**SSRV\_CORE:  
Subset RISCV 32-bit Core**

HAS: High-Level Architecture Specifications

Revision HAS 0.3 – draft  
Amichai Ben-David ([amichai.ben.david@intel.com](mailto:amichai.ben.david@intel.com))

Table of Contents

[1 General Description 4](#_Toc77860076)

[1.1 Block Diagram 4](#_Toc77860077)

[2 Top Level Interface 5](#_Toc77860078)

[3 SSRV\_CORE Pipe Stages 6](#_Toc77860079)

[3.1 Other HW Blocks 6](#_Toc77860080)

[4 SSRV ISA – Instruction Set Architecture 7](#_Toc77860081)

[4.1 RV32I Base Instruction Formats 7](#_Toc77860082)

[4.2 Integer Computational Instructions 8](#_Toc77860083)

[4.3 Load And Store Instructions 10](#_Toc77860084)

[4.4 Instruction Set Listing – Subset of RV32I 11](#_Toc77860085)

[5 Hazards 12](#_Toc77860086)

[5.1 Data Hazard 12](#_Toc77860087)

[5.2 Load Hazard 12](#_Toc77860088)

[6 Example use case of SSRC32I\_CORE within the Fabric: 13](#_Toc77860089)

[6.1 Fabric Execution Flow: 13](#_Toc77860090)

[7 Assembly Programmer Notes 14](#_Toc77860091)

Figures

[Figure 1 - SSRV\_CORE Block Diagram 4](#_Toc77860092)

[Figure 2 - Fabric Block diagram 13](#_Toc77860093)

Tables

[**Table 1 - Revision History** 2](#_Toc77860226)

[**Table 2 - Related Documents** 2](#_Toc77860227)

[**Table 3 - Glossary** 3](#_Toc77860228)

[**Table 4 - SSRV\_CORE Parameters** 5](#_Toc77860229)

[**Table 5 - SSRV\_CORE general interface** 5](#_Toc77860230)

[**Table 6 - SSRV\_CORE standard interface** 5](#_Toc77860231)

[**Table 7 - Instruction Formats** 7](#_Toc77860232)

[**Table 9 - Register-Immediate Instructions** 8](#_Toc77860233)

[**Table 10 - NOP instruction** 8](#_Toc77860234)

[**Table 11 - Register-Register Instructions** 9](#_Toc77860235)

[**Table 12 - LOAD instruction** 10](#_Toc77860236)

[**Table 13 – STORE instruction** 10](#_Toc77860237)

[**Table 15 - Instruction Formats** 11](#_Toc77860238)

[**Table 16 – SSRV – Subset of the RISC-V Base Instruction Set** 11](#_Toc77860239)

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| R*ev. No.* | Who | Description | Rev. Date |
| 0.3 | Amichai Ben-David | Initial SSRV\_CORE HAS | 18 July 2021 |
| 0.35 | Amichai Ben-David | First Review with Ori – Simplify HW Spec.  - Remove EBREAKE - simplify the Immediate sign extension explanation. - Remove “Excluded Instructions.” Section - Remove the Hazards "possible solution" | 22 July 2021 |

**Table 1 - Revision History**

Related Documents

|  |  |  |
| --- | --- | --- |
| Name | Path | Description |
| riscv\_isa\_spec.pdf | TODO | The full RISCV Unprivileged Specification file.  Including the RV32I Baseline ISA |
| SSRV\_TILE\_HAS.pdf | TODO | The HAS for core & memory & io\_ctrl Integration model. |

