f8 manual

Philipp Klaus Krause

2026-01-29

Chapter 1

Architecture

1.1 Introduction

The f8 is an 8/16-bit architecture based on lessons learned from many years of working with existing 8/16-bit architectures, their strengths and weaknesses, in particular when targeted by a C compiler. It emphasizes efficient use of memory (for both the program and data), and is meant for use cases where the power of a 32-bit ARM or RISC-V is not needed. For the lower end, there is also the f8l variant, which has a smaller instruction set, and can be implemented with fewer gates / less silicon area.

At a high level, this means:

- An efficient stackpointer-relative addressing mode for efficient handling of local variables
- A unified address space for efficient pointer access
- Having a few data / pointer registers for temporary storage
- Hardware multithreading and support for atomics to replace peripheral hardware

1.2 Safety and Security

With a 16-bit logical address space, and the intended use, the f8 cannot afford to use virtual memory, which would be necessary for guard pages. Instead we use a simple mitigation for memory safety issues, that resets the f8 on three error conditions:

- Attempts to write via null pointers reset the f8 (via an I/O register at address 0x0000).

- Attempts to execute 0-initialized memory reset the f8 (via the trap instruction that has opcode 0x00).
- A simple watchdog can reset the f8.

The f8 is little-endian. The stack grows downward. There is a 16-bit flat address space. Memory reads have no side-effects. All instructions execute atomically.

1.3 Memory Map

I/O is mapped starting from 0x0000 (as required by the safety / security feature regarding null pointer reads). RAM is mapped up to 0x3fff. (P)ROM/Flash from 0x4000. The idea is to allow for up to 48 KB of (P)ROM/Flash and up to 8 KB of RAM directly in the 16-bit address space.

1.4 Registers

There is an 8-bit flag register `f`, which contains the half-carry flag `h`, the carry flag `c`, the negative flag `n`, the zero flag `z`, the overflow / parity flag `o`, and three reserved bits. Unless otherwise noted, instructions leave the reserved flags in an undefined state. The reserved bits should not be written by the user except via the `xch f, (n, sp)` instruction.

After reset, `pc` has the value 0x4000 the value of the other registers mentioned in this section after reset is unspecified.

0					7
`h`	`c`	`n`	`z`	`o`	reserved

There are a 16-bit program counter `pc` and a 16-bit stack pointer `sp`.

0 … 15
`pc`

0 … 15
`sp`

There are three 16-bit general-purpose registers, each consisting of two 8.bit registers.

0 … 7	8 … 15
`x`	
`xl`	`xh`

0 … 7	8 … 15
`y`	
`yl`	`yh`

<table>
<tr><td>0</td><td>7 8</td><td>15</td></tr>
<tr><td colspan="3">z</td></tr>
<tr><td>zl</td><td colspan="2">zh</td></tr>
</table>

1.5 Instructions

The lightweight f8 instruction subset f8l is meant for smaller cores. This simplification comes at the cost of a reduction in performance and an increase in code size.

Instructions have up to 3 source and up to 2 destination operands. At most one source and one destination operand are in memory. All destination operands in general-purpose registers need to be one 16-bit register or part of the same 16-bit general-purpose register.

Each instruction is encoded by 1 to 4 bytes: an optional prefix byte is followed by the opcode byte and 0 to 2 operand bytes.

There are 8 prefix bytes:

Prefix	semantics	group
`swapop`	swap operands	0
`altacc1`	alternative accumulator `xh` instead of `xl`	1
`altacc2`	alternative accumulator `yl` instead of `xl`, `z` instead of `y`	2
`altacc4`	alternative accumulator `yh` instead of `xl`, `z` instead of `y`	2
`altacc3`	alternative accumulator `zl` instead of `xl`, `x` instead of `y`	2
`altacc5`	alternative accumulator `zh` instead of `xl`	2

1.6 Addressing Modes

`xl`, `xh`, `yl`, `yh`, `zl`, `zh`, `f`	8-bit register
`x`, `y`, `z`, `sp`	16-bit register
`#i`	8-bit immediate
`#ii`	16-bit immediate
`#d`	8-bit immediate sign-extended to 16-bit
`mm`	direct
`(n, sp)`, `(n, y)`	indexed with 8-bit offset
`(nn, z)`	indexed with 16-bit offset
`(x)`, `(y)`, `(z)`	indirect

Chapter 2

Instructions

op8_2	Any of xh, yl, yh, zl, #i, mm, (n, sp), (nn, z).
op8_2ni	Any of xh, yl, yh, zl, mm, (n, sp), (nn, z).
altacc8	Any of xh, yl, yh, zl, zh.
op16_2	Any of x, #ii, mm, (n, sp).
op16_2ni	Any of x, mm, (n, sp).
altacc16	Any of x, z.
op8_1	Any of xl, mm, (n, sp), (n, y).
op16_1	Any of y, mm, (n, sp), (nn, z).

2.1 8-bit 2-operand instructions

Instructions where the same location is used for altacc8 and op8_2 operand are not valid.

2.1.1 adc: 8-bit addition with carry

Assembler code	Operation	f8l
adc xl, op8_2	xl = xl + op8_2 + c	Yes
adc altacc8, op8_2	altacc8 = altacc8 + op8_2 + c	Yes
adc op8_2ni, xl	op8_2ni = op8_2ni + xl + c	Yes

Affected Flags

hcnzo

2.1.2 add: 8-bit addition

Assembler code	Operation	f8l
add xl, op8_2	xl = xl + op8_2	Yes
add altacc8, op8_2	altacc8 = altacc8 + op8_2	Yes
add op8_2ni, xl	op8_2ni = op8_2ni + xl	Yes

Affected Flags

hcnzo

2.1.3 and: 8-bit bitwise and

Assembler code	Operation	f8l
`and xl, op8_2`	`xl = xl & op8_2`	Yes
`and altacc8, op8_2`	`altacc8 = altacc8 & op8_2`	Yes
`and op8_2ni, xl`	`op8_2ni = op8_2ni & xl`	Yes

Affected Flags

nz

2.1.4 cp: 8-bit comparison

Subtraction where the result is used to update the flags only.

