

Baby42

A simple RISC processor

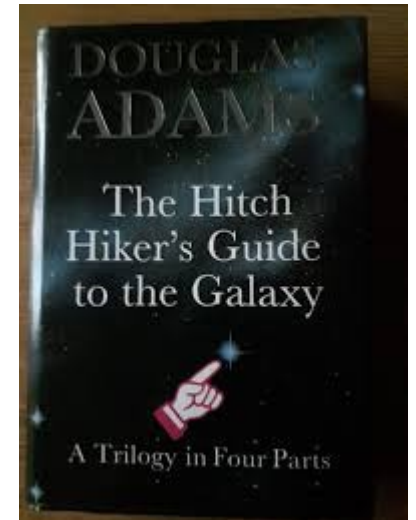
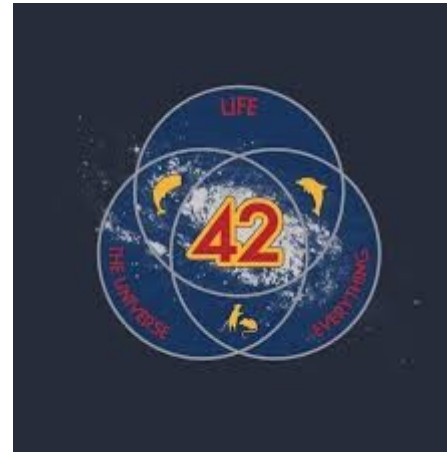
Jecel Mattos de Assumpção Jr
August 2022

name



- The Manchester Baby (Small-Scale Experimental Machine – SSEM) ran a stored memory program in June 1948

- 4 bytes (32 bits) of data with 2 bytes of instruction
- And:



state

General purpose
registers

Program Counter

	31	24	23	16	15	8	7	0
r15								
r14								
r13								
r12								
r11								
r10								
r9								
r8								
r7								
r6								
r5								
r4								
r3								
r2								
r1								
r0								

31	24	23	16	15	8	7	0

17 words = 68 bytes =
136 hexadecimal digits
+ external memory

Manchester Baby: 3 registers
+ 32 words of memory of 32
bits each

RISC style

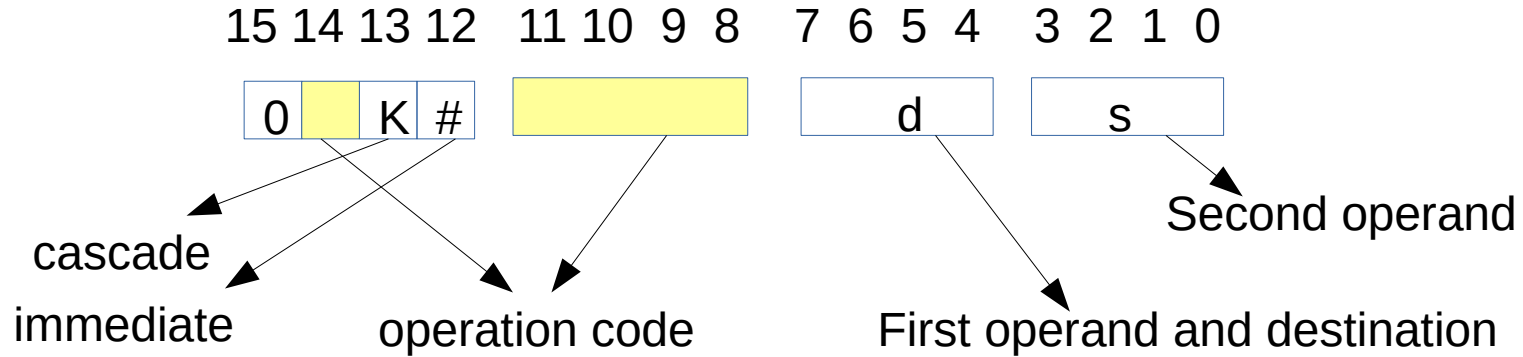
Two instruction formats:

- Data manipulation instructions only operate on registers, or a register and an immediate value present in the instruction itself
- Data storage transfer a register to/from memory (jump and link has the same format, though it should be considered a third kind of instruction)

RISC: how many instructions?

- Counting all variations in assembly syntax we have 108 standard instructions and 12 optional ones (multiplication)
- Not counting the variations we have 31 standard instructions and 3 optional ones
- The control unit deals with only five instruction types:
 - 1)ALU (Math, Logic and Multiply)
 - 2)Comparison
 - 3)Load
 - 4)Store, Store Byte
 - 5)Jump and Link

Data Manipulation Instructions



30 of the 32 possible op codes are used, but due to # and K nearly all instructions have 4 variations.

