Femtium Reference Manual

Femtium Engineering Laboratories

2020-v1 Edition

1 Femtium Architecture

DRAFT VERSION - FOR INTERNAL REVIEW ONLY

Not for distrubution outside Femtium Engineering Laboratories!

(((here goes introduction text after approval from technical marketing)))

In most explanatory text in this manual, the following color coding is used:

- `register`
- `labels`
- `numbers`

2 Registers

The Femtium architecture has 64 registers, each 32 bits wide. Exactly 2 registers have a hardware-mandated special purpose.

The first register, `r0` (also known as `zz`), is hard-wired to zero. Any write to this register is discarded, and all reads return `0`.

The last register, `r63` (also known as `pc`), is the *program counter*. Writing to this value will cause the processor to execute the instruction at that address next. When reading `pc`, the value returned will be the instruction *after* the currently executing instruction. For example, if the instruction at adress copies `pc` to `t0`, the value of `t0` will be after the instruction finishes.

Except for `r0` and `r63`, no registers are considered special by the Femtium *hardware*. When writing programs directly for the hardware, all 62 general-purpose registers can be used at will.

However, only the hardware itself allows this degree of flexibility. When using the standard compiler, Fenix kernel, c library, or other existing software, the standard register layout must be adhered to. In the table below, all 64 registers are described, including the intended use designation for each register.

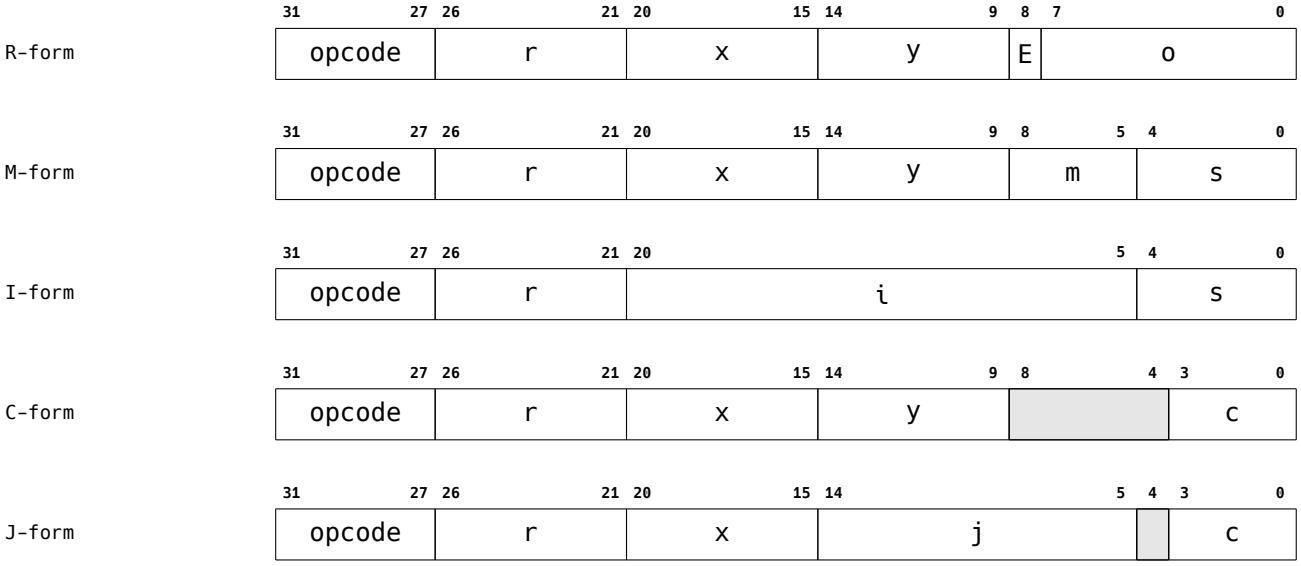
name	index	description
zz	0	zero register
v0	1	(legacy) result 0
v1	2	(legacy) result 1
a0	3	argument / result 0
a1	4	argument / result 1
a2	5	argument 2
a3	6	argument 3
a4	7	argument 4
a5	8	argument 5
a6	9	argument 6
s0	10	callee-save 0
s1	11	callee-save 1
s2	12	callee-save 2
s3	13	callee-save 3
s4	14	callee-save 4
s5	15	callee-save 5
s6	16	callee-save 6
s7	17	callee-save 7
s8	18	callee-save 8
s9	19	callee-save 9
s10	20	callee-save 10
s11	21	callee-save 11
s12	22	callee-save 12
s13	23	callee-save 13
s14	24	callee-save 14
s15	25	callee-save 15
s16	26	callee-save 16
s17	27	callee-save 17
s18	28	callee-save 18
s19	29	callee-save 19

