AVRCIS

AVRCIS is the simple include file for avrasm2 that is activating complex instruction set mode.

```
Defreg
```

CONTENT

3.9. LPM
3.10. ELPM
3.11. IN
3.12. OUT
3.13. MOV
3.14. MOVW
3.15. BST
3.16. BLD
3.17. SBRS
3.18. SBRC
3.19. ROL
3.20. ROR
3.21. LSL
3.22. LSR
3.23. ASR
3.24. INC
3.25. DEC
3.26. AND
3.27. ANDI
3.28. OR
3.29. ORI
3.30. EOR
3.31. EORI
3.32. NEG
3.33. COM
3.34. ADIW
3.35. SBIW
3.36. ADD
3.37. ADDI

1. AVRCIS: как это работает?

в avrasm (и вероятно не только в нём) есть возможность перехватить инструкции — т. е. перенаправить инструкцию в указываемый макрос:

пример перехвата инструкции '**nop**':

```
#define nop nop_                        ; redirect nop to nop_
.macro nop_
            .message "NOP catched!"
            ; but how we can performs an original 'nop' instruction ?
            #undef nop
                  nop                   ; original 'nop' instruction
            #define nop nop_
      .endm
nop
NOP catched!
```

avrasm2 позволяет объявить single-line макрос с тем-же именем что и у мнемоники. этот макрос перенаправляет выполнение в уже multi-line макрос 'nop_' внутри которого помимо

собственных дел надо будет выполнять и оригиналную инструкцию 'nop'. это становится возможным благодаря директиве #undef которая в совокупности с последующим #define временно андефинирует single-line макрос 'nop' в промежутке осуществляя таким образом оригинальную инструкцию 'nop'.

используя этот трюк AVRCIS перехватывает большую часть инструкций процессора (все требующие хотя-бы одного операнда в виде регистра). для чего?

2. КОМПЛЕКСНЫЕ РЕГИСТРЫ

представьте себе что регистр(ы) в инструкциях - это не имена конкретных регистров, а просто имена с каждым из которых ассоциировано число — 32-битная регистровая маска — по одному биту на каждый физический регистр процессора. если бит в маске равен 1 значит соответствующий ему физический регистр процессора является частью *комплексного регистра* представляемого маской.

например:

```
defreg  xl, 1 << 26             ; объявляем регистр xl соотвесттвующий регистру r26
defreg  xh, 1 << 27             ; объявляем регистр xh соотвесттвующий регистру r27

defreg  x, 1 << 26 | 1 << 27    ; объявляем регистр x как регистры: r26 и r27
```

общее количество физических регистров выбранных по маске может быть любым: от 0 до 32. их комбинации никак не ограничены:

<table>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>r0</td><td></td><td></td><td rowspan="2">r</td><td>rl</td><td></td><td></td><td></td><td rowspan="2">User Reated</td></tr>
<tr><td>r1</td><td></td><td></td><td>rh</td><td></td><td></td><td></td></tr>
<tr><td>r2</td><td colspan="8" rowspan="6">Undefined (free)</td></tr>
<tr><td>r3</td></tr>
<tr><td>r4</td></tr>
<tr><td>r5</td></tr>
<tr><td>r6</td></tr>
<tr><td>r7</td></tr>
<tr><td>r8</td><td></td><td></td><td></td><td>FLAG2</td><td></td><td></td><td></td><td>?</td></tr>
<tr><td>r9</td><td></td><td></td><td></td><td>zero</td><td></td><td></td><td></td><td>Instruction Set Related</td></tr>
<tr><td>r10</td><td></td><td></td><td rowspan="2">TCNTH</td><td>TCNTHL</td><td></td><td></td><td></td><td rowspan="2">System Timer</td></tr>
<tr><td>r11</td><td></td><td></td><td>TCNTHH</td><td></td><td></td><td></td></tr>
<tr><td>r12</td><td></td><td></td><td></td><td>CSPL</td><td></td><td></td><td></td><td>Core Stack Pointer Low</td></tr>
<tr><td>r13</td><td></td><td></td><td></td><td>iff</td><td></td><td></td><td></td><td rowspan="5">Interrupt Related</td></tr>
<tr><td>r14</td><td></td><td></td><td rowspan="2">g</td><td>gl</td><td></td><td></td><td></td></tr>
<tr><td>r15</td><td></td><td></td><td>gh</td><td></td><td></td><td></td></tr>
<tr><td>r16</td><td></td><td></td><td rowspan="2">i</td><td>il</td><td></td><td></td><td></td></tr>
<tr><td>r17</td><td></td><td></td><td>ih</td><td></td><td></td><td></td></tr>
</table>

r18				FLAG0				System Flag Register 0
r19				tmp				Instruction Set Related
r20		ab	a	al				User Reated
r21				ah				
r22			b	bl				
r23				bh				
r24		tx	t	tl				
r25				th				
r26	xy		x	xl				
r27				xh				
r28		yz	y	yl				
r29				yh				
r30			z	zl				
r31				zh				

МАКРООПРЕДЕЛЕНИЕ

тело макроопределения выделяется из следующей после заголовка x-зоны

```
`name::
        ...
```

- .next в .end
- spcs in def macro title
- : в ::

- ! как name символ

asm3d

- name's begin and end

в теле макроопределения возможны ссылки на метку '.next' которая всегда указывает на конец макроопределения («за» макроопределение):

```
: name
        ...
        brne    .next
        ...
    .next:
```

```
push    ^t
```

```
push    ^t
```

An interrupt handler entering should look like:

```
in      iff, SREG
mov     g, z

...

mov     z, g
out     SREG, iff
reti
```

объявляем регистр *name* как совокупность регистров: r0, r7, r30 и r31

```
defreg  name, (1 << 0) | (1 << 7) | (1 << 30) | (1 << 31)

push    name            →       push   r31
                                push   r30
                                push   r7
                                push   r0

pop     name            →       pop    r0
                                pop    r7
                                pop    r30
                                pop    r31
```

следует однако помнить что маска «не помнит» последовательности физических регистров в которой они присваивались объявляемому комплексному регистру:

```
defreg      a, (1 << 0) | (1 << 1)

defreg      b, (1 << 1) | (1 << 0)
```

^ регистр 'a' – то-же что и регистр 'b': регистры r0 и r1

3. КОМПЛЕКСНЫЕ ИНСТРУКЦИИ

а теперь представьте себе что инструкции «видят» и «умеют» работать с такими комплексными регистрами и генерируют код в соответствии со своей к*омплексной интерпретацией* — т. е. тем, как описывается данная инструкция если её распределить по разрядности на несколько регистров.

одиночная инструкция процессора, перехваченная AVRCIS, генерирует последовательность команд которая работают с физическими регистрами процессора указанными в маске, и которая описывают операцию в той разрядности, которая соответствует общему количеству физических регистров по маске:

example:

```
defreg xl, (1 << 26)
defreg xh, (1 << 27)
```

```
defreg x, reg( xl ) | reg( xh )

push   xl                  →      push   r26

push   x                   →      push   r27
                                  push   r26

ldi    x, 0x2211           →      ldi    r26, 0x11
                                  ldi    r27, 0x22

pop    x                   →      pop    r26
                                  pop    r27
```

перехваченные инструкции умеют «видеть» и работать с такими комплексными регистрами генерируя код выполняющий нужные операции, с нужными регистрами и с нужной разрядностью:

вот некоторые примеры таких инструкций:

```
defreg  t, ( 1 << 24 ) | ( 1 << 25 )
defreg  x, ( 1 << 26 ) | ( 1 << 27 )

defreg  tx, reg( t ) | reg( x )



push   t                   →      push   r24
                                  push   r25

push   x                   →      push   r26
                                  push   r27

ldi    x, 0x1234           →      ldi    r26, 0x34
                                  ldi    r27, 0x12

lds    t, ( adr )          →      lds    r24, ( adr )
                                  lds    r25, ( adr + 1 )

add    t, x                →      add    r24, r26
                                  adc    r25, r27

ror    t                   →      ror    r25
                                  ror    r24

bst    t, 13               →      bst    r25, 5

sts    ( adr ), t          →      sts    ( adr + 1 ), r25
                                  sts    ( adr + 0 ), r24

pop    x                   →      pop    r27
                                  pop    r26

pop    t                   →      pop    r25
                                  pop    r24
```

> ИНТЕРЕСНЫЙ ВОПРОС: какой разрядности система команд процессора для которого написан пример выше?

в примере мы вначале объявляем некоторый комплексный регистр с именем '**t**' которому присваиваем маску: (1 << 24) | (1 << 25) которая говорит нам что регистр 't' это два физических регистра: r24 и r25.

также, мы объявляем регистр '**x**' которому присваиваем маску (1 << 26) | (1 << 27) которая говорит нам что регистр '**x**' это два физических регистра: r26 и r27.

и наконец мы объявляем регистр '**tx**' соответствующий регистру '**t**' и регистру '**x**' — т. е. регистрам: r24, r25, r26 и r27

соответственно например такую инструкцию как '**push**', выполненую с регистром '**t**' в качестве операнда можно интерпретировать как:

```
push   t                    →       push   r24
                                    push   r25
```

понятно что ответная команда '**pop**' выполненная с той-же маской должна соответствовать своему антиподу:

```
pop    t                    →       pop    r25
                                    pop    r24
```

обратите внимание на различную последовательность отработки регистров по маске: '**push**' идёт снизу-вверх (low-high), а '**pop**' наоборот — сверху-вниз (high-low)

но как видно из примера выше — комплексно можно интерпретировать не только инструкции '**push**' и '**pop**', но и также практически все, другие инструкции, работающие с регистрами ”ложатся” в комплексную интерпретацию.

ещё раз возвращаясь к вышеприведённому примеру следует заметить что можно объявить регистр 'tx' и присвоить ему регистр 't' и регистр 'x' и в таком случае не делать вначале два '**push**' и в конце два '**pop**', а лишь по одному:

```
defreg t, ( 1 << 24 ) | ( 1 << 25 )

defreg x, ( 1 << 26 ) | ( 1 << 27 )

defreg tx, reg( t ) | reg( x )


push   tx                   →       push   r24
                                    push   r25
                                    push   r26
                                    push   r27

ldi    x, 0x1234            →       ldi    r26, 0x34
                                    ldi    r27, 0x12

lds    t, ( adr )           →       lds    r24, ( adr )
                                    lds    r25, ( adr + 1 )

add    t, x                 →       add    r24, r26
                                    adc    r25, r27

ror    t                    →       ror    r25
                                    ror    r24

bst    t, 13                →       bst    r25, 5

sts    ( adr ), t           →       sts    ( adr + 1 ), r25
                                    sts    ( adr ), r24

pop    tx                   →       pop    r27
                                    pop    r26
                                    pop    r25
                                    pop    r24
```

на самом деле практически все инструкции процессора «ложаться» на комплексный лад. и вот именно эта мультирегистерная интерпретация команды и называется комплексной

инструкцией — работающей со всеми регистрами по маске

AVRCIS перехватывает все инструкции требующие как минимум одного регистрового параметра. Сюда НЕ входят:

- инструкции без параметра: nop, cli, wdr, sleep, ijmp, ret, …
- инструкции для доступа к IO: sbis, sbic, sbi и cbi
- инструкции переходов: br*xx*, (r/i)jmp, (r/i)call

командами не имеющими очевидной комплексной интерпретации являются на текущий момент:

1.4. PACKAGE

AVRCIS package's available for download from my git repository https://github.com/digics/AVRCIS

the package contains two forms of include file:

cis.inc - normal(human readable) version: ~600KB

cis0.inc - the shortest version(recommended): ~400KB

user should include one of the file form above in it's own project as earlier as possible:

.include ".../cis0.inc"

I also be glad to publish the AVRCIS source file but it's requiring an external _s3m-parser to be compiled into avrasm2's friendly form. please contact me if you're intresting of how to do that. however take a look at the sources in:

cis.src - AVRCIS _s3m-source: ~45 KB

the _s3m-parser is powered by gawk. it's allows to placing a commands inside an assembler's commentaries. The commands including defining/calling special s3m-macros and supporting an duplication command.

finally this document is also included in AVRCIS package:

cis.doc

РАЗРЯДНОСТЬ ОПЕРАЦИИ

обычно разрядность операции определяется в мнемонике. я сам долгое время

использовал макросы типа:

```
ldiw   r26, r27, val
```

а позже, такие как:

```
ldix   val
ldiy   val
```

в обоих вариантах в самой мнемонике закладывается разрядость производимой операции. в первом примере это буква 'w' указывающая на то что операция имеет разрядность 16 бит.во втором случае это имя 16-битного регистра в конце мнемоники.