The assembly language syntax is C-like, with an addition being indicated by

`rD += rS`

- Immediate values

"immediate" changes the second operand to be a number from -6 to 7 instead of a register. The value -7 (9 in hex) indicates that the actual operand is the next 32 bits and -8 (8 in hex) indicates 16 bits

In assembly, a # character is placed before the second operand to indicate that it is an immediate. Otherwise the numbers 0 to 15 indicate a register.

K - Cascade

"cascade" changes the destination to be the first operand of the following instruction. A sequence like

```
r3 <<= #2 ; multiply by four  
r3 += r1
```

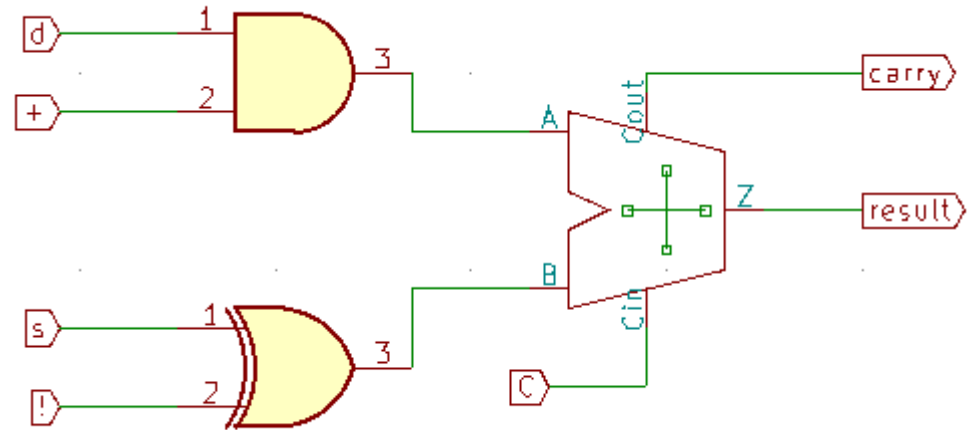
can be implemented with cascade as

```
r5 = {r3 << #2} + r1
```

Unlike the original code fragment, these two instructions don't destroy the value in r3. So though the architecture is a two address in general, it can have the functionality of three addresses (actually four) when needed. The second instruction's syntax is changed from "d op= s" to "d = {..} op s" and the cascaded instruction is placed in the curly brackets also in infix form. It is possible to have more than one level of cascade, but then the d field of intermediate instructions is wasted.

Math: op code is 0 0+!C

- + means that the destination is added to the source
- ! means that the source is bitwise inverted (one's complement)
- C means that the carry in is set

