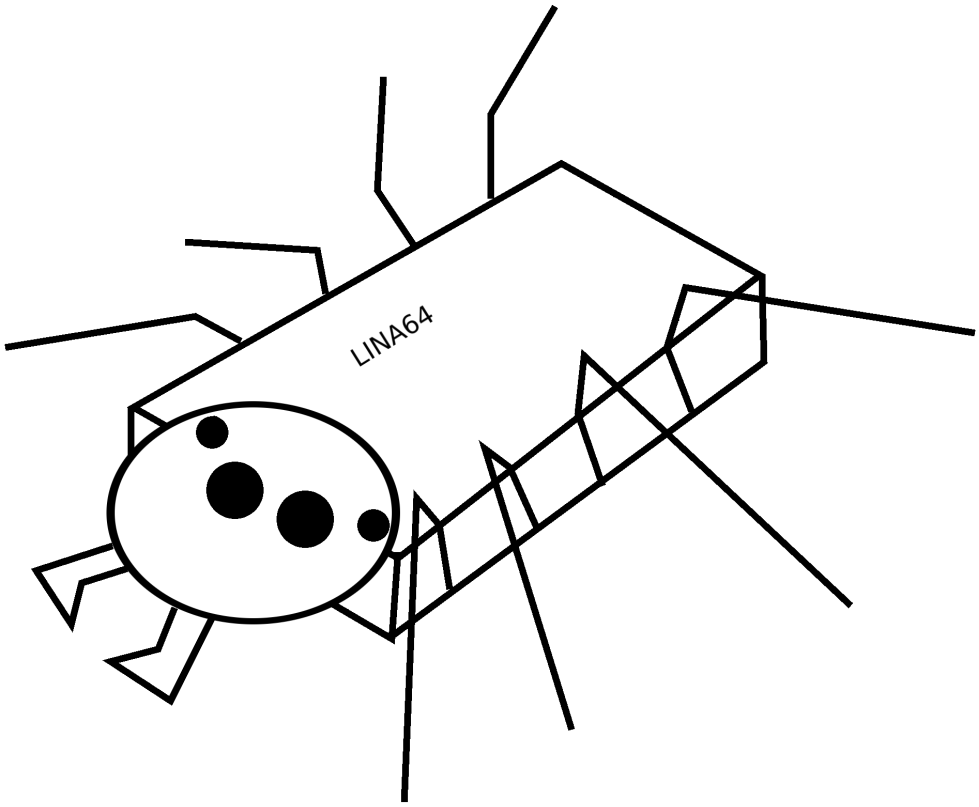
Instruction set architecture of Spiderchip64

Version 0.1

Aurélien Casteilla

14-05-2021



Contents

1 Register file

1.1 Regular registers

Table 1: User register set

R0	R1	R2	R3	R4	R5	R6	R7
R8	R9	R10	R11	R12	R13	R14	PC
R16	R17	R18	R19	R20	R21	R22	R23
R24	R25	R26	R27	R28	R29	R30	LR

Table 2: Interrupt register set

R0	R1	R2	R3	R4	R5	R6	R7
R8	R9	R10	R11	R12	R13	R14	PC
RLIi	R17i	R18i	R19i	R20i	R21i	R22i	R23i
R24i	R25i	R26i	R27i	R28i	R29i	R30i	LRi

R0 is hard-wired to zero : loading a value from it returns 0, storing a value in it does nothing.
RLIi : Last instruction executed before the interrupt
LR : link register
LRi : supervisor link register
PC : program counter, always point the next instruction to be executed
Registers R0 to R14 and PC are shared between the sets.
All registers are 64 bit-wide

1.2 Status register and conditional code register

The whole status register is directly accessible only in supervisor mode, with the instructions TRS and TSR. In user mode, the current condition codes can be accessed with the instructions TRC and TCR. The condition codes bits are bit 7 to bit 0. The condition codes bits can be used to perform conditional operations, such as branches. All interrupts move the current status register to the saved status register. And RTI moves back the saved status to the current status. RTI can be executed only in supervisor mode.

