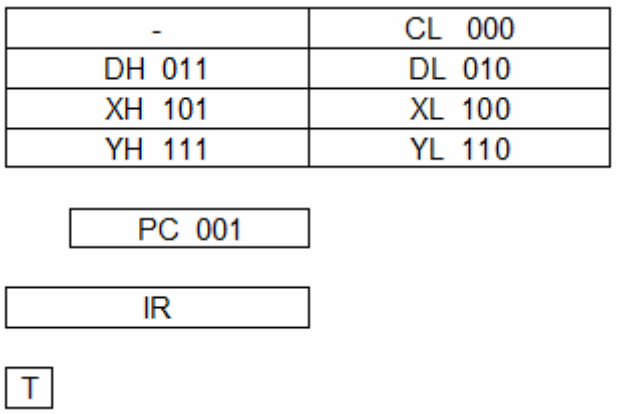
RISC Relay CPU. Architecture and Instruction List

R.J.H. 20190106

This is a Harvard architecture with separate memories for instructions and data. Registers, instructions and both memories are 16 bits wide. Both memory sizes are 64K words. Most instructions need only one cycle (defined as memory read followed by ALU operation), but the 32 bit instructions, and executed jump instructions need 2 cycles.
It is intended that the design will be implemented without using microcode.

GENERAL INSTRUCTION INFORMATION

REGISTERS



- There are eight user-visible registers, one of them is the PC.
- All registers are 16 bit. In the implementation, the PC is only 12 bit.
- Three of these registers (CL, XL, YL) can be used as an address register.
- Six registers can be paired to form three 32-bit registers D, X and Y
- There is a 16-bit instruction register IR and a single condition bit called T.

The picture shows the registers, their names and their 3-bit code.

INSTRUCTION FORMATS

This is a 2-operand design. The first operand is a register (RRR) and the second operand is:

- A register (SSS) in the register format, or
- An Immediate operand (Immediate format), or
- The contents of a memory location (in Memory and Zero page formats).

The result of the operation is put in the RRR register.

FORMATS	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Zero page	0	MMM			0	RRR			L	0	X	Z				
Zero page logic	0	MMM			0	RRR			L	1	X	Z				
Immediate	0	MMM			1	RRR			Immediate data							
Memory	1	MMM			0	RRR			L	A		D				
Register	1	MMM			1	RRR			0	S	S	L	Q	Q	Q	S
Test & Branch	1	BBB			1	BBB			1	S	S	Branch distance				

A: 2-bit address register select
B: 6-bit specification of Test-and-Branch instructions
D: 5-bit displacement. The displacement is OR-ed with the address register contents.
L: 32-bit operation flag
M: 3-bit operation code (opcode)
Q: 3-bit extra operation code
R: 3-bit destination register
S: 3-bit source register or 2-bit source register pair
X: unused
Z: 5-bit zero-page memory address.

The register instruction format has three additional operation code bits (Q), that can be used to define a shift for the operand.

Allmost all instructions have two operands, like this:

```
<instruction> <dst>,<src>  ; comment

ADD    DL,[CL+2]      ; Add contents of location (CL+2) to the destination register DL.

ST     [CL+4],DL      ; Store the DL register at location (CL+4).
```

INSTRUCTIONS

The possible instructions depend on the instruction format:

	instruction format					
Opcode	**Zero page**	**Zero page logic**	**Memory**	**Register**	**Immediate (except PC)**	**Immediate PC**
000	TEQ	7segment	TEQ	TEQ	TEQ	
001	LD	XOR	LD	LD	OR/PREFIX	JP
010	ST		ST	TSTB	TSTB	
011	ADD	OR	ADD	ADD	ADD	BR
100	-		-	SUB / TGT	XOR	BRF
101	DADD		DADD	DADD	DADD	JPF
110	-	AND	-	DSUB / DTGT	AND	BRT
111	DMPY		DMPY		LD	JPT

TEQ	test if register equal to operand. Only affects flag T.
LD	load register with operand
ST	store register in memory
TSTB	testbit, test if (register AND operand) is zero. Only affects flag T.
TGT	test greater than (subtract without storing result)
ADD	add operand to register (binary)
DADD	add operand to register (decimal)
DSUB	subtract operand from register (decimal)
DTGT	decimal test greater than (subtract without storing result)
DMPY	multiply-step (conditional add, to be defined later)
AND	register <- register AND operand
OR	register <- register OR operand
XOR	register <- register XOR operand

The TBRZ/TBRNZ instructions have a different encoding and are not in this table.