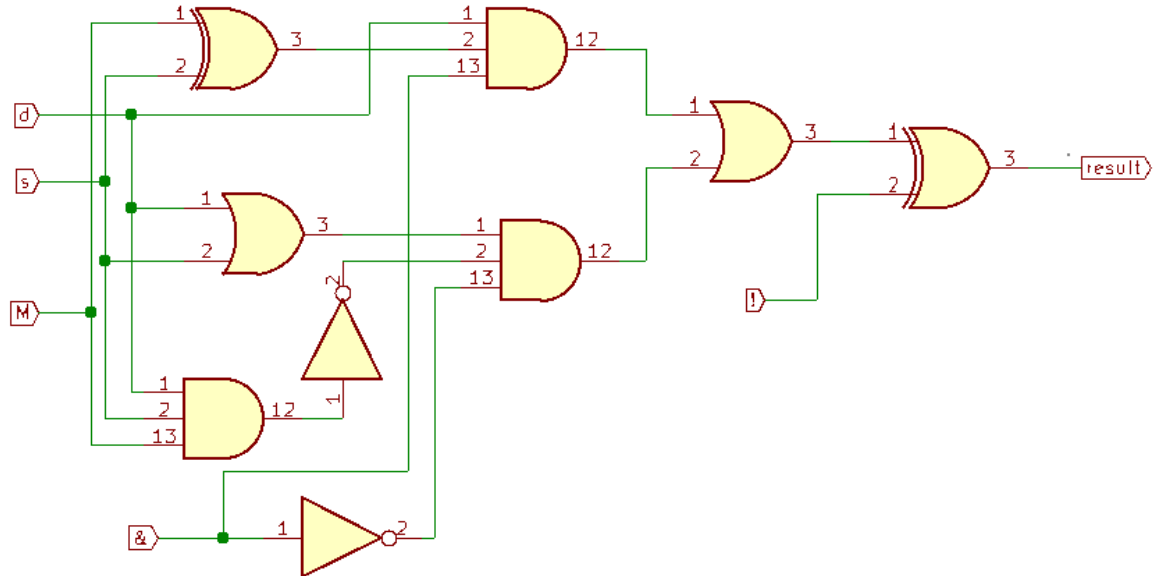


Math

		0?ds	1?ds	2?ds	3?ds
?0ds	move	$d = s$	$d = \#s$		
?1ds	increment	$d = s+1$	$d = \#s+1$		
?2ds	invert	$d = !s$	$d = !\#s$		
?3ds	negate	$d = \sim s$	$d = \sim \#s$		
?4ds	add	$d += s$	$d += \#s$	$\{d + s\}$	$\{d + \#s\}$
?5ds	add forcing carry	$d += s+1$	$d += \#s+1$	$\{d + s+1\}$	$\{d + \#s+1\}$
?6ds	subtract forcing borrow	$d -= s-1$	$d -= \#s-1$	$\{d - s-1\}$	$\{d - \#s-1\}$
?7ds	subtract	$d -= s$	$d -= \#s$	$\{d - s\}$	$\{d - \#s\}$

Logic: op code is 0 1&!M

- & means an AND operation, otherwise it is an OR
- ! means that output is bitwise inverted (one's complement)
- M means a modified version of the instruction



Logic (the 4 bits are ds d!s !ds !d!s)

		0?ds	1?ds	2?ds	3?ds
?8ds	1110 or	$d \mid= s$	$d \mid= \#s$	$\{d \mid s\}$	$\{d \mid \#s\}$
?9ds	0110 exclusive or	$d \wedge= s$	$d \wedge= \#s$	$\{d \wedge s\}$	$\{d \wedge \#s\}$
?Ads	0001 nor	$d \nmid= s$	$d \nmid= \#s$	$\{d \nmid s\}$	$\{d \nmid \#s\}$
?Bds	1001 equivalence	$d \nmid \wedge= s$	$d \nmid \wedge= \#s$	$\{d \nmid \wedge s\}$	$\{d \nmid \wedge \#s\}$
?Cds	1000 and	$d \&= s$	$d \&= \#s$	$\{d \& s\}$	$\{d \& \#s\}$
?Dds	0100 and invert	$d \&!= s$	$d \&!= \#s$	$\{d \&! s\}$	$\{d \&! \#s\}$
?Eds	0111 nand	$d \nmid \&= s$	$d \nmid \&= \#s$	$\{d \nmid \& s\}$	$\{d \nmid \& \#s\}$
?Fds	1011 nand invert	$d \nmid \&!= s$	$d \nmid \&!= \#s$	$\{d \nmid \&! s\}$	$\{d \nmid \&! \#s\}$

Logic: remaining rules

0000 clear	10d0	$d = \#0$
1100 destination	00dd	$d = d$
0010 other and invert	2Edd 0Cds	$d = \{d \text{ !\& } d\} \& s$
1010 source	00ds	$d = s$
0101 not source	02ds	$d = !s$
1101 other nand invert	2Edd 0Eds	$d = \{d \text{ !\& } d\} \text{ !\& } s$
0011 not destination	02dd	$d = !d$
1111 set	10dF	$d = \#-1$

Shift: op code is 1 00\$<

		4?ds	5?ds	6?ds	7?ds
?0ds	shift right	d >>= s	d >>= #s	{d >> s}	{d >> #s}
?1ds	shift left	d <<= s	d <<= #s	{d << s}	{d << #s}
?2ds	signed shift right	d \$>>= s	d \$>>= #s	{d \$>> s}	{d \$>> #s}
?3ds	rotate left	d <>= s	d <>= #s	{d <> s}	{d <> #s}

$a = "<" ? s : 32 - s$
 $x = ("\$" \& !"<") \& 32\{d[31]\}$
 $z[63:0] = \{x, d\} \ll a$
 $result = (z[31:0] \& "<") \mid (z[63:32] \& !(!"\$" \& "<"))$

Barrel shifter: $x[63:0] \ll a[4:0]$
 $z0[63:0] = a[0] ? \{x[62:0], 0\} : x$
 $z1[63:0] = a[1] ? \{z0[61:0], 2\{0\}\} : z0$
 $z2[63:0] = a[2] ? \{z1[59:0], 4\{0\}\} : z1$
 $z3[63:0] = a[3] ? \{z2[55:0], 8\{0\}\} : z2$
 $z[63:0] = a[4] ? \{z3[47:0], 16\{0\}\} : z3$