Assembler code	Operation	f8l
`cp xl, op8_2`	`xl + ~op8_2 + 1`	Yes
`cp altacc8, op8_2`	`altacc8 + ~op8_2 + 1`	Yes
`cp op8_2, xl`	`op8_2 + ~xl + 1`	No

Affected Flags

hcnzo

2.1.5 or: 8-bit bitwise or

Assembler code	Operation	f8l
`or xl, op8_2`	`xl = xl | op8_2`	Yes
`or altacc8, op8_2`	`altacc8 = altacc8 | op8_2`	Yes
`or op8_2ni, xl`	`op8_2ni = op8_2ni | xl`	Yes

Affected Flags

nz

2.1.6 sbc: 8-bit subtraction with carry

Assembler code	Operation	f8l
`sbc xl, op8_2ni`	`xl = xl + ~op8_2ni + c`	Yes
`sbc altacc8, op8_2ni`	`altacc8 = altacc8 + ~op8_2ni + c`	Yes
`sbc op8_2ni, xl`	`op8_2ni = op8_2ni + ~xl + c`	No

Affected Flags

hcnzo

2.1.7 sub: 8-bit subtraction

Assembler code	Operation	f8l
`sub xl, op8_2ni`	`xl = xl + ~op8_2ni + 1`	Yes
`sub altacc8, op8_2ni`	`altacc8 = altacc8 + ~op8_2ni + 1`	Yes
`sub op8_2ni, xl`	`op8_2ni = op8_2ni + ~xl + 1`	No

Affected Flags

`hcnzo`

2.1.8 xor: 8-bit bitwise exclusive or

Assembler code	Operation	f8l
`xor xl, op8_2`	`xl = xl ^ op8_2`	Yes
`xor altacc8, op8_2`	`altacc8 = altacc8 ^ op8_2`	Yes
`xor op8_2ni, xl`	`op8_2ni = op8_2ni ^ xl`	Yes

Affected Flags

`nz`

2.2 16-bit 2-operand-instructions

Todo: Document possible altacc prefixes.

2.2.1 adcw: 16-bit addition with carry

Assembler code	Operation	f8l
`adcw y, op16_2`	`y = y + op16_2 + c`	No
`adcw op16_2ni, y`	`op16_2ni = op16_2ni + y + c`	No

Affected Flags

`cnzo`

2.2.2 addw: 16-bit addition

Assembler code	Operation	f8l
`addw y, op16_2`	`y = y + op16_2`	No
`addw op16_2ni, y`	`op16_2ni = op16_2ni + y`	No

Affected Flags

`cnzo`

2.2.3 orw: 16-bit bitwise or

todo: do we really want the effect on o here? If yes, why not on the 8-bit logic ops?

Assembler code	Operation	f8l
`orw y, op16_2`	`y = y | op16_2`	No
`orw op16_2ni, y`	`op16_2ni = op16_2ni | y`	No

Affected Flags

`nzo`

2.2.4 sbcw: 16-bit subtraction with carry

Assembler code	Operation	f8l
`sbcw y, op16_2ni`	`y = y + ~op16_2ni + c`	No
`sbcw op16_2ni, y`	`op16_2 = ~op16_2ni + y + c`	No

Affected Flags

`cnzo`

2.2.5 subw: 16-bit subtraction

Assembler code	Operation	f8l
`subw y, op16_2ni`	`y = y + ~op16_2ni + 1`	No
`subw op16_2ni, y`	`op16_2 = ~op16_2ni + y + 1`	No

Affected Flags

`cnzo`

2.2.6 xorw: 16-bit bitwise exclusive or

todo: do we really want the effect on o here? If yes, why not on the 8-bit logic ops?

Assembler code	Operation	f8l
`xorw y, op16_2`	`y = y ^ op16_2`	No
`xorw op16_2ni, y`	`op16_ni2 = op16_2ni ^ y`	No

Affected Flags

`nzo`

2.3 8-bit 1-operand-instructions

2.3.1 clr: 8-bit clear

Assembler code	Operation	f8l
`clr op8_1`	`op8 = 0x00`	Yes, except (n, y)
`clr altacc8`	`altacc8 = 0x00`	Yes

Affected Flags

none

Rationale

Initializing or setting an object, or parts thereof, to 0 is common, so having a dedicated instruction is worth it vs. using `ld`.

2.3.2 dec: 8-bit decrement

Assembler code	Operation	f8l
`dec op8_1`	`op8 = op8 + -1`	Yes, except (n, y)
`dec altacc8`	`altacc8 = altacc8 + -1`	Yes

Affected Flags

`hcnzo`

2.3.3 inc: 8-bit increment

Assembler code	Operation	f8l
`inc op8_1`	`op8 = op8 + 1`	Yes, except (n, y)
`inc altacc8`	`altacc8 = altacc8 + 1`	Yes

Affected Flags

`hcnzo`

2.3.4 push: 8-bit push onto stack

Assembler code	Operation	f8l
`push op8_1`	`(--sp) = op8`	Yes, except (n, y)
`push altacc8`	`(--sp) = altacc8`	Yes

Affected Flags

none

Rationale

8-bit stack parameters can be passed easily via this instruction. Registers can be saved temporarily (e.g. for the duration of a function call, or in the middle of a complex computation). Not affecting flags makes the instruction more useful for saving registers in the middle of a long addition/subtraction/comparison/multiplication.

2.3.5 sll: 8-bit shift left logical

Assembler code	Operation	f8l
`sll op8_1`	`c = (op8 & 0x80) >> 7` `op8 = op8 << 1`	Yes, except (n, y)
`sll altacc8`	`c = (op8 & 0x80) >> 7` `altacc8 = altacc8 << 1`	Yes

Affected Flags

`cnz`

2.3.6 srl: 8-bit shift right logical

Assembler code	Operation	f8l
`srl op8_1`	`c = op8 & 0x01` `op8 = op8 >> 1`	Yes, except (n, y)
`srl altacc8`	`c = op8 & 0x01` `altacc8 = altacc8 >> 1`	Yes

Affected Flags

`cnz`

2.3.7 rlc: 8-bit rotate left through carry

Assembler code	Operation	f8l

`rlc op8_1`	`tc = (op8 & 0x80) >> 7`
	`op8 = (op8 << 1) | c`
	`c = tc`
		Yes, except (n, y)

`rlc altacc8`	`tc = (altacc8 & 0x80) >> 7`
	`altacc8 = (altacc8 << 1) | c`
	`c = tc`
		Yes

Affected Flags

`cnz`

2.3.8 rrc: 8-bit rotate right through carry

Assembler code	Operation	f8l

`rrc op8_1`	`tc = op8 & 0x01`
	`op8 = (op8 >> 1) | (c << 7)`
	`c = tc`
		Yes, except (n, y)

`rrc altacc8`	`tc = altacc8 & 0x01`
	`altacc8 = (altacc8 >> 1) | (c << 7)`
	`c = tc`
		Yes

Affected Flags

`cnz`

2.3.9 tst: 8-bit test

Set n and z flags according to value of operand, o flag by parity, reset c.

Assembler code	Operation	f8l
`tst op8_1`	`op8`	Yes, except (n, y)
`tst altacc8`	`altacc8`	Yes

Affected Flags

`cnzo`

Rationale

Testing a variable for zero or being nonnegative is common. We also want a way to check parity and reset the carry flag. Making that a side-effect in this instructions saves opcodes for other uses.