**Table 2 - Related Documents**

Glossary

|  |  |
| --- | --- |
| Term | Description |
| ISA | Instruction Set Architecture. (such as X86, ARM, RISC-V etc.) |
| IO | Input & output. |
| IP | Intellectual Property. In this case, RTL building block that can be consumed. |
| HAS | High Level Architecture Specifications. (This document) |
| MAS | Micro Architecture Specifications. Document with the microarchitecture details. |
| I\_MEM | Instruction memory – where the program is loaded and ready for execution. |
| D\_MEM | Data Memory – where the LOAD & STORE instructions read/write Data. |
| Pipeline | Common Way to parallel and utilize Hardware  <https://en.wikipedia.org/wiki/Instruction_pipelining> |
| RISC | Reduce Instruction Set Computer. (Unlike CISC -Complex Instruction Set Computer) <https://en.wikipedia.org/wiki/Reduced_instruction_set_computer> |
| Thread | A "hardware thread" is a physical CPU or core that ca run a program. |
| RISC-V | A relatively new open and free ISA. (comparable to intel X86, ARM)  <https://en.wikipedia.org/wiki/RISC-V> |
| RV32I | “RISC-V 32-bit Integer” The RISC-V baseline compatible ISA (no extensions M/A/F etc.) |
| Standard interface | Functional characteristics to allow the exchange of information between two systems |
| Word | 32-bits of data - 4 Bytes. The size of an integer in RV32I ISA. |
| Hazard | Potential source of harm. in this document when reading Outdated data, or wrongly executing Instruction. |
| Strap | Tie signals to constant value (1’b0 or 1’b1) |
| MSFF | Main & Secondary Flip Flop. (AKA Master Slave Flip Flop) |
| Clock Gating | Logic that allows to condition the MSFF clock. Functionality and power reasons. |
| Polling | Actively sampling the status of an external device. |

**Table 3 - Glossary**

# General Description

RISC-V is an Open & Free ISA that is used in academia and industry.  
The RISC-V eco system has all the SW needed to program, compile and creating RISC-V assembly & executable RISC-V machine code.  
Unlike other ISAs, anyone can write compatible RISC-V Core without going through a bureaucracy of licenses and fees.

The HAS describes the High-Level-Architecture of the ssrv\_core.  
A single thread, general purpose core that supports a **subset** of the RV32I ISA.

The ssrv\_core sets the PC (program counter) with an address and gets the corresponding instruction from the I\_MEM (instruction memory).  
The Instructions loaded from the I\_MEM (AKA the “program”) will be executed in a pipeline fashion.

The program will interact with the data memory (D\_MEM) using LOAD & STORE Instruction.

## Block Diagram



Figure - SSRV\_CORE Block Diagram

\* The HAS describes the SSRV\_CORE High-Level-Architecture-Specification   
and not the memory module.

# Top Level Interface

**Default Parameter Values:**

|  |  |  |
| --- | --- | --- |
| Name | Default Value | Description |
| XLEN | 32 | Integer Size - RV32I Spec |
| REGFILE\_NUM | 32 | Number of registers in the register file - RV32I Spec is 32 |

**Table 4 - SSRV\_CORE Parameters**

**General interface signals:**

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Size | Direction | Description |
| ClkQH | 1 | Input | Q Clock – a single clock domain. 100Mhz - 2Ghz |
| ResetQnnnH | 1 | Input | Active High |

**Table 5 - SSRV\_CORE general interface**

**Standard interface signals:**

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Size | Direction | Description |
| Core < --- > I\_MEM | | | |
| PcQ100H | XLEN | Output | The program Counter. Used as the “read address pointer” which reads the current Instruction to be executed for the I\_MEM. |
| InstructionQ101H | 32 | Input | The instruction that was read from I\_MEM  to be executed. The size is according to RISCV spec |
| Core < --- >D\_MEM | | | |
| AddressDmQ103H | XLEN | Output | 1) The LOAD address. Read address D\_MEM 2) The STORE address. Write address D\_MEM |
| WrDataDmQ103 | XLEN | Output | The data to STORE in the D\_MEM |
| RdEnDmQ103H | 1 | Output | In case of a LOAD instruction – Read Enable |
| WrEnDmQ103H | 1 | Output | In case of a STORE instruction – Write Enable |
| RdDataDmQ104H | XLEN | Input | The data to LOAD from D\_MEM to Register file. |