Table 3: Current status register and saved status register

	bit No.	name	description
unused	63-32	-	
saved status	31	R	interrupt register set selected
	30	I	interrupt request disabled
	29	D	map directly the memory
	28	W	wait interrupt
	27	P	privileged instructions allowed
	26	-	
	25	-	
	24	-	
	23	N	negative
	22	V	overflow
	21	-	
	20	H	higher than (H = C./Z)
	19	g	greater than or equal (g = /N./V + N.V)
	18	G	greater than (G = /N./V./Z + N.V./Z)
	17	Z	zero
	16	C	carry
current status	15	R	interrupt register set selected
	14	I	interrupt request disabled
	13	D	map directly the memory
	12	W	wait interrupt
	11	P	privileged instructions allowed
	10	-	
	9	-	
	8	-	
current status and conditional codes	7	N	negative
	6	V	overflow
	5	-	
	4	H	higher than (H = C./Z)
	3	g	greater than or equal (g = /N./V + N.V)
	2	G	greater than (G = /N./V./Z + N.V./Z)
	1	Z	zero
	0	C	carry

- R : if set, R14i to LRi are used instead of R14 to LR
- I : if set, interrupt requests are disabled
- D : if set, the MMU doesn't translate the addresses
- W : if set, the CPU waits until an interrupt is received. If I is set, the processor continues the

execution without affecting the PC nor the status (except W which is cleared). Otherwise, it performs a regular interrupt.

- P : if set, the CPU can perform privileged instructions and memory access
- N : this status is set if the result of an instruction is negative.
- V : this status is set if a signed overflow occurs. A signed overflow has occurred when the result has not the expected sign.
- H : this status is set if the unsigned result is strictly higher than zero in case of a subtraction
- g : this status is set for a signed subtraction if the first operand is greater than or equal the second operand.
- G : this status is set for a signed subtraction if the first operand is strictly greater than the second operand.
- Z : this status is set when the result is zero.
- C : this status is set when an addition has an unsigned result that overflows. In case of a subtraction, this status is set if the unsigned result is higher than or equal than zero.

2 Interrupts

For all interrupts : The CPU starts executing instruction at a predefined address, swaps the register set, and saves the current PC in LRi, saves the current status to the saved status and dumps the last instruction tried into the RLIi register. The status bits D, I, R and P are set, and W is cleared. An interrupt servicing routine has to be quitted with the instruction RTI to restore the PC and the saved status. For multicore systems, $100 multiplied by the core number is added to the jump address. The highest priority number is the least prioritised interrupt.

Table 4: interrupt table

name	jump address	Priority	short description	long description
Reset	`$FFFFFFFF FFFFFFFC`	0	Reset	This interrupt performs a cold reset of the system. This interrupt has to be triggered on power on. After the reset, only the core 0 is started. For the other cores, the status and the PC are initialised, but the W status is left set.
ME	`$00000000 00000000`	2	memory error	This interrupt is triggered when the MMU detects an error or a privilege violation. It aborts the current instruction. The faulty instruction is dumped into RLIi. **Warning :** we assume that no memory error will occur during an interrupt servicing routine. However, if it happens anyway, the CPU will overwrite the LRi. (But if a memory error happen in an ISR, your system is probably already really f**ked up)
AE	`$00000000 00000004`	3	Address error	This interrupt happen when the processor tries to access an address not aligned. It aborts the current instruction. The faulty instruction is dumped into RLIi. **Warning :** we assume that no address error will occur during an interrupt servicing routine. However, if it happens anyway, the CPU will overwrite the LRi. Thus, address error should be ABSOLUTELY avoided in an ISR.
ILG	`$00000000 00000008`	4	Illegal instruction	This interrupt happen when an undefined instruction is executed or privileged instruction in user mode is executed The faulty instruction is dumped into RLIi.
SYS	`$00000000 0000000C`	4	System call	This interrupt happen when the SYS opcode is executed. The SYS opcode is stored in the RLIi.
IRQ	`$00000000 00000010`	1	Interrupt request	This interrupt is the general purpose hardware interrupt. This interrupt is maskable with the status I. The last instruction executed is dumped in the RLIi.