при использовании AVRCIS разрядность операции определяется выбором регистра.

```
ldi    x, val
```

это значительно улучшает все показатели в сравнении с двумя первыми вариантами:

```
ldi    x, val
ldi    y, val
```

ARCHITECTURE

архитектура AVRCIS предполагает что из 32ух физических регистра процессора — два регистра используются для обеспечения её работы:

- регистр '**tmp**' (r19) который используют в качестве временного хранилища инфомации

  внимание! использование временного регистра при имплементации содержит в себе предупреждение

  прерывание может произойти между командой загрузки во временный регистр **tmp** значения и командой его использующей. Обработчики прерываний либо не должны включать в себя команды использующие временный регистр(псевдоинструкции), либо сохранять\восстанавливать регистр tmp на входе\выходе обработчика прерывания.

  данный регистр позволяет

- регистр '**zero**' (r9) содержащий всегда ноль

я являюсь человеком жадным до регистров и, всё-же, после взвешивания всех факторов эти два регистра были включены в имплементацию.

с помощью регистра 'tmp' мы закрываем «бреши» в системе команд AVR и по другому это вообще-то всё равно никак не сделать

регистр хранящий нулевое значение тоже весьма полезен позволяя в некоторых(на самом деле не таких уж и редких) случаях облегчить имплементацию той или иной инструкции. К тому-же его мы можем расположить в нижнем регистер сэте

INSTRUCTION SET

набор команд AVR содержит очевидные «дыры».

например:

- существует команда вычитания из регистра числа, а команды аналогичного сложения отсутствуют; тоже с флагом переноса
- существует команда EOR обеспечивающая ксоринг регистра и регистра, но отуствует команда ксоринга регистра и числа
- практически все инструкции работающие с регистром и числом — могут работать только с 16тью старшими регистрами процессора

при разработке AVRCIS, такие «дыры» в инстракшен сэте множественно вызывали недоумение по поводу того что-же с этим делать. и было принято решение «выровнять» ситуацию в этом плане с помощью описания псевдоинструкций - «умеющих» делать то, что не заложено в оригинальном instruction set.

по большей части это «умение» основано на загрузке в регистр tmp некоторого значения и осуществления требуемой операции с использованием этого регистра:

```
ldi     tmp, 100
add     xl, tmp
```

Ещё одной «дырой» в архитектуре AVR является очевидная оплошность разработчиков в том, как сохраняется адрес возврата в стэке при использоании команды (r)call или при возникновении прерывания:

AVR “пушит» в стэк сначала младший байт адреса возврата, затем старший. Учитывая что указатель стэка декрементальный получается так что в памяти вначале располагается старший байт, а затем младший в то время как всё остальное в AVR работает наоборот — сначала младший и затем старший. это явная ошибка разработчиков платформы AVR которая говорит об уровне их профессионализма (НЕДАЁЛКИЕ ЛОХИ)

данная бага вызывает к сожалению нехорошие последствия. Ведь это означает в том числе например что и 'push' с 'pop' должны выполнять инструкции в противоположном направлении - «пушить» вначале младший регистр и затем старший.

```
        call    l0
        . . .
l0:
```

```
pop     z
```

А это приводит к тому что в памяти

AVRCIS перехватывет большую часть инструкций процессора направляя их в макросы-перехватчики которые «умеют» генерировать код инструкции в комплексной интерпретации.

что такое комплексная интерпретация инструкции?

это версия того, что и как должна делать та или иная инструкция если её результат находится более чем в одном регистре

за основу комплексного инстракшен сэта взята модель в которой объявленый комплексный регистр — это 32ух битная *регистровая маска* присваиваемая имени регистра. Эта маска покрывает все 32 физических регистра процессора. если бит в маске равен 1 значит соответствующий физический регистр процессора — часть представляемого маской комплексного регистра:

пример объявления комплексного регистра:

```
defreg    t, ( 1 << 24 ) | ( 1 << 25 )
```

^ объявляем регистр t с маской: (1 << 24) | (1 << 25) - что соответствует двум физическим регистрам процессора: r24 и r25

комплексная инструкция «умеет» генерировать код с нужной разрядностью операции и работающий именно с теми регистрами которые выбраны маской:

команда push в своей комплексной интерпретации должна подавать свою мнемонику для каждого физического регистра выбранного по маске. то-же касается и команды pop — однако - что важно — для pop это делается в обратной последовательности:

```
push   t                        →      push   r25
                                       push   r24

pop    t                        →      pop    r24
                                       pop    r25
```

другой пример — команда add – для неё мы объявим ещё один регистр: 'x' охватывающий регистры: r26 и r27:

```
defreg    x, ( 1 << 26 ) | (1 << 27 )

add    t, x                     →      add    r24. r26
                                       adc    r25, r27
```

add обладает двойной мнемоникой — для первой и для всех последующих операций

перехваченные инструкции умеют «видеть» и работать с такими комплексными регистрами генерируя код выполняющий нужные операции, с нужными регистрами и с нужной разрядностью:

все ассемблерные примеры в данной документации предполагают объявленые регистры согласно таблице в разделе 2. AVRCIS CONTROL

инклуд файл получился не маленький

в урезанной версии инклуда удалены все ненужные пробелы и табуляции и её вес составляет порядком 400 KB.

полная версия инклуда «весит» около ~ 0 KB

такой большой объём обусловлен спецификой конкретной задачи, а именно — девственной глубиной стэка макросов в avrasm2 — равной всего-лишь 16 уровням (на полшишечки). вот тот придурок, который ридумал эту ересь и решил что этого будет достаточно попросту шарлатан

короче говоря вся имплементация выполнена с использоывнием приёмов линейного программирования и это позволяет свести затраты макро стэка к одному единственному уровню. использование псевдоинструкций требует ещё одного дополнительного уровня макро стэка.

СОМНИТЕЛЬНЫЕ ОПЕРАЦИИ

не все инструкции получается хорошо истолковать на комплексный лад. ряд инструкций

- **cpc**

  возможно — линейное сравнение даст общий C-флаг

- **cpse**

  со скипами вообще всё не просто так

- **swap**

  поменять ниблы во всех физических регистрах комплексног регистра??

все три команды «умеют» работать с комплексными регистрами просто повторяя свою мнемонику для каждого физического регистра из маски

```
cpc     t, x
```

1. **AVRCIS**

AVRCIS is the source include file “_.avrcis” that if included activates in avrasm2 the complex instruction set mode.

user should include file “_.avrcis” at the top of it's own project

the include file is large (short: about 400K) because the most of preprocessor code is linear and do not use macro stack level.

1.1. **AVRCIS ENGINE TWEAK**

AVRCIS implementation based on the fundamental avrasm's tweak allowing to catch cpu instructions and replace it's code generating by running its own preprocessor code.

here is the simple example of how this tweak may be applied for an '**nop**' instruction:

```
example:
        ; catching the NOP!
        #define nop nop_
        .macro nop_
```

```
                        .message "NOP catched!"
                        ; how to perform the original 'nop' instruction?
                        #undef nop
                               nop    ; the original 'nop' instruction
                        #define nop nop_
                .endm
        nop

        NOP catched!
```

the example above is purely demonstrates of how an instruction may be catched.
see 6. **CATCH ME!** for details.

based on this feature AVRCIS catches all cpu instructions requiring at least a single register as an operand and turns its starting be an complexary.

the complex(ary) instruction definition is based on the another definition of the complex registers:

1.2. **COMPLEX REGISTERS**

complex register – is the 32-bit mask covering all 32 cpu physical registers. If a bit in register-mask is equals to 1 then corresponding cpu physical register is the part of providing complex register.

this open the gates for possibilities to provide instruction's register parameters as the numbers – the register masks.

user is allows to define it's own complex register(s) with an any name and iany register mask using '**defreg**' :

```
        defreg      name, mask
example:
        defreg   x, ( 1 << 26 ) | ( 1 << 27 )
```

here we're defining complex register 'x' by assigning its the register mask covering two physical registers: r26 and r27

the register mask may be any.

1.3. **COMPLEX INSTRUCTION**

complex instruction is the instruction that is generates it's code by register mask(s).

```
example:

        defreg x, ( 1 << 26 ) | ( 1 << 27 )     ; assigning complex register 'x' for
                                                ; two physcal registers: r26 and r27

        push   x                        →       push    r27     ; push complex register x
                                                push    r26

        pop    x                        →       pop     r26     ; pop complex register x
                                                pop     r27
```

AVRCIS implementation replaces most cpu instructions by it's own preprocessor code generating instruction code using register mask.

AVRCIS programming design is including an elements of linear programming solutions that is minimizing macro stack usage to the single level. It is important because of the virgin's deep of the avrasm's macro stack that is equals to max 16 (dumbs!)

