Baby8

A tiny processor for FPGA i/o

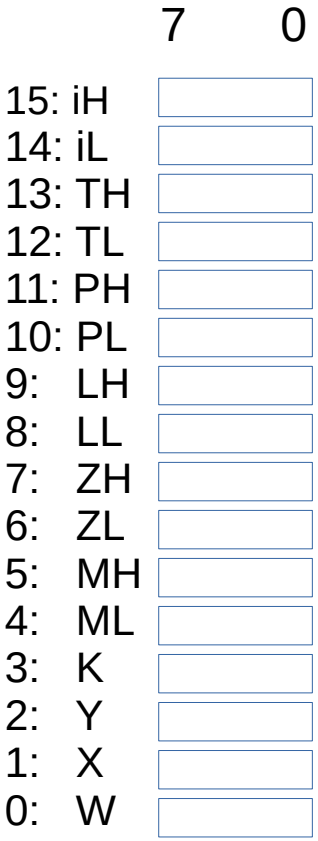
Jecel Mattos de Assumpção Jr
September 2023

name

- 8 bit processor

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

state



internal registers

Flags: Z, N, C and V
They don't persist from one intruction to the next but can be saved with the test instructions

16 bytes
+ external memory

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

Basic Syntax

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

```
rD += rS
```

K - Cascade

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

```
X += [20] ; increment by the value pointed to by location 20
W &= X
```

can be implemented with cascade as

```
Y :=  W & {X + [20]}
```

Unlike the original code fragment, these two instructions don't destroy the value in X. 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 := s op {..}" 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, specially when we don’t want to save a result used for a test.

Source and Destination

The two bit ss and dd fields in the instruction use values 0, 1 and 2 to encode the registers W, X and Y respectively. A value of 3 indicates that a byte follows which encodes the actual address. When both ss and dd are 3 then the first extra byte is for the source and the second for the other source and destination. In the case of immediate instructions the immediate value is the first extra byte and the destination follows that.

When the extra byte is less than 128 then it indicates a byte in the “zero page” and the syntax is just the address. If the top bit is set then the bottom 7 bits indicate a pair of bytes in the zero page with the address of the actual byte value. The even byte is the low part of the address and the odd byte the high part. The bottom bit is ignored and instead used to indicate (when 1) that the two bytes will be incremented after used as the address. The syntax is the value in square brackets, like [addr], when bit zero is 0 or followed by a plus sign, like [addr]+, when bit zero is 1 indicating an increment.

Source and Destination

In theory, the zero page allows access to bytes 0 to 127 of the 256 byte page selected by register ZH. In practice, addresses 0 to 3 actually access the four input and four output ports instead. More i/o ports can be memory mapped to other addresses if needed.

Zero page addresses 4 to 15 actually access the internal registers ML to iH. This allows LL and LH to be saved and restored if more than one level of subroutines are needed. ZH can also be changed so the “zero page” can be relocated to anywhere in memory, like on the 6809 and 65816 processors.

When TL is changed the timer is started if TL and TH are not both zero and it is stopped if they are zero.

Trying to change PL or PH directly is very tricky and should be avoided.

Basic Instructions

add	0000 ssdd	D += S	0001 ssdd	{D + S}
subtract	0010 ssdd	D -= S	0011 ssdd	{D - S}
move	0100 ssdd	D = S	0101 ssdd	{D = S}
test	0110 cccc	cond ? ….	0111 cccc	{cond}
and	1000 ssdd	D &= S	1001 ssdd	{D & S}
or	1010 ssdd	D |= S	1011 ssdd	{D | S}
exclusive or	1100 ssdd	D ^= S	1101 ssdd	{D ^ S}
See next table	111f ffdd			

Immediate Instructions

add	1110 00dd	D += #
subtract	1110 01dd	D -= #
move	1110 10dd	D = #
See next table	1110 11ff	
and	1111 00dd	D &= #
or	1111 01dd	D |= #
exclusive or	1111 10dd	D ^= #
See next table	1111 11ff	

Control Flow Instructions

jump	1110 1100	===> ##	PH,PL := ##
call	1110 1101	===> ##*	LH,LL := PH,PL; PH,PL := ##
branch	1110 1110	===> .#	PH,PL += #
return	1110 1111	<===	PH,PL := LH,LL
jump	1111 1100	===> [{}]	PH,PL := [K]
call	1111 1101	===> [{}]*	LH,LL := PH,PL; PH,PL := [K]
branch	1111 1110	===> .{}	PH,PL += K
table	1111 1111	<=== {}	PH,PL := [LH,LL + K]

