Baby42

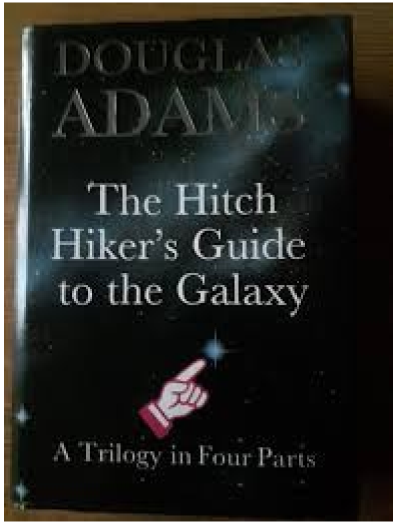
A simple RISC processor

Jecel Mattos de Assumpção Jr
April 2020

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

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

Program Counter

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

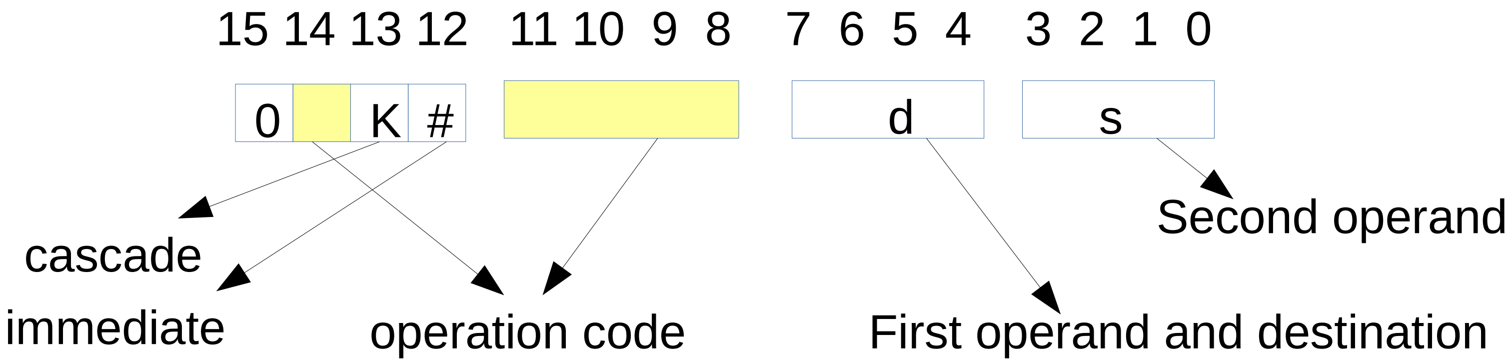
Two instruction formats:

- 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 106 standard instructions and 12 optional ones (multiplication)
- Not counting the variations we have 30 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
  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