In the second half of this document, all instructions will be presented individually.

CONDITION FLAG

After an add or subtract instruction, the T flag indicates the state of the carry. After other instructions (including LD, the compare instruction TEQ and bit-test instruction TSTB), the T flag indicates if the ALU result was zero. (The TEQ instruction is an XOR, where the result is not written to the register.)
Branches or calls can be conditional (ending with T or F), these instructions test the T flag.

ADDRESS REGISTERS

The memory format specifies an address register. The registers CL, XL and YL can be used as address register. The register is specified by the two A bits in the instruction.

00 CL
01 unused
10 XL
11 YL

Memory addressing always has a small (5 bit) displacement within the instruction opcode. This facilitates addressing of variables in a stack frame, or addressing of structure members. This displacement is not really added, but or'ed to the address register contents. This saves an adder. The programmer or compiler must be aware that the address register contents must be properly aligned.

IMMEDIATE PREFIX

The immediate instructions have only a 8 bit data field. Normally, bits 8-15 of the data will get the same value as bit 7 (sign-extension).
If a full 16-bit immediate is needed, then use the prefix instruction to put 8 bits in DH (automatically generated by the assembler). The prefix instruction will set a flag (not accessible by the user), that indicates to the following (immediate) instruction that data bits 8-15 must be copied from DH bit 0-7.
This is also used for 16-bit jump and call instructions.

If you want to load DH with an immediate value, without the intention to use it as the higher 8 bits in the next instruction, then use a regular "LD immediate" instead of PREFIX.

32-BIT INSTRUCTIONS

An instruction handles 32 bits if the L bit is 1. Bit 0 and bit 8 must be 0. For immediate instructions, the 32 bit mode is not available.

The 32 bit instruction needs two cycles:

1) The 32 bit instruction will first handle the lower 16 bits just like a regular 16 bit instruction, with XL, YL or DL registers (and an even memory address). It will also set an internal flag. It wil not increment the program counter, so the next executed instruction will be the same one, although the internal flag will set bit 0 and bit 8 in the instruction register.

2) So now, it will handle the higher 16 bits, with XH, YH or DH registers (and the next memory address). If it is an add or sub instruction, the carry will be used, so that this becomes a real 32-bit add or sub. The internal flag will be cleared and the PC will be incremented again so the execution will continue with the next instruction.

OPERAND SHIFTING

The register instruction format has some bits Q that can be used to define a shift for the operand, or for ignoring the ALU result (for TGT / DTGT).

Q	Q	Q	
D3	**D2**	**D1**	
0	**0**	**0**	SHL4 shift left by 4 bit positions
0	**0**	**1**	SHR4 shift right by 4 bit positions
0	**1**	**0**	SHR8 shift right by 8 bit positions
0	**1**	**1**	SHR12 shift right by 12 bit positions
1	**0**	**0**	Normal (not shifted)
1	**0**	**1**	
1	**1**	**0**	Not store result (used for TGT / DTGT)
1	**1**	**1**	

Note that the shift operates on the 32 bits of a register pair.

The shift can be useful in combination with the instructions LD, ADD, SUB, DADD, DSUB and their 32-bit versions. When used with the 32-bit mode, it can shift a 32-bit value and add or subtract that to/from another 32 bit value, all with a single instruction.

PROGRAM COUNTER

While the instruction executes, the PC will be incremented and the next instruction will be fetched at the same time. This is possible because the program memory is separated from the data memory. This makes it possible to execute every instruction in a single cycle.

This would also mean, that the instruction that follows a jump will always be executed (also if the jump is taken), because this instruction was already fetched (pipeline effect). To prevent this, this instruction (that is fetched immediately after a jump is taken) will be automatically changed to a NOP, so it will not have an effect. Due to this mechanism, a jump that is taken will take two cycles.

There is also a mechanism to save the return address in case of a CALL instruction. A CALL is identical to a LD PC instruction. Every LD to the PC will place a return address in register DH (if it is a normal jump that should not save the PC, use the relative jump. The relative jump will not write to DH).

