SPECT

Programmers Guide

Version: 0.5

Git tag: v0.2

Tropic Square

August 7, 2023

Version history

Version Tag	Date	Author	Description
0.1	12.10.2022	Ondrej Ille	Initial version
0.2	7.11.2022	Ondrej Ille	Fix semantics of LSR instruction.
0.3	8.11.2022	Ondrej Ille	Fix SCB semantics.
0.4	14.11.2022	Ondrej Ille	Fix semantics of ST instruction (op1 instead of op2). Add note about modular instruction operands.
0.5	23.11.2022	Ondrej Ille	Add description of SW toolchain.

Contents

1 Glossary

- **CPU** - Central Processing Unit
- **ECC** - Eliptic Curve Cryptography
- **SPECT** - Secure Processor of Eliptic Curves for Tropic

2 Register field types

Meaning of Register field types is following:

- **RW** - Read-Write field
- **RO** - Read-only field
- **WO** - Write-only field
- **RW W1C** - Read-Write field, Write 1 to clear
- **RW W0C** - Read-Write field, Write 0 to clear
- **RW W1S** - Read-Write field, Write 1 to set
- **RW W0S** - Read-Write field, Write 0 to set
- **RW W1T** - Read-Write field, Write 1 to toggle
- **RW W0T** - Read-Write field, Write 0 to toggle

3 Introduction

This document provides a programmer's guide for SPECT. SPECT is a domain specific processing unit targeted for calculations of Elliptic Curve Cryptography (ECC). SPECT provides instructions for calculation with 256 bit numbers and modular arithmetics. SPECT is useful to implement operations/algorithms such as:

- ECDSA - Elliptic Curve Digital Signature Algorithm
- ECDH - Elliptic Curve Diffe-Hellman

