

# Learn RISC-V CPU Implementation and BSV

(BSV: a High-Level Hardware Design Language)

Rishiyur S. Nikhil

L2: Overview of the RISC-V ISA



# Reminders

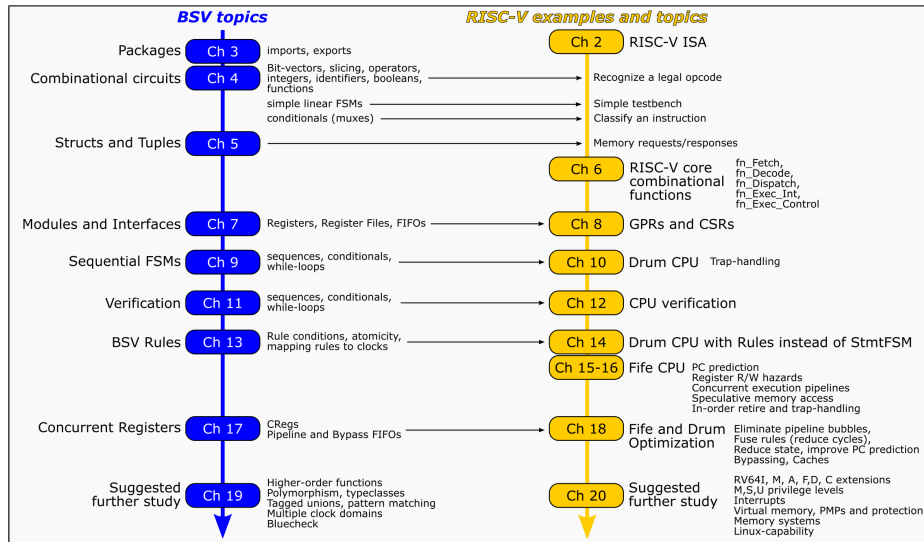
Please git clone or git pull: [https://github.com/rsnikhil/Learn\\_Bluespec\\_and\\_RISCV\\_Design](https://github.com/rsnikhil/Learn_Bluespec_and_RISCV_Design)

```
./Book_BLang_RISCV.pdf
  Slides/
    Slides_01_Intro.pdf
    Slides_02_ISA.pdf
    ...
  Doc/Installing_bsc_Verilator_etc.{adoc,html}
  Exercises/
    Ex_03_B_Top_and_DUT/
    Ex_03_A_Hello_World/
    ...
  Code/
    src_Common/
    src_Drum/
    src_Fife/
    src_Top/
    ...
```

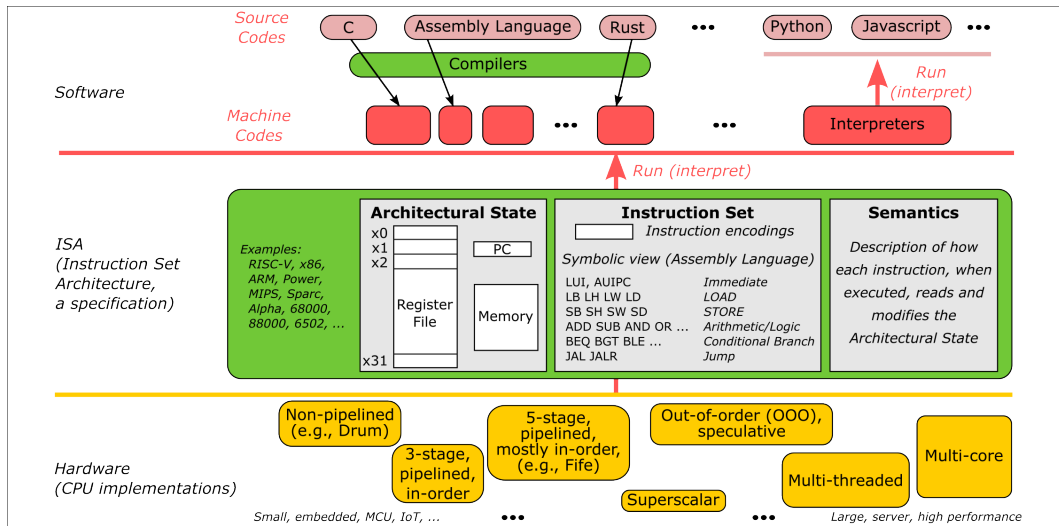
To compile and run the code for exercises, Drum and Fife, please make sure you have installed:

- *bsc* compiler (see <https://github.com/B-Lang-org/bsc>)
- Verilator compiler (see <https://www.verilator.org/>)

# Chapter Roadmap



# What is an ISA?



# Architectural State

The “architectural state” is the state that is visible to instructions. For RV32I, these are:

- The PC (program counter)
- 32 General-Purpose Registers (GPRs, the “register file”)
- Memory (byte-addressed)

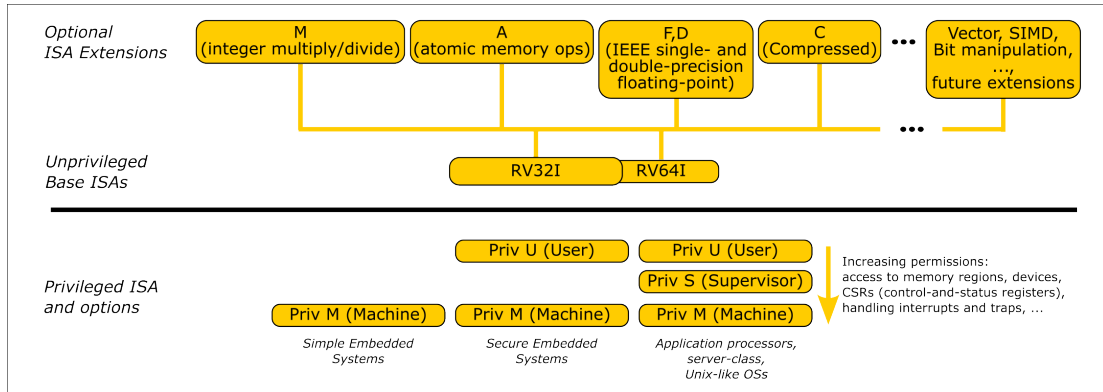
(More architectural state is defined for RV64I, and for most extensions A, F, D, Vector, ...)

The architectural state *does not include* other registers, buffers, FIFOs, memories that may be present in an implementation (they are not visible to instructions).

As such, the architectural state is present in every RISC-V implementation, from tiny CPUs for IoT devices to massive warehouse-scale servers.

Compilers only care about/know about architectural state.

# Modularity of the RISC-V ISA



# RISC-V Instruction Encodings

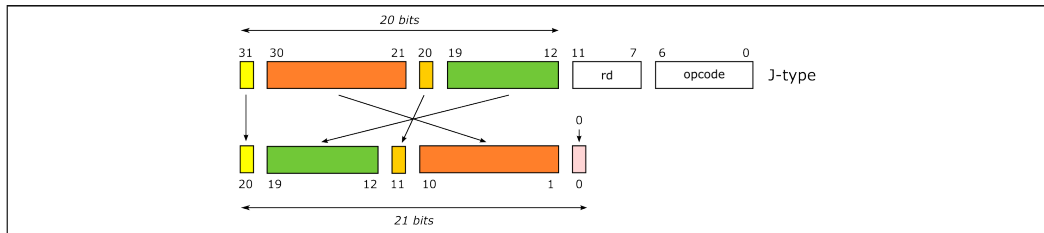
From the RISC-V specification documents:

130

*Volume I: RISC-V Unprivileged ISA V20191213*

| 31                    | 27 | 26 | 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 |
| imm[12 10:5]          |    |    |    | rs2 |    | rs1 |    | funct3 |    | imm[4:1 11] |   | opcode |   | B-type |
| imm[31:12]            |    |    |    |     |    |     |    |        |    | rd          |   | opcode |   | U-type |
| imm[20 10:1 11 19:12] |    |    |    |     |    |     |    |        |    | rd          |   | opcode |   | J-type |

# RISC-V Instruction Encodings; J-type immediates

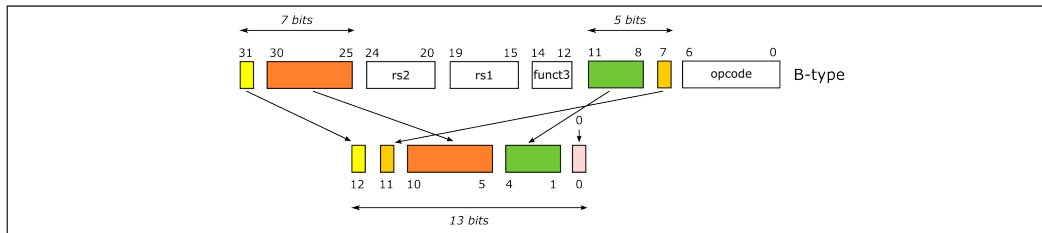


For JAL instruction



# RISC-V Instruction Encodings; B-type immediates

For BRANCH set of instructions (BEQ, BNE, BLT, BGE, BLTU, BGEU)



