Introduction to RISC-V

SK Kang, Computer Architecture and Memory Systems Lab

Table of Contents

What is RISC-V?

- RISC-V Overview
- Types of the RISC-V ISA

RISC-V

- RISC-V [risk-five], is an **open source*** ISA based on **RISC** philosophy

- RISC-V aims...[1]
  - Greater innovation via free-market competition
  - Shared open core designs
  - Affordable processors for more devices

- RISC-V supports...
  - both 32-bit and 64bits address space variants
  - multicore/manycore implementations
  - virtualization and ease hypervisor development
  - optional variable-length instructions to expand encoding space
  - optional dense instructions for performance, code size, and energy efficiency

*Open under BSD liscence

[1] Instruction Sets Should Be Free: The Case For RISC-V, D.Patterson

RISC-V ISA Overview

- RISC-V ISA = base integer ISA (mandatory) + optional extensions (optional)
- Two base integer ISAs (named **[I]**)
  - consists of basic interger computation, load/store, control flow
  - distinguished by width of integer registers and user address space
    32-bit (**RV32I**), 64-bit (**RV64I**), 128-bit (**RV128I**, oncoming)
- Optional extension ISAs
  - Standard extensions
    - **[M]** Mutliplication/Division, **[A]** Atomic read/modify/write, **[F]** floating point, **[D]** double-precision
    - **[G]** All ISAs above (**RV32G** = **RV32IMAFD**)
  - Non-standard extensions
    - Users can add ISAs for user-specific functionalities
    - Non-standard extensions may overwrite standard extensions
- All lab assignments are based on a subset of **RV32I**

RISC-V ISA Example

TAIGA: A NEW RISC-V SOFT-PROCESSOR FRAMEWORK ENABLING HIGH PERFORMANCE CPU ARCHITECTURAL FEATURES

Eric Matthews, Lesley Shannon

In this work, we present Taiga: a RISC-V, 32-bit, soft-processor architecture supporting the RISC-V Multiply/Divide and Atomic operations extensions (RV32IMA) designed to support Linux-based shared-memory systems.

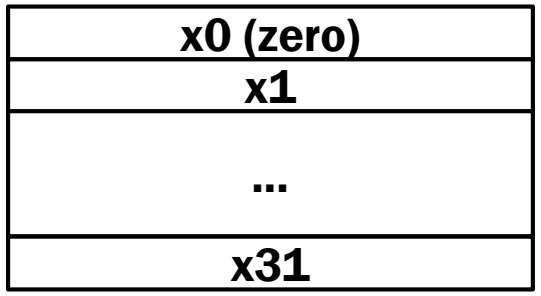
32 32-bit
[I] Integer Base ISA
[M] Multiplication/Division Extension
[A] Atomic Extension

How does the processor see code?

- Programmer's View
- Encoding Levels

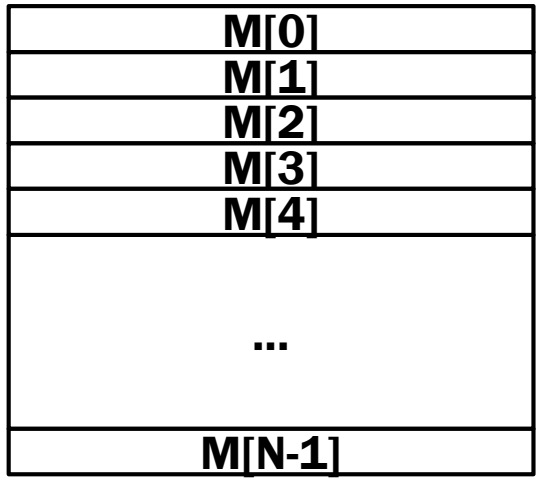
RISC-V Programmer Visible State

- 32 General Purpose Registers
  - Indexed using 5-bit register #
  - 32-bit or 64-bit
  - x0 is always zero (un-writable)
- Program Counter
  - Memory address of execution
  - 32-bit or 64-bit
- Memory
  - 32-bit or 64-bit address
  - Max 4GB or 16EB



General Purpose Register File

Program Counter



Memory

Encoding Levels

- Assembly code
  - What the programmer/compiler sees
  - Consists of operation type, register #, immediates

ADDI 2, 1, 0x10

- ISA
  - What is written in the compiled binary
  - What the processors sees at instruction memory
  - Each line must be multiple of 16 bits (exactly 32 bits in all labs)