2.4 16-bit 1-operand instructions

2.4.1 adcw: 16-bit addition with carry

Assembler code	Operation	f8l
`adcw op16_1`	`op16 = op16 + c`	No
`adcw altacc16`	`altacc16 = altacc16 + c`	No

Affected Flags

`cnzo`

Rationale

In additions, often one operand is a small integer. This instructions allows an efficient implementation of the handling of the upper bits.

2.4.2 clrw: 16-bit clear

Assembler code	Operation	f8l
`clrw op16_1`	`op16 = 0x0000`	Yes
`clrw altacc15`	`altacc16 = 0x0000`	Yes

Affected Flags

none

Rationale

Initalizing or setting an object, or parts thereof, to 0 is common, so having a dedicated instruction is worth it vs. using `ldw`.

2.4.3 incw: 16-bit increment

Assembler code	Operation	f8l
`incw op16_1`	`op16 = op16 + 1`	Yes
`incw altacc16`	`altacc16 = altacc16 + 1`	Yes

Affected Flags

`cnzo`

Rationale

Incrementing a variable is common, so having a dedicated instruction is worth it vs. using `addw`. Affecting the carry flag makes this instruction less useful for incrementing pointers in the middle of wider arithmetic operations, but makes it more useful for incrementing wider variables.

2.4.4 pushw: 16-bit push onto stack

Assembler code	Operation	f8l
`pushw op16_1`	`sp -= 2; (sp) = op16`	Yes
`pushw altacc16`	`sp -= 2; (sp) = altacc16`	Yes

Affected Flags

none

Rationale

16-bit stack parameters can be passed easily via this instruction. Registers can be saved temporarily for a function call, or in the middle of wider arithmetic operations; for the latter use, it is important that this instruction does not affect any flags.

2.4.5 sbcw: 16-bit subtraction with carry

Assembler code	Operation	f8l
`sbcw op16_1`	`op16 = op16 + 0xffff + c`	No
`sbcw altacc16`	`altacc16 = altacc16 + 0xffff + c`	No

Affected Flags

`cnzo`

Rationale

In subtractions, often one operand is a small integer. This instructions allows an efficient implementation of the handling of the upper bits.

2.4.6 tstw: 16-bit test

Set n and z flags according to value of operand, o flag by parity, set c.

Assembler code	Operation	f8l
`tstw op16_1`	`op16`	Yes
`tstw altacc16`	`altacc16`	Yes

Affected Flags

`cnzo`

Rationale

Testing a variable for zero or being nonnegative is common. We also want a way to check parity and set the carry flag. Making that a side-effect in this instructions saves opcodes for other uses.

2.5 8-bit loads

2.5.1 ld: 8-bit load from memory

Assembler code	Operation	f8l
`ld xl, #i`	`xl = #i`	Yes
`ld altacc8, #i`	`altacc8 = #i`	Yes
`ld xl, mm`	`xl = mm`	Yes
`ld altacc8, mm`	`altacc8 = mm`	Yes
`ld xl, (n, sp)`	`xl = (n, sp)`	Yes
`ld altacc8, (n, sp)`	`altacc8 = (n, sp)`	Yes
`ld xl, (nn, z)`	`xl = (nn, z)`	Yes
`ld altacc8, (nn, z)`	`altacc8 = (nn, z)`	Yes
`ld xl, (y)`	`xl = xh`	Yes
`ld altacc8, (altacc16)`	`altacc8 = (altacc16)`	Yes
`ld xl, (n, y)`	`xl = (n, y)`	No
`ld altacc8, (n, y)`	`altacc8 = (n, y)`	No

Affected Flags

`nz`

Rationale

To be able to handle 8-bit data efficiently, we need a variety of 8-bit load instructions. Often, data is being tested for being (non)zero or (non)negative after being loaded from memory, so having `ld` update the `n` and `z` flags can save a `tst` instruction.

2.5.2 ld: 8-bit load from register

Assembler code	Operation	f8l
`ld xl, xh`	`xl = xh`	Yes
`ld xh, xl`	`xh = xl`	Yes
`ld altacc8, xh`	`altacc8 = xh`	Yes
`ld xl, yl`	`xl = yl`	Yes
`ld yl, xl`	`yl = xl`	Yes
`ld altacc8, yl`	`altacc8 = yl`	Yes
`ld xl, yh`	`xl = yh`	Yes
`ld yh, xl`	`yh = xl`	Yes
`ld altacc8, yh`	`altacc8 = yh`	Yes
`ld xl, zl`	`xl = zl`	Yes
`ld zl, xl`	`zl = xl`	Yes
`ld altacc8, zl`	`altacc8 = zl`	Yes
`ld xl, zh`	`xl = zh`	Yes
`ld zh, xl`	`zh = xl`	Yes
`ld altacc8, zh`	`altacc8 = zh`	Yes
`ld mm, xl`	`mm = xl`	Yes
`ld mm, altacc8`	`mm = altacc8`	Yes
`ld (n, sp), xl`	`(n, sp) = xl`	Yes
`ld (n, sp), altacc8`	`(n, sp) = altacc8`	Yes
`ld (nn, z), xl`	`(nn, z) = altacc8`	Yes
`ld (nn, z), altacc8`	`(nn, z) = altacc8`	Yes
`ld (y), xl`	`(y) = xl`	Yes
`ld (altacc16), altacc8`	`(altacc16) = altacc8`	Yes
`ld (n, y), xl`	`(n, y) = xl`	No
`ld (n, y), altacc8`	`(n, y) = altacc8`	No

Affected Flags

none

Rationale

To be able to handle 8-bit data efficiently, we need a variety of 8-bit load instructions.

2.5.3 ldi: 8-bit load with increment

Flags according to old `(z)`.

Assembler code	Operation	f8l
`ldi (n, y), (z)`	`(n, y) = (z); z += 1;`	No

Affected Flags

`nz`

Rationale

Copying larger blocks of data is a very common operation, both explicitly via `memcpy`, and when assigning larger variables. While `ldwi` has higher throughput, this instruction can be used for the first or last byte when copying an odd number of bytes. Due to its effect on the `z` flag, it is also useful, when the value of the individual copied byte matters, in particular for implementing `strlen` and `strnlen`.

2.6 16-bit loads

2.6.1 ldw: 16-bit load from memory

Assembler code	Operation	f8l
`ldw y, #ii`	`y = #ii`	Yes
`ldw altacc16, #ii`	`altacc16 = #ii`	Yes
`ldw y, mm`	`y = mm`	Yes
`ldw altacc16, mm`	`altacc16 = mm`	Yes
`ldw y, (n, sp)`	`y = (n, sp)`	Yes
`ldw altacc16, (n, sp)`	`altacc16 = (n, sp)`	Yes
`ldw y, (nn, z)`	`y = (nn, z)`	Yes
`ldw altacc16, (nn, z)`	`altacc16 = (nn, z)`	Yes
`ldw y, (n, y)`	`y = (n, y)`	No
`ldw altacc16, (n, y)`	`altacc16 = (n, y)`	No
`ldw y, (y)`	`y = (y)`	Yes
`ldw altacc16, (altacc16)`	`altacc16 = (altacc16)`	Yes
`ldw y, #d`	`y = #d`	Yes
`ldw altacc16, #d`	`altacc16 = #d`	Yes
`ldw x, (y)`	`x = (y)`	Yes
`ldw y, (z)`	`y = (z)`	Yes
`ldw z, (x)`	`z = (x)`	Yes
`ldw z, (y)`	`z = (y)`	Yes

Affected Flags

`nz`

Rationale

To be able to handle 16-bit data efficiently, we need a variety of 16-bit load instructions. Often, data is being tested for being (non)zero or (non)negative after being loaded from memory, so having `ldw` update the `n` and `z` flags can save a `tstw` instruction.