- that generated code should selects it's registers using register mask
- that whole operation may have different bit-wide

2. **AVRCIS CONTROL**

typical user's register map

(actual for all assembler examples in this documentation)

REG	r31	r30	r29	r28	r27	r26	r25	r24	r23	r22	r21	r20	r19	r18	r17	r16	r15	r14	r13	r12	r11	r10	r9	r8	r7	r6	r5	r4	r3	r2	r1	r0
RL																																1
R																															1	1
RH																															1	
AL												1																				
A											1	1																				
AH											1																					
BL										1																						
B									1	1																						
BH									1																							
TL								1																								
T							1	1																								
TH							1																									
XL						1																										
X					1	1																										
XH					1																											

YL				1																											
Y			1	1																											
YH			1																												
ZL		1																													
Z	1	1																													
ZH	1																														
ALL	1	1	1	1	1	1	1	1	1	1	1	1																	1	1	1	1
TZ	1	1					1	1																							
XY			1	1	1	1																									
R0 TZ	1	1					1	1																							1
R1 XY			1	1	1	1																								1	
ALL	1	1	1	1	1	1	1	1	1	1	1	1																	1	1	1	1

2.1. **DEFREG**

```
type:               directive
operand0:           register name
operand1:           register mask (expression)
```

define (main) register *name* by register *mask*:

```
        defreg      name, mask
```

the register *mask* is provided as an expression form and will not be censored or modified.

user's may operates with the registers masks of the already defined registers using func-macro **reg**() that is returning register mask:

```
        defreg      al, reg( r20 )
        defreg      ah, reg( r21 )

        defreg      a, reg( al ) | reg( ah )
```

the AVRCIS implementation includes amount of predefined registers:

```
        r0 - r31    as the appropriated cpu physical register

        all         as @USEREG
```

defreg internally stores the *mask* of the register at:

@REG_*name* = *mask*

in addition **defreg** calculates and stores some extra *mask*-related information at:

@REGL_*name* = (*width*) the total number of 1 in *mask*

@REGB_*name* = number of the first(*lower*) physical register selected by *mask*

@REGE_*name* = number of the last(*higher*) physical register selected by *mask*

if register is already defined by the *another* mask then fatal error will be occured in case of it's redefenition. in case if defined register mask is exactly the same as defined before then no error will be produced:

```
defreg xl, reg( r26 )

defreg xl, reg( r26 )            ← OK

defreg xl, reg( r27 )            ← ERROR
```

main register definition is also overriding all autodefined shadow- and sub-registers
in case of name conflicts:

```
defreg t, reg( r24 ) | reg( r25 )

defreg t0, …                     ← OK
```

defreg internally requiring 2 macro stack levels.

```
defreg      al, reg( r20 )
defreg      al, reg( r20 )       ← no error

defreg      al, reg( r21 )       ← fatal error
```

user's may to define any name by the any combination of the physical cpu registers to be assign it. but there is important to remember that the mask isn't 'remembering' the order in which registers was been assigned while definition. register mask just contains information about what physical registers is the part of provided register name and whats is not:

```
defreg      a, reg( r1 )  | reg( r0 )
defreg      b, reg( r0 )  | reg( r1 )
```

a is the same as b: registers r0 and r1

the order in which cpu physical registers are handled is depending from instruction. see complex instruction set for more specific.

If provided register mask contains one or more registers from the user's part (marked in the @USEREG as 1) then extra registers will also be autodefined:

– shadow register *_name*

if @DEFREG_SHADOWREG is true:

the shadow register defined with the leading underscore character in register name and by the negative(inversed) register mask. it is calculated as follows:

```
shadow register mask = @USEREG & ( ~ mask )
```

example:

```
defreg t, reg( r24 ) | reg( r25 )
```

will define:

```
register t   00001100 00000000 00000000 00000000
register _t  11110011 11110000 00000000 00000011

@USEREG      11111111 11110000 00000000 00000011
```

– subregisters *name*0, *name*1, *name*2, …

if @DEFREG_SUBREG is true:

if register mask is containing more than one physical register then subregisters will be defined with the numeric suffix in register name and with the mask appropriated to every single register:

```
defreg t, reg( r24 ) | reg( r25 )
```

will define:

```
register t   00001100 00000000 00000000 00000000
register t0  00000100 00000000 00000000 00000000
register t1  00001000 00000000 00000000 00000000
```

all (auto)defined registers (including sub and shadow registers) have the following mask-related information at:

```
@REG_name          = register mask
@REGL_name         = total register-length
@REGB_name         = number of lowest register in mask
@REGE_name         = number of highest register in mask
```

2.1.1. **SUBREGISTER**s

if defined register mask contains more than one physical registers then subregisters associated with the every physical register from *mask* will also be defined:

the subregisters are defined only in case if register mask contains at least single register from the user part that is determinated by @USEREG global var.

the subregisters is using the same name as main defined register with the sub-register incremental number in the name suffix:

example:

```
defreg      tx, reg( r24 ) | reg( r25 ) | reg( r26 ) | reg( r27 )
```

this will cause that the following registers are will be defined:

```
tx     main         register definition as r27:r26:r25:r24

tx0    sub          register definition as r24
tx1    sub          register definition as r25
tx2    sub          register definition as r26
tx3    sub          register definition as r27

_tx    shadow       as all from @USEREG except r27:r26:r25:r24
                    (see further)
```

2.1.2. **SHADOW REGISTER**

any register definition leads to its shadow (*not*-) register definition will also be performed.

the shadow register is defined using the same register name as for main register definition with leading underscore character.

the mask that is assigned to an shadow register is the negative(inversed) *mask* with non-user part registers exclusion. it is calculated as follows:

```
shadow mask = ( mask XOR 0xFFFFFFFF ) AND @USEREG
```

@USEREG is the constant that is containing global register mask. this mask determinates what registers is the part of the user part registers group and what is the part of the system related registers group.

if a bit in @USEREG is equals to 1 – then appropriated cpu physical register is relative to user part registers group. otherwise if bit equals to 0 then appropriated cpu physical register is relative to system register group.

at definition the shadow register is taken original register *mask*, inverse it and **AND** its with the @USEREG constant so that system registers will not the part of the shadow registers.

shadow registers is useful if dev needs to specify all user registers except thoses.

for example 'push _tl' will leads to pushing to stack all user registers except register 'tl':

```
.set @USEREG =       0b11111111 11111100 00000000 00001111

defreg r,      0b00000000 00000000 00000000 00000011        ; r1:r0
      _r ==    0b11111111 11111100 00000000 00001100

push   _r                  →      push   r31          ; except r0, r1
                                  push   r30
                                  push   r29
                                  push   r28
                                  push   r27
                                  push   r26
                                  push   r25
                                  push   r24
                                  push   r23
                                  push   r22
                                  push   r21
                                  push   r20
                                  push   r3
                                  push   r2
```

2.2 **ISREG**() / **_ISREG**()

returns true / false if provided parameter is the name of defined register.

to getting know if parameter is the name of defined register perform:

```
.if defined( @REG_@0 )
```

where @0 is the name of potential register

or user may use func-macros: **isreg**() and **_isreg**() that is doing the same:

example:

… .if **isreg**(*r10*) ...

note that providing an expression as the parameter will cause to avrasm2 syntax error.

2.3 **REG**()

returns mask of the provided register

… **reg**(*name*) …

example:

```
reg( r10 ) == ( 1 << 10 )
```

2.4 **@AVRCIS**

controls avrcis code generation mode by assigning special flags to @AVRCIS var

the @AVRCIS flags:

0	== 0	- exact instruction mode
	== 1	- complex instructions enabled
1	== 0	- pseudo-instructions not allowed
	== 1	- pseudo-instructions enabled
2	== 0	-
	== 1	-

0000000000000000000

3. **COMPLEX INSTRUCTION SET**

- If an instruction's register0 or register1 is register **nul** then instruction will be skipped – /??check wrong registers

– RAM/IO write access performing in high-low register order (except stack operation)

– RAM/IO read access performing in low-high register order (except stack operation)

– STACK operations performed in low-high register order for output(write) and with the high-low register order for input(read)

– instruction is have the same bit-wide as its register 0-wide:

```
mov    xl, yl          →      mov    xl, yl

mov    xl, y           →      mov    xl, yl

mov    y, xl           →      mov    yl, xl
                              ldi    yh, 0  <- suffix

mov    x, y            →      movw   xl, yl
```

prefix op suffix

\+ - changing according to instruction

– - no change

* - changed not according to instruction

instructios 'sbr', 'cbr' is also defined and works we

'bset' and 'bclr' instructions are not catchedstays in original form, tst

sixth column: '**ITHS VNZC'** shows how an instruction affects on cpu flags:

- if flag state is specified by the green character:

  then flag state is according to the normal cpu instruction

- if flag state is specified by the red character:

  then flag state isn't according to the normal cpu instruction

- if flag state is specified by the orange character:

  both cases mixed

3.1. **PUSH**

```
mnemonic:     push
direction:    low-high
operand:      register(0)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
push tl	1	2	2	push tl	---- ----
push tz	4	8	8	push tl push th push zl push zh	---- ----

'**push**' register to stack. avr pushes data at exact SPH:SPL address. after writing SPH:SPL is post-decremented.

'**push**' is performing using low-high register order:

```
push   t                    →       push   r24
                                    push   r25
```

see also: **pop**

3.2. **POP**

```
mnemonic:     pop
direction:    high-low
operand:      register(0)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
pop tl	1	1	2	pop tl	---- ----
pop tz	2	3	6	pop zh pop zl pop th pop tl	---- ----

'**pop**' register from stack. the stack pointer is pre-incremented by 1 before the '**pop**' reads each data bytes.

'**pop**' is performing using high-low register order:

```
pop    t                    →       pop    r25
                                    pop    r24
```

see also: **push**

3.3. **LDS**

```
mnemonic:     lds
direction:    low-high
operand0:     register(0)
operand1:     address (immediate)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
lds tl, 0x0200	1	2	4	lds tl, 0x0200	---- ----

lds tz, 0x0200	4	8	16	lds tl, 0x0200
				lds th, 0x0201
				lds zl, 0x0202
				lds zh, 0x0203
					---- ----

read ram at immediate address to register

as an read operation '**lds**' is performing using low-high register order:

```
lds    t, adr          →      lds    r24, adr
                              lds    r25, adr + 1
```

see also: **sts**

3.4. **STS**

```
mnemonic:     sts
direction:    high-low
operand0:     address (immediate)
operand1:     register(0)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
sts 0x0200, tl	1	1	4	sts 0x0200, tl	---- ----
sts 0x0200, tz	4	8	16	sts 0x0203, zh sts 0x0202, zl sts 0x0201, th sts 0x0200, tl	---- ----

write register(0) to ram at immediate address

as an write operation '**sts**' is performing using high-low register order:

```
sts    adr, t          →      sts    adr + 1, r25
                              sts    adr, r24
```

see also: **lds**

3.5. **LDD**

```
mnemonic:     ldd
direction:    low-high
operand0:     register(0)
operand1:     address register + offset (immediate)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
ldd tl, z + 5	1	1	2	ldd tl, z + 5	---- ----
ldd ty, z + 5	4	8	8	ldd tl, z + 5 ldd th, z + 6 ldd yl, z + 7 ldd yh, z + 8	---- ----

read ram at address register with displacement to register:

as an read operation '**ldd**' is performing using low-high register order:

```
ldd    t, z + offestr      →      ldd    r24, z + offset
                                  ldd    r25, z + offset + 1
```

see also: **std**

3.6. **STD**

```
mnemonic:     std
direction:    high-low
operand0:     address register + offset (immediate)
operand1:     register(0)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
std z + 5, tl	1	2	2	std z + 5, tl	---- ----
std z + 5, ty	4	8	8	std z + 8, yh std z + 7, yl std z + 6, th std z + 5, tl	---- ----

write register to ram at address register with displacement:

as an write operation '**std**' is performing using high-low register order:

```
std    reg + offset , t    →      std    reg + offset + 1 , r25
                                  std    reg + offset, r24
```

see also: **ldd**

3.7. **LD**

```
mnemonic:     ld
direction:    low-high
operand0:     register(0)
operand1:     address register
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
ld tl, z+	1	2	2	ld tl, z+	---- ----

ld ty, z+	4	8	8	ld tl, z+
				ld th, z+
				ld yl, z+
				ld yh, z+
					---- ----

read ram at address register to register.

as an read operation '**ld**' is performing using low-high register order:

```
ld     t, z+          →     ld     r24, z+
                            ld     r25, z+
```

see also: **st**

3.8. **ST**