Multiply (optional)

		4?ds	5?ds	6?ds	7?ds
?4ds	multiply high	d *= s	d *= #s	{d * s}	{d * #s}
?5ds	multiply low	d *>= s	d *>= #s	{d *> s}	{d *> #s}
?6ds	multiply signed high	d \$*= s	d \$*= #s	{d \$* s}	{d \$* #s}
?7ds					

unsigned

signed

4 bit examples
with 8 bit
result:

hex (decimal)

4 (4)
x F (15)

3C (60)

C (12)
x F (15)

B4 (180)

4 (4)
x F (-1)

FC (-4)

C (-4)
x F (-1)

04 (4)

Comparison (calculation)

Greater than
 $d-s ==> !z \& c$

Equal
 $d-s ==> z$

Greater than or equal
 $d-s ==> c$

Not equal
 $d-s ==> !z$

Signed greater than
 $d-s ==> !z \& (n!^v)$

Carry on add
 $d+s ==> c$

Signed greater than or equal
 $d-s ==> n!^v$

$c = \text{result}[32]$
 $z = !(| \text{result})$
 $n = \text{result}[31]$
 $v = (n \& !d[31] \& !s[31]) \mid (!n \& d[31] \& s[31])$

Comparison

		4?ds	5?ds	6?ds	7?ds
?8ds	Greater than	$d > s ? \dots$	$d > \#s ? \dots$	$\{d > s\}$	$\{d > \#s\}$
?9ds	Greater or equal	$d \geq s ? \dots$	$d \geq \#s ? \dots$	$\{d \geq s\}$	$\{d \geq \#s\}$
?Ads	Signed greater than	$d \$> s ? \dots$	$d \$> \#s ? \dots$	$\{d \$> s\}$	$\{d \$> \#s\}$
?Bds	Signed greater or equal	$d \$\geq s ? \dots$	$d \$\geq \#s ? \dots$	$\{d \$\geq s\}$	$\{d \$\geq \#s\}$
?Cds	Equal	$d == s ? \dots$	$d == \#s ? \dots$	$\{d == s\}$	$\{d == \#s\}$
?Dds	Not equal	$d != s ? \dots$	$d != \#s ? \dots$	$\{d != s\}$	$\{d != \#s\}$
?Eds	Carry on add	$d ++ s ? \dots$	$d ++ \#s ? \dots$	$\{d ++ s\}$	$\{d ++ \#s\}$
?Fds					

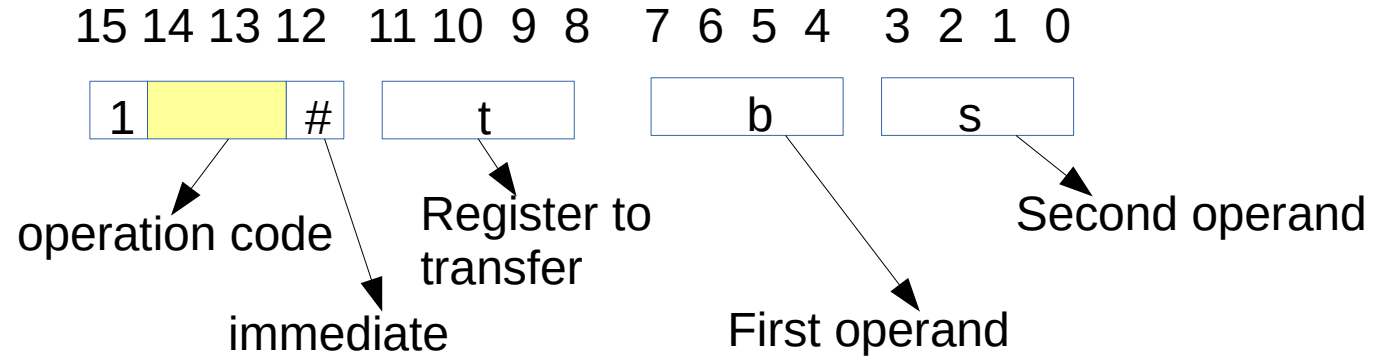
Comparison (trivia)

The comparison instructions will produce a 0 or a 1 result for the next instruction when cascade is used or will skip the next instruction in the normal case when the comparison fails (note that d is not modified)

Note that a comparison that wants less than results can use greater than and swap the operands. This doesn't work if the second operand is an immediate value, but since "if (r3<200) x1 else x2" does the same thing as "if (r3>=200) x2 else x1" it is normally not a problem.

