POPPY RISC

AS IS

May 16, 2017

SONNE-16 – ARCHITECTURE, INSTRUCTION SET AND IMPLEMENTATION

MICHAEL MANGELSDORF, WALDKIRCH

Sonne-16 is a 16-bit MPU instruction set architecture which describes a 3-operand load-store machine developed as part of a personal project. Its registers are memory locations that can be held in a separate RAM of arbitrary depth to offload local data from the main RAM. Each interrupt and subroutine nesting level has its own set of registers. These sets are referred to as register frames. The frames form two stacks controlled by two frame pointers, one for interrupts and one for subroutine calls. Through the use of a device referred to as subroutine frame key, a calling subroutine can conceptually pass four of its registers by reference to the called subroutine. The four registers appear to the called function as alias registers. Each of the 24 instructions has a fixed addressing mode, of which there are four.
Apart from describing the Sonne-16 ISA in more detail, this text also introduces a simulation software called Hen, a firmware for it called 8T3 including the self-assembling assembler and pattern-matcher Rey, and Paverho, a reference microarchitecture for Sonne-16 written in Verilog HDL, which runs on an FPGA device.

PROJECT NOTE

The Sonne-16 ISA and the Paverho micro-architecture together represent the "RISC" module of the Poppy project. Project source and documentation can be downloaded from the author's website, http://ok-schalter.de/poppy.

Part I.
Architecture

1 STORAGE

This section describes the memory and register entities used.

1.1 *Storage Overview*

Instruction and data words are 16 bits wide with no provision for byte-size access. Two separate memory channels can be used, one for the storage of programs and general purpose 16-bit data, and another one for storing 16-bit registers in shadow frames of 9 (subroutine frame) and 4 (interrupt frame) registers respectively, each frame being local to an interrupt service routine or subroutine.

1.2 *Registers*

1.2.1 *Hardware Registers*

The ISA uses four internal registers called PC, SFP, IFP and CFK which control execution of the MPU and are not visible to the programmer. The PC register (Program Counter) holds the memory address of the next instruction to be carried out. The SFP register (Subroutine Frame Pointer) holds the base address in frame stack memory of the current/local subroutine register frame of 9 consecutive 16-bit words. The IFP register (Interrupt Frame Pointer) holds the base address in frame stack memory of the current/local interrupt register frame of 4 consecutive 16-bit words. The subroutine frame pointer is decremented by 9 during a subroutine call, and is incremented by 9 during a return from subroutine. The interrupt frame pointer is incremented by 4 during an interrupt request, and decremented by 4 when the interrupt has been handled. The CFK register (Current Frame Key) holds four consecutive 3-bit register indices, each selecting a target for an alias register.

1.2.2 *Alias Registers*

The indices stored in the frame key register reference a selection of four subroutine frame registers in the caller's subroutine frame. The four subroutine frame registers that are thus selected are accessible in the called subroutine as virtual registers called A1, A2, A3 and A4.

1.2.3 *Subroutine Frame Registers*

Of the nine 16-bit words in the current ("local") subroutine register frame, eight are visible to the programmer and are labelled R, L1, L2, L3, L4, L5, L6, L7. The R register (Return) receives the return address during subroutine calls. The ninth 16-bit word, which is not programmer-visible, receives a copy of the frame key of the caller frame during subroutine calls.

1.2.4 *Interrupt Frame Registers*

The four 16-bit words of the current ("local") interrupt register frame are accessible through machine code instructions and are referenced as D, E, SP and DP. The D register (Default) is used as an implicit register in several instructions. Registers E (Error/Ergebnis), SP (Stack Pointer) and DP (Data Pointer) are general purpose registers but with a prescriptive name.

1.3 *Frame stacks*

As interrupts and subroutine calls occur, new interrupt frames and new subroutine frames are created, creating two stacks of frames. The interrupt stack grows towards high memory addresses, and the subroutine frame stack grows towards low memory addresses. The following figures illustrate the layout of frame memory. Each frame in these stacks resembles what is termed a shadow register bank in other architectures.

Address	...	IFP-4	IFP-3	IFP-2	IFP-1	IFP	IFP+1	IFP+2	IFP+3	...
Value	...	D	E	SP	DP	D	E	SP	DP	...
		Previous interrupt frame				Current interrupt frame				

Address	...	SFP	SFP+1	...	SFP+7	SFP+8	SFP+9	...	SFP+17	SFP+18
Value	...	R	L1	...	L7	(FK)	R	...	L7	(FK)
		Current subroutine frame					Previous subroutine frame			

2 INSTRUCTION WORDS

Sonne-16 instruction words are 16 bits wide with a fixed layout, logically composed of five constituents, referred to as G (Guide Code), L (Left-Hand), R2_MSB (Right-Hand 2 MSB), R2_LOR (Right-Hand 2 Low-Order) and R1 (Right-Hand 1). The 4-bit concatenation of R2_MSB and R2_LOR is referred to as R2. The top row of the following table refers to bit positions of the respective constituent, with the most significant bit of the instruction word being bit 15. Left-hand or right-hand refers to the order in which these constituents appear in Sonne-16 assembly language, typically using the format L `XYZ` R1 R2, where `XYZ` is an instruction mnemonic.

15-12 (4 bits)	11-8 (4 bits)	7 (1 bit)	6-4 (3 bits)	3-0 (4 bits)
G	L	R2_MSB	R2_LOR	R1
		R2		

2.1 *Guide Code*

The guide code represents the 4 high-order bits of the 5-bit operation code for a given instruction.

2.2 *R2_MSB*

R2_MSB represents the least significant bit of the 5-bit operation code for a given instruction.

2.3 *Operation Codes*

Sonne-16 operation codes do not directly appear in the instruction word. Instead, the 5-bit operation code of each instruction is derived from two constituents of the instruction word – the 4-bit guide code (high-order bits of the opcode) and R2_MSB (least significant bit of the opcode). While

numerically present in every operation code, R2_MSB is not always used for decoding; in some cases, this bit is used to extend the R2 operand range instead. In these cases, one instruction occupies two consecutive operation codes.

2.4 *Selectors*

Selectors are 4-bit groups which select one of the 16 programmer visible registers (four interrupt frame registers, eight subroutine frame registers, four alias registers). The following table lists the binary selector values together with the entities selected.