```
mnemonic:     st
direction:    high-low
operand0:     address register
operand1:     register(0)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
st -z, tl	1	2	2	st -z, tl	---- ----
st -z, ty	4	8	8	st -z, yh st -z, yl st -z, th st -z, tl	---- ----

write register0 to ram at address register.

as an write operation '**st**' is performing using high-low register order:

```
st     -z, t          →     st     -z, r25
                            st     -z, r24
```

see also: **ld**

3.9. **LPM**

```
mnemonic:     lpm
direction:    low-high
operand0:     register(0)
operand1:     address register
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
lpm tl, z+	1	3	2	lpm tl, z+	---- ----
lpm ty, z+	4	12	8	lpm tl, z+ lpm th, z+ lpm yl, z+ lpm yh, z+	---- ----

read flash from address register to the specified register.

as an read operation '**lpm**' is performing using low-high register order:

```
lpm    t, z+          →      lpm    r24, z+
                             lpm    r25, z+
```

see also: **elpm**

3.10. **ELPM**

```
mnemonic:     elpm
direction:    low-high
operand0:     register(0)
operand1:     address register
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
elpm tl, z+	1	3	2	elpm tl, z+	---- ----
elpm ty, z+	4	12	8	elpm tl, z+ elpm th, z+ elpm yl, z+ elpm yh, z+	---- ----

extended read flash from address register to the specified register.

as an read operation '**elpm**' is performing using low-high register order:

```
elpm   t, z+          →      elpm   r24, z+
                             elpm   r25, z+
```

see also: **lpm**

3.11. IN

```
mnemonics:    in
direction:    low-high
operand0:     register(0)
operand1:     port (immediate)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
in tl, 0x3F	1	1	2	in tl, ox3F	---- ----
in tl, 0x60	1	2	4	lds tl, 0x60	---- ----
in t, 0x3E	2	2	4	in tl, 0x3E in th, 0x3F	---- ----
in t, 0x3F	2	3	6	in tl, 0x3F lds th, 0x60	---- ----
in t, 0x60	2	4	8	lds tl, 0x60 lds th, 0x61	---- ----

read from given io location to a register

as an read operation '**in**' is performing using low-high register order:

```
in     t, port          →     in     r24, port
                              in     r25, port + 1
```

instruction uses both methods for accessing to io: by using '**in**' or '**lds**' command. complex instruction determinates that's which ones to use for the each byte read:

```
in     t, 0x3F          →     in     tl, 0x3F
                              lds    th, 0x0060
```

see also: **out**, **io definitions**

3.12. **OUT**

```
mnemonics:    out / sts
direction:    high-low
operand0:     port (immediate)
operand1:     register(0)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
out 0x3F, tl				out 0x3F, tl	---- ----
out 0x60, tl				sts 0x60, tl	---- ----
out 0x3E, t				out 0x3F, th out 0x3E, tl	---- ----
out 0x3F, t				sts 0x60, th out 0x3F, tl	---- ----
out 0x60, t				sts 0x61, th sts 0x60, tl	---- ----

write register(s) to a given io location

as an write operation '**out**' is performing using high-low register order:

```
out    port, t          →      out    port + 1, r25
                               out    port + 0, r24
```

instruction uses both methods for accessing to io: by using '**out**' and '**sts**' commands. complex instruction determinates that's which ones to use for the each byte write:

```
out    0x3F, t          →      sts    0x0060, th
                               out    0x3F, tl
```

see also: **in**, **outi**

3.13. **MOV**

```
mnemonics:    mov / movw
direction:    low-high
operand0:     register(0)
operand1:     register(1)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
mov tl, xl	1	1	2	mov tl, xl	---- ----
mov tl, x	1	1	2	mov tl, xl	---- ----
mov x, tl	2	2	4	mov xl, tl mov xh, r9	---- ----
mov x, t	2	1	2	movw xl, tl	---- ----
mov tz, xy	4	2	4	movw tl, xl movw zl, yl	---- ----

mov tz, xl	4	4	8	mov tl, xl
				mov th, r9
				mov zl, r9
				mov zh, r9
					---- ----

mov tz, x	4	3	6	movw tl, xl
				mov zl, r9
				mov zh, r9
					---- ----

mov t, _all	2	2	4	mov tl, r9 mov th, r9	---- ----

copy contents of register(1) to register(0).

if register0 is wider than register1 then register0 high part will be zeroed:

```
mov    tz, xl          →      mov    tl, xl
                              mov    th, r9
                              mov    zl, r9
                              mov    zh, r9
```

this have side effect that using register _all in operand1 will leads to clearing register0:

```
mov    t, _all         →      mov    tl, r9
                              mov    th, r9
```

if register1 is wider than register0 then only parts of register0 will be handled:

```
mov    xl, tz          →      mov    xl, tl
```

'**mov**' have optimization using '**movw**' instruction if it's possible:

```
mov    t, x            →      movw   tl, xl

mov    tz, xy          →      movw   tl, xl
                              movw   zl, yl
```

see also: **movw**

3.14. **MOVW**

```
mnemonics:    movw
direction:    low-high
operand0:     register-pair(0)
operand1:     register-pair(1)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
movw tl, xl	2	1	2	movw tl, xl	---- ----
movw tl, x	2	1	2	movw tl, xl	---- ----
movw t, xl	2	1	2	movw xl, tl	---- ----
movw t, x	2	1	2	movw tl, xl	---- ----
movw tz, xy	4	2	4	movw tl, xl movw zl, yl	---- ----

movw tz, x				movw tl, xl
				mov zl, r9
				mov zh, r9
					---- ----

copy content of register-pair1 to register-pair0

as the register-pair both allowed: low register or pair itself:

```
movw   tl, xl          ↔      movw   t, xl
```

see also: **mov**

3.15. SER

```
mnemonic:     ser / or
direction:    low-high
operand0:     register(0)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
sbr tl, 3	1	1	2	ori tl, 1 << 3	---- ----
sbr tz, 23	4	1	2	ori zl, 1 << 7	---- ----
sbr rl, 5	1	2	4	ldi tmp, 1 << 5 or rl, tmp	

set register. this instruction is not exist in AVR cpu instruction set. Indeed that the '**ser**' is the form of the '**ldi**' instruction:

```
ser    reg                  →      ldi    reg, 0xFFFFFFFF...
```

see also: **clr**, **mov**

3.16. CLR

```
mnemonic:     clr / mov
direction:    low-high
operand:      register(0)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
clr tl	1	1	2	clr tl	---- ----
clr tz	4	1	2	clr tl clr th clr zl clr zh	---- ----
clr r	1	2	4	mov rl, r9 mov rh, r9	

clear register. this instruction is not exist in AVR cpu instruction set. Indeed that the '**clr**' is the form of the '**ldi**' instruction.

```
clr    reg                  →      ldi    reg, 0
```

see also: **ser**, **mov**

3.17. **BST**

```
mnemonic:     bst
operand0:     register(0)
operand1:     bit number
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
bst tl, 5	1	1	2	bst tl, 5	-T-- ----
bst tz, 22	4	1	2	bst zl, 6	-T-- ----

copy bit from specified register to the **T**-flag in SREG.

the bit number is linear for the specified complex register:

see also: **bld**

3.18. **BLD**

```
mnemonic:     bld
operand0:     register(0)
operand1:     bit number
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
bld tl, 5	1	1	2	bld tl, 5	-T-- ----
bld tz, 22	4	1	2	bld zl, 6	-T-- ----

copy **T**-flag in SREG to the specified bit / register.

the bit number is linear for the specified complex register:

see also: **bst**

3.19. **SBRS**

```
mnemonic:     sbrs
operand0:     register(0)
operand1:     bit number
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
sbrs tl, 5	1	1 2	2	sbrs tl, 5	---- ----
sbrs tz, 22	4	1 2	2	sbrs zl, 6	---- ----

skip next instruction in case if specified bit/register is set.

the bit number is linear for the specified complex register:

see also: **sbrc**

3.20. **SBRC**

```
mnemonic:     sbrc
operand0:     register(0)
operand1:     bit number
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
sbrc tl, 5	1	1 2	2	sbrc tl, 5	---- ----
sbrc tz, 22	4	1 2	2	sbrc zl, 6	---- ----

skip next instruction in case if specified bit/register is clear.

the bit number is linear for the specified complex register:

see also: **sbrs**

3.21. **ROL**

```
mnemonic:     rol
direction:    low-high
operand:      register(0)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
rol tl	1	1	2	rol tl	---- VNZC
rol tz	4	4	8	rol tl rol th rol zl rol zh	---- VNZC

shift left register through carry **C**-flag:

see also: **ror**

3.22. **ROR**

```
mnemonic:     ror
direction:    high-low
operand:      register(0)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
ror tl	1	1	2	ror tl	---- VNZC
ror tz	4	4	8	ror zh ror zl ror th ror tl	---- VNZC

shift right register through carry **C**-flag:

see also: **rol**

3.23. **LSL**

```
mnemonic:     lsl / rol
direction:    low-high
operand:      register(0)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
lsl tl	1	1	2	lsl tl	---- VNZC
lsl tz	4	4	8	lsl tl rol th rol zl rol zh	---- VNZC

shift left register through zero to carry **C**-flag:

see also: **rol**, **lsr**

3.24. **LSR**

```
mnemonic:     lsr / ror
direction:    high-low
operand:      register(0)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
lsr tl	1	1	2	lsr tl	---- VNZC
lsr tz	4	4	8	lsr zh ror zl ror th ror tl	---- VNZC

shift right register through zero to carry **C**-flag

see also: **ror**, **lsl**

3.25. **ASR**

```
mnemonic:     asr / ror
direction:    high-low
operand:      register(0)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
asr tl	1	1	2	asr tl	---- VNZC
asr tz	4	4	8	asr zh ror zl ror th ror tl	---- VNZC

arithmetic shift right register through zero to carry **C**-flag

see also: **ror**, **lsr**

3.26. **INC**

```
mnemonic:     inc / brne / adiw
direction:    low-high
operand:      register(0)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
inc tl	1	1	2	inc tl	---- VNZ-
inc a	2	3	6	inc al brne lx inc ah lx:	---- VNZ-
inc t	2	2	2	adiw tl, 1	
inc tz	4	3 5 7	14	inc tl brne lx inc th brne lx inc zl brne lx inc zh lx:	
inc r	2	3	6	inc r0 brne lx inc r1	

increment register

If register's wide is equals to 1 then normal '**inc**' instruction will be generated:

```
inc    tl          →      inc    tl
```

otherwise if register's wide is equals to 2 AND register is the one from the hi-pairs then '**adiw**' instruction will be generated:

```
inc    t           →      adiw   tl, 1
```

otherwise '**inc**'-'**brne**' sequence will be generated:

```
inc    a           →      inc    al
                          brne   lx
                          inc    ah
                      lx:

inc    tz          →      inc    tl
                          brne   lx
                          inc    th
                          brne   lx
                          inc    zl
                          brne   lx
                          inc    zh
                      lx:
```

see also: **dec**

3.27. **DEC**

```
mnemonic:     dec / subi / sbc / sec / sbiw
direction:    low-high
operand:      register(0)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
dec tl	1	1	2	dec tl	---- VNZ-
dec a	2	2	4	subi al, 1 sbc ah, r9	---- VNZ-
dec t	2	2	2	sbiw tl, 1	
dec tz	4	4	6	sbiw tl, 1 sbc zl, r9 sbc zh, r9	
dec r	2	3	6	sec sbc r0, r9 sbc r1, r9	

decrement register

If register's wide is equals to 1 then normal '**dec**' instruction will be generated:

```
dec    tl                 →      dec    tl
```

otherwise if first two registers of the provided mask is the one from hi-pairs then '**sbiw**'-'**sbc**' sequence will be generated:

```
dec    t                  →      sbiw   tl, 1

dec    tz                 →      sbiw   tl, 1
                                 sbc    zl, r9
                                 sbc    zh, r9
```

otherwise in case if lower physical register is in the range 0..15 then '**secsbc**'-'**sbc**' sequence will be generated:

```
dec    r                  →      sec
                                 sbc    r0, r9
                                 sbc    r1, r9
```

if lower physical register is in the range 16..31 then '**subi**'-'**sbc**' sequence will be generated:

```
dec    a                  →      subi   al, 1
                                 sbc    ah, r9

dec    ab                 →      subi   al, 1
                                 sbc    ah, r9
                                 sbc    bl, r9
                                 sbc    bh, r9
```

see also: **inc**

3.28. **AND**

```
mnemonic:     and / andi
direction:    low-high
operand0:     register(0)
operand1:     register(1) or immediate value
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
and tl, xl	1	1	2	and tl, xl	---- VNZ-
and tz, xy	4	4	8	and tl, xl and th, xh and zl, yl and zh, yh	---- VNZ-
and tz, x	2	2	4	and tl, xl and th, xh	
and x, tz	2	2	4	and xl, tl and xh, th	
and tl, 0x64	1	1	2	andi tl, 0x64	
and tz, 20	4	4	8	andi tl, 0x14 andi th, 0x00 andi zl, 0x00 andi zh, 0x00	
and tz, 0x44332211	4	4	8	andi tl, 0x11 andi th, 0x22 andi zl, 0x33 andi zh, 0x44	
and tz, 0xFF33FF11	4	2	4	andi tl, 0x11 andi zl, 0x33	
and tl, 0xFF	1	1	2	andi tl, 0xFF	

performs logic AND operation between two registers or register and immediate value. the result is stored back in register0.

if operand1 is the name of defined complex register then that's will be *register-register* operation using '**and**' cpu instruction:

```
and    tz, xy          →      and    tl, xl
                              and    th, xh
                              and    zl, yl
                              and    zh, yh
```

if register0 is wider than register1 then operation will be performed with the register1's bit-wide:

```
and    tz, x           →      and    tl, xl
                              and    th, xh
```

if register1 is wider than register0 then operation will be performed with the register0's bit-wide:

```
and    x, tz           →      and    xl, tl
                              and    xh, th
```

if operand1 is not the name of defined complex register then that's will be *register-immediate* operation using '**andi**' cpu instruction:

```
and    x, 0x2211       →      andi   xl, 0x11
                              andi   xh, 0x22
```

operand1 immediate should be a number but not an expression. an expression in operand1 will

produce error while compiling:

```
and    x, 2              →      andi   xl, 2        ok
                                andi   xh, 0

and    x, 1 + 2          →                          error
```

this is because of avrasm2 limitation that is producing an error if expression is provided as the parameter for the built-in **defined**() func-macro.

'**and**' *register-immediate* have the special rule that's excluding from the generated commands an instructions that is 'and' an register with the value 0xFF (due to it's senselless). however the first instruction is always present independetly from it's value:

```
and    tz, 0xFFFFFFFF    →      andi   tl, 0xFF

and    tz, 44FF33FF      →      andi   th, 0x33
                                andi   zh, 0x44
```

the flags in SREG will be appropriated to the last cpu instruction performed.

see also: **andi**, **sub**, **instruction tales**

3.29. ANDI

```
mnemonic:     andi
direction:    low-high
operand0:     register name
operand1:     immediate (expression)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
andi xl, 101	1	1	2	andi xl, 101	---- VNZ-
andi tz, 0x44332211	2	2	4	andi tl, 0x11 andi th, 0x22 andi zl, 0x33 andi zh, 0x44	---- VNZ-
andi xl, 0xFF				andi xl, 0xFF	
andi tz, 0xFF33FF11				andi tl, 0x11 andi th, 0xFF andi zl, 0x33	

performs logic AND operation between register(0) and immediate value. the result is stored back in register0.

'**andi**' supports an expression in operand1 immediate:

```
andi   x, 1 + 2          →      andi   xl, 0x03
                                andi   xh, 0x00
```

the flags in SREG will be appropriated to the last cpu instruction performed.

see also: **and**, **subi**

3.30. **OR**

```
mnemonic:     or /  ori
direction:    low-high
operand0:     register name
operand1:     register name or immediate
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
`or  tl, xl`	1	1	2	`or  tl, xl`	`---- VNZ-`

`or  tz, xy`		2	4	`or  tl, xl`
				`or  th, xh`
				`or  zl, yl`
				`or  zh, yh`
					`---- VNZ-`

`or  tz, x`				`or  tl, xl` `or  th, xh`	
`or  x, tz`				`or  xl, tl` `or  xh, th`	
`or  tl, 100`				`ori  tl, 0x64`	
`or  tz, 20`				`ori  tl, 0x14`	

`or  tz, 0x44332211`				`ori  tl, 0x11`
				`ori  th, 0x22`
				`ori  zl, 0x33`
				`ori  zh, 0x44`	

`or  tz, 0x00330011`				`ori  tl, 0x11` `ori  zl, 0x33`	
`or  tl, 0x00`				`ori  tl, 0x00`	

performs logic OR operation between two registers or register and immediate value. the result is stored back in register0.

if operand1 is the name of defined complex register then that's will be *register-register* operation using '**or**' cpu instruction:

```
or    tz, xy          →    or    tl, xl
                           or    th, xh
                           or    zl, yl
                           or    zh, yh
```

if register0 is wider than register1 then operation will be performed with the register1 bit-wide:

```
or    tz, x           →    or    tl, xl
                           or    th, xh
```

if register1 is wider than register0 then operation will be performed with the register0 bit-wide:

```
or    x, tz           →    or    xl, tl
                           or    xh, th
```

if operand1 is not the name of defined complex register then that's will be *register-immediate* operation using '**ori**' cpu instruction:

```
or    x, 0x2211       →    ori   xl, 0x11
                           ori   xh, 0x22
```

operand1 immediate should be a number but not an expression. an expression in operand1 will produce error while compiling:

```
or      x, 2              →      ori    xl, 2        ok
                                 ori    xh, 0
or      x, 1 + 2          →                          error
```

this is because of avrasm2 limitation that is producing an error if expression is provided as the parameter for the built-in **defined**() func-macro.

'**or**' *register-immediate* have the special rule that's excluding from the generated commands an instructions that is 'or' an register with the value 0x00 (due to it's senselless). however the first instruction is always present independetly from it's value:

```
or      tz, 0xFFFFFFFF    →      ori    tl, 0xFF

or      tz, 0x44003300    →      ori    th, 0x33
                                 ori    zh, 0x44
```

the flags in SREG will be appropriated to the last cpu instruction performed.

see also: **ori**, **add**, **instruction tales**

3.31. ORI

```
mnemonic:    ori
direction:   low-high
operand0:    register(0)
operand1:    immediate value (expression)
```

INSTRUC	WIDE	CLK	LEN	CODE	ITHS VNZC
ori tl, 0x00	1	1	2	ori tl, 0x00	---- VNZ-
ori x, 0x2211	2	2	4	ori xl, 0x11 ori xh, 0x22	---- VNZ-
ori x, 0x2200	2	1	2	ori xh, 0x22	---- VNZ-
ori x, 0x0011	2	1	2	ori xl, 0x11	---- VNZ-
ori x, 0	-	-	-	-	

ori r, 0x2211	2	4	8	ldi r19, 0x11
				or rl, r19
				ldi r19, 0x22
				or rh, r19
					---- VNZ-

ori tz, 0x44332211	4	4	8	ori tl, 0x11 ori th, 0x22 ori zl, 0x33 ori zh, 0x44	---- VNZ-
ori tz, 0x00330011	4	2	4	ori tl, 0x11 ori zl, 0x33	---- VNZ-

performs logic OR operation between register(0) and immediate value. the result is stored back in register0.

note that '**ori**' supports an expression in operand1:

```
ori     x, 0xFF00 + 3     →      ori    xl, 0x03
                                 ori    xh, 0xFF
```

'ori' will skip instructions that is OR an register with the zero value
the AVR instruction set do not allows '**ori**' for a lower 16 cpu registers. however 'in case if '**ori**' accessing lower 16 cpu registers then '**ori_**' pseudoinstruction will be used.

see also: **or**, **ori_**

3.32. **EOR / XOR**

```
mnemonic:     eor
direction:    low-high
operand0:     register(0)
operand1:     register(1) or immediate value
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
eor tl, xl	1	1	2	eor tl, xl	---- ----
eor tz, xy	2	3	6	eor tl, xl eor th, xh eor zl, yl eor zh, yh	---- ----
eor tz, x				eor tl, xl eor th, xh	
eor x, tz				eor xl, tl eor xh, th	
eor tl, 100				eori tl, 0x64	
eor tz, 0x44332211				eori tl, 0x11 eori th, 0x22 eori zl, 0x33 eori zh, 0x44	
eor tz, 0x00330011				eori tl, 0x11 eori zl, 0x33	
eor tl, 0x00				eori tl, 0x00	

performs logic XOR operation between two registers or register and immediate value. the result is stored back in register0.

note that there is no differences between '**eor**' and '**xor**' mnemonics. same as with '**eori**' and '**xori**' pseudo instructions.

if operand1 is the name of defined complex register then that's will be *register-register* operation using '**eor**' cpu instruction:

```
xor    tz, xy          →      eor    tl, xl
                              eor    th, xh
                              eor    zl, yl
                              eor    zh, yh
```

if register0 is wider than register1 then operation will be performed with the register1 bit-wide:

```
eor    tz, x           →      eor    tl, xl
                              eor    th, xh
```

if register1 is wider than register0 then operation will be performed with the register0 bit-wide:

```
eor    x, tz           →      eor    xl, tl
                              eor    xh, th
```

note that this behaviour is actual not for all instructions. see '**instruction tales**' and '**add**'.

if operand1 is not the name of defined complex register then that's will be *register-immediate* operation using '**xori**' pseudo instruction:

```
xor    x, 0x2211       →      xori   xl, 0x11
                              xori   xh, 0x22
```

operand1 immediate should be a number but not an expression. an expression in operand1 will produce error while compiling:

```
xor    x, 2              →      xori   xl, 2        ok
                                xori   xh, 0

xor    x, 1 + 2          →                          error
```

this is because of avrasm2 limitation that is producing an error if expression is provided as the parameter for the built-in **defined**() func-macro.

'**eor**'/'**xor**' *register-immediate* have the special rule that's excluding from the generated commands an instructions that is 'xor' an register with the value 0x00 (due to it's senselless). however the first instruction is always present independetly from it's value:

```
eor    tz, 0x00000000    →      xori   tl, 0x00

eor    tz, 0x44003300    →      xori   th, 0x33
                                xori   zh, 0x44
```

see also: **and**, **or**, **instruction tales**

3.33. EORI / XORI

```
pseudo:       eori / xori
mnemonics:    eor / com
direction:    low-high
operand0:     register(0)
operand1:     immediate value: 0, 0xFF supported
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
xori tl, 0	1	1	2	eor tl, r9	---- ----
xori tl, 0xFF	1	1	2	com tl	---- ----

xor register with immediate value.