Test

cond	name	code	syntax	alt	code	syntax	alt
Z	Equal	011k 0000	==	Z	011k 0001	!=	!Z
C	Greater equal	011k 0010	>=	C	011k 0011	<	!C
N	Negative	011k 0100	<0	N	011k 0101	>=0	!N
V	Overflow	011k 0110		V	011k 0111		!V
C&!Z	Greater	011k 1000	>		011k 1001	<=	
N==V	Signed greater equal	011k 1010	$>=		011k 1011	$<	
!Z&N==V	Signed greater	011k 1100	$>		011k 1101	$<=	
1	TRUE	011k 1110	true		011k 1111	false	

Shifts and Rotations

The missing multiplication and division instructions are to be expected for a very small processor (though they can be added as an i/o device if needed), but the lack of shift and rotate instructions might seem limiting. Adding a value to itself is equivalent to a one bit shift to the left with the carry indicating the removed bit. Shifting to the right (logical or arithmetic) an 8 bit value by N bits can be achieved by extending (zero or sign) to 16 bits, shifting that to the left by 8-N bits and using the top byte as the result.

```
  ; code to shift X right by 5 bits arithmetically
W = #0 ; default sign
C {X & #128} ? W = #255 ; extended sign for V
W += W ; shift left top half
X += X ; shift left bottom half
C ? W += #1 ; move carry to bottom bit
W += W
X += X
C ? W += #1 ; second bit
W += W
X += X
C ? W += #1 ; third bit. Now W is X $>> 5
```

Sending a byte with the least significant bit first might seem to need 7 such steps per bit, but the same result can be achieved by shifting a one bit mask to the left once per step to check each bit from least to most significant.

```
      W = #1 ; bit mask, start at least significant
SendBit:
      C {X & W} ? ===> sendOne
SendZero:
      …
      W += W
      C ? ===> sendBit
```

Interrupt

The interrupt mode uses iH,iL for instruction fetches instead of PH,PL (bit 2 of the register addresses is set) and zero page addresses go from 144 to 255 instead of from 16 to 127 (bit 7 of addresses is set).

The interrupt mode is entered on the next instruction fetch after the interrupt line goes high unless the previous instruction was a cascade, the current instruction is supposed to be skipped or the current instruction is a test. These conditions are indicated by internal states in the processor’s controller and can’t be saved or restored. The last condition can only be checked after the current instruction has been fetched, so if the interrupt mode is allowed then PL is restored (PH won’t have changed yet at this point) so the instruction can be fetched a second time after the interrupt mode is finished.

The interrupt mode ends on the next instruction fetch after the interrupt line goes low with the same restrictions as above.

Interrupt

Most processors have interrupt handlers start at fixed addresses or, more often, indicated in some table. In baby8 the address in iH,iL is simply the instruction following the one that caused the interrupt line to go low at the end of the previous interrupt. This is more like a coroutine scheme where an explicit “yield” instruction is executed. With careful programming it is possible to speed up response time by having the processor ready to execute in different places for different situations.

Any registers used by the interrupt handler should be saved and then restored right before the yield. Having use of the second half of the zero page makes this easier. Note that the first half can still accessed with indirect addressing.

Timer

Every instruction is executed in a known number of clock cycles, but to make precise delays even easier TH,TL implements a 16 bit timer. The timer mode is entered on the next instruction fetch after a write to TL which results in TH or TL having a non zero value. The restrictions are the same as entering or leaving the interrupt mode.

Instead of fetching instructions the processor simply decrements TL by one on every clock cycle. When TH,TL reaches zero, instruction fetch resumes. If TH is not 0 but TL is, then TL is decremented by two (going to 254) and on the next cycle TH is decremented.

If an interrupt arrives while in the timer mode, the interrupt is handled and execution is normal until the interrupt ends. The interrupt handler may clear TH,TL or reduce it by some amount if having a longer delay than expected in the user code is a problem.

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.