# RV32I Instructions

RV32I Base Instruction Set

|                    |  |      |       |       |             |         |         |  |   |
|--------------------|--|------|-------|-------|-------------|---------|---------|--|---|
| imm[31:12]         |  |      |       | rd    |             | 0110111 | LUI     | ↑ "load immediate"-kind: to load constant values into a register           |   |
| imm[31:12]         |  |      |       | rd    |             | 0010111 | AUIPC   |  |   |
| imm[20:10:11]19:12 |  |      |       | rd    |             | 1101111 | JAL     | ↑ "jump-and-link"-kind: subroutine calls and returns; distant jumps        |   |
| imm[11:0]          |  |      |       | rd    |             | 1100111 | JALR    |  |   |
| imm[12:10:5]       |  | rs2  | rs1   | 000   | imm[4:1:11] | 1100011 | BEQ     | ↑ "conditional branch"-kind: test and possibly jump up to ~0x1000 distance |   |
| imm[12:10:5]       |  | rs2  | rs1   | 001   | imm[4:1:11] | 1100011 | BNE     |  |   |
| imm[12:10:5]       |  | rs2  | rs1   | 100   | imm[4:1:11] | 1100011 | BLT     |  |   |
| imm[12:10:5]       |  | rs2  | rs1   | 101   | imm[4:1:11] | 1100011 | BGE     |  |   |
| imm[12:10:5]       |  | rs2  | rs1   | 110   | imm[4:1:11] | 1100011 | BLTU    |  |   |
| imm[12:10:5]       |  | rs2  | rs1   | 111   | imm[4:1:11] | 1100011 | BGEU    |  |   |
| imm[11:0]          |  |      | rs1   | 000   | rd          | 0000011 | LB      | ↑ "load data from memory into register" (rs1 and imm specify address)      |   |
| imm[11:0]          |  |      | rs1   | 001   | rd          | 0000011 | LH      |  |   |
| imm[11:0]          |  |      | rs1   | 010   | rd          | 0000011 | LW      |  |   |
| imm[11:0]          |  |      | rs1   | 100   | rd          | 0000011 | LBU     |  |   |
| imm[11:0]          |  |      | rs1   | 101   | rd          | 0000011 | LHU     |  |   |
| imm[11:5]          |  | rs2  | rs1   | 000   | imm[4:0]    | 0100011 | SB      | ↑ "store data from register rs2 to memory" (rs1 and imm specify address)   |   |
| imm[11:5]          |  | rs2  | rs1   | 001   | imm[4:0]    | 0100011 | SH      |  |   |
| imm[11:5]          |  | rs2  | rs1   | 010   | imm[4:0]    | 0100011 | SW      |  |   |
| imm[11:0]          |  |      | rs1   | 000   | rd          | 0010011 | ADDI    | ↑ "integer arithmetic operations (register-immediate)"                     |   |
| 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            |  |      | shamt | rs1   | 001         | rd      | 0010011 | SLLI   | ↑ "integer arithmetic operations (register-register)" |
| 0000000            |  |      | shamt | rs1   | 101         | rd      | 0010011 | SRLI   |   |
| 0100000            |  |      | shamt | rs1   | 101         | rd      | 0010011 | SRAI   |   |
| 0000000            |  |      | rs2   | rs1   | 000         | rd      | 0110011 | ADD  |   |
| 0100000            |  |      | rs2   | rs1   | 000         | rd      | 0110011 | SUB  |   |
| 0000000            |  |      | rs2   | rs1   | 001         | rd      | 0110011 | SLL  | ↑ "integer arithmetic operations (register-register)" |
| 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  |   |
| 0100000            |  |      | rs2   | rs1   | 101         | rd      | 0110011 | SRA  | ↑ "system" operations (ignore FENCE for now)          |
| 0000000            |  |      | rs2   | rs1   | 110         | rd      | 0110011 | OR   |   |
| 0000000            |  |      | rs2   | rs1   | 111         | rd      | 0110011 | AND  |   |
| fm                 |  | pred | succ  | rs1   | 000         | rd      | 0001111 | FENCE  |   |
| 000000000000       |  |      |       | 00000 | 000         | 00000   | 1110011 | ECALL  |   |
| 000000000001       |  |      |       | 00000 | 000         | 00000   | 1110011 | EBREAK   |   |