'**eori**'/ '**xori**' pseudoinstruction is based on the low-level macro '**eori_**' that is works with a single physical register.

note that '**eori_**' is implemented partially and currently supports immediate value 0 and 0xFF only.

see also: **addi, adci, movi**

3.34. **SBR**

```
mnemonic:     ori
direction:    low-high
operand0:     register(0)
operand1:     bit number
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
sbr tl, 3	1	1	2	ori tl, 1 << 3	---- ----
sbr tz, 23	4	1	2	ori zl, 1 << 7	---- ----
sbr rl, 5	1	2	4	ldi tmp, 1 << 5 or rl, tmp	

set bit in register. this instruction is not exist in AVR cpu instruction set. Indeed that the '**sbr**' is the form of the '**ori**' instruction:

```
sbr    reg, bit          →      ori    reg, 1 << bit
```

see also: ori, cbr

3.35. **CBR**

```
mnemonic:     andi
direction:    low-high
operand0:     register(0)
operand1:     bit number
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
cbr tl, 3	1	1	2	andi tl, ~ 1 << 3	---- ----
cbr tz, 23	4	1	2	andi zl, ~ 1 << 7	---- ----
cbr rl, 5	1	2	4	ldi tmp, ~ 1 << 5 and rl, tmp	

clear bit in register. this instruction is not exist in AVR cpu instruction set. Indeed that the '**cbr**' is the form of the '**andi**' instruction.

```
cbr    reg, bit          →      andi   reg, ~ 1 << bit
```

see also: andi, sbr

\

3.36. **NEG**

```
mnemonics:    neg / com
direction:    low-high
operand:      register
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
neg tl	1	1	2	neg tl	---- ----
neg tz	4	4	8	neg tl com th com zl com zh	---- ----

replaces the contents of register0 with its two's complement: register0 = 0 - register0

see also: **com**

3.37. **COM**

```
mnemonics:    com
direction:    low-high
operand:      register
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
com tl	1	1	2	com tl	---- ----
com tz	4	4	8	com tl com th com zl com zh	---- ----

performs a one's complement of register0. inverse it:

```
register = 0xFF - register
```

see also: **neg**

3.40. **ADD**

```
mnemonic:     add / adc / adiw
direction:    low-high
operand0:     register
operand1:     register or immediate
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
`add tl, xl`	1	1	2	`add tl, xl`	---- ----

`add tz, xy`	2	3	6	`add tl, xl`
				`adc th, xh`
				`adc zl, yl`
				`adc zh, yh`
					---- ----

`add tz, x`				`add tl, xl`
				`adc th, xh`
				`adc zl, r9`
				`adc zh, r9`	

`add x, tz`				`add xl, tl` `adc xh, th`	

`add tz, 0x3F`				`adiw tl, 0x3F`
				`adc zl, r9`
				`adc zh, r9`	

`add tz, 0x4433003F`				`adiw tl, 0x003F`
				`adci zl, 0x33`
				`adci zh, 0x44`	

`add tz, 64`				`ldi tmp, 64`
				`add tl, tmp`
				`adc th, nul`
				`adc zl, nul`
				`adc zh, nul`	

adds register0 and register1 or immediate value and store result back in register0.

if operand1 is the name of defined complex register then that's will be *register-register* operation using '**add**' and '**adc**' cpu instructions:

```
add    tz, xy          →      add    tl, xl
                              adc    th, xh
                              adc    zl, yl
                              adc    zh, yh
```

if register0 is wider than register1 then operation will be performed with the register0 bit-wide:

```
add    tz, x           →      add    tl, xl
                              adc    th, xh
                              adc    zl, r9        ←      tail
                              adc    zh, r9
```

if register1 is wider than register0 then operation will be performed with register0 bit-wide:

```
add    x, tz           →      add    xl, tl
                              adc    xh, th
```

if operand1 is not the name of defined complex register then that's will be *register-immediate* operation using '**addi**' and '**adci**' pseudo instructions ad cpu :

```
add    x, 2            →      addi   xl, 2         ok
                              adci   xh, 0
```

note that pseudo instructions are implemented partially. see **addi**.

'**add**' use '**adiw**' cpu instruction in case if its possible:

```
add    x,
```

operand1 immediate should be a number but not an expression. an expression in operand1 will produce an error while compiling:

```
add    x, 2              →      addi   xl, 2        ok
                                adci   xh, 0

add    x, 1 + 2          →                          error
```

this is because of avrasm2 limitation that is producing an error if expression is provided as the parameter for the built-in **defined**() func-macro.

see also: **sub**

3.37. ADDI

```
pseudo:       addi
mnemonics:    add / adc / sec
direction:    low-high
operand0:     register(0)
operand1:     immediate value: 0 and 1 supported
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
addi rl, 1	1	2	4	sec adc r0, r9	---- ----
addi al, 1	1	2	4	sec adc tl, r9	
addi a, 1	2	3	6	sec adc r20, r9 adc r21, r9	
addi t, 1	2	2	2	adiw tl, 1	
addi tz, 1	4	4	6	adiw tl, 1 adc zl, r9 adc zh, r9	
addi tzr0, 1	5	6	12	sec adc r0, r9 adc tl, r9 adc th, r9 adc zl, r9 adc zh, r9	

add immediate value to register.

'**addi**' instruction based on the '**addi_**' pseudoinstruction. see PSEUDOINSTRUCTIONS.

see also: **adci, eori/xori, movi**

3.38. **ADC**

```
mnemonic:     adc
direction:    low-high
operand0:     register(0)
operand1:     register(1)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
cpi tl, val	1	1	2	cpi tl, val	---- ----
cpi x, val	2	3	6	cpi xh, BYTE1(val) brne lx cpi xl, BYTE0(val) lx:	---- ----

adds register0 and register1 and **C**-flag in SREG and store result back in register0

see also: **sbc**

3.39. **ADCI**

```
pseudo:       adci
mnemonics:    add / adc / sec
direction:    low-high
operand0:     register(0)
operand1:     immediate value: 0 supported
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
adci tl, 0	1	1	2	adc tl, r9	---- ----

add register with immediate value and **C**-flag of SREG.

'**adci**' pseudoinstruction is based on the low-level macro '**adci_**' that is works with a single physical register.

note that '**adci_**' is implemented partially and currently supports immediate value 0 only.

see also: **addi, eori/xori, movi**

3.40. **SUB**

```
mnemonic:     sub / sbc / sbci / subi /sbiw
direction:    low-high
operand0:     register
operand1:     register or immediate
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
sub tl, xl	1	1	2	cpi tl, val	--HS VNZC
sub tz, xy	2	3	6	cpi xh, BYTE1(val) brne lx cpi xl, BYTE0(val) lx:	--HS VNZC
sub tz, x				sub tl, xl sbc th, xh sbci zl, 0 sbci zh, 0	--HS VNZC
sub x, tz				sub xl, tl sbc xh, th	--HS VNZC
sub tz, 0x3F				sbiw tl, 0x3F sbci zl, 0 sbci zh, 0	--HS VNZC
sub tz, 0x4433003F				sbiw tl, 0x3F sbci zl, 0x33 sbci zh, 0x44	--HS VNZC
sub tz, 0x40				subi tl, 0x40 sbci th, 0 sbci zl, 0 sbci zh, 0	--HS VNZC

subtract register1 from register0 or immediate and store result back in register0.

if operand1 is the name of defined complex register then that's will be *register-register* operation using '**sub**' and '**sbc**' cpu instructions:

```
sub    tz, xy          →      sub    tl, xl
                              sbc    th, xh
                              sbc    zl, yl
                              sbc    zh, yh
```

if register0 is wider than register1 then operation will be performed with register0 bit-wide:

```
sub    tz, x           →      sub    tl, xl
                              sbc    th, xh
                              sbci   zl, 0
                              sbci   zh, 0
```

if register1 is wider than register0 then operation will be performed with register0 bit-wide:

```
sub    x, tz           →      sub    xl, tl
                              sbc    xh, th
```

if operand1 is not the name of defined complex register then that's will be *register-immediate* operation using '**subi**' and '**sbci**' cpu instructions:

```
sub    x, 2            →      subi   xl, 2
                              sbci   xh, 0
```

'**sub**' use '**sbiw**' cpu instruction in case if its possible.

operand1 immediate should be a number but not an expression. an expression in operand1 will produce an error while compiling:

```
sub    x, 2              →      subi   xl, 2        ok
                                sbci   xh, 0

sub    x, 1 + 2          →                          error
```

this is because of avrasm2 limitation that is producing an error if expression is provided as the parameter for the built-in **defined**() func-macro.

see also: **add**

3.41. SUBI

```
mnemonic:     subi / sbci /sbiw
direction:    low-high
operand0:     register(0)
operand1:     immediate value
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
subi tl, val	1	1	2	subi tl, val	--HS VNZC
subi tz, val	4	4	8	subi tl, BYTE0(val) sbci th, BYTE1(val) sbci zl, BYTE2(val) sbci zh, BYTE3(val)	--HS VNZC
subi tz, 64	4	4	8	subi tl, 0x40 sbci th, 0 sbci zl, 0 sbci zh, 0	--HS VNZC
subi tz, 0x44332211	4	4	8	subi tl, 0x11 sbci th, 0x22 sbci zl, 0x33 sbci zh, 0x44	--HS VNZC
subi tz, 63	4	4	6	sbiw tl, 0x3F sbci zl, 0 sbci zh, 0	--HS VNZC
subi tz, 0x4433003F	4	4	6	sbiw tl, 0x3F sbci zl, 0x33 sbci zh, 0x44	--HS VNZC

subtract immediate value from register0. the result is stored back in register0.

'**subi**' allows expression in operand1.

see also: **sbci**

3.42. **SBC**

```
mnemonic:     sbc / sbci
direction:    low-high
operand0:     register(0)
operand1:     register(1) or immediate value
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
sbc tl, xl	1	1	2	sbc tl, xl	--HS VNZC
sbc tz, xy	4	4	8	sbc tl, xl sbc th, xh sbc zl, yl sbc zh, yh	--HS VNZC

subtract **C**-flag in SREG and register1 or immediate value from register0 and store result back in register0.

see also: **adc**

3.43. **SBCI**

```
mnemonic:     sbci
direction:    low-high
operand0:     register(0)
operand1:     immediate value
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
sbci tl, val	1	1	2	sbci tl, val	--HS VNZC
sbci tz, val	4	4	8	sbci tl, BYTE0(val) sbci th, BYTE1(val) sbci zl, BYTE2(val) sbci zh, BYTE3(val)	--HS VNZC

subtract **C**-flag in SREG and immediate value from register0 and store result back in register0.