2.6.2 ldw 16-bit load from register

Assembler code	Operation	f8l
`ldw y, x`	`y = x`	Yes
`ldw y, z`	`y = z`	Yes
`ldw z, x`	`z = x`	Yes
`ldw x, z`	`x = z`	Yes
`ldw x, y`	`x = y`	Yes
`ldw z, y`	`z = y`	Yes
`ldw mm, y`	`mm = y`	Yes
`ldw mm, altacc16`	`mm = altacc16`	Yes
`ldw (n, sp), y`	`(n, sp) = y`	Yes
`ldw (n, sp), altacc16`	`(n, sp) = altacc16`	Yes
`ldw (nn, z), y`	`(nn, z) = y`	Yes
`ldw (nn, z), altacc16`	`(nn, z) = altacc16`	Yes
`ldw (y), x`	`(y) = x`	Yes
`ldw (z), y`	`(z) = y`	Yes
`ldw (x), z`	`(x) = z`	Yes
`ldw (y), z`	`(y) = z`	Yes
`ldw (n, y), x`	`(n, y) = x`	No
`ldw y, sp`	`y = sp`	Yes
`ldw sp, y`	`sp = y`	Yes
`ldw altacc16, sp`	`altacc16 = sp`	Yes
`ldw ((d, sp)), y`	`(d, sp) = y`	No
`ldw ((d, sp)), altacc16`	`(d, sp) = altacc16`	No

Affected Flags

none

Rationale

To be able to handle 16-bit data efficiently, we need a variety of 16-bit load instructions.

2.6.3 ldwi: 16-bit load with increment

Flags according to old `(z)`.

Assembler code	Operation	f8l
`ldwi (n, y), (z)`	`(n, y) = (z); z += 2;`	No

Affected Flags

`nz`

Rationale

Copying larger blocks of data is a very common operation, both explicitly via `memcpy`, and when assigning larger variables. This instruction substantially increases throughput vs. using individual loads and stores. The effect on the `z` flag makes it suitable for copying zero-terminated UTF-16 strings.

2.6.4 sex: sign-extend

Assembler code	Operation	f8l
`sex y, xl`	`y = (int8_t)xl`	No
`sex altacc16, altacc8`	`altacc16 = (int8_t)altacc8`	No

Affected Flags

`nz`

Rationale

When aiming for memory efficiency, it is important to be able to chose the smallest type that can hold the data without incurring a code size or performance penalty. This instruction allows efficient up-casts of signed numbers.

2.6.5 zex: zero-extend

Assembler code	Operation	f8l
`zex y, xl`	`y = xl`	No
`zex altacc16, altacc8`	`altacc16 = altacc8`	No

Affected Flags

`z`

Rationale

When aiming for memory efficiency, it is important to be able to chose the smallest type that can hold the data without incurring a code size or performance penalty. This instruction help implement efficient up-casts of unsigned numbers. Its benefits are not as big as those of `sex` per individual upcast, but on the other hand, unsigned numbers are used more commonly, thus unsigned upcasts are more common.

2.7 Other 8-bit instructions

2.7.1 bool: 8-bit cast to bool

Todo: Remove from f8l subset?

Assembler code	Operation	f8l
`bool xl`	`xl = (bool)xl`	Yes
`bool altacc8`	`altacc8 = (bool)altacc8`	Yes

Affected Flags

`z`

Rationale

This instruction allows the efficient implementation of explicit casts of 8-bit numbers to bool and, together with the `xor` instruction, of the negation operator for 8-bit numbers.

2.7.2 cax: 8-bit compare and exchange

z is set according to the old value of (y) - zl.

Assembler code	Operation	f8l
`cax (y), zl, xl`	`if ((y) == zl) (y) = xl; else zl = (y);`	Yes
`cax (y), zl, xh`	`if ((y) == zl) (y) = xh; else zl = (y);`	Yes
`cax (y), zl, zh`	`if ((y) == zl) (y) = zh; else zl = (y);`	Yes

Affected Flags

`z`

Rationale

This instruction is essential for the implementation of 8-bit lock-free atomics.

2.7.3 da: decimal adjust

Decimal adjust for addition / subtraction - binary coded decimal semantics.

todo: describe details!

Assembler code	Operation	f8l
`da xl`		Yes
`da altacc8`		Yes

Rationale

While the binary-coded-decimal (BCD) representation of numbers is mostly obsolete today, this instruction still has a use: it allows efficient conversion from binary to BCD, and thus to ASCII. This can substantially speed up the printing of numbers, considering that the f8 does not have division or modulo hardware.

Affected Flags

`hcnzo`

2.7.4 mad: multiply and add

Assembler code	Operation	f8l
`mad x, mm, yl`	`x = mm * yl + xh + c`	No
`mad x, (n, sp), yl`	`x = (n, sp) * yl + xh + c`	No
`mad x, (nn, z), yl`	`x = (nn, z) * yl + xh + c`	No
`mad x, (z), yl`	`x = (z) * yl + xh + c`	No

Affected Flags

`nz`

Rationale

Multiplication hardware is expensive. We need it for the `mul` instruction. However, on multiplications of larger numbers, if we only had `mul`, we'd spend a lot of cycles moving and adding, and the multiplication hardware would be idle for many cycles. This instruction speeds up multiplications of large numbers substantially, so that every other instruction actually uses the multiplication hardware.

2.7.5 msk: mask

z flag set according to old value of (y) & #i.

Assembler code	Operation	f8l
`msk (y), xl, #i`	`(y) = xl & #i	(y) & ~#i`	Yes
`msk (altacc16), altacc8, #i`	`(altacc16) = altacc8 & #i	(altacc16) & ~#i`	Yes

Affected Flags

`z`

Rationale

Bit-fields are an important tool to reduce data memory usage. This instruction allows for substantially better code for writing bit-fields, and for writing parts of I/O registers. Due to its effect on the z flag, it also can be used as a single bit exchange instruction, which can be useful on memory-mapped I/O.

