AASM Manual (with School of CS, U. of Nottm., additions)

(This is more of a set of notes than a true manual.)

Beta - release!

Compiling the assembler

This has already been done for you within the School of CS at Nottingham.

However if you take the source code ("aasm.c"), or binary ("aasm") for use at home on your own Linux machine, then note that when running "aasm" you need the following to be accessible:

binary file (aasm), token specifier file (mnemonics), source file (aasm.c)

If you collect the correct (64-bit or 32-bit) binary then it should work under Linux at home but if you do need to recompile then note that the source is in a single file. To compile it, simply use your favourite C compiler:

e.g. "gcc -O2 -o aasm aasm.c"

Command line format

```
aasm [-s[d,v][{l, p}], <file>, -l <file>, -h <file>, -e <file>] <sourcefile>

-s dumps the symbol table to the specified file.
      the default is to dump in alphabetical order
      the -sd option dumps symbols in order of Definition
      the -sv option sorts symbols into ascending Value
            adding an 'l' will dump the Local labels too
            adding a 'p' will dump the literal Pools too
-l dumps the list output to the specified file.
      the -ls option will list the symbol table too
      the -lk option will list the symbol table and insert a "KMD" identifier
-h dumps ASCII hexadecimal to the specified file.
-e dumps ELF to the specified file.
omitting the filename (or substituting '-') directs to stdout.
```

Automated settings

The Komodo emulator has a button at the top right called "Compile". When activated with your chosen input filename (e.g. myfile.s) this obeys a Unix shell script called kmd_compile. Here is the version of kmd_compile I am using at home:

```
      #!/usr/local/bin/bash
      KOMODO_HOME=/home/dfb/komodo
      FLNME=`echo $1 | sed s/[.]s$//`
      $KOMODO_HOME/aasm -lk ${FLNME}.kmd $1
```

Notice that the issue of having access to the files aasm, aasm.c and mnemonics at assembly time (as mentioned in the first section above) is handled by having them all be present in a directory held in the shell variable KOMODO_HOME. Notice too how the last two lines of the script ensure that if your input file is called file1.s then the binary

(i.e. 'executable') file produced will have thesamename but with extension '.kmd'.

A similar *kmd_compile* setup is present in the School of Computer Science, University of Nottingham, but *kmd_compile* is located in /usr/local/bin rather than in your home directory.

Emulator SWIs

~~~~~~~~~~~~~

Currently the only feature requested by the kmd emulator is a terminal. To help you in putting out results and/or reading in data the following calls are available which are NOT true assembler primitives. Instead they are implemented in the emulator itself to make life easier for you.

The following SWI calls are trapped from causing the normal SWI behaviour.

SWI 0 Outputs the least significant byte of R0 to the terminal
SWI 1 Inputs character typed in the terminal window to the least significant byte of R0
SWI 2 Halts execution
SWI 3 Prints a string, pointed to by R0
SWI 4 Prints the value of R0 in decimal
Other SWIs trap to a handler, as would be expected.

(These functions were implemented to support teaching laboratories at Univ. of Manchester. They are equally useful for our purposes. They are implemented purely in the emulator back end.)

Output information

~~~~~~~~~~~~~~~~~~

Each assembler pass attempts to define and refine label values. On each pass information is echoed indicating "Label changes":

"defined" is the number of labels which were defined for the first time on that pass.

"value changed" is the number of labels whose definition has been altered on that pass; this is normally due to the object code changing size as offsets are adjusted.

"read while undefined" indicates references to labels which have not yet had any definition.

Iteration continues until all these values are zero - at which time a final pass generates code - or the assembler becomes fed up, which implies that the source code is not sensible. In this case labels still undefined are reported.

Errors are reported as they are determined and, if a pass contains errors, assembly will be aborted at the end of the pass.

Source file input format

Standard assembly language: optional label followed by a mnemonic - with addressing mode(s) as required - followed by an optional comment.

Any field may be omitted where sensible.

No restrictions on mnemonic/label formatting (1st column, etc.)

Mnemonics (including directives) are case insensitive.

Labels are case sensitive.

Lines are terminated by <LF> or <EOF>.

A line is truncated to a limit of 255 characters. (Alterable in source file.)