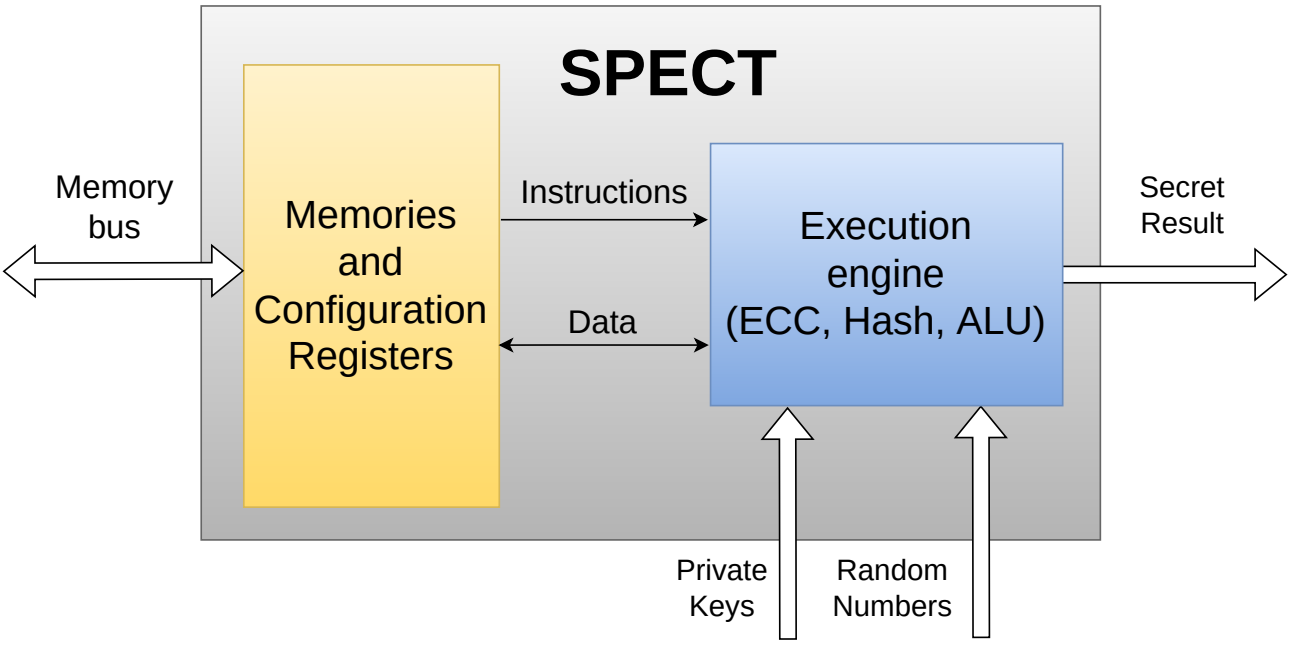


Figure 1: SPECT - Block diagram

4 Programmer's model

SPECT programmer's model consists of:

- 32 x 256 bit general purpose registers (**R0** - **R31**).
- **PC** - Program counter.
- Zero (**Z**) and Carry (**C**) flag.
- HW **RAR** (Return Address Register) stack for nested procedure calls.
- **SRR** (Secret Results Register) for handling secret results.
- 2048 B read-write memory space in address range 0x0000 – 0x07FC.
- 512 B write-only memory space in address range 0x1000 – 0x11FC.
- 2048 B read-only memory space in address range 0x3000 – 0x37FC.

Note

SPECTs address space is 32 bit word organized. **LD** and **ST** instructions works with 256 bit values and it always uses 8 consecutive words in the memory. E.g. 0x0020 - 0x003C.

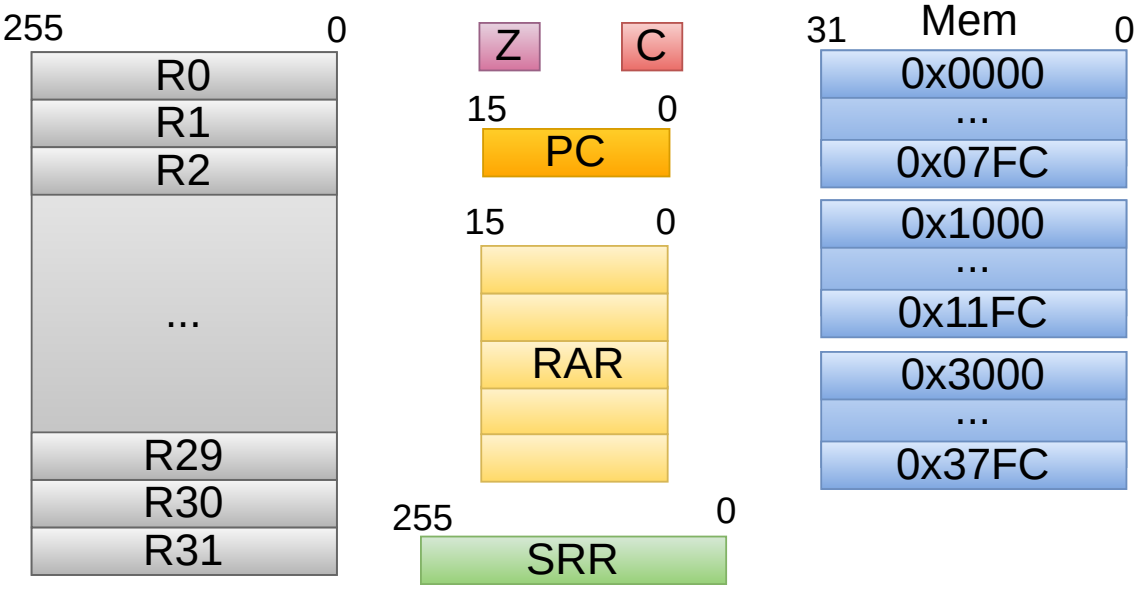


Figure 2: SPECT - Programmers model

4.1 Subroutine calls

SPECT contains HW **RAR** stack, and it pushes return address from subroutine to RAR stack each time when it executes CALL instruction. When SPECT executes RET instruction, it pops value from RAR stack and updates **PC**. HW **RAR** stack supports up to 5 nested subroutine calls.