2.7.6 pop: 8-bit pop from stack

Assembler code	Operation	f8l
`pop xl`	`xl = (sp++)`	Yes
`pop altacc8`	`altacc8 = (sp++)`	Yes

Affected Flags

none

Rationale

Registers that were saved temporarily via a `push` can be restored by this instruction. Not affecting flags makes the instruction more useful for restoring registers after a comparison before a conditional jump, or in the middle of a long addition/subtraction/multiplication.

2.7.7 push: 8-bit push onto stack

Ignores all flags, changes no flags, not even the reserved ones.

Assembler code	Operation	f8l
`push #i`	`(--sp) = #i`	Yes

Affected Flags

none

Rationale

8-bit stack parameters can be passed easily via this instruction. Not affecting any flags makes this instruction, together with `xch f, (n,sp)`, suitable for saving the flags at the beginning of an interrupt handler.

2.7.8 rot: 8-bit rotate

Assembler code	Operation	f8l
`rot xl, #i`	`xl = (xl << #i) | (xl >> (8 - #i))`	No
`rot altacc8, #i`	`altacc8 = (altacc8 << #i) | (altacc8 >> (8 - #i))`	No

Affected Flags

`nz`

Rationale

8-bit rotations happen in code. Together with `and`, this instruction can be used to efficiently do shifts by more than 2. Another important use is shuffling bits around for bit-field reads and writes (and bit-fields are an important tool to reduce data memory usage).

2.7.9 sra: 8-bit shift right arithmetic

Assembler code	Operation	f8l
`sra xl`	`c = op8 & 0x01` `xl = (xl >> 1) | xl & 0x80`	Yes
`sra altacc8`	`c = op8 & 0x01` `altacc8 = (altacc8 >> 1) | altacc & 0x80`	Yes

Affected Flags

cnz

Rationale

This instruction is used for right-shift of signed integers, which is also relevant to implementing signed division by powers of two.

2.7.10 thrd

Get current hardware thread number.

Assembler code	Operation	f8l
`thrd xl`	`xl` = current hardware thread number	Yes
`thrd altacc8`	`altacc8` = current hardware thread number	Yes

Affected Flags

z

Rationale

Getting the hardware thread number efficiently is useful for implementing thread-local storage. While `thrd` will not be a common instruction is typical programs, the alternative is doing a search for the current value of `sp` in a list of stack pointer ranges, which would be quite inefficient.

2.7.11 xch: 8-bit exchange

Assembler code	Operation	f8l
`xch yl, yh`	`t = yl; yl = yh; yh = t`	No
`xch xl, xh`	`t = xl; xl = xh; xh = t`	No
`xch zl, zh`	`t = zl; zl = zh; zh = t`	No
`xch xl, (n, sp)`	`t = (n, sp); (n, sp) = xl; xl = t`	No
`xch altacc8, (n, sp)`	`t = (n, sp); (n, sp) = altacc8; altacc8 = t`	No
`xch xl, (y)`	`t = (y); (y) = xl; xl = t`	Yes
`xch altacc8, (altacc16)`	`t = (altacc16); (altacc16) = altacc8; altacc8 = t`	Yes
`xch f, (n, sp)`	`t = (n, sp); (n, sp) = f; f = t`	Yes

Affected Flags

All, including reserved ones (`xch f, (n, sp)`) or none (all others).

Rationale

The instruction with register and stack parameters is useful for shuffling data in registers and on the stack around, allowing for substantially more efficient register and stack allocation. The `xch xl, (y)` instruction and its variant `xch` `altacc8, (altacc16)` are useful for implementing 8-bit atomics. `xch f, (n, sp)` together with `push #i` and `addw sp, #d` is suitable for saving and restoring the flags for interrupt handlers.

2.8 Other 16-bit instructions

2.8.1 addw: 16-bit addition

`addw sp, #d` ignores all flags, changes no flags, not even the reserved ones.

Assembler code	Operation	f8l
`addw sp, #d`	`sp = sp + #d`	Yes
`addw y, #d`	`y = y + #d`	Yes
`addw altacc16, #d`	`altacc16 = altacc16 + #d`	Yes

Affected Flags

none (`addw sp, #d`) or `cnzo` (all others).

Rationale

This instruction allows to efficiently adjust the stack pointer, which is useful for the setup of the stack at the beginning of functions and stack cleanup at the end of a function or after a function call.

2.8.2 boolw: 16-bit cast to bool

Assembler code	Operation	f8l
`boolw y`	`y = (bool)y`	No
`boolw altacc16`	`altacc16 = (bool)altacc16`	No

Affected Flags

`z`

Rationale

This instruction allows the efficient implementation of explicit casts of 16-bit numbers to bool and, together with the `xor` instruction, of the negation operator for 16-bit numbers.

2.8.3 caxw: 16-bit compare and exchange

z is set according to the old value of (y) - z.

Assembler code	Operation	f8l
`caxw (y), z, x`	`if ((y) == z) (y) = x; else z = (y);`	Yes

Affected Flags

z

Rationale

This instruction is essential for the implementation of 16-bit lock-free atomics.

2.8.4 cpw: 16-bit comparison

Subtraction where the result is used to update the flags only.

Assembler code	Operation	f8l
`cpw y, #ii`	`y + ~#ii + 1`	No
`cpw #ii, y`	`#ii + ~y + 1`	No
`cpw altacc16, #ii`	`altacc16 + ~#ii + 1`	No

Affected Flags

cnzo

Rationale

This instruction allows the efficient implementation of sparse switch statements, and of some if-else chains with a 16-bit or wider condition.

2.8.5 decw: 16-bit decrement

Assembler code	Operation	f8l
`decw (n, sp)`	`(n, sp) = (n, sp) + -1`	No

Affected Flags

cnzo

Rationale

Decrement is a common special case of subtraction, though not as common as increment as a special case of addition.

2.8.6 incnw: 16-bit increment without carry update

Ignores all flags, changes no flags (except possibly the reserved ones).

Assembler code	Operation	f8l
`incnw y`	`y = y + 1`	No
`incnw altacc16`	`altacc16 = altacc16 + 1`	No

Affected Flags

none

Rationale

Incrementing pointers is common. When needing to do so in the middle of wider or arbitrary-width arithmetic operations, the carry flag needs to be preserved across the increment.

2.8.7 negw: 16-bit negation

Assembler code	Operation	f8l
`negw y`	`y = ~y + 1`	No
`negw altacc16`	`altacc16 = ~altacc16 + 1`	No

Affected Flags

`cnzo`

Rationale

Negation is a common special case of subtraction.

2.8.8 mul: multiplication

Clears carry.

Assembler code	Operation	f8l
`mul y`	`y = yl * yh`	No
`mul x`	`x = xl * xh`	No
`mul z`	`z = zl * zh`	No

Affected Flags

`cnz`

Rationale

Multiplications are common, both explicitly and in array indexing. For efficient use of data memory, structs should not be padded, thus accessing arrays of structs often requires multiplications with factors that are not a power of two. This instruction allows to do these multiplications efficiently. The effect on the carry flag is motivated by the use of this instruction together with `mad` for wider multiplications.