'**sbci**' allows expression in operand1.

see also: **subi**

3.38. **ADIW**

```
mnemonic:     adiw
operand0:     high register-pair(0)
operand1:     immediate value
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
adiw tl, 63	2	2	2	adiw tl, 63	---- ----
adiw x, 63	2	2	2	adiw xl, 63	---- ----

adds an immediate value (0-63) to a register-pair and write back result to register-pair.

as the high register-pair low register of a high-pairs may be specified or hugh register-pair itself:

```
adiw   tl, 1

adiw   t, 1
```

'**adiw**' supports an expression in operand1 immediate:

```
adiw   x, 1 + 2          →      adiw   xl, 3
```

see also: **sbiw**

3.39. **SBIW**

```
mnemonic:     sbiw
operand0:     high register(0)-pair
operand1:     immediate value
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
sbiw tl, 63	2	2	2	sbiw tl, 63	---- ----
sbiw x, 63	2	2	2	sbiw xl, 63	---- ----

subtracts immediate value (0-63) from a register-pair and write back result to register-pair.

as the register-pair low register of a high-pairs may be specified or register-pair itself:

```
sbiw   xl, 1

sbiw   x, 1
```

'**sbiw**' supports an expression in operand1 immediate:

```
sbiw   x, 1 + 2          →      sbiw   xl, 3
```

see also: **adiw**

3.46. **SWAP**

```
mnemonic:     swap
direction:    -
operand0:     register(0)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
swap tl	1	1	2	swap tl	---- ----
swap t	2	2	4	swap tl swap th	---- ----

swap nibbles in register0

3.47. **CPC**

```
mnemonics:    cpc
direction:    high-low
operand0:     register(0)
operand1:     register(1)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
cpc tl	1	1	2	cpc tl	--HS VNZC
cpc tz	4	4	8	cpc tl cpc th cpc zl cpc zh	--HS VNZC

zzzz

see also: **neg**

3.48. **CPSE**

```
mnemonics:    cpse
direction:    high-low
operand0:     register(0)
operand1:     register(1)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
cpse tl, al	1	1	2	cpse tl, al	---- ----
cpse t, x	2	2	4	cpse tl, xl cpse th, xh	---- ----

zzzz

see also: **neg**

3.44. **CP**

```
mnemonic:     cp
direction:    high-low
operand0:     register(0)
operand1:     register(1) or immediate value
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
cp al, tl	1	1	2	cp al, tl	--HS VNZC
cp al, t	1 2	4	8	sec clz cpse th, r9 cp al, tl lx:	--HS VNZC
cp a, tl	1 2	3	6	cpse ah, r9 cp al, tl lx:	--HS VNZC
cp a, t	2	3	6	cp ah, th brne lx cp al, tl lx:	--HS VNZC
cp tz, xy	4	3 5 7	14	cp zh, yh brne lx cp zl, yl brne lx cp th, xh brne lx cp tl, xl lx:	--HS VNZC

compare register with register or immediate.

the comparing operation is exactly the same as the alu-subtraction operations: '**sub**' or '**subi**' - except that the result of subtraction is not stored anywhere. however cpu flags contains appropriated values as the result of subtraction pefrformed.

if operand1 is a defined register name then *reg - reg1* comparing operation will be performed:

```
cp     tz, xy          →      cp     zh, yh
                              brne   lx
                              cp     zl, yl
                              brne   lx
                              cp     th, xh
                              brne   lx
                              cp     tl, xl
                          lx:
```

if register0 is wider than register1 then higher part or register0 is checking for zero value. the low part will then compared with the register1:

```
cp     tz, x           →      or     zh, zh
                              brne   lx
                              or     zl, zl
                              brne   lx
                              cp     th, xh
                              brne   lx
                              cp     tl, xl
                          lx:
```

if register1 is wider than register0 then higher part of register1 is checking for zero value. the low part will then compared with the register:

```
cp     x, tz              →      or     zh,zh
                                 brne   lx
                                 or     zl, zl
                                 brne   lx
                                 cp     th, xh
                                 brne   lx
                                 cp     tl, xl
                             lx:
```

otherwise if operand1 is not a defined register name then *reg* - *value* comparing operation will be performed:

```
cp     ty, 0x44332211     →      cpi    r29, 0x44
                                 brne   lx
                                 cpi    r28, 0x33
                                 brne   lx
                                 cpi    r25, 0x22
                                 brne   lx
                                 cpi    r24, 0x11
                            lx:
```

note that expression isn't allowed in operand1 (avrasm limitation)

see also: **sub**, **cpi**

3.45. CPI

```
mnemonic:     cpi
direction:    high-low
operand0:     register(0)
operand1:     immediate value (expression)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
cpi tl, val	1	1	2	cpi tl, val	--H- VNZC
cpi x, val	2	3	6	cpi xh, BYTE1(val) brne lx cpi xl, BYTE0(val) lx:	--H- VNZC

cpi r, 0x1234	2	4 5	10	ldi r19, 0x34
				cp rl, r19
				brne lx
				ldi r19, 0x12
				cp rh, r19
					--H- VNZC

compare register and immediate value

note that operand1 is allows to be an expression:

```
cpi    tl, 0x2A + 0x80     →      cpi    tl, 0xAA
```

see also: **cp**, **subi**

3.43. **LDI**

```
mnemonic:     sbci
direction:    low-high
operand0:     register(0)
operand1:     immediate value
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
sbci tl, val	1	1	2	sbci tl, val	--HS VNZC
sbci tz, val	4	4	8	sbci tl, BYTE0(val) sbci th, BYTE1(val) sbci zl, BYTE2(val) sbci zh, BYTE3(val)	--HS VNZC

subtract **C**-flag in SREG and immediate value from register0 and store result back in register0.

'**sbci**' allows expression in operand1.

see also: **subi**

3.49. MUL

```
mnemonic:     mul
direction:    -
operand0:     register(0)
operand1:     register(1)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
mul tl, al	1	2	2	mul tl, al	---- --ZC

multiply unsigned

multiregisters isn't supported.

register0 and register1 should be in range: r0 – r31.

see also: **fmul**

3.50. MULS

```
mnemonic:     muls
direction:    -
operand0:     register(0)
operand1:     register(1)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
muls tl, al	1	2	2	muls tl, al	---- --ZC

multiply signed

multiregisters isn't supported.

register0 and register1 should be in range: r0 – r31.

see also: **fmuls**

3.51. MULSU

```
mnemonic:     mulsu
direction:    -
operand0:     register(0)
operand1:     register(1)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
mulsu al, bl	1	2	2	mulsu al, bl	---- --ZC

multiply signed with unsigned

multiregisters isn't supported.

register0 and register1 should be in range: r0 - r23

see also: **fmulsu**

3.52. FMUL

```
mnemonic:     fmul
direction:    -
operand0:     register(0)
operand1:     register(1)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
fmul al, bl	1	2	2	fmul al, bl	---- --ZC

fractional multiply unsigned

multiregisters isn't supported.

register0 and register1 should be in range: r0 - r23

see also: **mul**

3.53. FMULS

```
mnemonic:     fmuls
direction:    -
operand0:     register(0)
operand1:     register(1)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
fmuls al, bl	1	2	2	fmuls al, bl	---- --ZC

fractional multiply signed

multiregisters isn't supported.

register0 and register1 should be in range: r0 - r23

see also: **muls**

3.54. FMULSU

```
mnemonic:     fmulsu
direction:    -
operand0:     register(0)
operand1:     register(1)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
fmulsu al, bl	1	2	2	fmulsu al, bl	---- --ZC

fractional multiply signed with unsigned

multiregisters isn't supported.

register0 and register1 should be in range: r0 - r23

see also: **rol**

4. PSEUDOINSTRUCTIONS

псевдоинструкции являются отдельной темой в данной имплементации.

псевдоинструкции призваны «покрыть пробелы» в инструкшен сэте процессора. Для реализации этого было принято решение об использовании двух физических регистров процессора в особых целях:

zero - регистр r9

всегда содержит значение 0

использует: ? мнемоник

изменение содержимого регистра **zero** не рекоммендовано. в любом случае оно может производится только при выключенных прерываниях или при условии не-использования обработчиками прерываний псевдоинструкций использующих данный регистр или-же его сохранения\восстановления на входе\выходе обработчика прерываний.

использование регистра **zero** несколько понижает общую устойчивость кода в работе т. к. изменение его значения на не равное 0 — приведёт к практически гарантированному сбою во всех точках кода использующих данный регистр (неправильные значения, неверная арифметика и др.)

достаточно эффективен при операциях загрузки значений, ALU и некоторых других (вообщем-то нередких) случаях.

блокирующие флаги:

PSEUDOFF, **PSZEROFF**

tmp - регистр r19

временный регистр

используется в различных целях в том числе и для описания псевдоинструкций типа: регистр-значение, порт-значение и других

использует: ? мнемоник

использование регистра **tmp** существенно расширяет инстракшен сэт для пользователя однако его использование содержит в себе и угрозу: использование этого регистра возможно только либо при выключенных прерываниях, либо при условии не-использования обработчиками прерываний псевдоинструкций использующих данный регистр или-же его сохранения\восстановления на входе\выходе обработчика прерываний.

эффективен во многих случаях для многих типов инструкций.

блокирующие флаги:

PSEUDOFF, **PSTMPOFF**

макрос псевдоинструкции имеет тоже имя что и у инструкции его использующей с дополнительным символом подчёркивания в конце:

subi_ addi_

макрос псевдоинструкции работает только с конкретным физическим регистром процессора и с конкретным значением.

16-ти (и более) битные инструкции имеют специальный суффикс в имени в зависимости от разрядности операции:

w	2	outiw
r	3	outir
d	4	outid

В основном работа псевдоинструкций сводится к предзагрузке некоторого значения в регистр tmp и затем использования этого регистра вместо immediate value в паре с основным

регистром инструкции. При этом определённые псевдоинструкции «умеют» производить операцию с конкретным значением каким-то иным особым способом. Так например все псевдоиснтрукции «умеют» произвоить свою операцию со значением 0 следующим образом:

```
addi   r0, 0              →      add    r0, zero
```

в то время как подавляющее число псевдоиснтрукций производят свою операцию со значением как:

```
addi   r0, 10             →      ldi    tmp, 10
                                 add    r0, tmp
```

смотрите описание каждой псевдоинструкции отдельно.

note that using pseudoinstructions requires extra macro stack level.

4.1. LDI_

```
pseudo:       ldi_
mnemonics:    ldi / mov
operand0:     register(0)
operand1:     immediate value
```

COMPLEX INSTRUC	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
ldi_ r1, 0	1	2	mov r1, r9	---- ----
ldi_ r1, val	2	4	ldi r19, val mov r1, r19	---- ----
ldi_ r, val				

loads register by immediate value.

block flags:

PSEUDOFF

4.2. CPI_

```
pseudo:       cpi_
mnemonics:    cp / ldi
operand0:     register(0)
operand1:     immediate value
```

COMPLEX INSTRUC	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
cpi_ r1, 0	1	2	cp r1, r9	---- ----
cpi_ r1, val	2	4	ldi r19, val cp r1, r19	---- ----

compare register with immediate value.