The "d >= s" instruction can be used as a "borrow on subtract" to complement the "d ++ s" (carry on add) instruction.

Data Storage Instructions



Store	8tbs	$b[s] = t$	9tbs	$b[\#s] = t$
Load	Atbs	$t = b[s]$	Btbs	$t = b[\#s]$
Store Byte	Ctbs	$b[s] \text{ B} = t$	Dtbs	$b[\#s] \text{ B} = t$
Jump and Link	Etbs	$\text{====> } b[s]/t$	Ftbs	$\text{====> } b[\#s]/t$

Load/Store

The address of the memory operation will be the sum of b and s (unless b is 0, in this case it will be just s and the "b" can be omitted):

8t0s $[s] = t$
9t0s $[\#s] = t$
At0s $t = [s]$
Bt0s $t = [\#s]$

This allows access to global variables and i/o ports without having to use up a register for the base.

Store byte instruction

```
        ; code to load a byte from memory
Loadb:
        ; r1 is the address
        ; r2 is the destination
        ; r3 is scratch
r2 = r1[#0]
r3 = {r1 & #3} << #3
r2 = {r2 >> r3} & #255
```

This isn't too bad. But since load rotates the word it read on a non aligned address only two instructions are actually needed:

```
        ; code to load a byte from memory
Loadb:
r2 = r1[#0] ; might be rotated
r2 &= #255
```

```
        ; code to store a byte to memory
Storeb:
        ; r1 is the address
        ; r2 is the byte to be stored
        ; r3 and r4 are scratch
r4 = r1[#0] ; previous word
r3 = {r1 & #3} << #3
r2 &= #255 ; make sure it is a byte
r5 = #255
r5 <=< r3
r4 &~= r5 ; clear destination byte
r4 = {r2 << r3} | r4
r1[#0] = r4
```

This is why we defined the remaining data storage instruction to be a store byte:

```
        ; code to store a byte to memory
Storeb:
        r1[#0] B= r2
```

Jump and Link

In the case of JL the t register will hold the previous value of the PC (unless t is 0, in which case the old value of the PC is discarded. "/t" can be omitted):

E0bs ====> b[s]
F0bs ====> b[#s]

The new value of the PC will be the sum of b and s (unless b is 0, in this case it will be the sum of s and the previous value of the PC. The "b" can be omitted):

Et0s ====> [s]/t
Ft0s ====> [#s]/t
E00s ====> [s]
F00s ====> [#s]

Syntactic sugar for the assembler allows "====> label" and "====> label/t" to be written in place of "====> [#label-.]" and "====> [#label-.]/t" respectively.

Interrupts

The focus in this design is simplicity, even at a cost in performance. Nearly all functionality is implemented in the external interrupt controller. An “irq” signal goes to the processor and an “iack” signal goes back to the controller.

The controller has a visible register for each interrupt level it can handle plus two extra ones for the non existing “level 0” and “level 1” (L0 and L1). The value of these registers is the address for the corresponding handler routine.

When some interrupt is pending the controller raises irq until the processor executes an iack cycle. The cycle looks like a memory read where the address is the value of the next PC (which is stored in L1) and the data read is the value to be loaded into the PC. Interrupts do not nest so even if a new, higher level request arrives it won't do anything until the current handler returns.

Return from Interrupt

While register L1 holds the address saved during the iack cycle, L0 has a constant value in bits 16 to 31 such that a jump to \$ircontroller+2 will execute a jump back to the user program.

When bit 0 of L0 is set interrupts are disabled. This happens automatically in an iack cycle and is cleared on a fetch from \$ircontroller+2, but can also be set or cleared by software to create a protected region. Bit 1 is always 0 while bits 2 to 15 are set if interrupt levels 2 to 15, respectively, are pending. Levels beyond 15 can exist, but register L0 can't be used to check on them.

Bit 0 of L2, L3, L4, etc must be cleared for interrupts at that level to be accepted. This will happen if they hold a valid address since instructions are aligned on even bytes. On reset bit 0 of all registers are set.

Note that while the interrupt controller saves the user's PC, any other registers must be explicitly saved and then restored before the return.

Video and Sound

A write-only register configures the beginning of the video buffer.

Bits 1 and 0 select the number of visible lines: none, 192, 384 or 768. The number of pixels per line is always $\frac{4}{3}$ of the number of lines.