2.8.9 popw: 16-bit pop from stack

Assembler code	Operation	f8l
`popw y`	`y = (sp); sp += 2`	Yes
`popw altacc16`	`altacc16 = (sp); sp += 2`	Yes

Affected Flags

none

Rationale

This instruction is useful to restore 16-bit registers that were saved temporarily via `pushw`. It is also a code-size efficient way of adjusting the stack pointer by 2 (but does a memory read).

2.8.10 pushw: 16-bit push onto stack

Assembler code	Operation	f8l
`pushw #ii`	`sp -= 2; (sp) = #ii`	Yes

Affected Flags

none

Rationale

16-bit stack parameters can be passed easily via this instruction. This is common enough to make it worth having this instruction. Compared to using `ldw` followed by a `pushw` with a register operand, we save one byte of code size, some execution time, and do not need a free 16-bit register (which might not be easily available at calls to functions that also have register parameters).

2.8.11 rlcw: 16-bit rotate left through carry

Assembler code	Operation	f8l

`rlcw y`	`tc = (y & 0x8000) >> 15`
	`y = (y >> 1) | (c << 15)`
	`c = tc`
		No

`rlcw (n, sp)`	`tc = ((n, sp) & 0x8000) >> 15`
	`(n, sp) = ((n, sp) >> 1) | (c << 15)`
	`c = tc`
		No

`rlcw altacc16`	`tc = (altacc16 & 0x8000) >> 15`
	`altacc16 = (altacc16 >> 1) | (c << 15)`
	`c = tc`
		No

Affected Flags

`cnz`

Rationale

This instruction is useful to implement wider shifts.

2.8.12 rrcw: 16-bit rotate right through carry

Assembler code	Operation	f8l

`rrcw y`	`tc = y & 0x0001`
	`y = (y >> 1) | c`
	`c = tc`
		No

`rrcw (n, sp)`	`tc = (n, sp) & 0x0001`
	`(n, sp) = ((n, sp) << 1) | c`
	`c = tc`
		No

`rrcw altacc16`	`tc = altacc16 & 0x0001`
	`altacc16 = (altacc16 << 1) | c`
	`c = tc`
		No

Affected Flags

`cnz`

Rationale

This instruction is useful to implement wider shifts.

2.8.13 sllw: 16-bit shift left logical

Assembler code	Operation	f8l
`sllw y`	`c = y & (0x8000 >> 15); y = y << 1`	No
`sllw altacc16`	`c = altacc16 & (0x8000 >> 15); altacc16 = altacc16 << 1`	No
`sllw y, xl`	`y = y << xl`	No
`sllw altacc16, altacc8`	`altacc16 = altacc16 << altacc8`	No

Affected Flags

`cnz` (`sllw y` and `sllw altacc16`) or `nz` (others).

Rationale

This instruction is useful to implement shifts of 16 or more bits.

2.8.14 sraw: 16-bit shift right arithmetic

Assembler code	Operation	f8l
`sraw y`	`c = y & 0x0001; y = y >> 1 | y & 0x8000`	No
`sraw altacc16`	`c = y & 0x0001; altacc16 = altacc16 >> 1 | altacc16 & 0x8000`	No

Affected Flags

`cnz`

Rationale

This instruction is useful to implement shifts of 16 or more bits.

2.8.15 srlw: 16-bit shift right logical

Assembler code	Operation	f8l
`srlw y`	`c = y & 0x0001; y = y >> 1`	No
`srlw altacc16`	`c = y & 0x0001; altacc16 = altacc16 >> 1`	No

Affected Flags

`cnz`

Rationale

This instruction is useful to implement shifts of 16 or more bits.

2.8.16 xchw: 16-bit exchange

Assembler code	Operation	f8l
`xchw x, (y)`	`t = x; x = (y); (y) = t`	Yes
`xchw y, (z)`	`t = y; y = (z); (z) = t`	Yes
`xchw z, (x)`	`t = z; z = (x); (x) = t`	Yes
`xchw z, (y)`	`t = z; z = (y); (y) = t`	Yes
`xchw y, (n, sp)`	`t = y; y = (n, sp); (n, sp) = t`	No
`xchw altacc16, (n, sp)`	`t = altacc16; altacc16 = (n, sp); (n, sp) = t`	No

Affected Flags

none

Rationale

This instruction is useful to shuffle data around, and to implement 16-bit atomic exchange.

Affected Flags

`z`

2.9 Jumps

2.9.1 call

`call #ii` ignores all flags, changes no flags, not even reserved ones.

Assembler code	Operation	f8l
`call #ii`	`sp -= 2; (sp) = pc; pc = #ii`	Yes
`call y`	`sp -= 2; (sp) = pc; pc = y`	Yes
`call altacc16`	`sp -= 2; (sp) = pc; pc = altacc16`	Yes

Affected Flags

none

Rationale

Calling and returning from functions using a return address on the stack is common. This instruction helps implement it efficiently. Not affecting flags, not even reserved ones, makes `call #ii` suitable for as a software interrupt.

2.9.2 dnjnz: decrement without carry update amd jump if not zero

Assembler code	Operation	f8l
`dnjnz yh, #d`	`if(--yh) pc += #d`	No
`dnjnz xh, #d`	`if(--xh) pc += #d`	No
`dnjnz zh, #d`	`if(--zh) pc += #d`	No

Affected Flags

none

Rationale

This instruction can be used to implement while loops instead of using `dec` followed by `jr nz`. Not affecting flags makes it suitable for implementing arbitrary-length arithmetic (`dec` would not preserve the carry flag, thus complicating its use). The choice of operands is motivated by the use-case of arbitrary-length multiplications via `mad`.

2.9.3 jp: jump

`jp #ii` ignores all flags, changes no flags, not even reserved ones.

Assembler code	Operation	f8l
`jp #ii`	`pc = #ii`	Yes
`jp y`	`pc = y`	Yes
`jp altacc16`	`pc = altacc16`	Yes

Affected Flags

none

Rationale

A jump instruction that can reach any target is very useful to implement control-flow. Not affecting flags, not even reserved ones, makes `jp #ii` instruction suitable for use at the interrupt vector.

2.9.4 jr: jump

`jr #d` ignores all flags, changes no flags, not even reserved ones.

Assembler code	Operation	f8l
`jr #d`	`pc += #d`	Yes

Affected Flags

none

Rationale

Having a jump instruction is very useful to implement control-flow. Jumps are common, and most of them have a nearby target, making it worth having a relative jump instruction.

2.9.5 jrc: jump on carry

Assembler code	Operation	f8l
`jr #d`	`if (c) pc += #d;`	Yes

Affected Flags

none

Rationale

Conditional jumps depending on the carry flag are useful for implementing common unsigned comparisons, and control-flow depending thereon. Since most jump targets are nearby, it makes sense to only have the relative conditional jump, as further jumps can still be implemented by inverting the condition and using an unconditional jump.