0x00080B49

- Decoded values (ex immediates)
  - What the internal hardware sees
  - Usually 32-bit or 64-bit, depending on architecture
  - Some values must be inflated into 32-bits before using

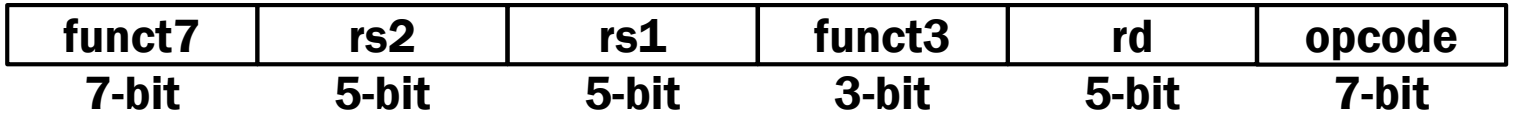
op: OP-IMM
funct: ADDI
rd: 0b00010
rs1: 0b00001
imm: 0x00000010

How are the Instructions Formatted?

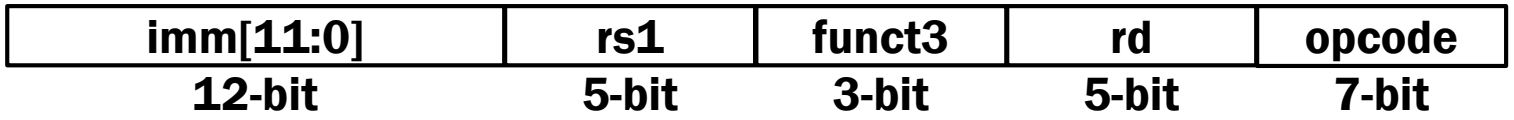
- RV32I Instruction Format
- Immediate Construction

RV32I Instruction Formats

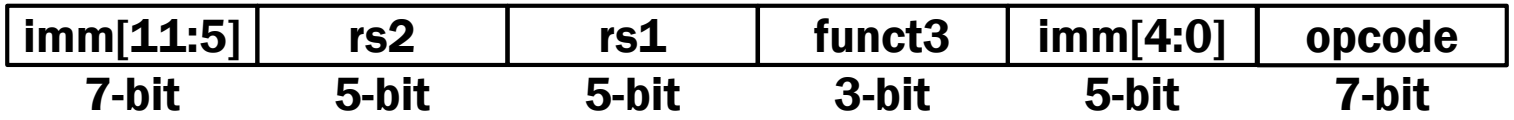
- **R-type**, 3 register operands



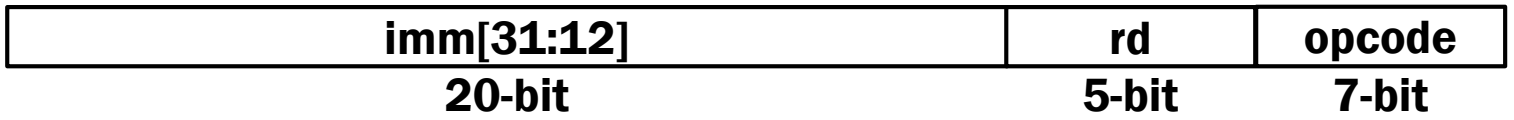
- **I-type**, 2 register operands and 12-bit immediate operand