	00	01	10	11	
00XX	D	E	SP	DP	Interrupt frame registers
01XX	A1	A2	A3	A4	Alias registers
10XX	R	L1	L2	L3	Subroutine frame registers
11XX	L4	L5	L6	L7	

2.5 *Addressing Modes*

Each instruction uses a specific addressing mode, of which there are four in total, listed in the following table. For the two composite literals, the C pseudo-code for their derivation is provided, with `iw` referring to the binary value of the instruction word. Recall that slot G of the instruction word invariably contains the guide code.

1: Mode "SEVEN"
L is a selector,
R1 and R2_LOR combine
into a 7-bit literal
(`SEVEN = (R2_LOR<<4) + R1`)

2: Mode "OFFS"
L and R1 are selectors,
R2_LOR is a 3-bit literal

3: Mode "TWELVE"
L, R1 and R2 combine
into a 12-bit literal
(`iw & 0x0FFF`)

4: Mode "PAIR"
L, R1 and R2 are selectors

The following table lists the 32 nominal operation codes (binary value in parentheses), the addressing mode, and the corresponding instruction mnemonic. Note that in the second block (operation codes greater than 15), pairs of successive operation codes map to a single instruction, so that there are 24 instructions in total. This reflects the fact that R2_MSB carries no decoding information for these full-range instructions and is used to extend the R2 operand range. The underscore character separates the four bit guide code from R2_MSB.

0	(0000_0)	1	CPU	16	(1000_0)	2	GET
1	(0000_1)	1	SET	17	(1000_1)		
2	(0001_0)	1	RBF	18	(1001_0)	3	JSR
3	(0001_1)	1	RBT	19	(1001_1)		
4	(0010_0)	1	DTC	20	(1010_0)	3	RTS
5	(0010_1)	1	LTL	21	(1010_1)		
6	(0011_0)	1	EQL	22	(1011_0)	4	AND
7	(0011_1)	1	GTL	23	(1011_1)		
8	(0100_0)	2	JMP	24	(1100_0)	4	IOR
9	(0100_1)	2	LTR	25	(1100_1)		
10	(0101_0)	2	EQR	26	(1101_0)	4	EOR
11	(0101_1)	2	GTR	27	(1101_1)		
12	(0110_0)	2	LOD	28	(1110_0)	4	ADD
13	(0110_1)	2	STO	29	(1110_1)		
14	(0111_0)	2	SHL	30	(1111_0)	4	SUB
15	(0111_1)	2	SHR	31	(1111_1)		

For reference and complementing the above table, the following list presents the instruction mnemonic and a short description of each instruction. Recall that SEVEN refers to the 7-bit concatenation of R2_LOR and R1, OFFS is the sum of the register selected by R1 and the literal R2_LOR, and SXOFFS is that same sum, but with sign-extended R2_LOR.

`CPU` Transfer register L via hardware bus using SEVEN as request code
`SET` Store SEVEN into register L
`RBF` PC relative branch by signed SEVEN if L is zero (FALSE)
`RBT` PC relative branch by signed SEVEN if L non-zero (TRUE)
`DTC` Enter direct threaded code at PC + 1 using SEVEN as parameter
`LTL` Set D to 1 if L greater than SEVEN, else 0
`EQL` Set D to 1 if L less than SEVEN, else 0
`GTL` Set D to 1 if L equal to SEVEN, else 0
`JMP` Store current PC into register L, absolute branch to OFFS
`LTR` Set D to 1 if L less than SXOFFS
`EQR` Set D to 1 if L equal to SXOFFS
`GTR` Set D to 1 if L greater than SXOFFS
`LOD` Load memory cell at address OFFS into register L
`STO` Store register L into memory cell at address OFFS
`SHL` Shift register R1 left arithmetically by R2_LOR bits, result into L
`SHR` Shift register R1 right arithmetically by R2_LOR bits, result into L

`GET` Add sign-extended R2 to register R1, result in L
`JSR` Jump to subroutine using TWELVE as frame key
`RTS` Return from subroutine using TWELVE as return code into D
`AND` Bitwise AND between registers R1 and R2, result in L
`IOR` Bitwise inclusive OR between registers R1 and R2, result in L
`EOR` Bitwise exclusive OR between registers R1 and R2, result in L
`ADD` Add registers R1 and R2, result into L, carry into D
`SUB` Add negated register R2 to register R1, result into L, carry into D

Part II.
Instruction Set

1 INSTRUCTION SEMANTICS

The following subsections detail the operation and instruction word layout of the 24 Sonne-16 instructions.

1.0.1 *Instruction "CPU"*

The CPU instruction is used for interfacing the MPU to its surrounding hardware or emulation environment. It represents a link between the instruction set architecture and the micro-architecture used. The CPU instruction encodes a 7-bit request code and a selector for the source or destination register of the requested operation. The CPU instruction has an operation code of 0. In typical systems, memory cells will be zero after power-up, and hence by a system-specific definition of request code 0 ("SYS_boot"), a suitable start-up sequence can be implemented, since the first instruction executed will be "`D CPU SYS_boot`" at address 0000h.

Mode	Guide	L	R2_MSB	R2_LOR	R1
1	0000	Transfer Register	0	Request Code	

1.0.2 *Instruction "SET"*

The SET instruction is used to set a register to a small unsigned literal value (for example an ASCII code). If the 7-bit literal is equal to 7Fh, the literal following the current instruction in memory is loaded into register selected by L and PC is incremented by 2. If the 7-bit literal is not equal to 7Fh, the 7-bit literal is loaded into the register selected by L and PC is incremented by 1.

Mode	Guide	L	R2_MSB	R2_LOR	R1
1	0000	Target Register	1	Immediate Value	

1.0.3 *Instruction "RBF"*

The RBF (Relative Branch if False) instruction compares the value of the register selected by L to zero. If its value is zero (condition FALSE), the sign-extended 7-bit literal is added to the program counter. If its value is non-zero, the program counter is incremented by one. Due to sign-extension of the literal, the target address for the branch must be in the range [PC-64,PC+63].