Note

Behavior of SPECT when number of nested subroutine calls is exceeded is undefined.

4.2 Using private keys

Private keys are typically sensitive information in cryptographic systems. Therefore, it is good to minimize the time for which they are present in such system. When SPECT Program needs to use private key which is stored elsewhere in the system, it executes GPK instruction which reads 256 bit private key from outside of SPECT. GPK instruction supports key identification in case of multiple private keys are available in the system.

4.3 Using random numbers

Proper random numbers with uniform distribution are critical for cryptographic calculations. Such random numbers can be typically generated by TRNG (True Random Number Generator). When SPECT Program needs a random number, it executes GRV instruction which obtains random number from outside of SPECT.

4.4 Modular arithmetics

SPECT provides instructions for finite field arithmetic such as addition, subtraction and multiplication with 256 bit operands stored in general purpose registers. SPECT supports fast multiplication in Ed25519 and P-256 curves finite fields via dedicated instructions – MUL25519 and MUL256. Modular arithmetics with generic modulus specified by value in **R31** is supported by instructions ADDP, SUBP, MULP. SPECT also supports modular reduction of 512 bit number with REDP instruction.
When programming with modular instructions, one needs to be careful about input operands of such instructions. Following conditions must be met:

- op2 < P_{25519} and op3 < P_{25519} for MUL25519 instruction.
- op2 < P_{256} and op3 < P_{256} for MUL256 instruction.
- op2 < R31 and op3 < R31 for ADDP, SUBP instructions.
- R31 != 0 and R31 != 1 for ADDP, SUBP, MULP, REDP instructions.

if these conditions are not met when invoking such modular instruction, result of instruction calculation is undefined (value in op1).

Note

Performance of MULP when **R31** = P_{25519} / P_{256} is lower than performance of MUL25519 / MUL256.

4.5 Hash calculation

SPECT supports SHA512 Hash calculation as specified in FIPS 180-4. SPECT can calculate SHA512 hash from arbitrarily long data stream. When SPECT executes HASH_IT instruction, it resets context in its execution engine to initialization vector as specified in FIPS 180-4. Each execution of HASH instruction processes 1024 bit block, and executes next round of SHA512 calculation.

Note

SPECT does not perform padding of input data as specified in section 5.1 of FIPS 180-4. It is responsibility of SPECT program or external system to perform such padding.

4.6 Handling secret results

Cryptographic calculations frequently result in a shared secret (e.g. in case of ECDH calculation), which shall only be visible to certain other objects in the system, but it shall not be globally accessible on system level memory bus. When SPECT program finishes its execution, it stores these results into **SRR** register. After the program execution has finished, external blocks within a system can read this secret result from the SPECT via dedicated handshake.

4.7 Scalar blinding

SPECT supports scalar blinding by a random number as side-channel counter-measure by SCB instruction. It blinds the scalar sc using group scalar randomization method as defined in [1] with 256 bit random number. The random number rng shall be obtained in advance by GRV instruction as described above. The group order q shall be present in **R31**.
SCB performs this exact function:

$$Blind(sc, rng, q) = q \times (rng|(2^{255} + 2^{223})) + sc$$

4.8 SPECT invocation

SPECT is invoked by external system which has access to its memory space via memory bus as shown in following figure:

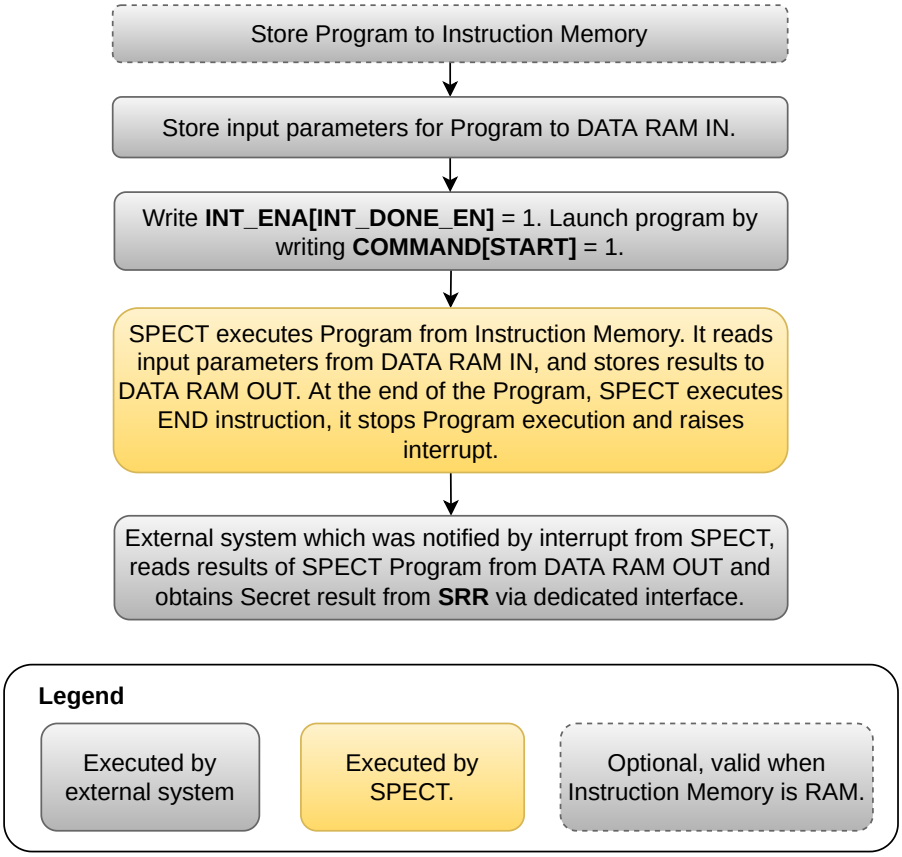


Figure 3: SPECT - Invocation

Note

Address of first Program instruction executed by SPECT after **COMMAND[START]** = 1 is written, is fixed and defined by a system which integrates SPECT.

4.9 Invalid instructions

When SPECT attempts to execute invalid instruction, it aborts program execution and sets **STATUS[ERR]** = 1.

Note

Invalid instruction means invalid opcode or not matching parity bit in the instruction code. Unless a fault, usual cause of this is e.g. missing RET instruction in subroutine or END instruction at the end of program.

4.10 Soft Reset

SPECT can be reset by extenal system by writing **COMMAND[RST]** = 1. When SPECT is reset, it aborts program execution and resets its internal state.

4.11 Interrupts

SPECT program execution can't be interrupted by an external event (other than Soft reset). SPECT itself can generate following interrupts for external system:

- Program Done - Enabled when **INT_ENA[INT_DONE_EN]** = 1. Generated when SPECT program executes END instruction, or it detects error.
- Program Error interrupt - Enabled when **INT_ENA[INT_DONE_EN]** = 1. Generated when SPECT attempts to execute invalid instruction.

5 Instruction set

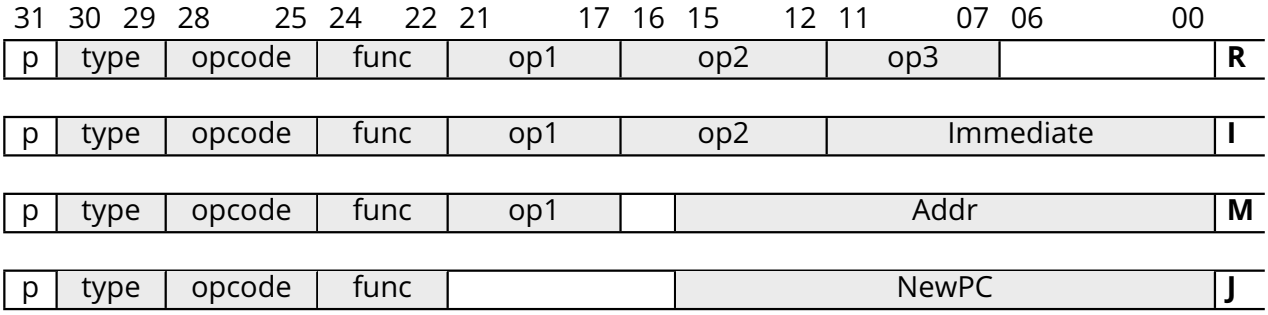
SPECT provides 4 types of instructions:

- **R** - Register
- **I** - Immediate
- **M** - Memory
- **J** - Jump

5.1 Operand interpretation

All operands are considered as 256 bits unsigneds. Arithmetic instructions working only with 32 bit operands ignores the 224 MSBs of input and clears them in the result. Logic instructions working only with 32 bit operands also ignores the 224 MSBs of input, but passes the 224 MSBs of op2 to the result.

5.2 Instruction Format



5.3 Symbols

Following symbols are used in description of instructions:

- || – Bitwise concatenation
- $P_{25519} = 2^{255} - 19$
- $P_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
- F – Flags set by the instruction
- #C – Number of cycles the instruction takes to execute

5.4 R instructions

Mnemonic	Name	Semantics	F	#C
32 bit arithmetic instructions				
ADD op1,op2,op3	32 bit adition	op1 = op2 + op3	Z	11
SUB op1,op2,op3	32 bit subtraction	op1 = op2 - op3	Z	11
CMP op2,op3	32 bit comparison	op2 - op3	Z	9
32 bit logic instructions				
AND op1,op2,op3	32 bit bitwise AND	op1 = op2 & op3	Z	11
OR op1,op2,op3	32 bit bitwise OR	op1 = op2 | op3	Z	11
XOR op1,op2,op3	32 bit bitwise Exclusive OR	op1 = op2 ^ op3	Z	11
NOT op1,op2	32 bit bitwise NOT	op1 = ~op2	Z	10
Shift Instructions				
LSL op1,op2	Logic shift left	op1 = op2[254:0] || 0	C	10
LSR op1,op2	Logic shift right	op1 = 0 || op2[255:1]	C	10
ROL op1,op2	Rotating shift left	op1 = op2[254:0] || op2[255]	C	10
ROR op1,op2	Rotating shift right	op1 = op2[0] || op2[255:1]	C	10
ROL8 op1,op2	Rotating byte shift left	op1 = op2[247:0] || op2[255:248]		10
ROR8 op1,op2	Rotating byte shift right	op1 = op2[7:0] || op2[255:8]		10
SWE op1,op2	Swap endianity	op1[255:248] = op2[7:0] op1[247:240] = op2[15:8] ... op1[7:0] = op2[255:248]		10
Modular arithmetic instructions				
MUL25519 op1,op2,op3	Multiplication in $GF(P_{25519})$	op1 = (op2 * op3) % P_{25519}		91
MUL256 op1,op2,op3	Multiplication in $GF(P_{256})$	op1 = (op2 * op3) % P_{256}		139
ADDP op1,op2,op3	Generic Modular Addition	op1 = (op2 + op3) % R31		16
SUBP op1,op2,op3	Generic Modular Subtraction	op1 = (op2 - op3) % R31		16

Mnemonic	Name	Semantics	F	#C
MULP op1,op2,op3	Generic Modular Multiplication	op1 = (op2 * op3) % R31		597
REDP op1,op2,op3	Generic Modular Reduction	op1 = (op2 || op3) % R31		528
Other Instructions				
MOV op1,op2	Move register	op1 = op2		7
CSWAP op1,op2	Conditional swap	***if*** **C** == 1 ***then:*** op1 = op2 op2 = op1		11
HASH op1,op2	Hash	tmp = *SHA512*(op2+3||op2+2||op2+1||op2) op1 = tmp[255:0] op1+1 = tmp[511:256]		347
GRV op1	Get Random Value	op1 = Random number		–
SCB op1,op2,op3	Blind scalar	B = *Blind*(op2, op3, R31) op1 = B[255:0] op1+1 = B[511:256]		88