Examples of PC instructions:

- Relative forward jumps. This is done by simply adding a small immediate value to PC. The relative jump will take only a single 16 bit instruction.

```
BR  40          //  jump 40 instructions forward
```

- Fast absolute jumps or calls: A call to a location (up to $7F) can be done with:

```
TRAP  0x34    // jump or call to 0x0034
```

SUBROUTINE STACK FRAME

The CPU does not have a stack pointer in the classic sense (you could use an address register and increment or decrement it before or after every use, but this is not very efficient). It is the intention that one of the registers, let's say CL, is used as a frame pointer, that points to a section of memory that contains the local variables of the current executing subroutine (P). This section can also be used to store temporary results.

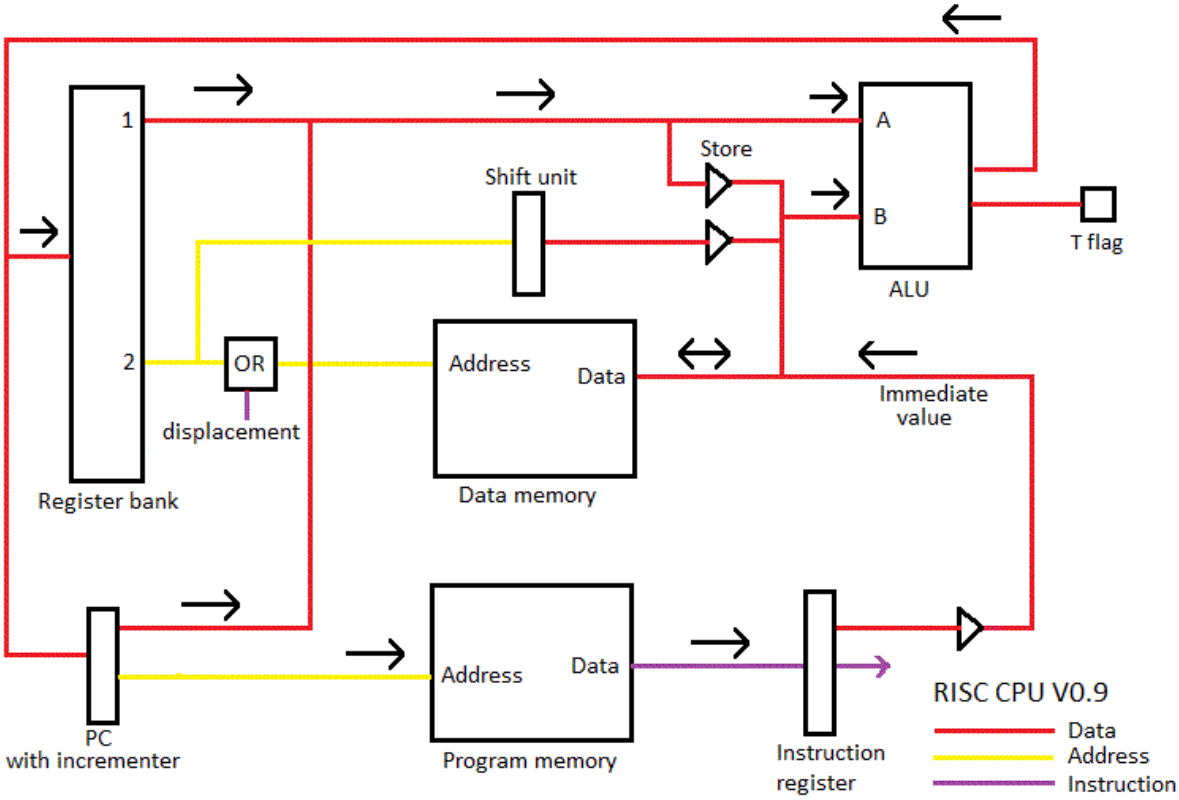
When a subroutine (Q) is called, the frame pointer CL should get a new value, to point to a free section of memory. When Q returns control to P, CL must again get its original value. The simplest way to do this, is to increment or decrement CL by a fixed value of 32 (see next example). Of course you can also use a frame size of 16 or 8. With some effort, you could also design a mixed-size frame system.

```
        CALL PC,$1F28       // call address $1F28 (the assembler will also generate
                            //         an 8-bit immediate prefix instruction)
        ......


$1F28:  ST  [CL+31], DH     // store the return address at top of the caller's frame
        ADD CL,#32          // select next frame
        ......              // 31 locations (CL+0) up to (CL+30) are now available
        .......             // to this subroutine for local storage.
        ADD     CL,# -32    // subtract 32 to go back to previous frame
        JP      [CL+31]     // load PC with return address (will also write PC to DH, but
                            //         this DH value will not be used)
```

If a subroutine is a leaf (does not call other subroutines), it could use zero-page locations for its variables and it is not needed to save DH.

