

Example 2.

```
ldr r0, .L16+4
```

loads the *the address of the label* `.L16` *plus 4* into the `r0` register.

Example 3.

```
ldr r1, [fp, #-12]
```

loads the data using an indirect address expression. This example loads the value *stored in the frame pointer* (`fp`) *minus 12* into the `r0` register.

8.4.8 *mov*

Copies a value into a register. Note that, because of space reasons, `mov` is limited to register-to-register copies, or 16-bit immediate values. To copy larger values, like addresses, use `ldr`.

Syntax.

`mov` s *cond* ␣ *rdst* , *num2*
or `mov` *cond* ␣ *rdst* , #*imm16*

where:

`s`
: is an optional suffix. If `s` is appended, condition flags are updated on the result of the operation.

cond
: is an optional condition code.

rdst
: is the destination register.

num2
: is either a constant or a register with optional shift.

imm16
: is any value in the range 0-65535.

The `mov` instruction copies the value of `num2` or `#imm16` into `rdst`. In certain circumstances, the assembler may substitute `mvn` for `mov`, or `mov` for `mvn`. Be aware of this when reading disassembly listings.

Example.

```
mov r0, #0
```

stores 0 into the `r0` register.

8.4.9 *pop*

Pops registers off of a full-descending stack.

Syntax.

`pop` *cond* ␣ *regset*

where:

cond

is an optional condition code.

regset

is a non-empty set of registers, enclosed in curly braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range. The order that `pop` processes pops is *register order*.

Be aware of the order that `pop` processes values. A simple mnemonic to remember the order is “low addresses go in low registers.” In other words, the value at the *top* of the stack (the lowest address in a full-descending stack) goes in the register with the lowest register number in the given *regset*.

Example.

```
pop {fp, pc}
```

pops two values off the stack and stores them in the `fp` and `pc` registers. Since `fp` (register 11) comes before `pc` (register 15) in register order, `pop` stores the first pop in `fp` and the second pop in `pc`. Here, the contents of `sp` will be stored in `fp`, the contents of `sp`+4 will be stored in `pc`, and `sp` will be updated to `sp`+8.

8.4.10 *push*

Pushes registers onto a full-descending stack.

Syntax.

`push` *cond* ␣ *regset*

where:

cond
is an optional condition code.

regset
is a non-empty list of registers, enclosed in curly braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range. The order that `push` processes pushes is *reverse register order*.

Be aware of the order that `push` processes values. A simple mnemonic to remember the order is "low addresses go in low registers." This is the same rule that `pop` uses. In other words, the register with the lowest register number in the given *regset* will be stored at the *top* of the stack (the lowest address in a full-descending stack).

Example.

```
push {fp, lr}
```

pushes the `fp` and `lr` registers onto the stack. Since `lr` (register 14) comes after `fp` (register 11) in register order, `push` pushes `lr` first and `fp` second. The contents of `lr` will be stored at `sp`-4, the contents of `fp` will be stored at `sp`-8, and `sp` will be updated to `sp`-8.

8.4.11 *str*

Copies a value from a register into memory.

Syntax.

str <u>*type*</u> <u>*cond*</u> ␣ *rsrc* , ␣ *addr*

where:

type
  can be any one of

- `B`, an unsigned byte (zero extended to 32 bits on loads);
- `H`, an unsigned halfword (zero extended to 32 bits on loads); or
- *omitted*, the default, which is a 32-bit word.

cond
  is an optional condition code.

rsrc
  is the register to load the value from.

addr
  is a label or a numeric value, denoting the location to store the loaded value.

Example.

```
str r0, [fp, #-8]
```

stores the value in the `r0` register into the address *stored in the frame pointer* (`fp`) *minus 8*.

8.4.12 *sub*

Subtract without carry.

Syntax.

```
   sub s cond ␣ rdst, ␣ rnum1, ␣ num2
or sub cond ␣ rdst, ␣ rnum1, ␣ #imm12
```

where:

`s`
: is an optional suffix. If `s` is appended, condition flags are updated on the result of the operation.

cond
: is an optional condition code.

rdst
: is the destination register.

rnum1
: is the register holding the first operand.

num2
: is either a constant or a register with optional shift.

imm12
: is any value in the range 0-4095.

The `sub` instruction subtracts the value of *num2* or *imm12* from the value in *rnum1*. In certain circumstances, the assembler may substitute one instruction for another. Be aware of this when reading disassembly listings.

Example.

```
sub sp, sp, #16
```

subtracts 16 from the contents of the `sp` register and stores the result in the `sp` register.

8.4.13 *uxtb*

Zero extend byte.

Syntax.

`uxtb` *cond* ␣ *rdst* `,` ␣ *rnum* `,` ␣ *rot*

where:

`s`
: is an optional suffix. If `s` is appended, condition flags are updated on the result of the operation.

cond
: is an optional condition code.

rdst
: is the destination register.

rnum
: is the register holding the byte.

rot
: can be any one of

- `ror #8`, meaning that *rnum* is rotated right 8 bits;
- `ror #16`, meaning that *rnum* is rotated right 16 bits;
- `ror #24`, meaning that *rnum* is rotated right 24 bits; or
- *omitted*, for no rotation.

`utxb` extends an 8-bit value to a 32-bit value. It does this by

1. rotating the value from *rnum* right by 0, 8, 16, or 24 bits;
2. extracting bits [7:0] from the value obtained; and
3. zero extending to 32 bits.

Example.

```
uxtb r3, r3
```

zero-extends the byte stored in register `r3` and stores the result in register `r3`.