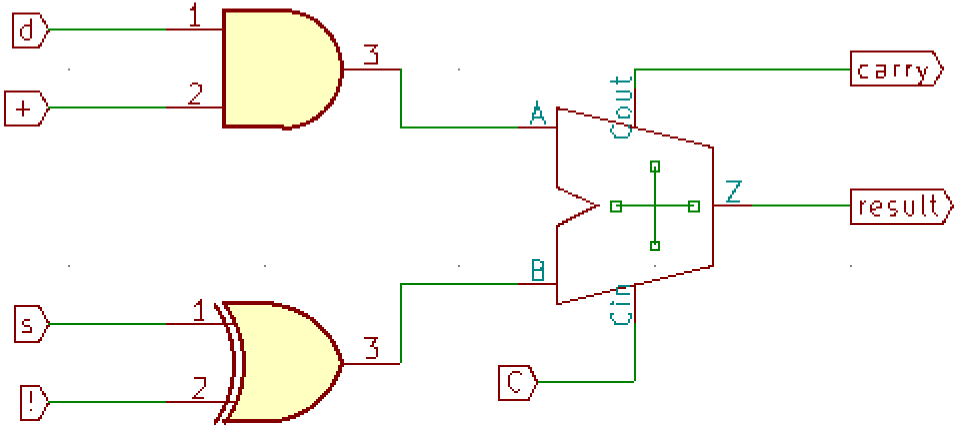
```
r5 = {r3 << #2} + r1
```

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

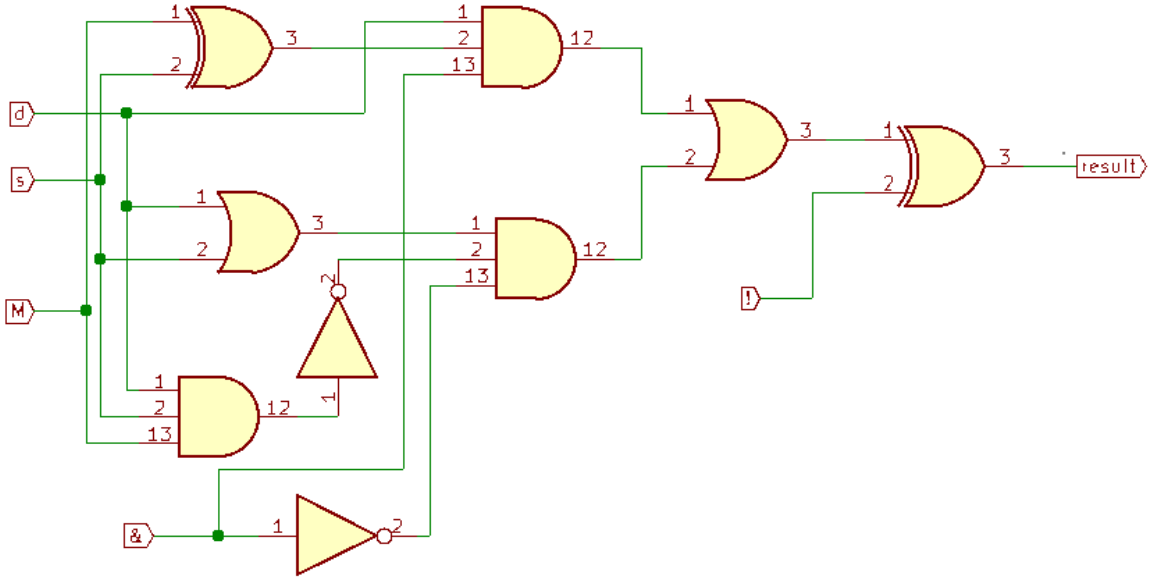


Math

		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 = ~s	d = ~#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 !s!d s!d !sd sd)

		0?ds	1?ds	2?ds	3?ds
?8ds	0111 or	d |= s	d |= #s	{d | s}	{d | #s}
?9ds	0110 exclusive or	d ^= s	d ^= #s	{d ^ s}	{d ^ #s}
?Ads	1000 nor	d !|= s	d !|= #s	{d !| s }	{d !| #s}
?Bds	1001 equivalence	d !^= s	d !^= #s	{d !^ s}	{d !^ #s}
?Cds	0001 and	d &= s	d &= #s	{d & s}	{d & #s}
?Dds	0010 and invert	d &!= s	d &!= #s	{d &! s}	{d &! #s}
?Eds	1110 nand	d !&= s	d !&= #s	{d !& s}	{d !& #s}
?Fds	1101 nand invert	d !&!= s	d !&!= #s	{d !&! s}	{d !&! #s}

Logic: remaining rules

0000 clear	10d0	d = #0
0011 destination	00dd	d = d
0100 other and invert	2Edd 0Cds	d = {d !& d) & s
0101 source	00ds	d = s
1010 not source	02ds	d = !s
1011 other nand invert	2Edd 0Eds	d = {d !& d) !& s
1100 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 = (“$” & !“<”) & 32{d[31]}
z[63:0] = {x, d} << a
result = (z[31:0] & “<”) | (z[63:32] & !(!”$” & “<”))

Barrel shifter: x[63:0] << 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)        C (12)        4 (4)        C (-4)
x  F (15)    x  F (15)    x  F (-1)    x  F (-1)
-----        ------       ------       ------
  3C (60)      B4 (180)     FC (-4)      04 (4)
```

Comparison (calculation)

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

Greater than or equal
d-s ==> c

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

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

Equal
d-s ==> z

Not equal
d-s ==> !z

Carry on add
d+s ==> c

c = result[32]
z = !(|\result)
n = result[31]
v = (n&!d[31]&!s[31]) | (!n&d[31]&s[31])

Comparison

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

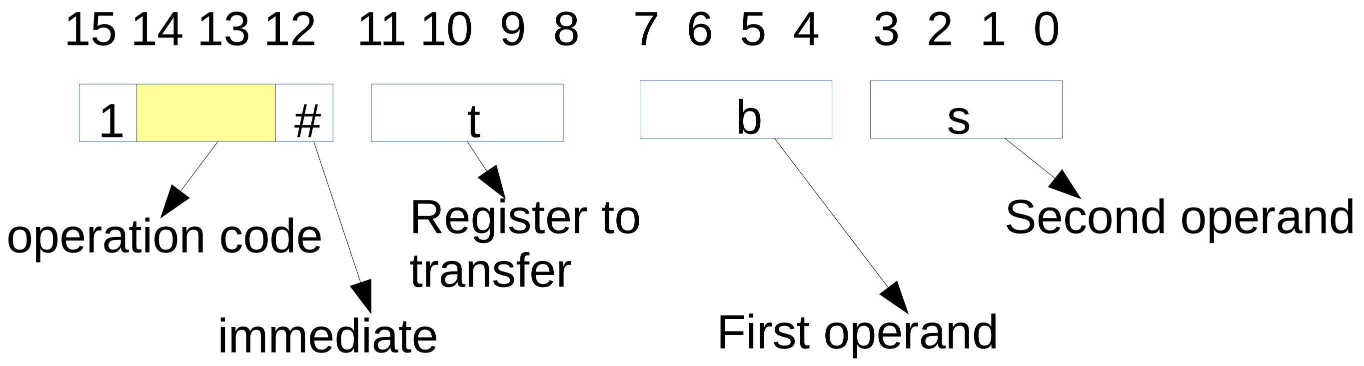
Comparison (trivia)

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]
Jump and Link	Ctbs	====> b[s]/t	Dtbs	====> b[#s]/t
	E???		F???	

Jump and Link

In the case of JL the t register will hold the previous value of the PC (unless t is 15, in which case the old value of the PC is discarded. "/t" can be omitted).

CFbs   ====> b[s]
DFbs   ====> b[#s]

The new value of the PC will be the sum of b and s (unless b is 15, in this case it will be the sum of s and the previous value of the PC. The "b" can be omitted).

CtFs   ====> [s]/t
DtFs   ====> [#s]/t
CFFs   ====> [s]
DFFs   ====> [#s]

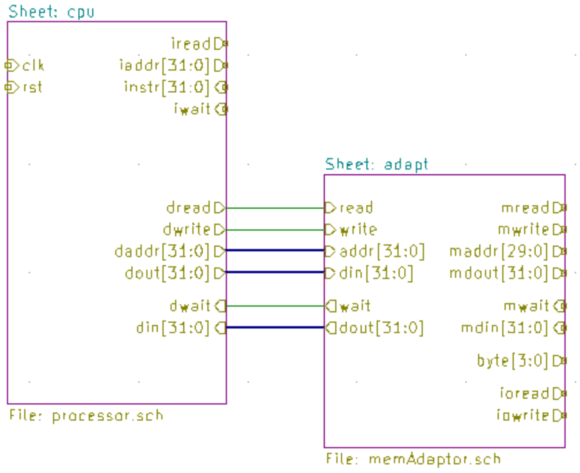
Syntatic sugar for the assembler allows "====> label" and "====> label/t" to be written in place of "====> [#label-.]" and "====> [#label-.]/t" respectively.

Bytes 1: Load/Store Adapter

Baby42 natively only transfers 32 bit words to and from memory. It is possible to have an adapter circuit between the processor and memory/cache to make 16 and 8 bit transfers possible, as well as allowing non aligned 32 bit transfers (addresses that are not a multiple of 4).

Bits 31 and 30 of an address indicate the size of the data to be fetched or stored while the rest of the address points to one of 1G bytes:

00 = 8 bit value
01 = 16 bit value
10 = 32 bit value
11 = I/O (32 bit value)



Bytes 2: 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. If we redefine load to rotate the word it read on a non aligned address then just two instructions would be 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
```