name	index	description
t0	30	caller-save 0
t1	31	caller-save 1
t2	32	caller-save 2
t3	33	caller-save 3
t4	34	caller-save 4
t5	35	caller-save 5
t6	36	caller-save 6
t7	37	caller-save 7
t8	38	caller-save 8
t9	39	caller-save 9
t10	40	caller-save 10
t11	41	caller-save 11
t12	42	caller-save 12
t13	43	caller-save 13
t14	44	caller-save 14
t15	45	caller-save 15
t16	46	caller-save 16
t17	47	caller-save 17
t18	48	caller-save 18
t19	49	caller-save 19
t20	50	caller-save 20
t21	51	caller-save 21
x	52	general-purpose 0
y	53	general-purpose 1
z	54	general-purpose 2
k0	55	kernel 0
k1	56	kernel 1
at0	57	assembler-temporary 0
at1	58	assembler-temporary 1
ra	59	return address
fp	60	frame pointer
gp	61	global pointer
sp	62	stack pointer
pc	63	program counter

3 Instruction encodings

	31 27	26 21	20 15	14 9	8	7 0
LDB	0 0 0 0 0	r	x	y	E	o
LDH	0 0 0 0 1	r	x	y	E	o
LDW	0 0 0 1 0	r	x	y		o
-	0 0 0 1 1	invalid opcode				
STB	0 0 1 0 0	r	x	y		o
STH	0 0 1 0 1	r	x	y		o
STW	0 0 1 1 0	r	x	y		o
-	0 0 1 1 1	invalid opcode				
ADD	0 1 0 0 0	r	x	y		o
MUL	0 1 0 0 1	r	x	y		o
DIV	0 1 0 1 0	r	x	y		o
NOR	0 1 0 1 1	r	x	y		o
MASK	0 1 1 0 0	r	x	y	m	s
-	0 1 1 0 1	invalid opcode				
-	0 1 1 1 0	invalid opcode				
-	0 1 1 1 1	invalid opcode				
MOVI	1 0 0 0 0	r	i			s
ADDI	1 0 0 0 1	r	i			s
CMOV	1 0 0 1 0	r	x	y		c
CMP	1 0 0 1 1	r	x	y		c
-	1 0 1 0 0	invalid opcode				
-	1 0 1 0 1	invalid opcode				
-	1 0 1 1 0	invalid opcode				
CJMP	1 0 1 1 1	r	x	j		c
IN	1 1 0 0 0	r	x	y		o
OUT	1 1 0 0 1	r	x	y		o
DSKR	1 1 0 1 0	r	x	y		o
DSKW	1 1 0 1 1	r	x	y		o
-	1 1 1 0 0	invalid opcode				
SYS	1 1 1 0 1					
IRET	1 1 1 1 0					
HALT	1 1 1 1 1	r				