**Table 6 - SSRV\_CORE standard interface**

# SSRV\_CORE Pipe Stages

Signals in the pipeline will be recognized using the suffix **QnnnH**:  
**‘Q’** -> name of Clock domain.  
**‘nnn’** -> Pipe stage number. Example: 100, 101, 102, 103, 104.  
**‘H’** -> Signal is a “positive edge” sensitive. Signal may change when clock transition Low to High.  
Example:

`SSRV\_MSFF ( OpcodeQ102H , OpcodeQ101H , QClk)

**Q100H) Instruction-Fetch**

PC (Program Counter) sends the current instruction pointer to I\_MEM.  
The instruction memory is synchronized memory, meaning the data will arrive at positive edge of next cycle.

**Q101H) Instruction-Decode**I\_MEM returns the 32’b Instruction.   
(see  
Core will decode the instruction and set control bits accordingly.

The program will use the “Register File” to hold intermediate & temporary variables.  
Register file read from both read ports – combinatorically.  
Note: Register File at entry 0 (RegFileQnnnH[0]) is tied low to XLEN’b0.  
This means all register reads “RegFileQnnnH[0]” will result in read\_data = 0.

**Q102H) Execute**Calculate LOAD/STORE address.   
Calculate the Register-Register/Register-Immediate operations.

**Q103H) Memory-Access**Send the LOAD/STORE address to D\_MEM.  
Opcodes with Register-Register/Register-Immediate operations  
will “passthrough” Q103H without any change.

**Q104H) Write-Back**Set the write back Data to register.   
mux LOAD data vs I/R type ALU output data.

## Other HW Blocks

**Hazard detection unit:**  
Used to detect hazards about to occur.   
will insert a NOP (Bubble) & backpressure the Pipe.

**Forwarding unit:**Due to “Write-Back” to register file takes place in Q104H,   
Reads from register file Q101H may read “old” data.  
Solution is to forward the Q103H/Q104H (post ALU) to Q102H (pre ALU)

# SSRV ISA – Instruction Set Architecture

This Chapter will detail the SSRV, a Subset of the RV32I ISA.  
For the complete RV32I instruction Details, please see the attached “riscv\_rv32i\_spec.pdf”

## RV32I Base Instruction Formats

In the Subset of the base RV32I ISA, there are 3 core instruction formats (R/I/S)  
All are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory.

The RISC-V ISA keeps the source (rs1 and rs2) and destination (rd) registers at the same position in all formats to simplify decoding. Immediates are always sign-extended and are packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity.

**Table 7 - Instruction Formats**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31:25 | 24:20 | 19:15 | 14:12 | 11:7 | 6:0 |  |
| funct7 | rs2 | rs1 | funct3 | rd | opcode | **R-Type** |
| imm[11:0] | | rs1 | funct3 | rd | opcode | **I-Type** |
| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | **S-Type** |

Note: Each immediate subfield is labeled with the bit position (imm[x]) in the immediate value being produced.

Both “I-Type” & “S-Type” Immediates bit length is [XLEN](#XLEN).   
This is achieved using a “sign-extend” on the MSB of the immediate (imm[12]) bits [31:12]

Sign extension always uses inst[31].  
Note: The immediates produced by each of the base instruction formats are labeled to show which instruction bit (inst[y]) produces each bit of the immediate value.

## Integer Computational Instructions

Integer computational instructions operate on XLEN bits of values held in the integer register file.   
Integer computational instructions are either encoded as

* **register-immediate** operations using the I-type format
* **register-register** operations using the R-type format.

The destination is register rd for both register-immediate and register-register instructions.   
No integer computational instructions cause arithmetic exceptions.  
Overflows are ignored, and the low XLEN bits of results are written to the destination rd.