3 Addressing modes

3.1 Memory operations

In this section EA stands for effective address

direct mode : (d) offset (15 bits signed — 32 KiB range)
EA = offset

index mode : (i) Rx, offset (10 bits signed — 1 KiB range)
EA = Rx + offset

base index shifted mode : (bi) Rx, Ry <<shift (shift amount on 5 bits)
EA = Rx + Ry <<shift

auto-indexing mode : (ai)

Rx, Ry+ Ry += data_size

Rx, Ry- EA = Rx + Ry then Ry -= data_size

Rx, +Ry Ry += data_size then EA = Rx + Ry

Rx, -Ry Ry -= data_size then EA = Rx + Ry

Important notes :

- Every memory access has to be aligned. If not, a MAM interrupt is raised.
  - If a double-word is accessed, it must be aligned on double-word (The least significant nibble is 0 or 8).
  - if a word is accessed, it must be word aligned (the least significant nibble of the address has to be either 0, 4, 8 or C).
  - If it is a half-word, the address has to be even.
  - For a byte, every address can be accessed.
- The PC is a valid register for addressing. In addition, the PC is always word aligned.
- The endianess is little.

3.2 ALU operations

register mode : (rr) `Rx, Ry [<< <left_shift_amount (4 bits)>]`
S1 = Rx, S2 = Ry (left shifted by left_shift_amount)

immediate mode : (rim) `Rx, #<immediate_value>`
S1 = Rx, S2 = immediate_value (9 bits signed)

3.3 Branch operations

relative to register mode : (r) offset
PC = Rx + offset (14 bits signed left shifted by 2 — 64 KiB range)

4 Instruction set

4.1 Memory operations

LDW — Load double-word
Loads a double-word into a register from memory

LWS — Load signed word
Loads a word into a register from memory Extends the sign bit.

LWU — Load unsigned word
Loads a word into a register from memory Pads the most significant bits with 0.

LHS — Load signed half-word
Loads a half-word into a register from memory Extends the sign bit.

LHU — Load unsigned half-word
Loads a half-word into a register from memory Pads the most significant bits with 0.

LBS — Load signed byte
Loads a byte into a register from memory Extends the sign bit.

LBU — Load unsigned byte
Loads a byte into a register from memory Pads the most significant bits with 0.

STD — Store double-word
Stores a double-word into the memory from a register (will cause a LLD / SCD on the same address to fail)

STW — Store word
Stores a word into the memory from a register stores only the least 32 least significant bits (will cause a LLD / SCD on the same address to fail)

STH — Store half-word
Stores a half-word into the memory from a register stores only the least 16 least significant bits (will cause a LLD / SCD on the same address to fail)

STB — Store byte
Stores a byte into the memory from a register stores only the least 8 least significant bits (will cause a LLD / SCD on the same address to fail)

Notes : For all memory operations, the suffix S can be added to update the status

4.2 Atomic memory operations

LLD — load link double-word
Loads a double-word from memory to a register. The load address is stored in a hidden register, the load link register, and the dirty bit of this register is cleared. If a store occures on this address, the dirty bit will be set.

SCD — store conditional double-word
Stores a double-word in memory from a register if the address of the target is in the load link register, and if the dirty bit is cleared. If V is set after the operation, the store has failed. Otherwise, it has succeeded. This operation will cause another LLD / SCD on the same address to fail. If the memory wasn't locked with LLD, the operation will fail. A core can't handle more than one LLD / SCD at a time.