Mode	Guide	L	R2_MSB	R2_LOR	R1
1	0001	Condition Register	0	Signed branch offset	

1.0.4 *Instruction "RBT"*

The RBT (Relative Branch if True) instruction compares the value of the register selected by L to zero. If its value is non-zero (condition TRUE), the sign-extended 7-bit literal is added to the program counter. If its value is zero, the program counter is incremented by one. Due to sign-extension of the literal, the target address for the branch must be in the range [PC-64,PC+63].

Mode	Guide	L	R2_MSB	R2_LOR	R1
1	0001	Condition Register	1	Signed branch offset	

1.0.5 *Instruction "DTC"*

The DTC (Direct Threaded Code) instruction sets the value of the register selected by L to the current value of the program counter. It then sets D to the 7-bit unsigned literal contained in the instruction, and loads the program counter from the memory address following the current instruction in memory.

Mode	Guide	L	R2_MSB	R2_LOR	R1
1	0010	Target register for PC	0	Immediate Value	

1.0.6 *Instruction "LTL"*

The LTL (Less Than Literal) instruction compares the value of the register selected by L to the 7-bit unsigned literal contained in the instruction. If its value is less than the literal, the value 1 is stored in the default register (D). If its value is not less than the literal, the value 0 is stored in D.

Mode	Guide	L	R2_MSB	R2_LOR	R1
1	0010	Condition Register	1	Comparison Value	

1.0.7 *Instruction "EQL"*

The EQL (EQual to Literal) instruction compares the value of the register selected by L to the 7-bit unsigned literal contained in the instruction. If its value is equal to the literal, the value 1 is stored in the default register (D). If its value is not equal to the literal, the value 0 is stored in D.

Mode	Guide	L	R2_MSB	R2_LOR	R1
1	0011	Condition Register	0	Comparison Value	

1.0.8 *Instruction GTL*

The GTL (Greater Than Literal) instruction compares the value of the register selected by L to the 7-bit unsigned literal contained in the instruction. If its value is greater than the literal, the value 1 is stored in the default register (D). If its value is not greater than the literal, the value 0 is stored in D.

Mode	Guide	L	R2_MSB	R2_LOR	R1
1	0011	Condition Register	1	Comparison Value	

1.0.9 *Instruction "JMP"*

The JMP (Jump) proceeds as follows. The current program counter is stored into the register selected by L. The sum of the register selected by R1 and the unsigned R2_LOR immediate value is stored into the program counter.

Mode	Guide	L	R2_MSB	R2_LOR	R1
2	0100	Target register for PC	0	xxx	xxxx

1.0.10 *Instruction "LTR"*

The LTR (Less Than Register) instruction compares the value of the register selected by L to the sum of the register selected by R1 and the sign-extended literal R2_LOR. If the value of the register selected by L is less than the sum, the value 1 is stored in the default register (D). If L is not less than the sum, the value 0 is stored in D.

Mode	Guide	L	R2_MSB	R2_LOR	R1
2	0100	Condition Register	1	xxx	xxxx

1.0.11 *Instruction "EQR"*

The EQR (EQual to Register) instruction compares the value of the register selected by L to the sum of the register selected by R1 and the sign-extended literal R2_LOR. If the value of the register selected by L is equal to the sum, the value 1 is stored in the default register (D). If L is not equal to the sum, the value 0 is stored in D.

Mode	Guide	L	R2_MSB	R2_LOR	R1
2	0101	Condition Register	0	xxx	xxxx

1.0.12 *Instruction "GTR"*

The GTR (Greater Than Register) instruction compares the value of the register selected by L to the sum of the register selected by R1 and the sign-extended literal R2_LOR. If the value of the register selected by L is greater than the sum, the value 1 is stored in the default register (D). If L is not greater than the sum, the value 0 is stored in D.

Mode	Guide	L	R2_MSB	R2_LOR	R1
2	0101	Condition Register	1	xxx	xxxx

1.0.13 *Instruction "LOD"*

The LOD (LOaD) instruction loads the memory value at the address of RAM into the register selected by L, where the address is the sum of the register selected by R1 and the unsigned literal R2_LOR. The program counter is incremented by one.

Mode	Guide	L selector	R2_MSB	R2_LOR	R1
2	0110	Target Register	0	xxx	xxxx

1.0.14 *Instruction STO*

The STO (STOre) instruction stores the register selected by L into RAM, where the destination address is the sum of the register selected by R1 and the unsigned literal R2_LOR. The program counter is incremented by one.

Mode	Guide	L selector	R2_MSB	R2_LOR	R1
2	0110	Source Register	1	xxx	xxxx

1.0.15 *Instruction "SHL"*

The SHL (SHift Left) instruction shifts the register selected by R1 left arithmetically by (R2_LOR+1) bits. The value of the register selected by R1 is not altered, the shift result is stored in the register selected by L. The program counter is incremented by one.

Mode	Guide	L selector	R2_MSB	R2_LOR	R1
2	0111	Target Register	0	Source Register	Count

1.0.16 *Instruction "SHR"*

The SHR (SHift Right) instruction shifts the register selected in R1 right arithmetically by (R2_LOR+1) bits. The value of the register selected by R1 is not altered, the shift result is stored in the register selected by L. The program counter is incremented by one.

Mode	Guide	L selector	R2_MSB	R2_LOR	R1
2	0111	Target Register	1	Source Register	Count

1.0.17 *Instruction "GET"*

The GET instruction adds the literal R2 sign-extended to 16-bit to the register selected in R1, and stores the sum in the register selected by L. Recall that GET is a full-range instruction, so that R2_MSB and R2_LOR combine into the 4-bit quantity R2.

Mode	Guide	L selector	R2	R1
2	1000	Target Register	Signed addend	Source Register

1.0.18 *Instruction "JSR"*

The JSR (Jump to SubRoutine) instruction decrements the subroutine frame pointer by 9. It then stores the sum (PC+2) into register R, and the current frame key into the subroutine at offset 8 (FK). It then sets the current frame key to the 12-bit immediate value specified in the instruction word. The program counter is then set to the memory value at address PC+1. If this memory value is zero, the program counter is set the value of the D register instead.

Mode	Guide	L, R1, R2
3	1001	12-bit frame key

1.0.19 *Instruction "RTS"*

The RTS (RTSurn) instruction increments the subroutine frame pointer by 9. The D register is set to the 12-bit immediate value specified in the instruction word. The program counter is set to the R register of the caller frame. The frame key is retrieved from offset 8 (FK) of the caller frame.