**Table 8 - Register-Immediate Instructions**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| immediate 31:20 | | rs1 19:15 | funct3 14:12 | rd 11:7 | opcode 6:0 |  |
| imm[11:0] | | src | ADD (000) | dest | OP\_IMM | **ADDI** |
| imm[11:0] | | src | SLT (010) | dest | OP\_IMM | **SLTI** |
| imm[11:0] | | src | SLTU (011) | dest | OP\_IMM | **SLTIU** |
| imm[11:0] | | src | XOR (100) | dest | OP\_IMM | **XORI** |
| imm[11:0] | | src | OR (110) | dest | OP\_IMM | **ORI** |
| imm[11:0] | | src | AND (111) | dest | OP\_IMM | **ANDI** |
| 0000000 | imm[4:0] (AKA shamt) | src | SLL (001) | dest | OP\_IMM | **SLLI** |
| 0000000 | imm[4:0] (AKA shamt) | src | SRL (101) | dest | OP\_IMM | **SRLI** |

**ADDI** adds the sign-extended 12-bit immediate to register rs1.   
Arithmetic overflow is ignored, and the result is simply the low XLEN bits of the result.   
Note: “**MV** rd, rs1” assembler pseudo instruction “ADDI rd, rs1, 0”

**SLTI** (set less than immediate) places the value 1 in register rd if register rs1 is less than the sign extended immediate when both are treated as signed numbers, else 0 is written to rd.

**SLTIU** is similar but compares the values as unsigned numbers (i.e., the immediate is first sign-extended to XLEN bits then treated as an unsigned number).   
Note: “**SEQZ** rd, rs” assembler pseudo instruction implemented as “SLTIU rd, rs1, 1”.  
(sets rd to 1 if rs1 equals zero, otherwise sets rd to 0)

**ANDI, ORI, XORI** are logical operations that perform bitwise AND, OR, and XOR on register rs1 and the sign-extended 12-bit immediate and place the result in rd.   
Note: “**NOT** rd, rs“ assembler pseudo instruction implemented as “XORI rd, rs1, -1”

**SLLI** is a logical left shift (zeros are shifted into the lower bits).

**SRLI** is a logical right shift (zeros are shifted into the upper bits).

**Table 9 - NOP instruction**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| immediate 31:20 | rs1 19:15 | funct3 14:12 | rd 11:7 | Opcode 6:0 |  |
| 0 | 0 | ADD | 0 | OP\_IMM | **NOP** |

**NOP** instruction does not change any architecturally visible state, except for advancing the pc and incrementing any applicable performance counters. NOP is encoded as ADDI x0, x0, 0.

**Table 10 - Register-Register Instructions**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| funct7 31:25 | rs2 24:20 | rs1 19:15 | funct3 14:12 | rd 11:7 | opcode 6:0 |  |
| 0000000 | src2 | src1 | ADD (000) | dest | OP | **ADD** |
| 0100000 | src2 | src1 | ADD (000) two's complement ADD | dest | OP | **SUB** |
| 0000000 | src2 | src1 | SLT (010) | dest | OP | **SLT** |
| 0000000 | src2 | src1 | SLTU (011) | dest | OP | **SLTU** |
| 0000000 | src2 | src1 | XOR (100) | dest | OP | **XORI** |
| 0000000 | src2 | src1 | OR (110) | dest | OP | **OR** |
| 0000000 | src2 | src1 | AND (111) | dest | OP | **AND** |
| 0000000 | src2 | src1 | SLL (001) | dest | OP | **SLL** |
| 0000000 | src2 | src1 | SRL (101) | dest | OP | **SRL** |

**ADD** performs the addition of rs1 and rs2.

**SUB** performs the subtraction of rs2 from rs1.

**SLT** perform signed compares, writing 1 to rd if rs1 < rs2, 0 otherwise.

**SLTU** perform unsigned compares, writing 1 to rd if rs1 < rs2, 0 otherwise.   
Note: “**SNEZ** rd, rs” assembler pseudo instruction implemented as “SLTU rd, x0, rs2”  
(sets rd to 1 if rs2 is not equal to zero, otherwise sets rd to zero (). )

**AND**, **OR**, and **XOR** perform bitwise logical operations.

**SLL** perform logical left shift on the Value in register rs1 by the shift amount held in the lower 5 bits of register rs2 (zeros are shifted into the lower bits).