2.9.6 jrgt: jump on greater

Assembler code	Operation	f8l
`jrgt #d`	`if (c && !z) pc += #d;`	Yes

Affected Flags

none

Rationale

See `jrle`.

2.9.7 jrle: jump on less or equal

Assembler code	Operation	f8l
`jrle #d`	`if (!c || z) pc += #d;`	Yes

Affected Flags

none

Rationale

Conditional jumps depending on the carry flag together with the z flag are useful for implementing common unsigned comparisons, and control-flow depending thereon. Since most jump targets are nearby, it makes sense to only have the relative conditional jump, as further jumps can still be implemented by inverting the condition and using an unconditional jump.

2.9.8 jrn: jump on negative

Assembler code	Operation	f8l
`jrn #d`	`if (n) pc += #d;`	Yes

Affected Flags

none

Rationale

Conditional jumps depending on the n flag are useful for implementing common unsigned comparisons with 0, some bit tests, and control-flow depending thereon. Since most jump targets are nearby, it makes sense to only have the relative conditional jump, as further jumps can still be implemented by inverting the condition and using an unconditional jump.

2.9.9 jrnc: jump on no carry

Assembler code	Operation	f8l
`jrnc #d`	`if (!c) pc += #d;`	Yes

Affected Flags

none

Rationale

See `jrc`.

2.9.10 jrnn: jump on nonnegative

Assembler code	Operation	f8l
`jrnn #d`	`if (!n) pc += #d;`	Yes

Affected Flags

none

Rationale

See `jrn`.

2.9.11 jrno: jump on no overflow

Assembler code	Operation	f8l
`jrno #d`	`if (!o) pc += #d;`	Yes

Affected Flags

none

Rationale

See `jro`.

2.9.12 jrnz: jump on nonzero

Assembler code	Operation	f8l
`jrnz #d`	`if (!n) pc += #d;`	Yes

Affected Flags

none

Rationale

See `jrz`.

2.9.13 jro: jump on overflow

Assembler code	Operation	f8l
`jro #d`	`if (o) pc += #d;`	Yes

Affected Flags

none

Rationale

Conditional jumps depending on the n flag are useful for implementing signed comparisons wider than the operands of the available compare and subtraction instructions, and control-flow depending thereon. Since most jump targets are nearby, it makes sense to only have the relative conditional jump, as further jumps can still be implemented by inverting the condition and using an unconditional jump.

2.9.14 jrsge: jump on signed greater or equal

Assembler code	Operation	f8l
`jrsge #d`	`if (!(n ^ o)) pc += #d;`	Yes

Affected Flags

none

Rationale

See `jrslt`.

2.9.15 jrsgt: jump on signed greater

Assembler code	Operation	f8l
`jrsgt #d`	`if (!z && !(n ^ o)) pc += #d;`	Yes

Affected Flags

none

Rationale

See `jrsle`.

2.9.16 jrsle: jump on signed less or equal

Assembler code	Operation	f8l
`jrsle #d`	`if (z || (n ^ o)) pc += #d;`	Yes

Affected Flags

Rationale

Conditional jumps depending on the z, n and o flags are useful for implementing signed comparisons, and control-flow depending thereon. Since most jump targets are nearby, it makes sense to only have the relative conditional jump, as further jumps can still be implemented by inverting the condition and using an unconditional jump.

2.9.17 jrslt: jump on signed less

Assembler code	Operation	f8l
`jrslt #d`	`if (n ^ o) pc += #d;`	Yes

Affected Flags

none

Rationale

Conditional jumps depending on the n and o flags are useful for implementing signed comparisons, and control-flow depending thereon. Since most jump targets are nearby, it makes sense to only have the relative conditional jump, as further jumps can still be implemented by inverting the condition and using an unconditional jump.

2.9.18 jrz: jump on zero

Assembler code	Operation	f8l
`jrz #d`	`if (z) pc += #d;`	Yes

Affected Flags

none

Rationale

Conditional jumps depending on the zero flag are useful for implementing common tests for 0, and control-flow depending thereon. Since most jump targets are nearby, it makes sense to only have the relative conditional jump, as further jumps can still be implemented by inverting the condition and using an unconditional jump.

2.9.19 ret: return

Assembler code	Operation	f8l
`ret`	`pc = (sp); sp += 2`	Yes

Affected Flags

none

Rationale

Calling and returning from functions using a return address on the stack is common. This instruction helps implement it efficiently.

2.9.20 reti: return from interrupt

Ignores all flags, changes no flags, not even reserved ones.

Assembler code	Operation	f8l
`reti`	`pc = (sp); sp += 2`	Yes

Affected Flags

none

Rationale

When returning from an interrupt handler, interrupts should be reenabled at the same time. This instruction is necessary to return and enable atomically. To ensure that all flags get restored to their state from before the interrupt handler, it may not affect any flags, not even reserved ones.

2.9.21 trap

Opcode 0x00. Trap reset.

Assembler code	Operation	f8l
`trap`	Trap reset	Yes

Rationale

Some bugs, including many security-relevant ones can lead to the execution of code from memory used for data. Many exploits actually rely on data commonly being zero, and `nop` having opcode 0. By making opcode 0 a `trap` instruction, we can mitigate the impact of such bugs, and make them easier to debug.

2.10 Non-instructions

A 16-bit bitwise and `andw` would not be as useful as `orw` and `xorw`: known 0x00 or 0xff bytes are more common for bitwise and, so the compiler will often use `ld`, `ldw`, `clr` and `clrw`, and handle the rest with 8-bit `and`.

Hardware multiplication is costly, so there are no instructions requiring a multiplier wider than 8 times 8 to 16. Instead, the `mad` instruction is provided for efficient use of the 8 times 8 to 16 multiplier when implementing wider multiplications. Division is less common than multiplication, but complex or costly to implement in hardware.

A `cpijz acc8, (z), #d` instruction could speed up `strlen`, `strnlen`, `memchr` and `memcmp`, but the gain is not as big as for `ldi` and `ldwi`. Furthermore, compilers would be unlikely to use `cpijz` in code generation, unlike `ldi` and `ldwi`. So `cpijz` would only be useful for the mentioned standard library functions.

An atomic bit-swap instruction `xchb` would not be used often enough to justify having it in addition to `msk`.