Notes : For all memory operations, the suffix S can be added to update the status

4.3 Generic ALU operations

General syntax for the documentation :
General 3 operands form : `FOO D, S1, S2`

ADD — Add
Does
D = S1 + S2

SUB — Subtract
Does
D = S1 - S2

AND — Bitwise and
Does
D = S1 & S2

ORR — Bitwise or
Does
D = S1 — S2

EOR — Bitwise exclusive or
Does
D = S1 ^S2

ASL / LSL — Arithmetic shift left / Logical shift left S2 times
Does
D = S1 <<S2

```
C<-xxxx...xxxx<-0
```

ASR — Arithmetic shift right S2 times
Does
D = S1 >>S2
where S1 is signed

```
+->xxxx...xxxx->C
|   |
+---+
```

LSR — Logical shift right S2 times
Does
D = S1 >>S2
where S1 is unsigned
```
0->xxxx...xxxx->C
```

ROL — rotate left S2 times
Does
D = S1 ROL S2
```
+-xxxx...xxxx<-+->C
|               |
+---------------+
```

ROR — rotate right S2 times
Does
D = S1 ROR S2
```
+->xxxx...xxxx-+->C
|               |
+---------------+
```

MUL — Multiply
Does
D = S1 * S2
Warning : only the 32 least significant bits of the operands are multiplied together to give a 64 bits result.

Notes : For all ALU operations, the suffix S can be added to update the status

4.4 Carried ALU operations

ADC — Add with carry
Does
D = S1 + S2 + C
where C is 1 if the carry is set, 0 otherwise

SBC — Subtract with borrow
Does
D = S1 - S2 + C - 1
where C is 1 if the carry is set, 0 otherwise

RLC — rotate left through carry S2 times
Does
D = S1 RLC S2
```
C<-xxxx...xxxx<-C
```

RRC — rotate right through carry S2 times
Does
D = S1 RRC S2
```
C->xxxx...xxxx->C
```

Notes : For all ALU operations, the suffix S can be added to update the status

4.5 SIMD ALU operations

For all SIMD operations, if S2 is immediate, then each S2 word, half-word or byte will be the same.
E.g :
This :

```
ADDH R3, R3, #-3
```

Is the same as this :

```
ADDH R3, R3, R4 ; if R4 = $FFFDFFFD FFFDFFFD
```

4.5.1 SIMD add

ADDD — Add double-words (is the same as ADD)

ADDW — Add words
Adds S1 and S2 to D, but doesn't carry the bit 31.

ADDH — Add half-words
Adds S1 and S2 to D, but doesn't carry the bits 47, 31 and 15.

ADDB — Add bytes
Adds S1 and S2 to D, but doesn't carry for the bits 55, 47, 39, 31, 23, 15 and 7.

4.5.2 SIMD subtract

SUBD — Subtract double-words (is the same as SUB)

SUBW — Subtract words
Subtracts S1 and S2 to D, but doesn't carry the bit 31.

SUBH — Subtract half-words
Subtracts S1 and S2 to D, but doesn't carry the bits 47, 31 and 15.

SUBB — Subtract bytes
Subtracts S1 and S2 to D, but doesn't carry for the bits 55, 47, 39, 31, 23, 15 and 7.

4.5.3 SIMD arithmetical or logical shift left

ASLD — Shift double-words (is the same as ASL)

ASLW — Shift words
Shifts S1 and S2 to D, but doesn't carry the bit 31.

ASLH — Shift half-words
Shifts S1 and S2 to D, but doesn't carry the bits 47, 31 and 15.

ASLB — Shift bytes
Shifts S1 and S2 to D, but doesn't carry for the bits 55, 47, 39, 31, 23, 15 and 7.

4.5.4 SIMD arithmetical shift right

ASRD — Shift double-words (is the same as ASR)

ASRW — Shift words
Shifts S1 and S2 to D, but doesn't carry to the bit 31.

ASRH — Shift half-words
Shifts S1 and S2 to D, but doesn't carry the bits 47, 31 and 15.

ASRB — Shift bytes
Shifts S1 and S2 to D, but doesn't carry for the bits 55, 47, 39, 31, 23, 15 and 7.

4.5.5 SIMD logical shift right

LSRD — Shift double-words (is the same as LSR)

LSRW — Shift words
Shifts S1 and S2 to D, but doesn't carry to the bit 31.

LSRH — Shift half-words
Shifts S1 and S2 to D, but doesn't carry the bits 47, 31 and 15.