Mode	Guide	L, R1, R2
3	1010	12-bit return code

1.0.20 *Instruction "AND"*

The AND instruction computes the bitwise AND function between the register selected by R2 and the register selected by R1. The result is stored in the register selected by L.

Mode	Guide	L	R2	R1
4	1011	Target Register	Register 1	Register 2

1.0.21 *Instruction "IOR"*

The IOR (Inclusive OR) instruction computes the bitwise (inclusive) OR function between the register selected by R2 and the register selected by R1. The result is stored in the register selected by L.

Mode	Guide	L	R2	R1
4	1100	Target Register	Register 1	Register 2

1.0.22 *Instruction "EOR"*

The EOR (Exclusive OR) instruction computes the bitwise exclusive OR function between the register selected by R2 and the register selected by R1. The result is stored in the register selected by L.

Mode	Guide	L	R2	R1
4	1101	Target Register	Register 1	Register 2

1.0.23 *Instruction "ADD"*

The ADD instruction adds the register selected by R2 to the register selected by R1. The sum is stored in the register selected by L. The carry (0 or 1) is stored in register D. If the L selector selects D, the carry value overwrites the sum.

Mode	Guide	L	R2	R1
4	1110	Target Register	Register 1	Register 2

1.0.24 *Instruction "SUB"*

The SUB instruction adds the two's complement of the register selected by R2 to the register selected by R1. The sum of this addition is stored in the register selected by L. The carry (0 or 1) of this addition is stored in D. If the L selector selects D, the carry value overwrites the sum.

Mode	Guide	L selector	R2	R1
4	1111	Target Register	Register 1	Register 2

Part III.
Implementation

1 OVERVIEW

The implementation part describes turning the abstract instruction set architecture for Sonne-16 into working systems or microarchitectures. The first microarchitecture described is a portable software simulation, the second system is a computer.

2 HEN

Hen is a small, portable C program that simulates a Sonne-16 console environment and is currently used as the main development tool for the Poppy RISC project in conjunction with a native binary image containing a self-assembling assembler and a corresponding text file with the assembler source code. Initially, an assembler written in C was used to produce the first iterations of the binary image until the native assembler was completed. Hen proceeds in four stages.

1. Load the binary image (in.egg) into (simulated) RAM
2. Load the assembler source code file (8T3.asm) into (simulated) RAM
3. Assemble the assembler in the binary image
4. Output the new iteration of the binary image file (out.egg)

The reason for the name is that the self-assembly mechanism forms a "chicken-and-egg" style succession, in which three components are vital to perpetuate the development cycle. If Hen is lost, it can easily be rewritten using the Sonne-16 specification. If the assembler source file (for the current binary image) is lost, a new native assembler must be written from scratch (the binary image can still be used together with Hen to assemble this new assembler). If the binary image is lost, a new cross assembler must be written to assemble the old source code and recreate the binary image. Hen has been used on macOS (Clang) and Windows (Pelle's C). The source code is available for download from the author's website.

2.1 *Structure*

Due to its trivial size, Hen mainly uses global variables to shorten signatures. The basic features of the program are the file input/output functions, the selector logic, the virtual input/output function, the main loop, and the execute function.

2.1.1 *File I/O*

Functions are provided that save and restore the (simulated) memory image in network byte order, and load the assembler source file into (simulated) memory.

2.1.2 *Virtual I/O*

The function io_switch() implements a case statement for each CPU request code. These case statements perform simulated input-output tasks.

2.1.3 *Selector Logic*

The selector logic consists largely of functions framekey() and ref(). The framekey() function returns an index value of the nth alias register into the caller frame, where n is specified by `slot_no`. Recall that the current frame key register contains the 12-bit concatenation of four 3-bit numbers, each representing an index into the caller frame. The ref() function turns the argument, a 4-bit selector value, into a pointer to the corresponding register entity. A selector value from 8 to 15 corresponds to a subroutine frame register. A selector value from 4 to 7 corresponds to an alias register. A selector value from 0 to 3 corresponds to an interrupt frame register.

```
word framekey( word slot_no )
{
  switch (slot_no) {
      case 0: return (cfk >> 9) & 7;
      case 1: return (cfk >> 6) & 7;
      case 2: return (cfk >> 3) & 7;
      default: return cfk & 7;
  }
}

word* ref( word regsel )
{
  word* r;
  if (regsel > 7) r = &fs_RAM[ sfp + (regsel-8) ];
  else if (regsel < 4) r = &fs_RAM[ ifp + regsel ];
  else r = &fs_RAM[ sfp + 9 + framekey(regsel-4) ];
  return r;
}
```

2.1.4 *Main Loop*

The following is the function that implements the main loop. It repeatedly fetches an instruction and decodes it into various global variables used by the execute function, which is then called to dispatch the operation code. The global variable assignments are used throughout the following subsection. Function `sxt()` does sign extension to the specified number of bits.

```
void run( void )
{
  init();
  while (!quit)
  {
    cycles++;
    iw = gp_RAM[pc]; // Fetch instruction word

    // Decode instruction
    G = (iw & 0xF000) >> 12; // Slot 0 Guide code
    L = (iw & 0xF00) >> 8;   // Slot 1 Left operand
    R2 = (iw & 0xF0) >> 4;   // Slot 2 Second right operand
    R2_MSB = (R2 & 8) >> 3;  // Slot 2 Most significant bit
    R2_LOR = R2 & 7;         // Slot 2 Remaining three bits
    R1 = iw & 0xF;           // Slot 3 First right operand

    SEVEN = (R2_LOR<<4) + R1;
    OFFS = *ref(R1) + R2_LOR;
    SXOFFS = *ref(R1) + sxt(R2_LOR,3);

    // Execute instruction
    execute( opcode = (G<<1) | R2_MSB );
  }
}
```

2.1.5 *Execute Function*

The execute function is called with an operation code as argument. The function contains a switch statement that runs the code fragment at the case-label that corresponds to the operation code. The case labels for all 24 instructions are listed in the following subsection.