4 Instruction formats



5 Reference constants

```
enum cmpmode:
    CMP_NZ  = 0b0000 /* test if x is Non-Zero */
    CMP_LEU = 0b0001 /* test if x is Less or Equal to y (unsigned) */
    CMP_LTU = 0b0010 /* test if x is Less Than y (unsigned) */
    CMP_EQ  = 0b0011 /* test if x is Equal to y */
    CMP_EZ  = 0b0100 /* test if x is Equal to Zero (the opposite of Non-Zero) */
    CMP_GTU = 0b0101 /* test if x is Greater Than y (unsigned) */
    CMP_GEU = 0b0110 /* test if x is Greater or Equal to y (unsigned) */
    CMP_NE  = 0b0111 /* test if x is Not Equal to y */
    CMP_X8  = 0b1000 /* RESERVED: do not use */
    CMP_LE  = 0b1001 /* test if x is Less or Equal to y (signed) */
    CMP_LT  = 0b1010 /* test if x is Less Than to y (signed) */
    CMP_X11 = 0b1011 /* RESERVED: do not use */
    CMP_X12 = 0b1100 /* RESERVED: do not use */
    CMP_GT  = 0b1101 /* test if x is Greater Than y (signed) */
    CMP_GE  = 0b1110 /* test if x is Greater or Equal to y (signed) */
    CMP_X15 = 0b1111 /* RESERVED: do not use */

enum shiftmode:
    SM_SHL  =   0b00 /* Shift left */
    SM_SHR  =   0b01 /* Shift right unsigned (logical shift) */
    SM_SAR  =   0b10 /* Shift right signed (arithmetic shift) */
    SM_X3   =   0b11 /* RESERVED: do not use */

enum blendmode:
    BM_MOV  =   0b00
    BM_AND  =   0b01
    BM_OR   =   0b10
    BM_XOR  =   0b11
```

```
enum opcode:
    LDB   = 0x00
    LDH   = 0x01
    LDW   = 0x02
    _LDE  = 0x03 /* RESERVED: do not use */
    STB   = 0x04
    STH   = 0x05
    STW   = 0x06
    _STE  = 0x07 /* RESERVED: do not use */
    ADD   = 0x08
    MUL   = 0x09
    DIV   = 0x0a
    NOR   = 0x0b
    MASK  = 0x0c
    _ALU5 = 0x0d /* RESERVED: do not use */
    _ALU6 = 0x0e /* RESERVED: do not use */
    _ALU7 = 0x0f /* RESERVED: do not use */
    MOVI  = 0x10
    ADDI  = 0x11
    CMOV  = 0x12
    CMP   = 0x13
    _MOV4 = 0x14 /* RESERVED: do not use */
    _MOV5 = 0x15 /* RESERVED: do not use */
    _MOV6 = 0x16 /* RESERVED: do not use */
    CJMP  = 0x17
    IN    = 0x18
    OUT   = 0x19
    DSKR  = 0x1a /* Obsolete. Reserved for backwards compatability */
    DSKW  = 0x1b /* Obsolete. Reserved for backwards compatability */
    _SYS0 = 0x1c /* RESERVED: do not use */
    SYS   = 0x1d
    IRET  = 0x1e
    HALT  = 0x1f
```

```
enum regname:
    REG_zz   =  0
    REG_v0   =  1
    REG_v1   =  2
    REG_a0   =  3
    REG_a1   =  4
    REG_a2   =  5
    REG_a3   =  6
    REG_a4   =  7
    REG_a5   =  8
    REG_a6   =  9
    REG_s0   = 10
    REG_s1   = 11
    REG_s2   = 12
    REG_s3   = 13
    REG_s4   = 14
    REG_s5   = 15
    REG_s6   = 16
    REG_s7   = 17
    REG_s8   = 18
    REG_s9   = 19
    REG_s10  = 20
    REG_s11  = 21
    REG_s12  = 22
    REG_s13  = 23
    REG_s14  = 24
    REG_s15  = 25
    REG_s16  = 26
    REG_s17  = 27
    REG_s18  = 28
    REG_s19  = 29
    REG_t0   = 30
    REG_t1   = 31
    REG_t2   = 32
    REG_t3   = 33
    REG_t4   = 34
    REG_t5   = 35
    REG_t6   = 36
    REG_t7   = 37
    REG_t8   = 38
    REG_t9   = 39
    REG_t10  = 40
    REG_t11  = 41
    REG_t12  = 42
    REG_t13  = 43
    REG_t14  = 44
    REG_t15  = 45
    REG_t16  = 46
    REG_t17  = 47
    REG_t18  = 48
    REG_t19  = 49
    REG_t20  = 50
    REG_t21  = 51
    REG_x    = 52
    REG_y    = 53
    REG_z    = 54
    REG_k0   = 55
    REG_k1   = 56
    REG_at0  = 57
    REG_at1  = 58
    REG_ra   = 59
    REG_fp   = 60
    REG_gp   = 61
    REG_sp   = 62
    REG_pc   = 63
```