Directives supported are:

```
INCLUDE <filename>            Include file as part of source
GET     <filename>            Synonym for "INCLUDE"
IMPORT  <filename>            Include binary file in output
EQU   <expr>                  Equate
EXPORT      <label>, <label>, ...   Marks labels for export (for expansion)
                        If <label> = "all", all labels are exported
ARM                     Assemble as 32-bit ARM code (default)
                        (Automatically aligns to word boundary)
THUMB                   Assemble as 16-bit Thumb code
ARCHITECTURE                  Set instruction set architecture (default "ALL")
                        {v3, v3M, v4, v4xM, v4T, v4TxM, v5, v5xM, v5T, v5TxM,
                                v5TE, v5TExP, all, any} (last 2 are synonyms)
ARCH                    Synonym for ARCHITECTURE
ORIGIN      <addr>                  Origin (defaults to 00000000 at start)
ORG                     Synonym for ORIGIN
ENTRY                   Mark code entry point
ALIGN [<expr>[, <fill>]]      Increment origin to next multiple of <expr>
                        Will default to 4-byte alignment
                        Note: any label will be at the first filled address
                        if filling the gap, but the aligned address if not.
DEFS  <size>[, <fill>]  Define space - space uninitialised if <fill>
                        byte omitted
DEFB  <expr>, <expr>, ...     Define byte
DEFB  "string", ...           may mix with above - possible delimiters " ' ` /
                        Accepts C-style escape codes
                        {
DEFH  <expr>, <expr>, ...     Define 16-bit word (also strings as DEFB)
DEFW  <expr>, <expr>, ...     Define 32-bit word (also strings as DEFB)
DCB                     Synonym for DEFB
DCW                     Synonym for DEFH
DCD                     Synonym for DEFW
LITERAL                       Dump any outstanding literal constants
                        Owing to the automatic alignment and
                        user invisibility of literals it is
                        inadvisable to label a LITERAL directive.
LITERALS                Synonym for LITERAL
POOL                    Synonym for LITERAL
LTORG                   Synonym for LITERAL
RN    <reg>             Alias a register name.
                        Note: must precede alias use in assembly

CN    <creg>                  Alias a coprocessor register name.
                        Note: must precede alias use in assembly
```

```
RECORD         [<expr>]       Start a data record [See section below]
STRUCTURE  [<expr>]           Synonym for RECORD
STRUCT         [<expr>]       Synonym for RECORD
ALIAS                   Identify a zero size data element
BYTE     [<expr>]       Identify one byte data element(s)
HALFWORD   [<expr>]           Identify two byte data element(s)
HALF     [<expr>]       Identify two byte data element(s)
WORD     [<expr>]       Identify four byte data element(s)
DOUBLE         [<expr>]       Identify eight byte data element(s)
DOUBLEWORD [<expr>]           Identify eight byte data element(s)
REC_ALIGN  [<expr>]           Align data field modulo <expr> (default 4)
STRUCT_ALIGN [<expr>]         Synonym for REC_ALIGN
```

Note: a value used in (e.g.) DEFB is truncated to the appropriate least significant bits **without** a warning message being generated.

Label rules
~~~~~~~~~~~

Labels are strings of alphanumerics, beginning with an alphabetic character. '_' is regarded as alphabetic. They are case sensitive. A label can be any length, but only the first 31 characters are stored. (This limit can be changed in the source file.) Longer labels are still likely to be distinguished by their hashes.

Local labels may be defined as decimal numbers. The same label may be defined repeatedly, if desired. A local label is referenced using a

```
999   beq   %b111
```

The reference may be specified as "%b##" (back) which searches for the nearest matching preceding definition in the assembly process, "%f##" (forward) which searches for the nearest matching succeeding definition or "%##" which searches backwards then forwards if not found. Both directions of search begin with the label on the current line, if any.

Expression rules
~~~~~~~~~~~~~~~~

An operand may be one of:

```
<label>           label
%<number>   local label
<number>    decimal number
:<number>   binary number                 ????
0b<number>  binary number
@<number>   octal number
$<number>   hexadecimal number
&<number>   hexadecimal number
0x<number>  hexadecimal number
```

Values are 32 bit, two's complement.

Any number may contain an arbitrary number of '_' characters. These are intended to be available as field separators (e.g. in long binary numbers) and are ignored by the assembler.

A monadic operator may be one of:

Symbols	Operation
+	Plus
-	Negative (2's complement)

~	NOT	(1's complement)
|	log	(i.e. the bit number of the MS '1' or -1 for 0)

Only one monadic operator is allowed per element.

A dyadic operator may be one of:

Symbols	Operation	Precedence	
<< LSL SHL	Left shift	7	(High)
>> LSR SHR	Right shift	7	
AND	Logical AND	6	
| OR	Logical OR	5	
^ EOR XOR	Logical XOR	5	
*	Multiply	4	
/ DIV	DIV	4	
MOD	MOD	4	
+	Add	3	
-	Subtract	3	
= EQ	is equal to	2	
<> NE !=	is not equal to	2	
> HI	is higher than	2	
>= HS	is higher or same	2	
< LO	is lower than	2	
<= LS	is lower or same	2	
GT	is greater than	2	
GE	is greater or equal	2	
LT	is less than	2	
LE	is less or equal	2	(Low)

Parentheses may be used to force priority.

Note the symbolic comparison operators (">", etc.) treat operands as unsigned.
Comparison operators return FFFFFFFF (-1) for true and 0 for false.