**SRL** perform logical right shifts on the Value in register rs1 by the shift amount held in the lower 5 bits of register rs2 (zeros are shifted into the upper bits).

## Load And Store Instructions

RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers. RV32I provides a 32-bit address space that is byte-addressed. The EEI will define what portions of the address space are legal to access with which instructions (e.g., some addresses might be read only, or support word access only).

Load and store instructions transfer a value between the registers and memory.

**Table 11 - LOAD instruction**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| immediate 31:20 | rs1 19:15 | funct3 14:12 | rd 11:7 | opcode 6:0 |  |
| Offset[11:0] | base | width | dest | LOAD | **LOAD** |

Loads are encoded in the I-type format.  
The effective address is obtained by adding register rs1 to the sign-extended 12-bit offset.  
Loads copy a value from memory to register rd.

The LW instruction loads a 32-bit value from memory into rd.

**Table 12 – STORE instruction**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| immediate  31:25 | rs2  24:20 | rs1  19:15 | funct3  14:12 | immediate  11:7 | opcode  6:0 |  |
| Offset[11:5] | src | base | width | Offset[4:0] | STORE | **STORE** |

Stores are encoded in the S-type format.   
The effective address is obtained by adding register rs1 to the sign-extended 12-bit offset.   
Stores copy the value in register rs2 to memory.

The SW instruction stores 32-bit values from register rs2 to memory.

## 

## Instruction Set Listing – Subset of RV32I

**Table 13 - Instruction Formats**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31:25 | 24:20 | 19:15 | 14:12 | 11:7 | 6:0 |  |
| funct7 | rs2 | rs1 | funct3 | rd | opcode | **R-type** |
| imm[11:0] | | rs1 | funct3 | rd | opcode | **I-type** |
| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | **S-type** |

**Table 14 – SSRV – Subset of the RISC-V Base Instruction Set**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31:25 | 24:20 | 19:15 | 14:12 | 11:7 | 6:0 | instruction |
| imm[11:0] | | rs1 | 010 | rd | 0000011 | **LW** |
| imm[11:5] | rs2 | rs1 | 010 | imm[4:0] | 0100011 | **SW** |
| imm[11:0] | | rs1 | 000 | rd | 0010011 | **ADDI** |
| imm[11:0] | | rs1 | 010 | rd | 0010011 | **SLTI** |
| imm[11:0] | | rs1 | 011 | rd | 0010011 | **SLTIU** |
| imm[11:0] | | rs1 | 100 | rd | 0010011 | **XORI** |
| imm[11:0] | | rs1 | 110 | rd | 0010011 | **ORI** |
| imm[11:0] | | rs1 | 111 | rd | 0010011 | **ANDI** |
| 0000000 | imm[4:0] (shamt) | rs1 | 001 | rd | 0010011 | **SLLI** |
| 0000000 | imm[4:0] (shamt) | rs1 | 101 | rd | 0010011 | **SRLI** |
| 0000000 | rs2 | rs1 | 000 | rd | 0110011 | **ADD** |
| 0100000 | rs2 | rs1 | 000 | rd | 0110011 | **SUB** |
| 0000000 | rs2 | rs1 | 001 | rd | 0110011 | **SLL** |
| 0000000 | rs2 | rs1 | 010 | rd | 0110011 | **SLT** |
| 0000000 | rs2 | rs1 | 011 | rd | 0110011 | **SLTU** |
| 0000000 | rs2 | rs1 | 100 | rd | 0110011 | **XOR** |
| 0000000 | rs2 | rs1 | 101 | rd | 0110011 | **SRL** |
| 0000000 | rs2 | rs1 | 110 | rd | 0110011 | **OR** |
| 0000000 | rs2 | rs1 | 111 | rd | 0110011 | **AND** |
| 000000000001 | | 00000 | 000 | 00000 | 1110011 | **EBREAK** |