Bits 3 and 2 select the number of bits per component: 1, 2, 4 or 8. Bit 4 selects RGB mode with three separate buffers. If we define a “block” as $\frac{1}{3}$ of the video lines, then in RGB mode the red buffer is blocks 0, 1 and 2, the green buffer blocks 3, 4 and 5 and the blue buffer blocks 6, 7 and 8. The sound buffer starts at block 9. When RGB is 0, the video is blocks 0, 1 and 2 and the sound starts at block 4.

Bits 11 to 8 define the interrupt level for indicating a change in the frame bit. Only levels 2 to 15 are valid so 0 or 1 mean no frame interrupts. The next highest interrupt level is used for keyboard and mouse button events.

Video and Sound (cont.)

The sound buffer includes 32 bit words that are output as two signed 16 bit values (left channel as the top bits) at the start of each horizontal synchronization pulse. The buffers includes samples for two frames (525 for 192 lines, 1050 for 384 lines and 2100 for 768 lines). The frame bit (when the register is read) indicates which one is currently being sampled.

The default value of 0 means no video, while F\$FFFF_4315 places the frame buffer at F\$FFFF_4000 with a resolution of 256x192 and 6 bits per pixel. The current frame is 0 and interrupt level 3 will indicate a change in that while interrupt level 4 will indicate keyboard events. The red buffer is at F\$FFFF_4000, green at F\$FFFF_7000 and blue at F\$FFFF_A000. The sound buffer goes from F\$FFFF_D000 to F\$FFFF_E067.

Keyboard and Mouse

The read-only register (same address as the video configuration register) indicates keyboard and mouse events.

Bits 6 to 0 indicate the first unread key code (or joystick or game pad button code) event, or 0 if all have been read. Bit 7 is 0 for key down and 1 for key up.

Bit 8 is 0 for even video frames and 1 for odd ones. It is inverted at the start of the vertical blank interval. The transition from 1 to 0 means that the first word in the sound buffer will be the next one to be output.

Bits 15 to 13 indicate the current state of the mouse buttons.

Bits 31 to 24 are a signed 8 bit number showing how much the mouse has moved horizontally since the register was previously read. Bits 23 to 16 are the same for vertical movement.

Assembly Language Expressions

Numeric literals are always integers represented by a sequence of digits, optionally separated by underscores to make them more readable. The size depends on the context in which the expression is used. Though there is a default radix, any number literal can specify its representation by starting with the value of the highest digit in the base (radix), followed by a “\$” character and then followed by the actual digits.

We can have binary numbers like 1\$1101_1001, decimal numbers like 9\$42, hexadecimal numbers like F\$4AC0 or even octal numbers like 7\$1776. The digits after “9” are “A” to “Z” (lower case is equivalent to upper case).

Defined symbols are treated as their values.

The operations are, in order of highest to lowest precedence: *, /, % then +, - then <<, >>, \$>>, <> then & then ^ then |. Parenthesis override the precedence and must be used inside cascades so the top level operator is the instruction.

Assembly language directives: define

Symbols can be defined by

name: expression

A period character in an expression indicates the value of the PC for that instruction. An empty expression is equivalent to just a period, so is equivalent of defining a label in other assemblers. This means that labels must be on a line of their own instead of coming before an instruction.

If a symbol is defined more than once, only the last definition is used in the whole program and a warning is issued.

Registers in an assembly instruction are just expressions that evaluate from 0 to 15 (0 to 7 in Baby042) so besides the default names r0 to r15 more meaningful names can be defined.

Assembly language directives: configuration

Symbol	Valid Values	Default	Description
\$processor	42, 22, 18	42	Baby42, Baby22 or Baby042 (9\$18=7\$42)
\$radix_dump	1\$1 to Z\$Z	F\$F	Base of numbers in addresses and data
\$radix_source	1\$1 to Z\$Z	9\$9	Base of numbers in the source code
\$kvmregister	Word aligned	F\$FFFF_FF00	Configure video and sound, read keyboard and mouse
\$ircontroller	Word aligned	F\$FFFF_FF00	Where the interrupt level registers are

Symbol names starting with “\$” are an error, unless they are in the list above and set to one of the indicated values. In that case they change the configuration of the assembler and/or the simulator.

Assembly language directives: include file

A line can be replaced with the contents of another file with:

```
< "file name"
```

As the assembler generates a single output and doesn't use a linker, the only way to develop with multiple files is to include the others into a top file. The tools can optionally show the contents of the file right below the directive adding "> " to each line. With multiple levels of include you can easily have lines starting with "> > > " or similar.