Chapter 3

Opcode Map

todo - see opcodemap.ods for now. This is still preliminary, and subject to ongoing optimization.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	trap	sub xl, mm	sub xl (n, sp)	sub xl (nn, z)	sub xl, xh	sub xl, zl	sub xl, yl	sub xl, yh	sbc xl, mm		sbc xl (n, sp)	sbc xl (nn, z)	sbc xl, xh	sbc xl, zl	sbc xl, yl	sbc xl, yh
1x	add xl, mm	add xl, #i	add xl (n, sp)	add xl (nn, z)	add xl, xh	add xl, yl	add xl, zl	add xl, yh	adc xl, mm	adc xl, #i	adc xl (n, sp)	adc xl (nn, z)	adc xl, xh	adc xl, zl	adc xl, yl	adc xl, yh
2x	cp xl, mm	cp xl, #i	cp xl (n, sp)	cp xl (nn, z)	cp xl, xh	cp xl, zl	cp xl, yl	cp xl, yh	or xl, mm	or xl, #i	or xl (n, sp)	or xl (nn, z)	or xl, xh	or xl, zl	or xl, yl	or xl, yh
3x	subw y, mm	ldw y, sp	subw y, (n, sp)	subw y, x	sbcw y, mm	ldw ((n,sp)), y	sbcw y, (n, sp)	sbcw y, x	addw y, mm	inc (n, sp)	addw y, (n, sp)	addw y, x	adcw y, mm	adcw y, #ii	adcw y, (n, sp)	adcw y, x
4x	sllw y		rrcw y	altacc3	sllw y, xl	sraw y	rrcw (n, sp)	rlcw (n, sp)	jrno #d	altacc5	jrsge #d	jrslt #d	jrsle #d		ldw x, (y)	jrle #d
5x	addw y, #ii	inc mm	inc xl	inc (n, y)	dec (n, sp)	dec mm	dec xl	dec (n, y)	rrc (n, sp)	rrc mm	rrc xl	rrc (n, y)	rlc (n, sp)	rlc mm	rlc xl	rlc (n, y)
6x	push (n, sp)	push mm	push xl	push (n, y)	jp y	jp #ii	call #ii	call y		srlw y					ldw y, z	
7x	and xl, mm	xor xl, #i	and xl (n, sp)	and xl (nn, z)	xor xl, xh	and xl, zl	and xl, yl	and xl, yh	xor xl, mm	and xl, #i	xor xl (n, sp)	xor xl (nn, z)	and xl, xh	xor xl, zl	xor xl, yl	xor xl, yh
8x	pushw (n, sp)	pushw mm	pushw (nn, z)	pushw y	tstw (n, sp)	tstw mm	tstw (nn, z)	tstw y	ldw (n, y), x	pushw #ii	addw sp, #d	addw y, #d	ldi (n, y), (z)	ldwi (n, y), (z)	sex y, xl	zex y, xl
9x	cpw y, #ii	jr #d	jrc #d	jrnz #d	jrnc #d	jrz #d	jrn #d	jrnn #d	mul y	msk (y), xl, #i	ret	reti	negw y	mad x, mm, yl	mad x, (nn, z), yl	mad x, (z), yl
ax	xch xl, (n, sp)	push #i	xch xl, (y)	xch yl, yh	incw (n, sp)	incw mm	sbcw (nn, z)	incw y	adcw (n, sp)	adcw mm	adcw (nn, z)	adcw y	sbcw (n, sp)	sbcw mm	incw (nn, z)	sbcw y
bx	ldw y, mm	ldw y, #ii	ldw y, (n, sp)	ldw y, (nn, z)	ldw y, (y)	ldw y, (n, y)	ldw y, x	ldw y, #d	ldw (n, sp), y	swapop	ldw (nn, z), y	ldw x, y	ldw (y), x	ldw z, y	popw y	xch f, (n, sp)
cx	ld xl, mm	bool xl	ld xl, (n, sp)	ld xl, (nn, z)	ld xl, (n, y)	ld xl, (y)	ld xl, xh	ld xl, yl	ld xl, zl	ld xl, yh	ld xl, zh	ld mm, xl	ld (nn, z), xl	ld (n, sp), xl	ld (y), xl	ld (n, y), xl
dx	clrw (n, sp)	clrw mm	clrw (nn, z)	clrw y	rot xl, #i	altacc4	sra xl	da xl	caxw (y), z, x	thrd xl	mad x, (n, sp), yl	boolw y	xorw y, mm	xorw y, #ii	xorw y, (n, sp)	xorw y, x
ex	orw y, mm	orw y, #ii	orw y, (n, sp)	orw y, x	xchw y, (n, sp)	xchw x, (y)	incnw y	decw (n, sp)	pop xl	ld xl, #i	dnjnz yh, #d	cax (y), zl, xl	altacc1	ldw mm, y	altacc2	rlcw y
fx	srl (n, sp)	srl mm	srl xl	srl (n, y)	sll (n, sp)	sll mm	sll xl	sll (n, y)	crl (n, sp)	crl mm	crl xl	crl (n, y)	tst (n, sp)	tst mm	tst xl	tst (n, y)

Chapter 4

Peripherals

Unless otherwise noted, the value of I/O registers on reset is unspecified.

4.1 Watchdog and Reset

The watchdog has an 8-bit configuration register and a 16-bit counter register.

When the watchdog is active, the system clock is divided by 16, and then used to increment the counter register.

The system is reset when a power-on reset happens, the watchdog counter register reaches 0xffff, the trap instruction is executed, or the byte at memory address 0x0000 is written.

Configuration Register

0	1	2	3	4–7
dog active	dog reset	trap reset	null reset	reserved

The lowest bit of the configuration register decides if the watchdog is active. It is 0 on reset. The following three bits give the reason of the latest reset. On a power-on-reset they are all 0.

4.2 Interrupt Controller

The interrupt controller has a 16-bit enable register, and a 16-bit active register.

0	1	2–15
t0ov	t0cp	reserved

When an interrupt happens and the corresponding bit in the enable register is set, the corresponding bit in the active register is set. When a bit in the active register is set, and no interrupt routine is currently executing, the program counter is put onto the stack and then set to 0x4004. From then on, an interrupt routine is considered to be executing until the `reti` instruction is executed.

Bit 0 of the enable register indicates that timer 0 overflow interrupts are enabled. Bit 0 of the active register indicates that a timer0 overflow interrupt is active. Bit 1 of the enable register indicates that timer 0 compare interrupts are enabled. Bit 1 of the active register indicates that a timer 0 compare interrupt is active. These bits are 0 on reset. All other bits are reserved.

4.3 Timer

The timer has an 8-bit configuration register and 16-bit counter, reload and comparison registers.

0–3	4–5	6–7
input clock	prescaler	reserved

The lowest 4 bits of the configuration register select the clock source (0 none, 1 system clock, 2 to 15 for other inputs), the next 2 select the prescaler factor (0 for 1, 1 for 4, 2 for 16, 3 for 64). All 6 bits are 0 on reset.

The timer increments the 16-bit counter register. When incrementing from 0xffff, a timer overflow interrupt happens, and the value from the reload register gets loaded into the counter register instead. When the timer register gets incremented to the value of the compare register, a timer compare interrupt happens.

4.4 GPIO

The GPIO has (up to 16-bit) data direction, output data, input data, pull-up registers.