# Hazards

## Data Hazard

When the first instruction generates a new register write,  
and the second/third instruction wants to read the new register value, but the register file write hasn’t accrued yet (write accrues in pipe stage Q104)

**Data Hazard Example:**  
(0) ADDI x2, x0, 5 //set x2 with value 32’d5 (x0 Hard-wired zero)  
(4) AND x3, x2, x2 //(0)is in Q102, (4) reading “old” x2 Q101.  
(8) OR x4, x2, x2 //(0)is in Q103, (8) reading “old” x2 Q101.

(C) ADD x5, $2, $2 //(0)is in Q104, (C) reading the Q104 write

## Load Hazard

Very similar to the Data Hazard.  
When the first instruction loads data from D\_MEM to register file,  
And the Second instruction wants to use the loaded data.  
**The problem:**  
We cannot forward the data from Q103 write to Q102 read due to the write data available only at Q104 (after the memory Access).

**Load Hazard Example:**(0) LW x2, 100(x0)//load to x2 the data in address 0x100  
(4) AND x3, x2, x2 //(0)is in Q102, (4) reading “old” x2 Q101.  
(8) OR x4, x2, x2 //(0)is in Q103, (8) reading “old” x2 Q101.   
 //cannot update (4) due to data available only on Q104

# **Example use case of SSRC32I\_CORE within the Fabric:**

For full details please see “SSRV\_TILE\_HAS.pdf”

The ssrv\_core is the main building block of the “TILE\_<#>”.  
Each tile contains the CORE, Memory & IO\_CTRL  
Many Tiles together make a multi-core fabric.



Figure - Fabric Block diagram

## Fabric Execution Flow:

1. Reset stage – Reset the Cores PC and Valid bits in the Fabric IO\_CTRL
2. When exiting the RESET - Freeze the CORE PC at address 0
3. SA (System Agent) will load the program to I\_MEM & the Data set to D\_MEM
4. Start the program by un-freezing the PC and letting it to increment.

# Assembly Programmer Notes

1. When CORE exits ResetQnnnH, the register file data values are unknown (AKA ‘x).  
   Using a simple SW reset routine is advised.  
   Example:

.global \_start

\_start:

/\* clear pipeline \*/

nop

nop

nop

nop

nop

/\* set all registers to zero \*/

mv x1, x0

mv x2, x0

mv x3, x0

…

…

…

mv x30, x0

mv x31, x0

1. They are excluded instruction from the RV32I ISA.   
   Must make sure they are not in-use in the loaded program.  
   **Upper Immediate instructions:**LUI, AUIPC - Not supporting U-Type instruction format

**ALL Jump & branch instructions:**JAL, JALR, BEQ, BNE, BLT, BGE, BLTU, BGEU  
Program must be continues – no Jumps & branches. (no loops, no function calls, no if-else)

**Partial LOAD/STORE instructions:**LB, LH, LBU, LHU, SB, SH  
Only support a full word, 32-bit, Load & Store Instructions from & to memory.

**Shift Right Arithmetic (Sight Extend):**SRAI, SRA - Only support Shift Right Logical – with Zero Extend (SRLI & SRL)

**Memory Ordering Instructions:**FENCE – No need to order I/O and physical access to memory device.

**Environment Call:**ECALL – No service request to executing environment

1. Due to not having any jump/branch instruction supported in the **Subset** rv32i,  
   the program must be sequential.   
   Meaning the program counter will can increment only PC=PC+4.