SIMPLIFIED CPU DIAGRAM



The register bank has one write port (at the left side) and two read ports (at the right side). The output of the ALU can only go to the PC or to one of the registers. A register can be stored in memory through the "Store" buffer.

ALU input A can be loaded from register port 1 or from the PC. ALU input B can be loaded from memory, from an immediate value, or from memory port 2 (through a shift unit, the shift amount can also be zero).

Note that this is a simplified diagram that does not show all details.

DECIMAL ADDITION

The following system is used to enable addition for binary coded decimals (BCD code):

The 4 nibbles that enter the ALU at port A will each be incremented by 6 by a special circuit.

The ALU has now on port A a number 0x06 - 0x0F, and on port B 0x00 - 0x09. These are added in the normal binary way. The adder will generate a carry at each nibble if the result in this nibble is higher than 0x0F, and this means the sum is greater than 9 decimal (because 6 was added). The carry of each nibble will be added to the next higher nibble in the same way as for binary addition.

So, for a result below 10, the ALU nibble output is 0x06 - 0x0F (6 too high) without carry, and for results of 10 or higher, the nibble output is 0x00 - 0x09 (that's ok) with the carry set.

When the nibble has no carry, the result must be corrected again, this time from 0x06 - 0x0F to 0x00 - 0x09. At the output of the ALU, there is a special circuit for each nibble that subtracts 6 from this nibble when there was no carry for that nibble.

DECIMAL SUBTRACTION

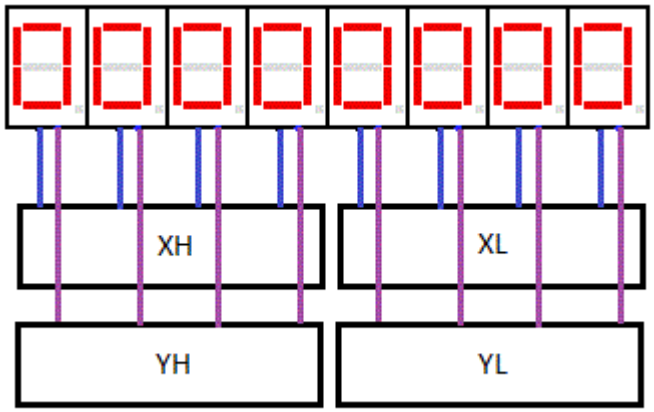
How subtraction works is not shown in the diagram. For subtraction it is needed to complement one of the ALU inputs. This does not have to be implemented in the ALU, but can be implemented in the register bank. The register bank can deliver complemented output on port 2. Setting the ALU function to ADD, and setting the ALU Carry-input to 1, will give the subtraction result of the two selected registers at the output of the ALU (for binary).

Now for decimal coded numbers.

The bitwise complement of a BCD number has a range from 0x06 - 0x0F, this means that the increment-6 circuit for ALU input A (described in decimal addition), must be switched off. The carry dependent subtract-6 is still needed.

7 SEGMENT DECODER

The design can be used for a calculator. The segments of the displays will be directly connected to CPU registers, to save hardware. This is possible because when the answer is shown at the display, the CPU will have finished its calculation, so the registers are not needed for something else at that moment. A specific I/O bit could be used to switch the displays off during calculation (not implemented).



Of each display, 4 segments are connected to the X register, and the other 3 segments are connected to the Y register. Note that each display is connected to the corresponding nibble.

The CPU has a special instruction that decodes all BCD nibbles from a source location to the corresponding segments in the X register, and another instruction that decodes the same BCD nibbles to the corresponding segments in the Y register.

When using 32-bit instructions, only two CPU instructions are needed to show all 8 BCD digits of a 32-bit number on the displays.

Another register is connected to the decimal points of the displays.