4.3. **ADDI_**

```
pseudo:       addi_
mnemonics:    add / sec / adc / ldi
operand0:     register(0)
operand1:     immediate value
```

COMPLEX INSTRUC	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
addi tl, 0	1	2	add tl, r9	---- ----
addi tl, 1	2	4	sec adc tl, r9	---- ----
addi tl, 2	2	4	ldi r19,2 add tl, r19	

add immediate value to register

4.4. **ADCI_**

```
pseudo:       adci_
mnemonics:    adc / ldi
operand0:     register(0)
operand1:     immediate value
```

COMPLEX INSTRUC	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
adci_ rl, 0	1	2	adc rl, r9	---- ----
adci_ rl, val	1	2	ldi r19, val adc rl, r19	---- ----

add immeduate value and C-flag to register

4.5. **SUBI_**

```
pseudo:       subi_
mnemonics:    sub / sec / sbc / ldi
operand0:     register(0)
operand1:     immediate value
```

COMPLEX INSTRUC	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
subi_ r1, 0	1	2	sub r1, r9	---- ----
subi_ r1, 1	1	2	sec sbc r1, r9	---- ----
subi_ r1, val	2	4	ldi r19, val sub r1, r19	

subtract immediate value from register

4.6. **SBCI_**

```
pseudo:       sbci_
mnemonics:    sbc / ldi
operand0:     register(0)
operand1:     immediate value
```

COMPLEX INSTRUC	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
sbci_ r1, 0	1	2	sbc r1, r9	---- ----
sbci_ t1, val	2	4	ldi r19, val sbc r1, r19	---- ----

subtract immediate value and C-flag from register

PSEUDOFF

PSZEROFF
PSTMPOFF

PSAND1FF
PSOR0FF
PSXOR0FF

4.7. **ANDI_**

```
pseudo:       andi_
mnemonics:    and / ldi
operand0:     register(0)
operand1:     immediate value
```

COMPLEX INSTRUC	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
andi r0, 0	1	2	and r0, zero	---- ----
andi r0, 10	2	4	ldi tmp, 10 and r0, tmp	---- ----

perform register logic AND operation with immediate value

the **andi_** pseudoinstruction skips its instruction in case if operand1 value is equals to 0xFF and

control flag **PSAND1FF** is set (default)

see also: **andi**

4.8. ORI_

```
pseudo:       ori_
mnemonics:    or / ldi
operand0:     register(0)
operand1:     immediate value
```

COMPLEX INSTRUC	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
ori r0, 0	1	2	or r0, zero	---- ----
ori r0, 0xAA	2	4	ldi tmp, 0xAA or r0, tmp	---- ----

perform register logic OR operation with immediate value.

pseudoinstructon called in case if register r0-r15 is served

the **ori_** pseudoinstruction skips its instruction in case if operand1 value is equals to 0 and control flag **PSOR0FF** is set (default)

see also: **ori**

4.9. XORI_

```
pseudo:       eori_
mnemonics:    eor / com / ldi
operand0:     register(0)
operand1:     immediate value
```

COMPLEX INSTRUC	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
xori tl, 0	1	2	eor tl, r9	---- ----
xori tl, 0xFF	1	2	com tl	---- ----
xori tl, 0xAA	2	4	ldi tmp, 0xAA eor t, tmp	---- ----

perform register logic XOR operation with immediate value

the **xori_** pseudoinstruction skips its instruction in case if operand1 value is equals to 0 and control flag **PSXOR0FF** is set (default)

see also: **xor**

4.10. **MOVI_**

```
pseudo:       movi_
mnemonics:    mov / ldi
operand0:     register(0)
operand1:     immediate value
```

COMPLEX INSTRUC	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
movi r0, 0	1	2	mov r0, zero	---- ----
PSZEROFF	1	2	eor r0, r0	---- ----
movi r0, 0xAA	2	4	ldi tmp, 0xAA mov r0, tmp	---- ----

4.10. **OUTI_**

```
pseudo:       outi_
mnemonics:    out / sts/ ldi
operand0:     port number
operand1:     immediate value
```

COMPLEX INSTRUC	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
outi 0x3F, 0	1	2	out 0x3F, zero	---- ----
outi 0x60, 0	2	4	sts 0x0060, zero	---- ----
outi 0x3F, 0xAA	2	4	ldi tmp, 0xAA out 0x3F, tmp	---- ----
outi 0x60, 0xAA	3	6	ldi tmp, 0xAA sts 0x0060, tmp	---- ----

performs output 8-bit value to the specified port.

mnemonic (**out** or **sts**) is selected depending from the port number:

0x00 – 0x3F low I/o space by **out** command

0x60 – ... extended I/o space by **sts** command

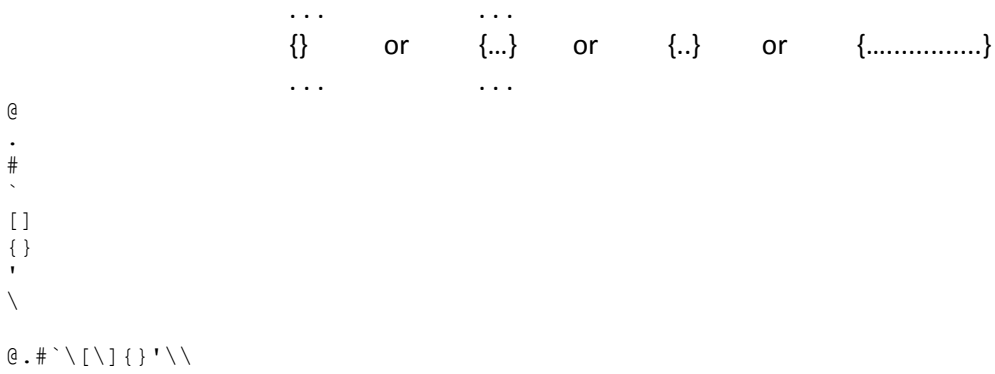
perform register logic XOR operation with immediate value

5. **CATCH ME!**

I believe that this tweak should works in any macro assembler that is supporting C-like single-line macros.

AVRASM2 tweaks:

- avrasm completely ignoring in source body expression like:
  - empty figure-parenthesis
    {}
  - single or more dot '.' or '@' characters seuqentially ignored
    ..@...@..@@@.@@...@
- both:
  { ..@@...@@@.@@ }



я думаю этот трюк прокатит в любом макроассемблере с поддержкой C-like single-line macors

6. **AFTERWORDS**

подобная реализация инстракшен сэта может быть актуальна для процессоров с большим набором регистров.

изучение и систематизация знаний об avr instruction set

инструкция обладает следующими свойствами:

- подача (мнемокода): одиночная/двойная

при генерации кода подача мнемоник может быть:

- одиночной - когда для всех регистров идёт одна и та-же мнемоника:

```
push   tz                  →      push   zh
                                  push   zl
                                  push   th
                                  push   tl
```

- двойной - когда для первого регистра идёт одна мнемоника, а для всех остальных — вторая:

```
lsl    tz                  →      lsl    tl
                                  rol    th
                                  rol    zl
                                  rol    zh
```

- направление(порядок отработки регистров по маске): прямое, обратное или n/a

инструкция отрабатывает регистры по маске в порядке:

- прямым направлением называется отработка регистров по маске от младших регистров к старшим (low-high):

```
add    tz, xy              →      add    tl, xl
                                  adc    th, xh
                                  adc    zl, yl
                                  adc    zh, yh
```

- обратным направлением называется последовательность отработки регистров по маске от старших к младшим (high-low):

```
lsr    tz                  →      lsr    zh
                                  ror    zl
                                  ror    th
                                  ror    tl
```

- n/a - когда свойство «направления» не актуально для инструкции:

```
bst    reg, bit          →      bst    …, bit

and    reg, ...
```

MOV:

самым кошмаром стала инструкция mov: требовалось пренести все регистры инструкциями mov, используя movw где-если это возможно. это самая объёмная инструция из всех.

mov отрабатывается двумя парами регистров. каждая пара имеет 3 состояния (нулевые не в счёт):

01 - только младший регистр задействован
10 - только старший регистр задействован
11 - старший и младший регистры задействованы

из девяти (3 * 3) состояний проблему прдставляют четыре :

3:1 и 3:2

1:3 и 2:3

здесь сразу стоит отметить проявление одного из основных свойств имеющихся у инструкций: направление перебора регистров.

в случае с 'push' это направление от старших регистров к младшим (high-low)

это связано с тем, что операция 'push' — является операцией записи, а следовательно выполняется в high-low последовательности.

в случае с 'pop' это направление от младших регистров к старшим (low-high)

это связано с тем, что операция 'pop' — является операцией чтения, а следовательно выполняется в low-high последовательности.

свойство направления определяется самой инструкцией — сутью её операции и того как она реализована

псевдоинструкция это макрос выполняющий ту или иную операцию которой нет в оригиальном интракшен сэте процессора. для AVR это как правило:

- новые инструкции: команды сложения и xora регистра и операнда

```
addi_  r0, val

eori_  r24, val
```

- дополняющие: команды с операндом и регистром из числа младших 16ти регистров процессора:

```
subi_  r0, val
```

псевдоинструкции призваны закрыть бреши в AVR инстракшен сэте касаемо того, что не со всеми регистрами можно всё делать. например инструкция '**subi**' — одна из нескольких инструкций способных работать только с верхними 16тью регистрами процессора. В случае если такой инструкции требуется доступ к младшим 16ти регистрам процессора — используется псевдоинструкция — в данном случае: '**subi_**'.

дополняющим типом псевдоинструкций пользуется например 'subi' использующий при доступе к старшим регистрам инстракшен сэт процессора, а при доступе к младшим 16ти регистрам использует одноименную псевдоинструкцию 'subi_'.

данный тип псевдоинструкций используется для частичного покрытия инстракшен сэта процессора — в качестве дополнения к инструкции процессора. этим они отличаются от полноценных псевдоинструкций —

псевдоинструкции: addi_ и adci_ являются полноценными псевдоинструкциями описывающими совершенно новую в инстракшен сэте операцию:

так например addi_ является проивоположностью к имеющейся в инстракшен сэте процессора команде subi, а adci соответственно к sbci

good questions about avr instruction set

SEI-CLI

we know that '**sei**' enables interrupt after next instruction. in other words – if interrupts are disabled and we're enabling its by the '**sei**' command then interrupts are disabled:

```
sei
cli
```

what the state of interrupts will be after code above is executed

7. **ABOUT** _____

AVRASM: проблемы

макропараметр не может быть соединён с цифрой справа

avrasm2 как известно поддерживает множественные (> 10) параметры, из-за этого их вызов

… @00 ← recognized as parameter 0

номер макропараметра идущий после символа '@' может занимать от одного до двух символов:

@0 .. @9 ← макропараметры номер 0 — 9
@10 .. @99 ← макропараметры номер 10 — 99

@00 ← ошибка

@01 ← 0@1

@12 ← 1@12

3.7. **TST**

```
mnemonic:     or
direction:    low-high
operand:      register(0)
```

COMPLEX INSTRUC	WIDE	CLK	LEN	CODE (CPU INSTRUC)	ITHS VNZC
tst tl	1	1	2	andi tl, ~ 1 << 3	---- ----
tst tz	4	1	2	andi zl, ~ 1 << 7	---- ----
tst rl	1	2	4	ldi tmp, ~ 1 << 5 and rl, tmp	

clear bit in register. this instruction is not exist in AVR cpu instruction set. Indeed that the '**tst**' is the form of the '**or**' instruction:

```
tst    reg                →      or     reg, reg
```

see also: **or**, **sbr**

2023.2.? 3730 +1000 == 4730 + 3 == 4760

2023.2.16? 4760 +740 == 5500 + 3 == 5530

2023.2.? 5530 + ==

5730 + 600 +4

2023.2.28 6370

2023.3.9 6370 + 1550 = 7920