2.2 *Instruction Implementation*

2.2.1 *Instruction "CPU"*

This instruction delegates to the virtual I/O function (see above).

```
case op_CPU:
     io_switch();
     pc++;
     break;
```

2.2.2 *Instruction "SET"*

The SET instruction sets L to a 7-bit immediate value, but if this value is equal to 7Fh, it sets L to the 16-bit word following the current instruction in memory.

```
case op_SET:
     if (SEVEN==0x7F) *ref(L) = gp_RAM[ ++pc ];
     else *ref(L) = SEVEN;
     pc++;
     break;
```

2.2.3 *Instruction "RBF"*

The `sxt()` function sign-extends its first argument, beginning at the specified bit position (counted from 1). RBF does a PC relative conditional branch if L is false (zero). `SEVEN` is defined as `(R2_LOR<<4) + R1`.

```
case op_RBF:
     if (*ref(L)==0) pc += sxt( SEVEN, 7 );
     else pc++;
     break;
```

2.2.4 *Instruction "RBT"*

The `sxt()` function sign-extends its first argument, beginning at the specified bit position (counted from 1). RBT does a PC relative conditional branch if L is true (non-zero).

```
case op_RBT:
     if (*ref(L)!=0) pc += sxt( SEVEN, 7 );
     else pc++;
     break;
```

2.2.5 *Instruction "DTC"*

The `sxt()` function sign-extends its first argument, beginning at the specified bit position (counted from 1). DTC threads control into the address following the current instruction in memory.

```
case op_DTC:
     *ref(L) = pc;
     fs_RAM[ifp+D] = SEVEN;
     pc = gp_RAM[pc+1];
     break;
```

2.2.6 *Instruction "LTL"*

LTL compares L to an immediate value and sets the default register to 1 if L is less than the value, or to 0 if L is not less than the value.

```
case op_LTL:
     fs_RAM[ifp+D] = (*ref(L)<SEVEN) ? 1 : 0;
     pc++;
     break;
```

2.2.7 *Instruction "EQL"*

EQL compares L to an immediate value and sets the default register to 1 if L is equal to the value, or to 0 if L is not equal to the value.

```
case op_EQL:
     fs_RAM[ifp+D] = (*ref(L)==SEVEN) ? 1 : 0;
     pc++;
     break;
```

2.2.8 *Instruction "GTL"*

GTL compares L to an immediate value and sets the default register to 1 if L is greater than the value, or to 0 if L is not greater than the value.

```
case op_GTL:
     fs_RAM[ifp+D] = (*ref(L)>SEVEN) ? 1 : 0;
     pc++;
     break;
```

2.2.9 *Instruction "JMP"*

The JMP instruction stores the current program counter in one register, and then loads the program counter with the sum of another (possibly the same) register and a small immediate value.

```
case op_JMP:
     *ref(L) = pc; // Order matters
     pc = OFFS;
     break;
```

2.2.10 *Instruction "LTR"*

LTR does an *unsigned* comparison. It compares L to the sum of a register and a small sign-extended immediate value, and sets the default register to 1 if L is less than that sum, or to 0 if L is not less than the sum.

```
case op_LTR:
     fs_RAM[ifp+D] = (*ref(L)<(SXOFFS)) ? 1 : 0;
     pc++;
     break;
```

2.2.11 *Instruction "EQR"*

EQR does an *unsigned* comparison. It compares L to the sum of a register and a small sign-extended immediate value, and sets the default register to 1 if L is equal to that sum, or to 0 if L is not equal to the sum.

```
case op_EQR:
     fs_RAM[ifp+D] = (*ref(L)==(SXOFFS)) ? 1 : 0;
     pc++;
     break;
```

2.2.12 *Instruction "GTR"*

GTR does an *unsigned* comparison. It compares L to the sum of a register and a small sign-extended immediate value, and sets the default register to 1 if L is greater than that sum, or to 0 if L is not greater than the sum.

```
case op_GTR:
     fs_RAM[ifp+D] = (*ref(L)>(SXOFFS)) ? 1 : 0;
     pc++;
     break;
```

2.2.13 *Instruction "LOD"*

The LOD instruction loads L from RAM at an address which is the sum of a register and a small unsigned immediate value.

```
case op_LOD:
     widesum = ram_hi*0x10000;
     if (widesum < 0x40000)
          *ref(L) = gp_RAM[widesum+OFFS];
     pc++;
     break;
```

2.2.14 *Instruction "STO"*

The STO instruction stores L in RAM at an address which is the sum of a register and a small unsigned immediate value.

```
case op_STO:
     widesum = ram_hi*0x10000;
     if (widesum < 0x40000)
         gp_RAM[widesum+OFFS] = *ref(L);
     pc++;
     break;
```

2.2.15 *Instruction "SHL"*

The SHL instruction shifts a register value left arithmetically by up to 8 bits and stores the result in L. The shift count is a small immediate value. The value 0 corresponds to a shift count of 1.

```
case op_SHL:
     *ref(L) = (*ref(R1) << (R2_LOR+1));
     pc++;
     break;
```

2.2.16 *Instruction "SHR"*

The SHR instruction shifts a register value right arithmetically by up to 8 bits and stores the result in L. The shift count is a small immediate value. The value 0 corresponds to a shift count of 1.

```
case op_SHR:
     *ref(L) = (*ref(R1) >> (R2_LOR+1));
     pc++;
     break;
```

2.2.17 *Instruction "GET"*

The GET instruction adds a small unsigned immediate value to a register and stores the result in L.

```
case op_GET:
case op_GET2:
     *ref(L) = *ref(R1) + sxt(R2,4);
     pc++;
     break;
```

2.2.18 *Instruction "JSR"*

The JSR instruction winds the subroutine frame stack by one level and loads a new current frame key.

```
case op_JSR:
case op_JSR1:
     sfp -= 9;
     fs_RAM[ sfp+8 ] = cfk;
     cfk = iw & 0x0FFF;
     fs_RAM[sfp] = pc + 2;
     pc = ram[ pc + 1 ];
     if (!pc) pc = fs_RAM[ifp+D];
     break;
```

2.2.19 *Instruction "RTS"*

The RTS instruction sets the default register to a 12-bit immediate value ("Return code") and unwinds the subroutine frame stack.

```
case op_RTS:
case op_RTS2:
      fs_RAM[ifp+D]= (L<<7) + SEVEN;
      pc = fs_RAM[sfp]; // Same as *ref(8,0) as R;
      cfk = fs_RAM[sfp+8];
      sfp += 9;
      break;
```