Assembly language directives: macros

Symbols can be defined as macros by

\name arg1 arg2 … argN ‘text replacement’

From then on whenever name is present in the program it is replaced by the indicated replacement text. Note that the text between ‘ characters can take up multiple lines. While expanding the text, any instances of arg1 and so on are replaced by the corresponding text on the line where name appeared. An argument can have spaces or even multiple lines if placed between ‘ characters. If the name of a macro is present while expanding another macro, another level of expansion is started.

When expanding a macro any / characters are ignored, but they still separate argument names. So arg1/loop when the first argument is ‘start’ will expand to startloop. Without the / it would be arg1loop which is not expanded.

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

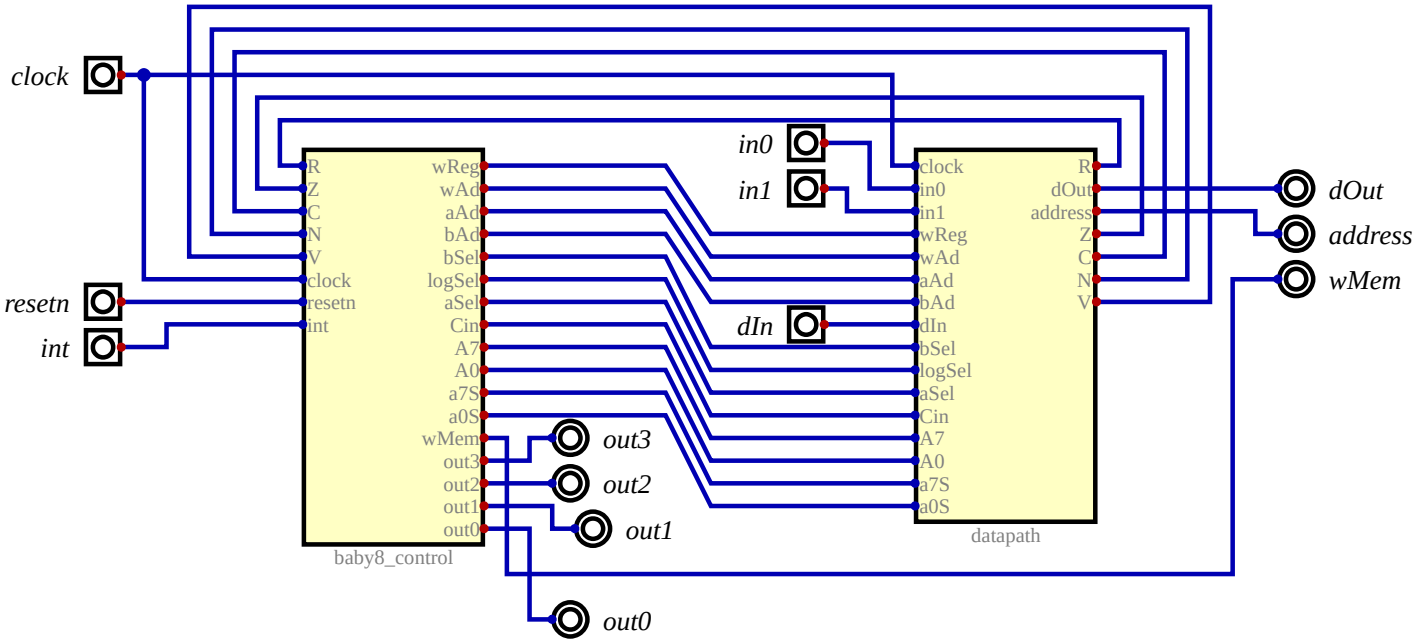
Constants can be inserted into the instruction stream with