LSRB — Shift bytes
Shifts S1 and S2 to D, but doesn't carry for the bits 55, 47, 39, 31, 23, 15 and 7.

4.5.6 SIMD rotate left

ROLD — Shift double-words (is the same as ROL)

ROLW — Shift words
Shifts S1 and S2 to D, but wrap around bit 63 to bit 32 and bit 31 to bit 0.

ROLH — Shift half-words
Shifts S1 and S2 to D, but wrap around bit 63 to bit 48, bit 47 to bit 32, bit 31 to bit 16 and bit 15 to bit 0.

ROLB — Shift bytes
Shifts S1 and S2 to D, but wrap around bit 63 to bit 56, bit 55 to bit 48, bit 47 to bit 40, bit 39 to bit 32, bit 31 to bit 24, bit 23 to bit 16, and bit 15 to bit 8, and bit 7 to bit 0.

4.5.7 SIMD rotate right

RORD — Shift double-words (is the same as ROR)

RORW — Shift words
Shifts S1 and S2 to D, but wrap around bit 32 to bit 63 and bit 0 to bit 31.

RORH — Shift half-words
Shifts S1 and S2 to D, but wrap around bit 48 to bit 63, bit 32 to bit 47, bit 16 to bit 31 and bit 0 to bit 15.

RORB — Shift bytes
Shifts S1 and S2 to D, but but wrap around bit 56 to bit 63, bit 48 to bit 55, bit 40 to bit 47, bit 32 to bit 39, bit 24 to bit 31, bit 16 to bit 23, and bit 8 to bit 15, and bit 0 to bit 7.

4.5.8 SIMD sum

SUMW — And words
Sums each word of S2 then adds it S1 to D.

SUMH — And half-words
Sums each half-word of S2 then adds it S1 to D.

SUMB — And bytes
Sums each byte of S2 then adds it S1 to D.

Notes :

- For all SIMD ALU operations, the suffix S can be added to update the status. However, these are pointless because it works the same way for double-words.
- For the SIMD barrel shifting operations, each shift amount are independents.

4.6 Branch instructions

BAL — branch always
Add a displacement to the PC.

BEQ — branch on equal
The branch is taken if Z is set

BNE — branch on not equal
The branch is taken if Z is cleared

BMI — branch on minus
The branch is taken if N is set

BPL — branch on plus
The branch is taken if N is cleared

BVS — branch on overflow set
The branch is taken if V is set

BVC — branch on overflow cleared
The branch is taken if V is cleared

BCS / BHQ — branch on carry set / branch on higher than or equal (Unsigned)
The branch is taken if C is set

BCC / BLO — branch on carry cleared / branch on lower than (Unsigned)
The branch is taken if C is cleared

BHI — branch on higher than (Unsigned)
The branch is taken if H is set

BLQ — branch on lower or equal (Unsigned)
The branch is taken if H is cleared

BGT — branch on greater than (Signed)
The branch is taken if G is set

BLE — branch on less than or equal (Signed)
The branch is taken if G is cleared

BGE — branch on greater than or equal (Signed)
The branch is taken if g is set

BLT — branch on less than (Signed)
The branch is taken if g is cleared

BLK — branch and link
This instruction branches then saves the old PC to LR.
The purpose of this instruction is to branch to a subroutine.

4.7 Conditional codes instructions

TCR — transfer conditional codes to register
This instruction transfers the conditional codes byte to a register.

TRC — transfer register to conditional codes
This instruction transfers a register to the conditional codes byte.

4.8 Status instructions (privileged)

TSR — transfer status to register
This instruction transfers the whole status word to a register.
This instruction will raise an ILG interrupt if it is executed in user mode.

TRS — transfer register to status
This instruction transfers a register to the whole status word.
This instruction will raise an ILG interrupt if it is executed in user mode.

RTI — return from interrupt
The purpose of this instruction is to end an interrupt servicing routine and return to the application program. It copies the LRi register into the PC and transfers back the saved status to the current status.
This instruction will raise an ILG interrupt if it is executed in user mode.