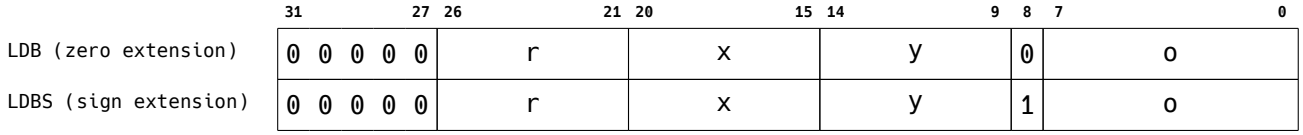
6 Example code

```
bool evaluate_condition(enum cmpmode cc, u32 a, u32 b)
{
    switch (cc) {
        case CMP_NZ:  return       0 !=       b;
        case CMP_LEU: return       a <=       b;
        case CMP_LTU: return       a <        b;
        case CMP_EQ:  return       a ==       b;
        case CMP_EZ:  return       0 ==       b;
        case CMP_GTU: return       a >        b;
        case CMP_GEU: return       a >=       b;
        case CMP_NE:  return       a !=       b;
        case CMP_LE:  return (i32) a <= (i32) b;
        case CMP_LT:  return (i32) a <  (i32) b;
        case CMP_GT:  return (i32) a >  (i32) b;
        case CMP_GE:  return (i32) a >= (i32) b;

        default: /* invalid instruction */;
    }
}
```

```
u32 evaluate_blend(u32 r, u32 x, enum blendmode bm, enum shiftmode sm, int shift)
{
    u32 maskval;
    switch (sm) {
        case SM_SHL: maskval =       x << shift; break;
        case SM_SHR: maskval = (u32) x >> shift; break;
        case SM_SAR: maskval = (i32) x >> shift; break;

        default: /* invalid instruction */
    }

    switch (bm) {
        case BM_MOV: return     maskval;
        case BM_AND: return r & maskval;
        case BM_OR:  return r | maskval;
        case BM_XOR: return r ^ maskval;

        default: /* invalid instruction */
    }
}
```

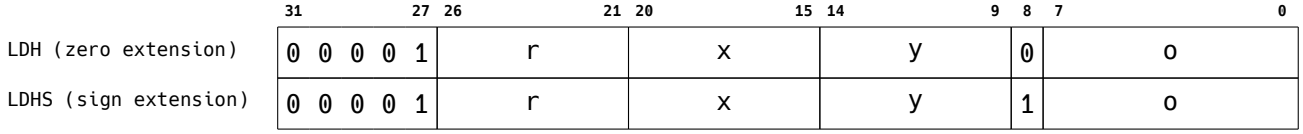
7 Hardware Instructions

7.1 LDB Load 1 byte



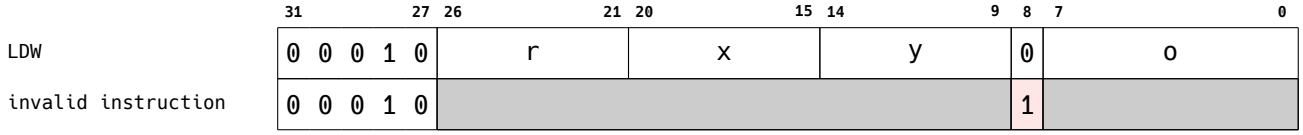

```
if E == 0:
    regs[r] := (u8) mem[regs[x] + regs[y] + o]
else:
    regs[r] := (i8) mem[regs[x] + regs[y] + o]
```

7.2 LDH Load 2 bytes



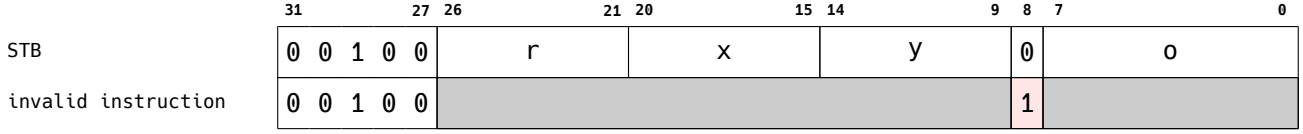

```
if E == 0:
    regs[r] := (u16) mem[regs[x] + regs[y] + o]
else:
    regs[r] := (i16) mem[regs[x] + regs[y] + o]
```