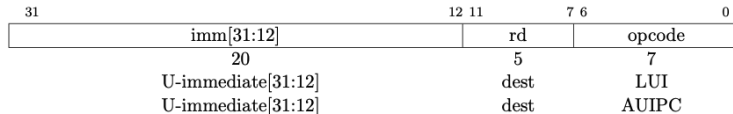
# Example specifications

Excerpt from text of ISA specification document for LUI and AUIPC instructions

*Volume I: RISC-V Unprivileged ISA V20191213*

19

SRLI is a logical right shift (zeros are shifted into the upper bits); and SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits).



LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register *rd*, filling in the lowest 12 bits with zeros.

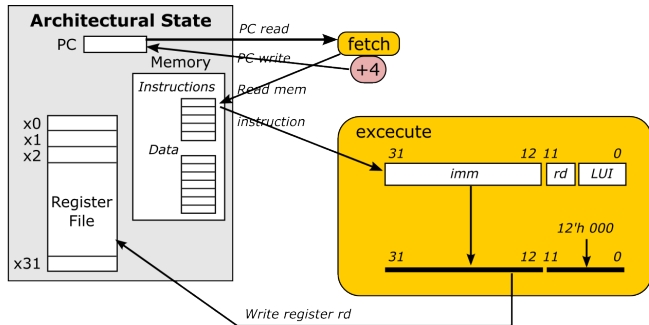
AUIPC (add upper immediate to *pc*) is used to build *pc*-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the address of the AUIPC instruction, then places the result in register *rd*.

# Execution semantics: example

## Execution semantics of LUI instruction

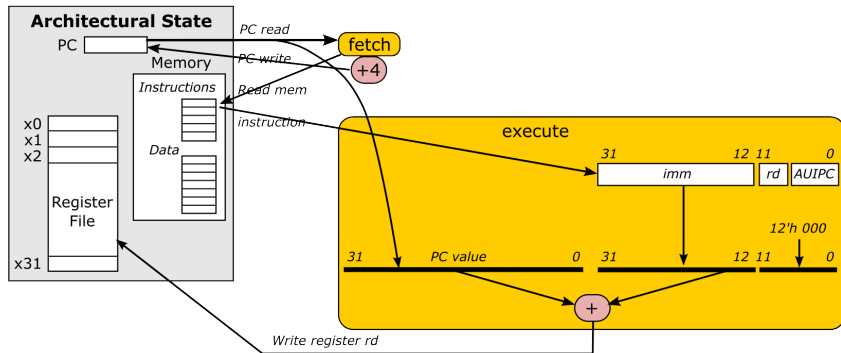
*CPU operation*

loop forever:  
instr = fetch (PC)  
execute (instr)