Conditional assembly
~~~~~~~~~~~~~~~~~~~~

Condition directives are as follows:

IF <expr>	Begin conditional assembly; following code produced if <expr> evaluates to non-zero.
ELSE	Toggle effect of preceding IF clause. (In practice 'ELSE' may be used repeatedly; this is not recommended.)
ENDIF	End conditional assembly following IF/ELSE.
FI	Synonym for ENDIF.

The expression used for the IF must be evaluated correctly on the
first pass, thus any variables used must be defined before it is
encountered.

Conditions may be nested up to a maximun depth (currently 10 levels).
** BUG at present **
These directives are not case sensitive.

Mnemonics
~~~~~~~~~

Standard ARM mnemonics and syntax are supported, with the following additions:

```
NOP                ARM no-operation (MOV R0, R0)
UNDEF              Undefined instruction (?C000010)
UNDEFINED          as above
ADR Rd, addr             Produce an ADD/SUB Rd, PC, ## so that Rd becomes addr
                   Exactly one instruction is generated.
ADRL Rd, addr            Produce a set of operations which sets up Rd
                   by adding or subtracting from the PC.  From
                   one to four instructions may be generated.
ADRn Rd, addr            Produce a set of operations which sets Rd to
                   addr, using exactly n instructions

LDR/STR Rd, addr  PC-relative addressing mode
LDR Rd, =##        MOV/MVN Rd, ## if possible, otherwise plant a
                   PC-relative load to a literal constant.  The
                   literals will be dumped at the end of the file,
                   or earlier with a LITERAL directive.
                   LDRH will generate half word literals.
                   Literals will be aliased to earlier constants
                   of the same value, where possible.
                   Literal pools are aligned automatically, as
                   deemed necessary by the assembler, and
                   terminate on word alignment.

ADDX<cc><s> Rd, Rn, #nn A number of 'extended' immediate operations
                   which are synthesized from real operations but
                   allow arbitrary sized immediate fields.  From
                   one to four instructions may be generated;
                   some substitution (e.g. SUB for ADD) will be
                   attempted if it shortens the sequence.
                   Operations supported are {MOVX, ADDX, ADCX,
                   SUBX, SBCX, RSBX, RSCX, ANDX, BICX, ORRX,
                   EORX}.
                   Conditional operation and flag setting is
                   optional; NOTE that an attempt to set the C or
                   V FLAG will result in an UNDEFINED state as
                   predicting where an overflow may occur is not
                   possible and the flags can only be set on the
                   last instruction in the sequence.
                   These sequences may provide a convenient means
                   of performing some operations but should be
                   used with caution.
                   An advantage of this mechanism over "LDR =" is
                   that the data is fetched from the code stream
                   rather than as data; thus the constants are
                   more easily integrated with instruction
                   caches, etc.

Notes for Thumb:
      NOP is MOV R8, R8 because MOV R0, R0 would affect the flags.
      Only ADR (and ADR1) are currently supported.
```

Data records
~~~~~~~~~~~~

These are primarily intended as an easy way to set up data offsets without planting code. The syntax is as follows:

```
            RECORD
element1    WORD

element2    WORD
element3    BYTE  20
```
~~~~~~~~~~~~

```
element4    HALFWORD
```

This will set the following labels (decimal):

```
element1     0
element2     4
element3     8
element4    28
```

There is no limit to the number of records. ‘RECORD’ may be followed by an expression which is used as the base or, as here, sets the origin to zero by default. Any element may be followed by a multiplier (as ‘BYTE’ is, above) or default to a single element.

Data sizes of 0, 1, 2, 4 and 8 bytes are supported. "ALIAS" is a means of defining a zero length element. Note this needs to *precede* the declaration which reserves the space.

‘REC_ALIGN’ allows realignment onto word (or other) boundaries as desired.

The free-format input style allows definitions to be indented differently, for clarity, if desired.

Register names
~~~~~~~~~~~~~~

Register names are case insensitive. The following register names are predefined:

Name	Name	Name	Value
r0	a1		0
r1	a2		1
r2	a3		2
r3	a4		3
r4	v1		4
r5	v2		5
r6	v3		6
r7	v4		7
r8	v5		8
r9	v6	sb	9
r10	v7	sl	10
r11	fp		11
r12	ip		12
r13	sp		13
r14	lr		14
r15	pc		15

Other aliases (also case insensitive) may be created with the "RN" directive, e.g.

```
acc         rn    r0
```

Coprocessor names are treated similarly. Predefined aliases are "C?" and "CR?" where ’?’ is a decimal number 0-15. The relevant directive is "CN".

Status register names are not aliasable.

Coprocessor names

~~~~~~~~~~~~~~~~~

Predefined coprocessor names are "P?" or "CP?" where ’?’ is a decimal number 0-15. They are case insensitive. Aliases may be defined with the "CP" directive.

Non-literal translation

Some instructions will be modified, if possible, if they cannot be assembled as specified. For example: "ADD R1, R2, #-1" becomes "SUB R1, R2, #1". There are two such sets of pairs:

ADD/SUB and CMP/CMN	Operand negated
AND/BIC, ADC/SBC and MOV/MVN	Operand inverted

As mentioned above, instructions of the form "LDR Rd, =###" will be translated to MOV or MVN if the immediate field will fit in the instruction.

Caveats

If using literal pools (i.e. the LDR R0, =&12345678 syntax) it is advisable to finish the code file with a "LITERAL" directive. Although this will be placed automatically this will ensure that any label(s) -apparently- marking the end of the code block do not exclude the remaining literal pool.

If setting the flags with the extended immediate operation sequences (e.g. "ADDXS") note that the carry and (for arithmetic operations) overflow flags are -undefined-.