7.3 LDW Load 4 bytes



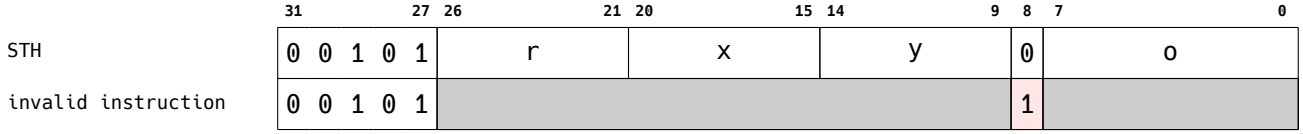

```
regs[r] := (u32) mem[regs[x] + regs[y] + o]
```

7.4 STB Store 1 byte



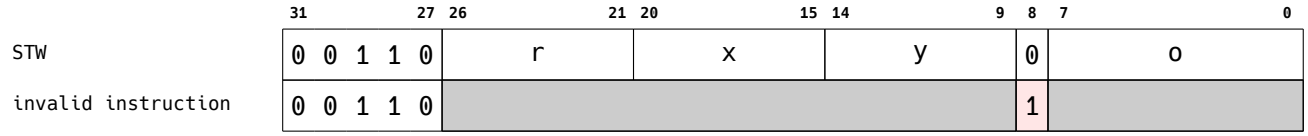

```
mem[regs[x] + regs[y] + o] := (u8) regs[r]
```

7.5 STH Store 2 bytes



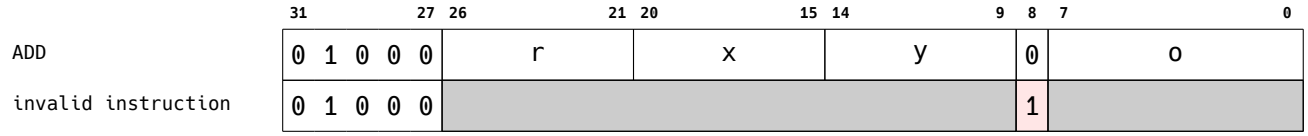

```
mem[regs[x] + regs[y] + o] := (u16) regs[r]
```

7.6 STW Store 4 bytes



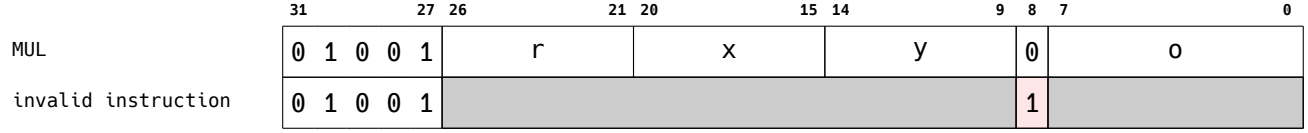

```
mem[regs[x] + regs[y] + o] := (u32) regs[r]
```

7.7 ADD Add



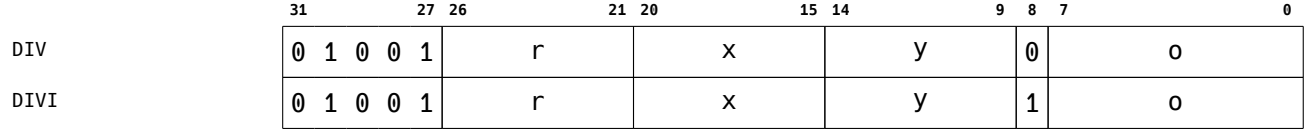

```
regs[r] := regs[x] + regs[y] + o
```

7.8 MUL Multiply



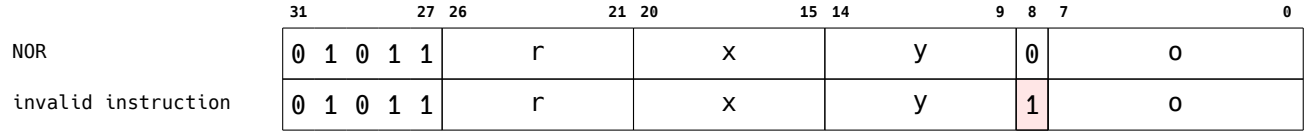

```
regs[r] := regs[x] * (regs[y] + o)
```

7.9 DIV Divide




```
if E == 0:
  regs[r] := (u32) regs[x] / (u32) (regs[y] + o)
else:
  regs[r] := (i32) regs[x] / (i32) (regs[y] + o)
```