2.2.20 *Instruction "AND"*

The AND instruction computes the bitwise AND function between two registers and stores the result in L.

```
case op_AND:
case op_AND2:
      *ref(L) = *ref(R2) & *ref(R1);
      pc++;
      break;
```

2.2.21 *Instruction "IOR"*

The IOR instruction computes the bitwise (inclusive) OR function between two registers and stores the result in L.

```
case op_IOR:
case op_IOR1:
      *ref(L) = *ref(R2) | *ref(R1);
      pc++;
      break;
```

2.2.22 *Instruction "EOR"*

The EOR instruction computes the bitwise exclusive OR function between two registers and stores the result in L.

```
case op_EOR:
case op_EOR1:
      *ref(L) = *ref(R2) ^ *ref(R1);
      pc++;
      break;
```

2.2.23 *Instruction "ADD"*

The ADD instruction adds two registers and stores the result in L. The carry (0 or 1) is stored in the default register. If the L selector selects D, the carry value overwrites the sum.

```
case op_ADD:
case op_ADD2:
      *ref(L) = *ref(R2) + *ref(R1);
      fs_RAM[ifp+D] = carry(*ref(R2), *ref(R1));
      pc++;
      break;
```

2.2.24 *Instruction "SUB"*

The SUB instruction adds the two's complement of one register to another register and stores the result in L. The carry (0 or 1) of this addition – not the borrow – is stored in the default register. If the L selector selects D, the carry value overwrites the sum.

```
case op_SUB:
case op_SUB2:
      tmp = (*ref(R1)^0xFFFF)+1;
      *ref(L) = *ref(R2) + tmp;
      fs_RAM[ifp+D] = carry( *ref(R2), tmp );
      pc++;
      break;
```

3 PAVERHO

Paverho is a Sonne-16 implementation in Verilog HDL. It provides a basic console system with VGA output and PS/2 keyboard input. The development environment is Altera Quartus II and the Terasic DE1-SoC board, which uses a Cyclone-V FPGA device. The design runs at the default 50 MHz memory clock of the board. The design files are available for download from the author's website.

3.1 *Structure*

The Paverho FPGA design consists of a top level design file, in which the Altera predefined memory components are instantiated and wired to the Paverho components, defined in separate files. A PS/2 keyboard driver for the PS/2 input on the DE1-SoC board was developed, as well as a VGA driver for the board's VGA circuit, with functionality for displaying text and graphics. The generated VGA signal displays 1024x768 at 60Hz refresh rate.

A single source file defines the Sonne-16 CPU core. The design is not pipelined, with currently three clock intervals per instruction. Paverho executes roughly 16 million instructions per second. During the first clock interval, an instruction is fetched from memory and the previous instruction is cleaned up. In the second clock interval, frame register selectors are resolved and the instruction is set up. During the third clock interval, the instruction is executed and results are written back.

3.1.1 *Definitions*

The Paverho core module defines the following identifiers which are used throughout this subsection, where `iw` is the 16-bit instruction word, `fs_RAM_q_b` is the output value requested from frame stack RAM port b, and gp_RAM_q is the output value requested from general purpose RAM.

```
‘define G    iw[15:12]
‘define L    iw[11:8]
‘define R2   iw[7:4]
‘define R1   iw[3:0]

‘define R2_MSB  iw[7:7]
‘define R2_LOR  iw[6:4]
‘define SEVEN   iw[6:0]

‘define OFFS    fs_RAM_q_b + iw[6:4]
‘define SXOFFS fs_RAM_q_b + {{13{iw[6]}},iw[6:4]}

‘define D  0 // Offsets from ifp
‘define E  1

‘define R  0 // Offsets from sfp
‘define FK 8
```

3.1.2 *Selector Logic*

This function, which corresponds to the `ref()` function in Hen, takes a 4-bit register selector and returns the memory address in Frame RAM of the selected register cell.

```
function [15:0] selector;
input [3:0] i;
if (i[3:3] == 1) selector = sfp + i[2:0]; // R–L7
else case(i[2:0])
 ’b000: selector = ifp + 0;                // D–DP
 ’b001: selector = ifp + 1;
 ’b010: selector = ifp + 2;
 ’b011: selector = ifp + 3;
 ’b100: selector = sfp + 9 + cfk[11:9];    // A1–A4
 ’b101: selector = sfp + 9 + cfk[8:6];
 ’b110: selector = sfp + 9 + cfk[5:3];
 ’b111: selector = sfp + 9 + cfk[2:0];
endcase
endfunction
```

3.1.3 *Main Loop*

The main loop is activated on each positive clock edge.

3.1.4 *Instruction Tasks*

For each instruction, there is a separate task that encapsulates the behaviour during the decode/execode stages. These tasks are listed in the following subsection.

The following subsections each list a Verilog HDL task implementing one of the 24 instructions. The case label 1 is active, when the CPU is in state 1. When the task is called, the instruction word (iw) has already been loaded from memory. The case label 2 is active when the CPU is in state 2.

3.2.1 *Instruction "CPU"*

Only one of the request code handlers in the case statements is listed. When a `CPU` instruction is executed with a literal value of "`GET_keycode`", during clock phase 1, either the value 0 (ASCII NUL) or the keyboard scancode is written into the register selected by L. The successful reading of the scancode is acknowledged by asserting the signal "gotkey". During clock phase 2, this edge triggered signal is deasserted, and the program counter is incremented by 1.

```
task op_CPU;
case( state )
 1: begin
     case (SEVEN)
      'KB_keycode_get:
      begin
       write_a( selector('L),
                scancode_ready ? keycode : 0);
       gotkey <= 1;
      end
     ...
     endcase
    end
  2: begin
      case (SEVEN)
       'KB_keycode_get: gotkey <= 0;
      ...
      endcase
      pc = pc + 1;
     end
endcase
endtask
```

3.2.2 *Instruction "SET"*

```
task op_SET;
case( state )
  1: if (SEVEN=='h7F) read_ram(pc+1);
  2: begin
      if (SEVEN!='h7F) begin
        write_a(selector('L),SEVEN);
        pc = pc + 1;
      end
      else begin
        write_a(selector('L),gp_RAM_q);
        pc = pc + 2;
      end
     end
endcase
endtask
```