MOVE INSTRUCTIONS

LD Load

LD		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Zero page	**10**	**0**	**001**			**0**	**R**			**L**	**0**		**Z**				
Immediate	**78**	**0**	**111**			**1**	**R**			**Immediate data**							
Memory	**90**	**1**	**001**			**0**	**R**			**L**	**A**		**D**				
Register	**98**	**1**	**001**			**1**	**R**			**0**	**S**	**S**	**L**	**Q**	**Q**	**Q**	**S**

The LDL instruction is a 32-bit version for move, the assembler will set the L bit in the opcode.

ST Store

ST		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Zero page	**20**	**0**	**010**			**0**	**R**			**L**	**0**		**Z**				
Memory	**A0**	**1**	**010**			**0**	**R**			**L**	**A**		**D**				

The STL instruction is a 32-bit version for ST, the assembler will set the L bit in the opcode.

LDC Load constant

Load constant from program space. The register defined by A points to a location in the program code. The 16 bit data at that position is moved to the DH register. This instruction takes 2 cycles.

LDC		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	09	**0**	**000**			**1**	**001**			**0**	**A**		**xxxxx**				

PREFIX

PREFIX		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	**1B**	**0**	**001**			**1**	**DH**			**Immediate data**							

The prefix instruction will be generated automatically by the assembler when a 16-bit constant is used. It loads the DH register with 8 bit immediate data.

CLR

CLR		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	**78**	**0**	**111**			**1**	**R**			**00000000**							

Puts value 0 in register R.

ARITHMETIC INSTRUCTIONS

ADD

Adds the source operand to register R.

ADD		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Zero page	30	**0**	**011**			**0**	**R**			**L**	**0**		**Z**				
Immediate	38	**0**	**011**			**1**	**R**			**Immediate data**							
Memory	B0	**1**	**011**			**0**	**R**			**L**	**A**		**D**				
Register	B8	**1**	**011**			**1**	**R**			**0**	**S**	**S**	**L**	**Q**	**Q**	**Q**	**S**

SUB

Subtracts register S from register R.

SUB		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register	C8	**1**	**100**			**1**	**R**			**0**	**S**	**S**	**L**	**Q**	**Q**	**Q**	**S**

TGT

Test Greater-Than. Subtract without storing result. Used to compare two numbers.

TGT		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register	C8	**1**	**100**			**1**	**R**			**0**	**S**	**S**	**L**	**1**	**1**	**0**	**S**

SUB immediate

The assembler will make the immediate operand negative and build an immediate ADD instruction.

SUB		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	38	**0**	**011**			**1**	**R**			**Neg (Immediate data)**							

INC

INC		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	38	**0**	**011**			**1**	**R**			**00000001**							

DEC

DEC		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	38	**0**	**011**			**1**	**R**			**11111111**							

DADD

Decimal ADD, uses binary coded decimal number format (4 digits in a 16 bit word).

DADD		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Zero page	54	**0**	**101**			**0**	**R**			**L**	**0**		**Z**				
Immediate	5C	**0**	**101**			**1**	**R**			**Immediate data**							
Memory	D0	**1**	**101**			**0**	**R**			**L**	**A**		**D**				
Register	D8	**1**	**101**			**1**	**R**			**0**	**S**	**S**	**L**	**Q**	**Q**	**Q**	**S**

DSUB

Decimal SUB, uses binary coded decimal number format

DSUB		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register	**E8**	**1**	**110**			**1**	**R**			**0**	**S**	**S**	**L**	**Q**	**Q**	**Q**	**S**

DTGT

Decimal Test Greater-Than

Decimal subtract without storing result. Used to compare two numbers.

DTGT		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register	**E8**	**1**	**110**			**1**	**R**			**0**	**S**	**S**	**L**	**1**	**1**	**0**	**S**

The ADDL, DADDL, SUBL, DSUBL, TGTL and DTGTL instructions are 32-bit versions of the arithmetic instructions, the assembler will set the L bit in the opcode.

DMPY

DMPY will do a conditional decimal ADD. It is intended to be used as a part of a multiplication. Mostly used in 32-bit version (DMPYL).