Assembly language directives: origin

The value of the PC can be changed with

`% expression`

This is equivalent of "org" in other assemblers. If 120 bytes need to be allocated, then "`% .+120`" will do the job.

Instructions should always be aligned on an even byte. This expression can force that to be the case:

`% ((.+1)/2)*2`

Assembly language directives: constants

Constants can be inserted into the instruction stream with

`# expr1, expr2, expr3, B# expr4, expr5, H# expr6, # expr7, expr8`

The `#` interprets the following expressions as 32 bit values, while `B#` inserts 8 bit values and `H#` 16 bit values.

Placing ASCII characters between two `"` has the same effect as `B#` followed by the list of the characters' numeric value.

Example 1

Jan Gray used this simple C program to illustrate the assembly language of his XR16 processor (<http://www.fpgacpu.org/papers/xsoc-series-drafts.pdf>)

```
typedef struct TN {
    int k;
    struct TN *left, *right;
} *T;

T search(int key, T p) {
    while (p && p->k != key)
        if (p->k < key)
            p = p->right;
        else
            p = p->left;
    return p;
}
```

; Example 1 Baby42 machine language and assembly with LCC style

k: 0

left: 4

right: 8

search: ; r3=key r4=p r9=scratch r2=return

F008 000E ====> L3

L2:

B940 r9 = r4[#k]

4893 F006 r9 > r3 ? ====> L5

B448 0008 r4 = r4[#right]

F002 ====> L6

L5:

B444 r4 = r4[#left]

L6:

L3:

0094 r9 = r4

5C90 F008 0008 r9 == #0 ? ====> L7

B940 r9 = r4[#k]

4D93 F008 FFE2 r9 != r3 ? ====> L2

L7:

0024 r2 = r4

L1:

F0D0 ====> r13[#0] ; return address was in r13

```
; Example 1 Baby42 machine language and assembly with better compiler
;      that optimizes jumps to jumps and jumps to returns
```

```
k:      0
```

```
left:   4
```

```
right:  8
```

```
search: ; r3=key  r4=p  r9=scratch  r4=return value
```

```
5C90 F0D0      r4 == #0 ? ====> r13[#0]  ; return
```

```
B940          r9 = r4[#k]
```

```
5C93 F0D0      r9 == r3 ? ====> r13[#0]
```

```
4893 F008 0008  r3 > r9 ? ====> L5
```

```
B448 0008      r4 = r4[#right]
```

```
F008 FFE8      ====> search
```

```
L5:
```

```
B444          r4 = r4[#left]
```

```
F008 FFE2      ====> search
```

Example 2

Here is the famous Sieve of Eratosthenes benchmark published in Byte magazine:

```
1 SIZE = 8190
2 DIM FLAGS(8191)
3 PRINT "Only 1 iteration"
5 COUNT = 0
6 FOR I = 0 TO SIZE
7   FLAGS (I) = 1
8 NEXT I
9 FOR I = 0 TO SIZE
10  IF FLAGS (I) = 0 THEN 18
11  PRIME = I+I + 3
12  K = I + PRIME
13  IF K > SIZE THEN 17
14  FLAGS (K) = 0
15  K = K + PRIME
16 GOTO 13
17 COUNT = COUNT + 1
18 NEXT I
19 PRINT COUNT," PRIMES"
```

```

size: 8190
true: r10
false: r11
flags: r12
flg:
???? .. ??    % .+8191
count: r1
i:    r2
prime: r3
k:    r4
text1:
    4F 6E6C    "Only 1 iteration", B# 0
7920 3120
6974 6572
6174 696F
6E00
text2:
2050 5249    " PRIMES", B# 0
4D45 5300
sieve:
10C0    flags = #flg
10A1    true = #1
10B0    false = #0
1008 1FFF    r0 = #text1

```

```

FD08 ????    ====> PrintText/r13
                ; library subroutine expects
                ; return address in r13
1010    count = #0
1020    i = #0
for1:
5828 1FFE    i > #size ? ====> endfor1
F008 0008
8AC2        flags[i] = true
1421        i += #1
F008 FFF0    ====> for1
endfor1:
1020        i = #0
for2:
5828 1FFE    i > #size ? ====> endfor2
F008 0028
A5A2        r5 = flags[i]
5C50 F008    r5 == #0 ? ====> L18
001A
2422 1433    prime = {i+i} + #3
2423 1440    k = {i + prime} + #0

```

```

L13:
5848 1FFE      k > #size ? =====> L17
F008 0008
8BA4          flags[k] = false
0443          k += prime
F008 FFF0      =====> L13

L17:
1411          count += #1

L18:
1421          i += #1
F008 FFD0      =====> for2
endfor2
0001          r0 = count
FD08 ????      =====> PrintInt/r13
1008 2010      r0 = #text2
FD08 ????      =====> PrintText/r13
F008 ????      =====> Stop

```

There is also a C version of the benchmark in the Wikipedia article (https://en.wikipedia.org/wiki/Byte_Sieve). But Basic, specially a version as primitive as this, corresponds a lot more directly to the assembly language implementation.