3.2.3 *Instruction "RBF"*

The complex-looking addend in curly braces effects a sign extension to obtain signed addition.

```
task op_RBF;
case( state )
  1: read_a( selector('L) );
  2: if (fs_RAM_q_a == 0) pc = pc + {{9{iw[6]}},SEVEN};
     else pc = pc + 1;
endcase
endtask
```

3.2.4 *Instruction "RBT"*

The complex-looking addend in curly braces effects a sign extension to obtain signed addition.

```
task op_RBT;
case( state )
  1: read_a( selector('L) );
  2: if (fs_RAM_q_a != 0) pc = pc + {{9{iw[6]}},SEVEN};
     else pc = pc + 1;
endcase
endtask
```

3.2.5 *Instruction "DTC"*

The complex-looking addend in curly braces effects a sign extension to obtain signed addition.

```
task op_DTC;
case( state )
  1: begin
       write_a( ifp + 'D, SEVEN );
       write_b( selector('L), pc );
       read_ram( pc + 1 );
     end
  2: pc = gp_RAM_q;
endcase
endtask
```

3.2.6 *Instruction "LTL"*

```
task op_LTL;
case( state )
  1: read_a( selector('L) );
  2: begin
       write_a( ifp + 'D, (fs_RAM_q_a < SEVEN) ? 1 : 0);
       pc = pc + 1;
     end
endcase
endtask
```

3.2.7 *Instruction "EQL"*

```
task op_EQL;
case( state )
  1: read_a( selector('L) );
  2: begin
       write_a( ifp + 'D, (fs_RAM_q_a == SEVEN) ? 1 : 0);
       pc = pc + 1;
     end
endcase
endtask
```

3.2.8 *Instruction "GTL"*

```
task op_GTL;
case( state )
 1: read_a( selector('L) );
 2: begin
      write_a( ifp + 'D, (fs_RAM_q_a > SEVEN) ? 1 : 0);
      pc = pc + 1;
    end
endcase
endtask
```

3.2.9 *Instruction "JMP"*

```
task op_JMP;
case( state )
  1: begin
       write_b( selector('L), pc );
       read_a( selector('R1) );
     end
  2: pc = fs_RAM_q_a + OFFS;
endcase
endtask
```

3.2.10 *Instruction "LTR"*

```
task op_LTR;
case( state )
  1: begin
       read_a( selector('L) );
       read_b( selector('R1) );
     end
  2: begin
       write_a( ifp + 'D, (fs_RAM_q_a < SXOFFS) ? 1 : 0);
       pc = pc + 1;
     end
endcase
endtask
```

3.2.11 *Instruction "EQR"*

```
task op_EQR;
case( state )
  1: begin
      read_a( selector('L) );
      read_b( selector('R1) );
     end
  2: begin
      write_a( ifp + 'D, (fs_RAM_q_a == SXOFFS) ? 1 : 0);
      pc = pc + 1;
     end
endcase
endtask
```

3.2.12 *Instruction "GTR"*

```
task op_GTR;
case( state )
  1: begin
      read_a( selector('L) );
      read_b( selector('R1) );
     end
  2: begin
      tmp4 = 'R2_LOR;
      write_a( ifp + 'D, (fs_RAM_q_a > SXOFFS) ? 1 : 0);
      pc = pc + 1;
     end
endcase
endtask
```

3.2.13 *Instruction "LOD"*

```
task op_LOD; case( state )
  1: read_b( selector('R1) );
  2: begin
      read_ram( OFFS );
      LOD_flag <= 1;
      pc = pc + 1;
     end
endcase
endtask
```

3.2.14 *Instruction "STO"*

```
task op_STO;
case( state )
  1: begin
      read_a( selector('L) );
      read_b( selector('R1) );
     end
  2: begin
      write_ram( OFFS, fs_RAM_q_a );
      pc = pc + 1;
     end
endcase
endtask
```

3.2.15 *Instruction "SHL"*

```
task op_SHL;
case( state )
  1: read_a( selector('R1) );
  2: begin
      write_a( selector('L), fs_RAM_q_a << ('R2_LOR+1) );
      pc = pc + 1;
     end
endcase
endtask
```

3.2.16 *Instruction "SHR"*

```
task op_SHR;
case( state )
  1: read_a( selector('R1) );
  2: begin
      write_a( selector('L), fs_RAM_q_a >> ('R2_LOR+1) );
      pc = pc + 1;
     end
endcase
endtask
```

3.2.17 *Instruction "GET"*

```
task op_GET;
case( state )
  1: read_a( selector('R1) );
  2: begin
      write_a( selector('L), fs_RAM_q_a + {{13{'R2_MSB}},'R2_LOR} );
      pc = pc + 1;
     end
endcase
endtask
```

3.2.18 *Instruction "JSR"*

```
task op_JSR;
case( state )
  1: begin
      sfp = sfp - 9;
      read_a(ifp + 'D);
      write_b( sfp +'FK, cfk );
      read_ram(pc+1);
     end
  2: begin
      write_a( sfp +'R, pc+2 );
      cfk = iw[11:0];
      if (gp_RAM_q != 0) pc = gp_RAM_q;
      else pc = fs_RAM_q_a;
     end
endcase
endtask
```

3.2.19 *Instruction "RTS"*

```
task op_RTS;
case( state )
  1: begin
      read_a( sfp + 'R );
      read_b( sfp + 'FK );
     end
  2: begin
      write_a( ifp + 'D, iw[11:0] );
      cfk = fs_RAM_q_b;
      pc = fs_RAM_q_a;
      sfp = sfp + 9;
end
endcase
endtask
```

3.2.20 *Instruction "AND"*

```
task op_AND;
case( state )
  1: begin
      read_a( selector('R2) );
      read_b( selector('R1) );
     end
  2: begin
      write_a( selector('L), fs_RAM_q_a & fs_RAM_q_b );
      pc = pc + 1;
     end
endcase
endtask
```

3.2.21 *Instruction "IOR"*

```
task op_IOR;
case( state )
  1: begin
      read_a( selector('R2) );
      read_b( selector('R1) );
     end
  2: begin
      write_a( selector('L), fs_RAM_q_a | fs_RAM_q_b );
      pc = pc + 1;
     end
endcase
endtask
```