DMPY		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Zero page	**70**	**0**	**111**			**0**	**1**	**R**		**L**	**0**		**Z**				
Memory	**F0**	**1**	**111**			**0**	**R**			**L**	**A**		**D**				

In the instruction, bit 2 and 3 determine which bits will determine what happens (note that these bits 2 and 3 are also part of the operand address):

If instruction.bit2=0 and instruction.bit3=0 :

- The result of the decimal ADD will be written to the register if bit 0 of DL is 1.
- The next instruction will only be executed if bit 1 of DL is 1.

If instruction.bit2=1 and instruction.bit3=1 :

- The result of the decimal ADD will be written to the register if bit 2 of DL is 1.
- The next instruction will only be executed if bit 3 of DL is 1.

DMPY USE

The DMPYL instruction is used for multiplying an 8-digit decimal number with a single decimal digit (using a sequence of only 4 instructions).

The decimal number is stored in memory, in four different versions (at four locations):

1. Location Z1, the decimal number itself.
2. Location Z2, the decimal number multiplied by 2.
3. Location Z4, the decimal number multiplied by 4.
4. Location Z8, the decimal number multiplied by 8.

The decimal digit is the right-most digit in the DL register. The result is added to the 8-digit X register. The algorithm is as follows:

- If bit 0 of DL is 1, add Z1 to X (32 bit decimal add)
- If bit 1 of DL is 1, add Z2 to X
- If bit 2 of DL is 1, add Z4 to X
- If bit 3 of DL is 1, add Z8 to X

This is the instruction sequence for the algorithm:

```
DMPYL X,[Z1]  ; If bit 0 of DL is 1, add Z1 to X. If bit 1 of DL is 0, skip next instruction

DADDL X,[Z2]  ; add Z2 to X. (This instruction will be skipped  if bit 1 of DL is 0)

DMPYL X,[Z4]  ; If bit 2 of DL is 1, add Z4 to X. If bit 3 of DL is 0, skip next instruction

DADDL X,[Z8]  ; add Z8 to X. (This instruction will be skipped  if bit 3 of DL is 0)
```

Note that Z1 and Z4 must be at certain memory positions in order to have the correct values for bit 2 and bit 3 of the instruction.

In worst case, 2 cycles per instruction are executed (8 cycles total). In the best case, the 2nd and 4th instruction are skipped, so only 2 instructions are executed (4 cycles total).

As an optimization, a DMPYL instruction that does not write the register, could skip the second half of the 32-bit instruction. Now, in best case, only a total of 2 cycles are needed (this happens when the multiplying digit is zero). This is not implemented.

LOGIC INSTRUCTIONS

AND

AND		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	68	**0**	**110**			**1**	**R**			**Immediate data**							
Zero page	60	**0**	**110**			**0**	**R**			**L**	**1**		**Z**				

OR

OR		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	18	**0**	**001**			**1**	**R**			**Immediate data**							
Zero page	30	**0**	**011**			**0**	**R**			**L**	**1**		**Z**				

The OR with immediate operand is not available for the DH register. That opcode is used for PREFIX.

XOR

XOR		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	48	**0**	**100**			**1**	**R**			**Immediate data**							
Zero page	10	**0**	**001**			**0**	**R**			**L**	**1**		**Z**				

TEQ Test equality

TEQ		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Zero page	**00**	**0**	**000**			**0**	**R**			**L**	**0**		**Z**				
Immediate	**08**	**0**	**000**			**1**	**R**			**Immediate data**							
Memory	**80**	**1**	**000**			**0**	**R**			**L**	**A**			**D**			
Register	**88**	**1**	**000**			**1**	**R**			**0**	**S**	**S**	**L**	**Q**	**Q**	**Q**	**S**

TSTB

Testbits. Set flag T according to the result of an AND between the register and the operand. The result of the AND is not stored.

TSTB		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	**28**	**0**	**010**			**1**	**R**			**Immediate data**							
Register	**A8**	**1**	**010**			**1**	**R**			**0**	**S**	**S**	**L**	**Q**	**Q**	**Q**	**S**

SEGA

SEGA		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Zero page	00	0	000			0	R			L	1	0	Z				

This will convert each nibble from the source location to segments F,C, D and B (for driving a 7-segment display) in the destination register. The destination register may be the same as the source register.

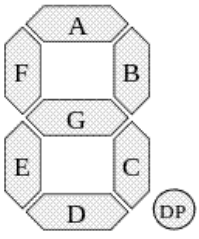
The result has inverted logic (a 0 means the segment is ON). Input nibbles above 9 are displayed as 2 to 7.