7.10 NOR Negated Logical-OR




```
regs[r] := ~(regs[x] | regs[y] | o)
```

7.11 MASK Masked bit manipulation

	31 27	26 21	20 15	14 9	8 5	4 0
MASK	0 1 1 0 0	r	x	y	m	s
SHL	0 1 1 0 0	r	x	y	0 0 0 0	s
SHR	0 1 1 0 0	r	x	y	0 0 0 1	s
SAR	0 1 1 0 0	r	x	y	0 0 1 0	s
invalid instruction	0 1 1 0 0				0 0 1 1	
SHLO	0 1 1 0 0	r	x	y	0 1 0 0	s
SHRO	0 1 1 0 0	r	x	y	0 1 0 1	s
SARO	0 1 1 0 0	r	x	y	0 1 1 0	s
invalid instruction	0 1 1 0 0				0 1 1 1	
SHLA	0 1 1 0 0	r	x	y	1 0 0 0	s
SHRA	0 1 1 0 0	r	x	y	1 0 0 1	s
SARA	0 1 1 0 0	r	x	y	1 0 1 0	s
invalid instruction	0 1 1 0 0				1 0 1 1	
SHLX	0 1 1 0 0	r	x	y	1 1 0 0	s
SHRX	0 1 1 0 0	r	x	y	1 1 0 1	s
SARX	0 1 1 0 0	r	x	y	1 1 1 0	s
invalid instruction	0 1 1 0 0				1 1 1 1	

```
/* Depending on the "m" field, one of the following: */

/* SHL  */ regs[r] :=                  regs[x] << (regs[y] + s)
/* SHR  */ regs[r] :=            (u32) regs[x] >> (regs[y] + s)
/* SAR  */ regs[r] :=            (i32) regs[x] >> (regs[y] + s)

/* SHLO */ regs[r] := regs[r] | (      regs[x] << (regs[y] + s))
/* SHRO */ regs[r] := regs[r] | ((u32) regs[x] >> (regs[y] + s))
/* SARO */ regs[r] := regs[r] | ((i32) regs[x] >> (regs[y] + s))

/* SHLA */ regs[r] := regs[r] & (      regs[x] << (regs[y] + s))
/* SHRA */ regs[r] := regs[r] & ((u32) regs[x] >> (regs[y] + s))
/* SARA */ regs[r] := regs[r] & ((i32) regs[x] >> (regs[y] + s))

/* SHLX */ regs[r] := regs[r] ^ (      regs[x] << (regs[y] + s))
/* SHRX */ regs[r] := regs[r] ^ ((u32) regs[x] >> (regs[y] + s))
/* SARX */ regs[r] := regs[r] ^ ((i32) regs[x] >> (regs[y] + s))
```

7.12 MOVI Assign immediate value to register

	31 27	26 21	20 5	4 0
MOVI	1 0 0 0 0	r	i	s

```
regs[r] := ((u16) i) << s
```

7.13 ADDI Add immediate value to register

	31 27	26 21	20 5	4 0
ADDI	1 0 0 0 1	r	i	s

```
regs[r] := regs[r] + (((u16) i) << s)
```

7.14 CMOV Conditional Move

	31 27	26 21	20 15	14 9		3 2 1 0
CMOV	1 0 0 1 0	r	x	y		c
CMNZ	1 0 0 1 0	r	x			0 0 0 0
CMLEU	1 0 0 1 0	r	x	y		0 0 0 1
CMLTU	1 0 0 1 0	r	x	y		0 0 1 0
CMEQ	1 0 0 1 0	r	x	y		0 0 1 1
CMEZ	1 0 0 1 0	r	x			0 1 0 0
CMGTU	1 0 0 1 0	r	x	y		0 1 0 1
CMGEU	1 0 0 1 0	r	x	y		0 1 1 0
CMNE	1 0 0 1 0	r	x	y		0 1 1 1
invalid instruction	1 0 0 1 0					1 0 0 0
CMLE	1 0 0 1 0	r	x	y		1 0 0 1
CMLT	1 0 0 1 0	r	x	y		1 0 1 0
invalid instruction	1 0 0 1 0					1 0 1 1
invalid instruction	1 0 0 1 0					1 1 0 0
CMGT	1 0 0 1 0	r	x	y		1 1 0 1
CMGE	1 0 0 1 0	r	x	y		1 1 1 0
invalid instruction	1 0 0 1 0					1 1 1 1

```
bool cond := evaluate_condition(c, regs[x], regs[y])

if (cond)
    regs[r] := regs[x]
```