3.2.22 *Instruction "EOR"*

```
task op_EOR;
case( state )
   1: begin
       read_a( selector('R2) );
       read_b( selector('R1) );
      end
   2: begin
       write_a( selector('L), fs_RAM_q_a ^ fs_RAM_q_b );
       pc = pc + 1;
      end
endcase
endtask
```

3.2.23 *Instruction "ADD"*

```
task op_ADD;
case( state )
  1: begin
      read_a( selector('R2) );
      read_b( selector('R1) );
     end
  2: begin
      carry = fs_RAM_q_a + fs_RAM_q_b; // 17-bit sum
      write_a( selector('L), carry[15:0] );
      if (selector('L) != 'D) write_b( 'D, carry[16] );
      pc = pc + 1;
     end
endcase
endtask
```

3.2.24 *Instruction "SUB"*

```
task op_SUB;
case( state )
  1: begin
      read_a( selector('R2) );
      read_b( selector('R1) );
     end
  2: begin
      tmp16 = ('hFFFF ^ fs_RAM_q_b) + 1;
      carry = fs_RAM_q_a + tmp16; // 17-bit sum
      write_a( selector('L), carry[15:0] );
      if (selector('L) != 'D) write_b( 'D, carry[16] );
      pc = pc + 1;
     end
endcase
endtask
```

There is currently a single source-file of around 2600 lines with native Sonne-16 code named "8T3.asm", referred to as the firmware. It contains basic utility and arithmetic functions, an extensible pattern-matcher and a native assembler. The object code size of the firmware is around 3000 cells (6 kb), including static data structures.

4.1 *Assembly Language*

The assembly language adopted for Sonne-16 has the following features.

- If a line does not start with a space, it is a comment until end-of-line
- Anything after a semicon is considered a comment until end-of-line
- The L register is placed to the left of the instruction mnemonic, except for JSR and RTS
- The R2 operand follows the R1 operand after the instruction mnemonic
- If the L register is omitted, it is assumed to be same as the R1 register
- If the R2 register is omitted, R2_LOR is assumed 0
- If the numeric operand of a SET instruction is omitted, it is assumed to be 7Fh (encoding for fetch immediate)
- Any "bare" upper-case identifier is considered a unique address label and is substitued by a 16-bit value
- The expression *DEF MixedCaseIdentifier <Number>* defines a numeric constant
- Mixed case identifiers resolve to numeric constants defined by the DEF keyword
- Any identifier with a preceding ">" placed outside the definition of an instruction is resolved to the 16-bit address of the nearest occurence of the identifier *following* the current object code address. When used inside the definition of the instruction, it is resolved in the same way, but relative to the current object code address
- Any identifier with a preceding "<" placed outside the definition of an instruction is resolved to the 16-bit address of the nearest occurence of the identifier *before* the current object code address. When used inside the definition of the instruction, it is resolved in the same way, but relative to the current object code address
- Any identifier preceded by "@" creates a 16-bit address symbol in the symbol table set to the current object code address
- Any numeric value placed outside an instruction is placed into the object code as a 16-bit number. The base indicators h (hexadecimal) or b (binary) may be appended to the number, and an optional plus or minus sign can be prefixed
- A double colon ("::") breaks the current lexical pattern, ensuring termination of the current instruction for instance
- Anything between inch marks (") is placed into the object code as a sequence of characters, one ASCII character per 16-bit word (least significant byte)

The 8T3 firmware includes a 2-pass native assembler called Rey. The source file is assembled line by line, populating a symbol table. Each line is tokenized into numbers, strings, register names, instruction mnemonics and other defined identifiers. The token sequence obtained is then matched against an extensible list of token patterns. Each pattern has an associated handler function, and the function that corresponds to the longest matching pattern for any given line is called, if there is at least one matching pattern. The pattern handlers generate the object code that corresponds to the source line.

4.2.1 *Pattern Example*

The following code fragment is the pattern for a line containing a branch instruction of the form "`L1 RBT >LABEL`". The first six hexadecimal numbers each encode a matching token type that must be present for the pattern to match. The four digits of each hexadecimal number correspond to the required index, group, subtype and type of the token. The pattern ends if all the digits of the hexadecimal number are 0. One can hence see that this pattern ends after three tokens. The first token required corresponds to a register name, the second token required corresponds to an instruction of subtype 4 (RBT/RBF), and the third token must have type 7 (code label). The number 22 is the pattern ID. It is possible to direct several patterns to the same handler function, and the ID allows the handler function to know why it was called. The expression "`>H_BRA`" is the code address of the handler function.

```
0101h 0149h 0107h 0000h 0000h 0000h 22 >H_BRA
```

The pattern matcher can be used for other purposes, as the patterns could for example identify commands on a commands line, which the pattern handlers can carry out.

4.3.1 *Multiplication*

The following subroutine multiplies two unsigned 16-bit numbers in alias registers A1 and A2. The low order result is returned in A3, the high order result is returned in A4. Due to the use of alias registers, the input operands A1 and A2 can be left unchanged, or they can be identified with A3 and A4 in the call signature, so that the arguments are overwritten by the result. This is why in the first two lines of code, A1 and A2 are assigned to subroutine frame registers rather than being used directly. Recall that alias registers reference any of the 8 subroutine frame registers of the caller frame, so the caller can freely choose argument and destination registers for the multiplication call.

```
@MUL  L6 GET A1          ; Copy multiplier
      L1 GET A2          ; Copy multiplicand
      A3 GET L6          ; Initialise low order result
      A4 SET 0           ; Initialise high order result
      L2 SET 16          ; Repeat 16 times
      L4 SET 1           ; Bit mask LSB
      L5 SET :: 8000h    ; Bit mask MSB

@REP  L7 AND A3 L4       ; Test multiplier bit 0
      L7 RBF >1
         ADD A4 L1       ; Add multiplicand if bit set
  @1  L3 AND A4 L4       ; Check high order bit 0
         SHR A4 1        ; Shift high order right 1 bit
         SHR A3 1        ; Shift low order right 1 bit
      L3 RBF >2
         IOR A3 L5       ; Import bit from high order
  @2  L2 GET L2 -1       ; Decrement counter
      L2 RBT <REP        ; Repeat
         RTS 0
```