SEGB

SEGB		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Zero page	00	0	000			0	R			L	1	1	Z				

SEGB works just as SEGA, but for segments G, C, A, E. Note that segment C is produced by both instructions.

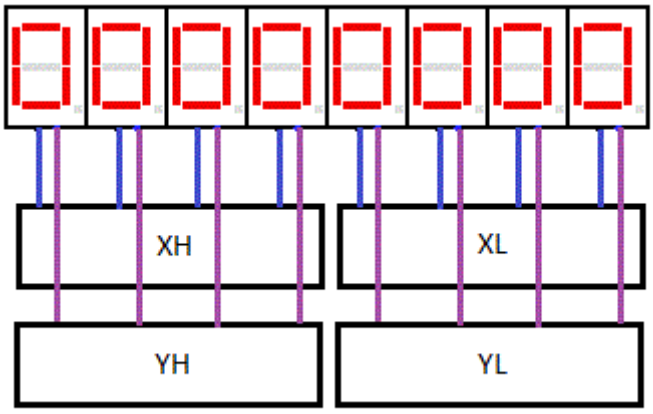
Digit	Input				SEGA result				SEGB result			
					F	C	D	B	G	C	A	E
	D3	D2	D1	D0	D3	D2	D1	D0	D3	D2	D1	D0
0	0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	1	1	0	1	0	1	0	1	1
2	0	0	1	0	1	1	0	0	0	1	0	0
3	0	0	1	1	1	0	0	0	0	0	0	1
4	0	1	0	0	0	0	1	0	0	0	1	1
5	0	1	0	1	0	0	0	1	0	0	0	1
6	0	1	1	0	0	0	0	1	0	0	0	0
7	0	1	1	1	1	0	1	0	1	0	0	1
8	1	0	0	0	0	0	0	0	0	0	0	0
9	1	0	0	1	0	0	0	0	0	0	0	1



DISPLAY SIGNALS

Display 1-12	12	11	10	9	8	7	6	5	4	3	2	1
	sign	mantissa								Exp sign	exponent	
GCAE	**C**	**XH**				**XL**				**C**	**C**	
FCDB	**DL**	**YH**				**YL**					**DL**	
bits	15-12	15-12	11-8	7-4	3-0	15-12	11-8	7-4	3-0	11	7-4	3-0
Decimal point (DH bit nr)		15	14	13	12	11	10	9	8			

The sign of the mantissa has all segments connected. The sign of the exponent has only the segment G connected.



The picture shows the connection of the mantissa.

The upper 8 bits of the DH register control the decimal points of the mantissa digits (not in the picture).

PROGRAM FLOW INSTRUCTIONS

Jumps or calls to a label (immediate instruction format) can have two sizes:

- Short distance. The jump address or displacement (-128 .. +127) is in the immediate field of the 16 bit instruction.
- Long distance. The assembler generates a prefix instruction for the upper 8 bits of the immediate operand, the lower 8 bits are in the immediate field of the instruction itself. This can not be used for conditional instructions because the prefix instruction will alter the T flag.

Two different methods can be used to determine the jump address:

- Relative jumps. The (signed) displacement is added to the PC. These instructions do not generate a return address.
- Absolute jumps. These place the return address in the DH register.

Note that, when a branch or jump is taken, the instruction that follows the branch or jump is converted to a NOP.

BR Branch (Jump relative)

BR		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Zero page	31	**0**	**011**			**0**	**001**			**L**	**0**		**Z**				
Immediate	39	**0**	**011**			**1**	**001**			**Immediate data**							
Memory	B1	**1**	**011**			**0**	**001**			**L**	**A**		**D**				
Register	B9	**1**	**011**			**1**	**001**			**0**	**S**	**S**	**L**	**Q**	**Q**	**Q**	**S**

BRT		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	**69**	**0**	**110**			**1**	**001**			**Immediate data**							

BRF		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	**49**	**0**	**100**			**1**	**001**			**Immediate data**							

Short distance jump to label: Use the BR assembly instruction. The assembler generates an Add-immediate-to-PC and will require a short jump distance. (The relative jump will not leave a return address).

Long distance jump to label: Use the JP assembly instruction. The assembler will generate a long-distance relative jump (The relative jump will not leave a return address).