7.15 `CMP` Compare values

	31 27	26 21	20 15	14 9		3 2 1 0
CMP	1 0 0 1 1	r	x	y		c
SNZ	1 0 0 1 1	r	x			0 0 0 0
SLEU	1 0 0 1 1	r	x	y		0 0 0 1
SLTU	1 0 0 1 1	r	x	y		0 0 1 0
SEQ	1 0 0 1 1	r	x	y		0 0 1 1
SEZ	1 0 0 1 1	r	x			0 1 0 0
SGTU	1 0 0 1 1	r	x	y		0 1 0 1
SGEU	1 0 0 1 1	r	x	y		0 1 1 0
SNE	1 0 0 1 1	r	x	y		0 1 1 1
invalid instruction	1 0 0 1 1					1 0 0 0
SLE	1 0 0 1 1	r	x	y		1 0 0 1
SLT	1 0 0 1 1	r	x	y		1 0 1 0
invalid instruction	1 0 0 1 1					1 0 1 1
invalid instruction	1 0 0 1 1					1 1 0 0
SGT	1 0 0 1 1	r	x	y		1 1 0 1
SGE	1 0 0 1 1	r	x	y		1 1 1 0
invalid instruction	1 0 0 1 1					1 1 1 1

```
bool cond := evaluate_condition(c, regs[x], regs[y])

if cond:
    regs[r] := 1
else:
    regs[r] := 0
```

7.16 `CJMP` Conditional Jump

	31 27	26 21	20 15	14 5	4	3 2 1 0
CJMP	1 0 1 1 1	r	x	j		c
BNZ	1 0 1 1 1	r	x	j		0 0 0 0
BLEU	1 0 1 1 1	r	x	j		0 0 0 1
BLTU	1 0 1 1 1	r	x	j		0 0 1 0
BEQ	1 0 1 1 1	r	x	j		0 0 1 1
BEZ	1 0 1 1 1	r	x	j		0 1 0 0
BGTU	1 0 1 1 1	r	x	j		0 1 0 1
BGEU	1 0 1 1 1	r	x	j		0 1 1 0
BNE	1 0 1 1 1	r	x	j		0 1 1 1
invalid instruction	1 0 1 1 1					1 0 0 0
BLE	1 0 1 1 1	r	x	j		1 0 0 1
BLT	1 0 1 1 1	r	x	j		1 0 1 0
invalid instruction	1 0 1 1 1					1 0 1 1
invalid instruction	1 0 1 1 1					1 1 0 0
BGT	1 0 1 1 1	r	x	j		1 1 0 1
BGE	1 0 1 1 1	r	x	j		1 1 1 0
invalid instruction	1 0 1 1 1					1 1 1 1

```
/* Note: CJMP uses (regs[r],regs[x]) while CMOV and CMP use (regs[x],regs[y]) */
bool cond := evaluate_condition(c, regs[r], regs[x])

/* The (signed) "j" field signifies how many instructions to jump */
if cond:
    regs[63] := regs[63] + (j * 4)
```

7.17 `IN` Read word from device

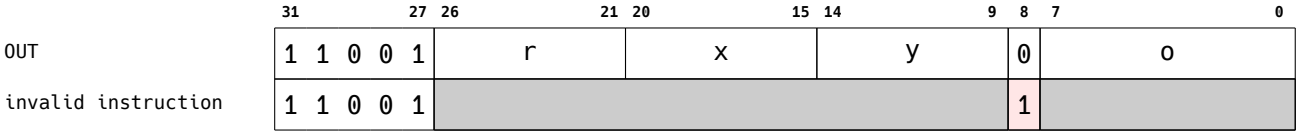
This instruction is only valid in privileged execution mode. For example, if you are running a general-purpose operating system (like Fenix), only the kernel can access this instruction.

	31 27	26 21	20 15	14 9	8	7 0
IN	1 1 0 0 0	r	x	y	0	o
invalid instruction	1 1 0 0 0				1	

```
var dev_id := regs[x]
var port_id := regs[y] + o
regs[r] := device[dev_id].port[port_id]
```

7.18 OUT Write word to device

This instruction is only valid in privileged execution mode. For example, if you are running a general-purpose operating system (like Fenix), only the kernel can access this instruction.




```
var dev_id := regs[x]
var port_id := regs[y] + o
device[dev_id].port[port_id] := regs[r]
```

7.19 DSKR Read sector from disk

This instruction is obsolete, and only exists for backwards compatibility with the U5 architecture. It must not be used in new programs.