5.5 I instructions

Mnemonic	Name	Semantics	F	#C
32 bit arithmetic instructions				
ADDI op1,op2,Immediate	32 bit addition	op1 = op2 + Immediate	Z	11
SUBI op1,op2,Immediate	32 bit subtraction	op1 = op2 - Immediate	Z	11
CMPI op2,Immediate	32 bit comparison	op2 - Immediate	Z	9
12 bit logic instructions				
ANDI op1,op2,Immediate	12 bit bitwise logic AND	op1 = op2 & Immediate	Z	11
ORI op1,op2,Immediate	12 bit bitwise logic OR	op1 = op2 | Immediate	Z	11
XORI op1,op2,Immediate	12 bit bitwise exclusive OR	op1 = op2 ^ Immediate	Z	11

Mnemonic	Name	Semantics	F	#C
Other Instructions				
CMPA op2,Immediate	comparison	***if*** op2 == Immediate ***then:*** Z = 1 ***else:*** Z = 0	Z	9
MOVI op1,Immediate	Move immediate	op1[11:0] = Immediate, op1[255:12] = 0		6
HASH_IT	Hash init	Reset hash calculation.		9
GPK op1, Immediate	Get Private Key	op1 = Private key, Key index = immediate		–

Due to not enought space in the 32 bit instruction format, the immediate operand is just 12 bit. Because of that, the logic instructions works only with the 12 LSBs of op2. E.g. 0xFF12 & 0xF0F = 0xFF02.

5.6 M instructions

Mnemonic	Name	Semantics	F	#C
LD op1,Addr	Load	op1[31:0] = Mem[Addr] op1[63:32] = Mem[Addr+0x4] ... op1[255:224] = Mem[Addr+0x1C]		21
ST op2,Addr	Store	Mem[Addr] = op1[31:0] Mem[Addr+0x4] = op1[63:32] = ... Mem[Addr+0x1C] = op1[255:224]		12

5.7 J instructions

Mnemonic	Name	Semantics	F	#C
CALL NewPC	Subroutine call	push(RAR, PC+0x4), PC = NewPC		5
RET	Return from subroutine	PC = pop(RAR)		5
BRZ NewPC	Branch on Zero	***if*** Z == 1 ***then:*** PC = NewPC		5
BRNZ NewPC	Branch on not Zero	***if*** Z == 0 ***then:*** PC = NewPC		5
BRC NewPC	Branch on Carry	***if*** C == 1 ***then:*** PC = NewPC		5
BRNC NewPC	Branch on not Carry	***if*** C == 0 ***then:*** PC = NewPC		5
JMP NewPC	Unconditional jump	PC = NewPC		5

6 SPECT Memory Map

Base Address: 0x0000 0000
End Address: 0x0000 9FFF

Memory region	Address offset range	Size
Data RAM IN	0x0000 0000 0x0000 07FF	2 KB
Data RAM OUT	0x0000 1000 0x0000 11FF	512 bytes
Configuration registers	0x0000 2000 0x0000 200F	16 bytes
Constants ROM	0x0000 3000 0x0000 37FF	2 KB
External Memory In	0x0000 4000 0x0000 403F	64 bytes
External Memory Out	0x0000 5000 0x0000 504F	80 bytes
Instruction Memory	0x0000 8000 0x0000 9FFF	8 KB

6.1 Configuration registers

Base Address: 0x0000 2000
End Address: 0x0000 200F

Address Offset	Register Name	Reset Value
0x0	BLOCK_ID	0x000-0030
0x4	COMMAND	0x00000000
0x8	STATUS	0x00000001
0xc	INT_ENA	0x00000000