We could define the remaining data storage instruction to be a store byte:

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

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.

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
DFF8 000E           ====> L3
                L2:
B940                r9 = r4[#k]
4893 DFF6           r9 > r3 ? ====> L5
B448 0008           r4 = r4[#right]
DFF2                ====> L6
                L5:
B444                r4 = r4[#left]
                L6:
                L3:
0094                r9 = r4
5C90 DFF8 0008      r9 == #0 ? ====> L7
B940                r9 = r4[#k]
4D93 DFF8 FFE2      r9 != r3 ? ====> L2
                L7:
0024                r2 = r4
                L1:
DFD0                ====> 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 DFD0           r4 == #0 ? ====> r13[#0]  ; return
B940                r9 = r4[#k]
5C93 DFD0           r9 == r3 ? ====> r13[#0]
4893 DFF8 0008      r3 > r9 ? ====> L5
B448 0008           r4 = r4[#right]
DFF8 FFE8           ====> search
                L5:
B444                r4 = r4[#left]
DFF8 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
DDF8 ????           ====> PrintText/r13
                     ; library subroutine expects
                     ; return address in r13
1010                count = #0
1020                i = #0
                for1:
5828 1FFE           i > #size ? ====> endfor1
DFF8 0008
8AC2                flags[i] = true
1421                i += #1
DFF8 FFF0           ====> for1
                endfor1:
1020                i = #0
                for2:
5828 1FFE           i > #size ? ====> endfor2
DFF8 0028
A5A2                r5 = flags[i]
5C50 DFF8           r5 == #0 ? ====> L18
001A
2422 1433           prime = {i+i} + #3
2423 1440           k = {i + prime} + #0
```

```
                L13:
5848 1FFE           k > #size ? ====> L17
DFF8 0008
8BA4                flags[k] = false
0443                k += prime
DFF8 FFF0           ====> L13
                L17:
1411                count += #1
                L18:
1421                i += #1
DFF8 FFD0           ====> for2
                endfor2
0001                r0 = count
DDF8 ????           ====> PrintInt/r13
1008 2010           r0 = #text2
DDF8 ????           ====> PrintText/r13
DFF8 ????           ====> 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 <= 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>>1
        BNE log_loop ; continue if n != 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 <= 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>>1
        r0 != #0 ? ====>log_loop ; continue if n != 0
        r2 == #1 ? r0 = #1 ; test m ==1
                          ; set m = 1 if true
log_rtn:
        ====>[lr]
```

26 bytes

What is missing?

- No floating point math
- No interrupts
- 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)

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.
- The top 3 bits select the instruction group: add, logic, shift, compare, move, load, store, jumpAndLink. The first four groups have 4 instructions each with normal and cascade versions (so half as many logic and compare as baby42). The move group is the first half of baby42 math instructions with no cascade versions.