The BRT instruction only does the branch if the T flag is 1. The BRF instruction only does the branch if the T flag is 0.

Alias instructions: BRC, BRNC, BRZ, BRNZ can also be used (but all test the same flag T)

CALL / TRAP

CALL		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Zero page	**11**	**0**	**001**			**0**	**001**			**L**	**0**		**Z**				
Immediate	**19**	**0**	**001**			**1**	**001**			**Immediate data**							
Memory	**91**	**1**	**001**			**0**	**001**			**L**	**A**		**D**				
Register	**99**	**1**	**001**			**1**	**001**			**0**	**S**	**S**	**L**	**Q**	**Q**	**Q**	**S**

CALLT		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	**79**	**0**	**111**			**1**	**001**			**Immediate data**							

CALLF		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	**59**	**0**	**101**			**1**	**001**			**Immediate data**							

Long distance call: Use the CALL assembly instruction. The assembler will generate a long-distance absolute jump (this jump will leave a return address in the DH register).

Call to zero-page or top-page: Use the TRAP assembly instruction. Can only be used if the destination address is 0..0x7F or 0xFF80..0xFFFF. The assembler will generate a absolute jump (this jump will leave a return address in the DH register). The subroutine address is part of the 16-bit instruction as a signed 8 bit value, so it is a fast and short way to call a subroutine.

The processor will, when the PC gets a new value, first increment the PC before it fetches the new instruction. This has as effect that the instruction after a call instruction is skipped, and this was not intended. As a workaround, it is recommended that every call is followed by a NOP. Also, when doing a computed jump (JP instruction), the first instruction will be skipped. This can be prevented by loading the PC with the instruction address minus one.

For jumps and branches to labels, the assembler will do the correction.

JP

JP		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Zero page	**11**	**0**	**001**			**0**	**001**			**L**	**0**		**Z**				
Memory	**91**	**1**	**001**			**0**	**001**			**L**	**A**		**D**				
Register	**99**	**1**	**001**			**1**	**001**			**0**	**S**	**S**	**L**	**Q**	**Q**	**Q**	**S**

The JP instruction can be used to load the PC with the contents of a register or the contents of a memory location as a source. There is no distinction between short or long distance. The address of the next instruction will be placed in the DH register. (If you use JP with a label (immediate mode), a relative jump will be generated as explained in the BR section).

TBRZ / TBRNZ

This instruction will test a specified bit in a register pair, and branch if the bit was zero or non-zero.

Bit to test	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**1**	**000**			**1**	**P**	**0**	**0**	**1**	**S**	**S**	**Branch distance**				
1	**1**	**000**			**1**	**P**	**1**	**0**	**1**	**S**	**S**	**Branch distance**				
2	**1**	**001**			**1**	**P**	**0**	**0**	**1**	**S**	**S**	**Branch distance**				
3	**1**	**001**			**1**	**P**	**1**	**0**	**1**	**S**	**S**	**Branch distance**				
4	**1**	**010**			**1**	**P**	**0**	**0**	**1**	**S**	**S**	**Branch distance**				
7	**1**	**010**			**1**	**P**	**1**	**0**	**1**	**S**	**S**	**Branch distance**				
15	**1**	**011**			**1**	**P**	**0**	**0**	**1**	**S**	**S**	**Branch distance**				
31	**1**	**011**			**1**	**P**	**1**	**0**	**1**	**S**	**S**	**Branch distance**				

P = 0 TBRZ Test Bit and Branch on Zero

P = 1 TBRNZ Test Bit and Branch on Non Zero

The branch is always forward, the distance is from 0 to 31.

SS selects one of the register pairs D, X, Y or the register CL.

If bit 8 in the instruction is 1, the instruction will operate on the CL register instead of the PC register (so it will conditionally add the 0-31 constant to the CL register).

The T flag is not involved in this instruction.

NOP

No operation (coded as TEQ CL,[0] so it might change the T flag)

NOP		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	00	**0**	**000**			**0**	**000**			**0**	**0000000**						

HLT

Halt. Will stop the CPU and wait for a key press. If a user key is pressed, the CPU will continue with the inverted key code in the DH register.

HLT		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	81	**1**	**000**			**0**	**001**			**x**	**xxxxxxx**						

--- end ---