Normally code that calls other subroutines should save the return address on the stack, but in this case we end with a jump instead of a return.

Since the address of flags happens to be 0 (it is a byte pointer), it would be possible to use "i[#FLG]" in the load and store instructions instead of "flag[i]", which would save one register. But that is not something that would normally happen, so it wasn't used in this example either.

Example 3: ARM and Baby42

```
AREA LOG, CODE, READONLY
EXPORT log
; r0 = input variable n
; r0 = output variable m (0 by default)
; r1 = output variable k ( $n \leq 2^k$ )
```

Log

```
MOV r2, #0 ; set m = 0
```

```
MOV r1, #-1 ; set k = -1
```

log_loop

```
TST r0, #1 ; test LSB(n) == 1
```

```
ADDNE r2, r2, #1 ; set m = m+1 if true
```

```
ADD r1, r1, #1 ; set k = k+1
```

```
MOVS r0, r0, LSR #1 ; set n =  $n \gg 1$ 
```

```
BNE log_loop ; continue if  $n \neq 0$ 
```

```
CMP r2, #1 ; test m == 1
```

```
MOVEQ r0, #1 ; set m = 1 if true
```

log_rtn

```
MOV pc, lr
```

END

40 bytes, hence Thumb

```
; AREA LOG, CODE, READONLY
```

```
; EXPORT log
```

```
; r0 = input variable n
```

```
; r0 = output variable m (0 by default)
```

```
; r1 = output variable k ( $n \leq 2^k$ )
```

Log:

```
r2 = #0 ; set m = 0
```

```
r1 = #-1 ; set k = -1
```

log_loop:

```
{r0 & #1} == #1 ? r2 += #1 ; test LSB(n) == 1
```

```
; set m = m+1 if true
```

```
r1 += #1 ; set k = k+1
```

```
r0 >>= #1 ; set n =  $n \gg 1$ 
```

```
r0 != #0 ? =====>log_loop ; continue if  $n \neq 0$ 
```

```
r2 == #1 ? r0 = #1 ; test m == 1
```

```
; set m = 1 if true
```

log_rtn:

```
=====>[lr]
```

26 bytes

Variation 1: baby22

- It is possible to have a 16 bit datapath so the processor would have both 2 byte instructions and 2 byte data. This is limited to 64KB of memory, but that is enough for many applications
- The only change from baby42 is that immediate value -7 does not indicate a 32 bit immediate

Variation 2: baby042

- Operating on groups of 6 bits instead of 8 bit bytes we could have 24 bit data and 12 bit instructions. It is more natural to use octal instead of hexadecimal numbers
- With 4 fewer bits in each instruction it would be better to have 8 registers instead of 16. Instead of a dedicated immediate bit, when the second operand is register 6 or 7 the actual value is a 24 or 12 bit immediate after the instruction.
- 0XDS: $d=s$, $d=s+1$, $d=!s$, $d=\sim s$,
- 1XDS: $d+=s$, $\{d+s\}$, $d+=s+1$, $\{d+s+s\}$, $d-=s-1$, $\{d-s-1\}$, $d-=s$, $\{d-s\}$
- 2XDS: $d|=s$, $\{d|s\}$, $d^=s$, $\{d^s\}$, $d\&s$, $\{d\&s\}$, $d!\&s$, $\{d!\&s\}$
- 3XDS: $d>>=s$, $\{d>>s\}$, $d<<=s$, $\{d<<s\}$, $d\$>>=s$, $\{d\$>>s\}$, $d<<=s$, $\{d<<=s\}$
- 4XDS: $d>s?$, $\{d>s\}$, $d>=s?$, $\{d>=s\}$, $d==s?$, $\{d==s\}$, $d!=s?$, $\{d!=s\}$
- 5TBS: $===> b[s]/t$
- 6TBS: $b[s] = t$
- 7TBS: $t = b[s]$

What is missing?

- No floating point math
- No exceptions (if the missing op codes are defined as either NOPs or redundant)
- No MMU (memory management unit) or supervisor mode
- No debug interface (JTAG or control panel)