4.9 Software interrupts

SYS — system call (software interrupt)
Interrupts the processor to execute the system call interrupt handler

ILG — Illegal instruction
Interrupts the processor to execute the illegal instruction interrupt handler

4.10 conditional prefix

A conditional prefix can be added to all instructions.

AL. — Always (equivalent to no prefix)
Always executed.

EQ. — Equal
Executed if Z is set.

NE. — Non-equal
Executed if Z is cleared.

MI. — Minus
Executed if N is set.

PL. — Plus
Executed if N is cleared.

VS. — Overflow set
Executed if V is set.

VC. — Overflow cleared
Executed if V is cleared.

CS. / HQ. — carry set / higher or equal (unsigned)
Executed if C is set.

CC. / LO. — carry cleared / lower than (unsigned)
Executed if C is cleared.

HI. — higher than (unsigned)
Executed if H is set.

LQ. — lower than or equal (unsigned)
Executed if H is cleared.

GT. — greater than
Executed if G is set.

LE. — less than or equal
Executed if G is cleared.

GE. — greater than or equal
Executed if g is set.

LT. — less than
Executed if g is cleared.

4.11 Meta-instructions

MOV — move data
Alias of `ADD <Rd> , R0, <Rx|#immediate_value>`

MVN — move negative
Alias of `SUB <Rd> , R0, <Rx|#immediate_value>`

NOT — move 1st complement
Alias of `EOR <Rd> , <Rx> , #-1`

CMP — compare
Alias of `SUBS R0, <Rx> , <Ry|#immediate_value>`

CMN — compare negative
Alias of `ADDS R0, <Rx> , <Ry|#immediate_value>`

BIT — test bits
Alias of `ANDS R0, <Rx> , <Ry|#immediate_value>`

NOP — the traditional No OPeration
Alias of `ADD R0, R0, R0`

PSH — push onto the stack
Alias of `STD <Rd>, -R30`

PLL — pull from the stack
Alias of `LDW <Rd>, R30+`

RTS — return from subroutine
Alias of `MOV PC, LR`

Note : Like ALU operations, the suffix S can be added to update the status

5 Assembler syntax

5.1 Memory operations

Rd is the register source in case of a store, Rd is the destination register in case of a load. The next values are for the addressing mode. options between <>mean required, options between [] mean optional

- `d :  FOO <Rd>, <offset(15 bits)>`
- `i :  FOO <Rd>, <Rx> [, offset(11 bits)]`
- `bi:  FOO <Rd>, <Rx>, <Ry> [, shift(6 bits)]`
- `ai:  FOO <Rd>, <Rx>+`
- `ai:  FOO <Rd>, <Rx>-`
- `ai:  FOO <Rd>, +<Rx>`
- `ai:  FOO <Rd>, -<Rx>`

5.2 ALU operations

Rd is the destination register. The next values are for the addressing mode. Options between <>mean required, options between [] mean optional

- `rr :  FOO <Rd>, <Rx>, <Ry> [<LSL|ASL|ROL|LSR|ASR|ROR> <shift_amount>]`
- `rim:  FOO <Rd>, <Rx>, #<immediate_value(12 bits)>`

General syntax for the documentation :
General form : `FOO D, S1, S2`

5.3 Branch operations

The values are for the addressing mode. Options between <>mean required, options between [] mean optional For 3 operands :

- `i :  BAR <Rx> [, offset]`
- `i :  BAR [PC,] <offset>`

5.4 Status instructions and conditional codes instructions

Rd is the destination for TSR and TCR, and Rd is the source for TRS and TCR. RTI does not have any addressing mode.

- `implied :  TRC <Rd>`
- `implied :  TCR <Rd>`
- `implied :  TRS <Rd>`
- `implied :  TSR <Rd>`
- `none :  RTI`

5.5 Software interrupts

Software interrupts don't have any addressing mode. However, a 19 bits numeric constant can be added to a SYS opcode.

- `none :  SYS [#constant]`
- `none :  ILG`