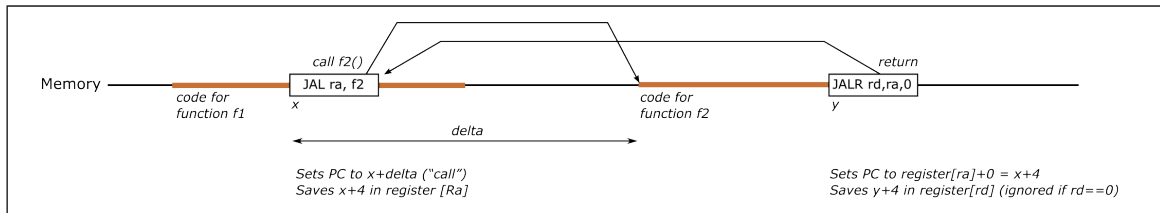
# Execution semantics: example

## Execution semantics of AUIPC instruction

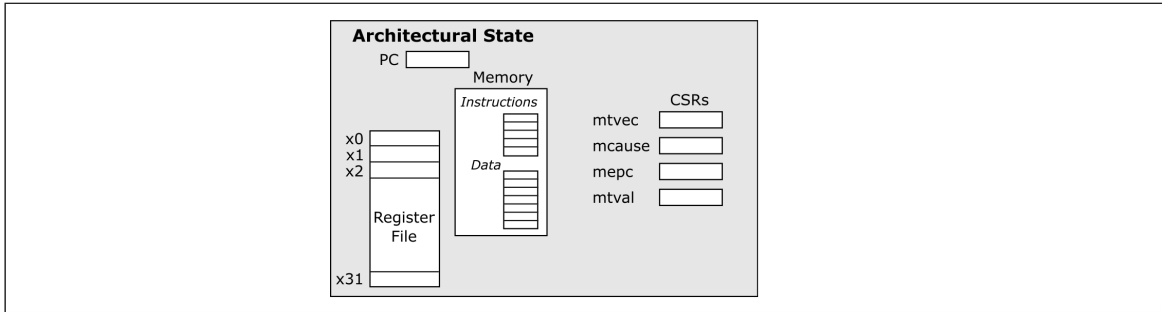


# Execution semantics: example

## Execution semantics of JAL and JALR instructions (unconditional jumps)

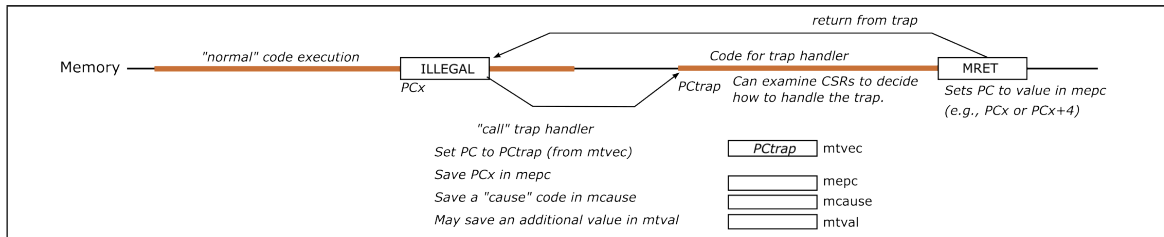


# Control and Status Registers (CSRs) for Trap-Handling



These CSRs are implicitly read and written when taking and returning from a trap.  
These CSRs can be explicitly read and written by CSRRxx instructions.

# Trap and Trap-return flow



There are many possible causes for exceptions and traps;  
the illustration is for an illegal instruction.



# Exception causes

On a trap, a cause-code is written into the `mcause` CSR:

| Exception-Cause code | Description                    |
|----------------------|--------------------------------|
| 0                    | Instruction address misaligned |
| 1                    | Instruction access fault       |
| 2                    | Illegal instruction            |
| 3                    | Breakpoint                     |
| 4                    | Load address misaligned        |
| 5                    | Load access fault              |
| 6                    | Store/AMO address misaligned   |
| 7                    | Store/AMO access fault         |
| ...                  | ...                            |
| 11                   | Environment call M-mode        |
| ...                  | ...                            |

# CSRRxx Instructions

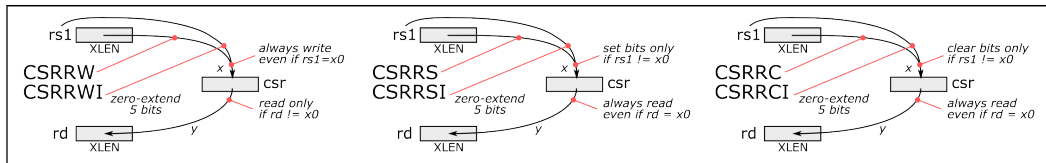
For reading and writing CSRs from RISC-V code  
(from the RISC-V ISA specifications document)

## 9.1 CSR Instructions

All CSR instructions atomically read-modify-write a single CSR, whose CSR specifier is encoded in the 12-bit *csr* field of the instruction held in bits 31–20. The immediate forms use a 5-bit zero-extended immediate encoded in the *rs1* field.

| 31          | 20 19 | 15 14     | 12 11  | 7 6  | 0      |
|-------------|-------|-----------|--------|------|--------|
| csr         |       | rs1       | funct3 | rd   | opcode |
| 12          |       | 5         | 3      | 5    | 7      |
| source/dest |       | source    | CSRRW  | dest | SYSTEM |
| source/dest |       | source    | CSRRS  | dest | SYSTEM |
| source/dest |       | source    | CSRRC  | dest | SYSTEM |
| source/dest |       | uimm[4:0] | CSRRWI | dest | SYSTEM |
| source/dest |       | uimm[4:0] | CSRRSI | dest | SYSTEM |
| source/dest |       | uimm[4:0] | CSRRCI | dest | SYSTEM |

# CSRRxx Instruction Semantics



- They all move potentially data into a CSR, and from a CSR to a GPR
- The non "I" variants take input from GPR[rs1] (unless rs1 is zero)
- The "I" variants use rs1 itself as input (unless rs1 is zero)