Register name:		BLOCK_ID		
Address:		0x2000		
Field	**Type**	**Reset value**	**Bits**	**Description**
ID_CODE	RO	0x30	15:0	Identification code
REV_CODE	RO	-	19:16	Revision code

Register name:		COMMAND		
Address:		0x2004		
Field	**Type**	**Reset value**	**Bits**	**Description**
START	WO W1S;	0x0	0:0	Starts SPECT FW operation
SOFT_RESET	WO	0x0	1:1	Stops FW execution and resets SPECT

Register name:		STATUS		
Address:		0x2008		
Field	**Type**	**Reset value**	**Bits**	**Description**
IDLE	RO	0x1	0:0	SPECT is in IDLE mode
DONE	RW W1C	0x0	1:1	Active when SPECT successfully completes the calculation
ERR	RW W1C	0x0	2:2	Active when SPECT ends the calculation with error

Register name:	INT_ENA

Address:		0x200c		
Field	**Type**	**Reset value**	**Bits**	**Description**
INT_DONE_EN	RW	0x0	0:0	Enables DONE interrupt
INT_ERR_EN	RW	0x0	1:1	Enables ERROR interrupt

6.2 Data RAM IN

Data RAM IN is a memory where external system stores parameters for SPECT Program before it starts its execution. SPECT Program sees it as read-write memory.

6.3 Data RAM OUT

Data RAM OUT is a memory where SPECT Program stores results of its calculation, and external system reads such results after SPECT Program execution ends. SPECT Program sees it as write-only memory.

6.4 Instruction Memory

Instruction memory contains the Program executed by SPECT. Based on SPECT manufacturing configuration, this memory might be readable/writable by external system (Instruction Memory is RAM), or not accessible by external system at all (Instruction Memory is ROM with fixed program). SPECT Program do not have access to this memory with LD and ST instructions.

6.5 Constant ROM

Constant ROM contains a ROM image with important cryptographic constants used by SPECT Program during program execution. SPECT program sees it as read-only memory.
Open Issue 1: Place SPECT Constant ROM content!

7 SW toolchain

SPECT contains SW toolchain intended for SW development and debugging. SPECT has following applications available:

- spect_compiler – A compiler/assembler which creates .hex file from .s assembly file.
- spect_iss – Instruction level simulator with simple command line debugger. It can simulate .s file as well as .hex file.

Options for each of the applications are described when using **- -help** command line option. Options available inside interactive shell of **spect_iss** are available with **- -help** command line oprion or **help** command.
SPECT assembler has support for following assembly language features:

- Function labels
- Constant definitions
- Include other assembly file

7.1 Tool requirements

SPECT SW toolchain requires following tools:

- CMAKE 3.18.2 or higher

Open Issue 2: What else?

7.2 Function labels

SPECT compiler allows definition of function labels, and passing them as NewPc of J instructions, e.g like so:

```
_start:
    CALL my_func
    END

my_func:
    ADD r0, r1, r2
    RET
```

7.3 Constant definitions

SPECT compiler allows definition of constants, and passing them as Addr of M instructions or Immediate operand of I type instructions like so:

```
threshold .eq 0x12

_start:
    ADDI r0, r0, threshold
```

```
p25519_addr .eq 0x3020

_start:
    LD r31, p25519_addr
```

Note

Currently, SPECT compiler does not support expression parsing, it only supports simple decimal, hexadecimal or binary value when defining constants.

7.4 Include other assembly file

Multiple .s assembly files can be connected together in SPECT source code via "include" directive, e.g. like so:

```
_start:
    NOP

.include <other_s_file>
    END
```

8 REFERENCES

References

[1] Danger, Jean-Luc et al. “A synthesis of side-channel attacks on elliptic curve cryptography in smart-cards.” Journal of Cryptographic Engineering 3 (2013): 241 - 265.

9 Open Issues

Document contains following open issues:

Open Issue 1: Place SPECT Constant ROM content!
Open Issue 2: What else?