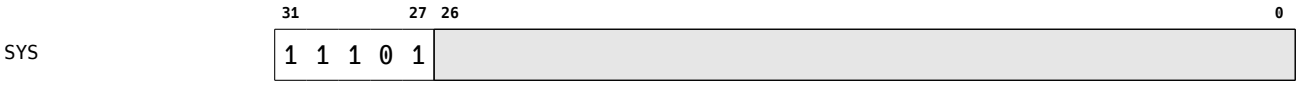
This instruction is only valid in privileged execution mode. For example, if you are running a general-purpose operating system (like Fenix), only the kernel can access this instruction.

7.20 DSKW Write sector to disk

This instruction is obsolete, and only exists for backwards compatibility with the U5 architecture. It must not be used in new programs.

This instruction is only valid in privileged execution mode. For example, if you are running a general-purpose operating system (like Fenix), only the kernel can access this instruction.

7.21 SYS Perform system call

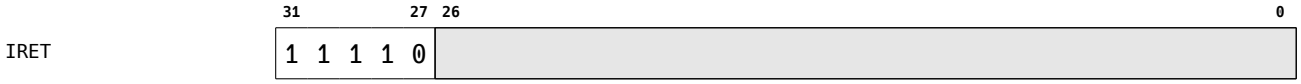


Perform a system call. Before calling SYS, prepare argument registers in accordance with the syscall calling convention. When invoking SYS, the syscall handler takes over execution.

```
saved_reg_62 := regs[62]
saved_reg_63 := regs[63]
regs[63] := address_of_system_call_handler
```

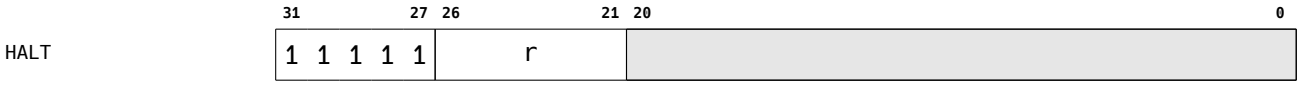
7.22 `IRET` Return from interrupt

This instruction is only valid in privileged execution mode. For example, if you are running a general-purpose operating system (like Fenix), only the kernel can access this instruction.




```
  regs[62] := saved_reg_62
  regs[63] := saved_reg_63
```

7.23 `HALT` Halt execution



Halts the CPU. The `r` register can be used to signify a return value. When running through an emulator, this register can be used to indicate a return value to the emulator. When running on physical hardware, the meaning of the `r` register depends on the debug facilities present in the CPU. If no such debug facilities are present, the register is ignored.

8 Pseudo-instructions

To improve the usability of Femtium, the assembler and compiler toolchains support a number of Pseudo-instructions. Unlike hardware instructions, these instructions do not have their own opcode or instruction encoding. They are aliases for hardware instructions, and are available for convenience.

8.1 MV Move value between registers

`MV A, B`

Moves the value from `B` into `A`. Expands to

`ADD A, B, r0, 0`

8.2 NOT Bitwise NOT

`NOT A, B`

Sets register `A` to the bit-wise negation of the value in register `B`.

Expands to

`NOR A, B, B, 0`

8.3 AND Bitwise AND

`AND A, B`

Sets register `A` to (`A` AND `B`). Expands to

`SHLA A, B, r0, 0`

8.4 OR Bitwise OR

`OR A, B`

Sets register `A` to (`A` OR `B`). Expands to

`SHLO A, B, r0, 0`

8.5 INC Increment register

`INC A`

Increments register `A` by `1`. Expands to

`ADD A, A, r0, 1`

8.6 DEC Decrement register

`INC A`

Decrements register `A` by `1`. Expands to

`ADD A, A, r0, -1`

8.7 `JMP` Unconditional jump

`JMP label`

Unconditionally jumps to `label`. Expands to

`BEQ r0, r0, label`

8.8 `RET` Return from function call

Usage:

`RET`

Return from function call. Expands to

`ADD pc, ra, r0, 0`