# RV64I instructions

From the RISC-V ISA specifications document

Volume I: RISC-V Unprivileged ISA V20191213

131

|           |    |    |    |     |    |     |    |        |    |          |   |        |   |        |
|-----------|----|----|----|-----|----|-----|----|--------|----|----------|---|--------|---|--------|
| 31        | 27 | 26 | 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 |

## RV64I Base Instruction Set (in addition to RV32I)

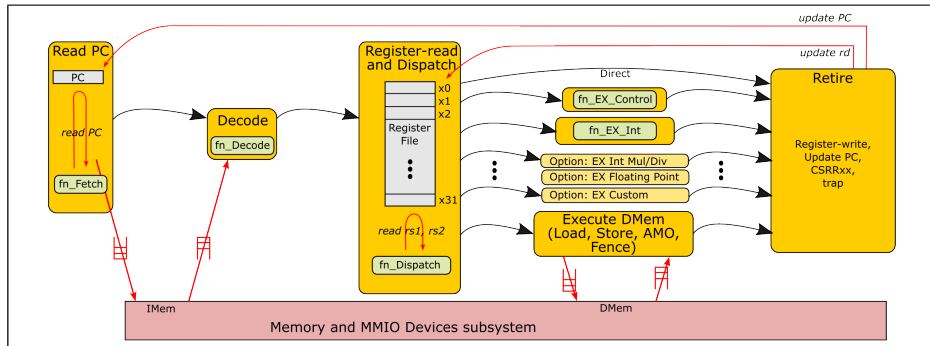
|           |  |  |  |       |  |     |  |     |  |          |  |       |
|-----------|--|--|--|-------|--|-----|--|-----|--|----------|--|-------|
| imm[11:0] |  |  |  | rs1   |  | 110 |  | rd  |  | 0000011  |  | LWU   |
| imm[11:0] |  |  |  | rs1   |  | 011 |  | rd  |  | 0000011  |  | LD    |
| imm[11:5] |  |  |  | rs2   |  | rs1 |  | 011 |  | imm[4:0] |  | SD    |
| 000000    |  |  |  | shamt |  | rs1 |  | 001 |  | rd       |  | SLLI  |
| 000000    |  |  |  | shamt |  | rs1 |  | 101 |  | rd       |  | SRLI  |
| 010000    |  |  |  | shamt |  | rs1 |  | 101 |  | rd       |  | SRAI  |
| imm[11:0] |  |  |  | rs1   |  | 000 |  | rd  |  | 0011011  |  | ADDIW |
| 0000000   |  |  |  | shamt |  | rs1 |  | 001 |  | rd       |  | SLLIW |
| 0000000   |  |  |  | shamt |  | rs1 |  | 101 |  | rd       |  | SRLIW |
| 0100000   |  |  |  | shamt |  | rs1 |  | 101 |  | rd       |  | SRAIW |
| 0000000   |  |  |  | rs2   |  | rs1 |  | 000 |  | rd       |  | ADDW  |
| 0100000   |  |  |  | rs2   |  | rs1 |  | 000 |  | rd       |  | SUBW  |
| 0000000   |  |  |  | rs2   |  | rs1 |  | 001 |  | rd       |  | SLLW  |
| 0000000   |  |  |  | rs2   |  | rs1 |  | 101 |  | rd       |  | SRLW  |
| 0100000   |  |  |  | rs2   |  | rs1 |  | 101 |  | rd       |  | SRAW  |

# RV64I instructions

**Architectural state:** In RV64, the 32 GPRs (General Purpose Registers) and the PC are each 64-bits wide.

- Most of the RV32I instructions have identical RV64I counterparts (here they operate on 64-bit values).
- A few instructions (SLLI, SRLI, SRAI) are slightly different (allowing 6 bits instead of 5 for the shift amount).
- A few instructions are new, to operate on 32-bit values in the 64-bit registers.
  - LWU to move a 32-bit value from memory into a 64-bit register
  - LD to load a 64-bit value from memory to a 64-bit register
  - SD to store a 64-bit value to memory from 64-bit register
  - ADDIW, SLLIW, ... SRAW to operate on 32-bits of 64-bit registers

# Abstract algorithm for interpreting an ISA



This is “abstract” in the sense that it just describes necessary functionality. Different implementations will make choices as to whether or not these functions are pipelined; if pipelined, how many stages; whether or not there are concurrent pipelines; *etc.*

End