    ## expr1, expr2, expr3, # expr4, expr5, ## expr6, # expr7, expr8

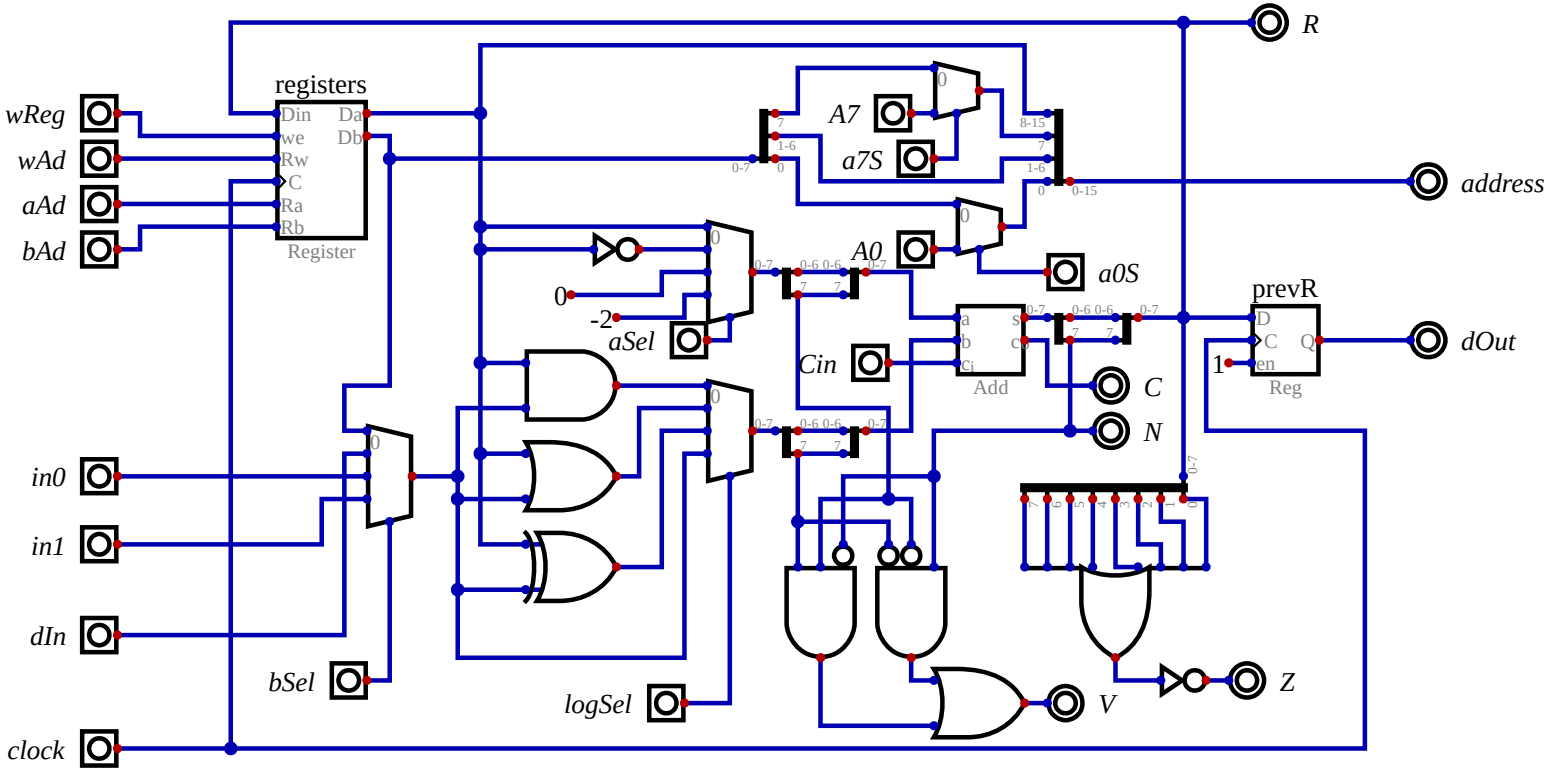
The # interprets the following expressions as 8 bit values, while ## inserts 16 bit values with the lowest byte first.

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

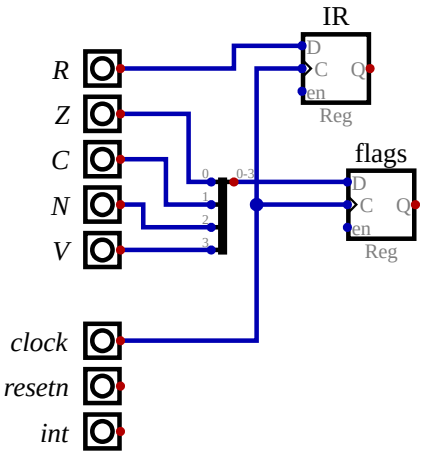
Implementation - CPU



Implementation - Datapath



Implementation – Control Unit



wReg
wAd
aAd
bAd

bSel
logSel
aSel

Cin

A7
A0
a7S
a0S

wMem

out3
out2
out1
out0