- **S-type**, 2 register operands and 12-bit immediate operand

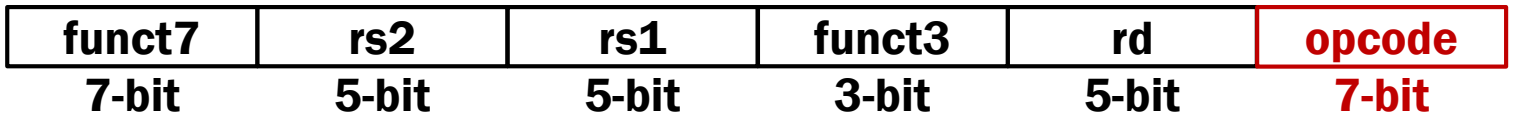


- **U-type**, 1 regitser operand and 20-bit immediate operand

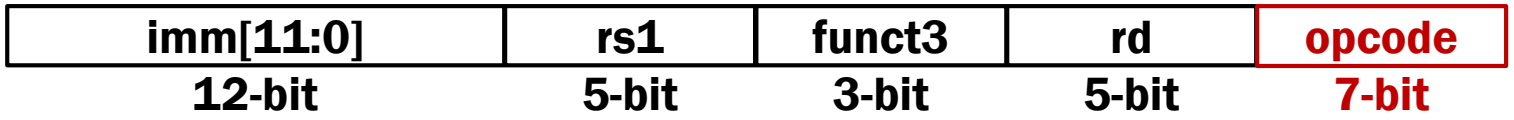


RV32I Intruction Formats

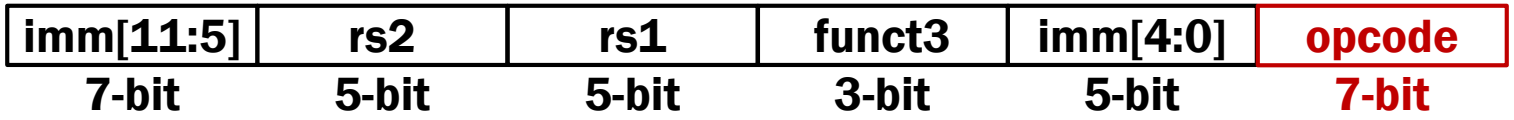
- **R-type**, 3 register operands



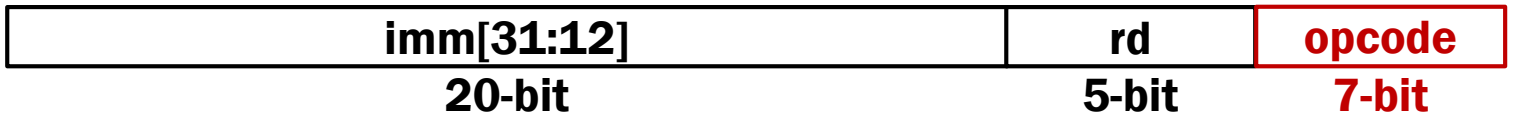
- **I-type**, 2 register operands and 12-bit immediate operand



- **S-type**, 2 register operands and 12-bit immediate operand

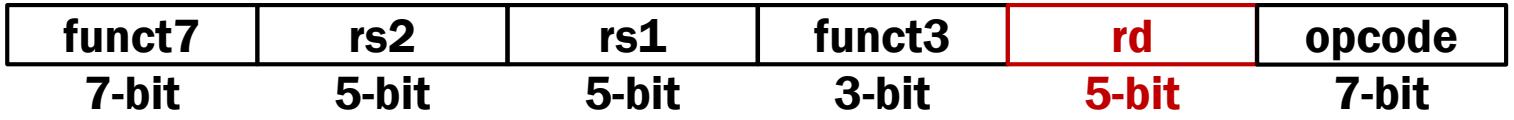


- **U-type**, 1 regitser operand and 20-bit immediate operand

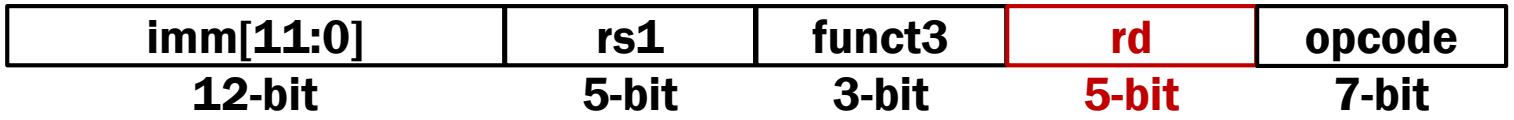


RV32I Intruction Formats

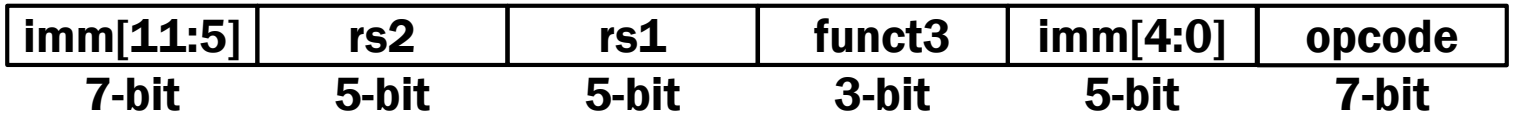
- **R-type**, 3 register operands



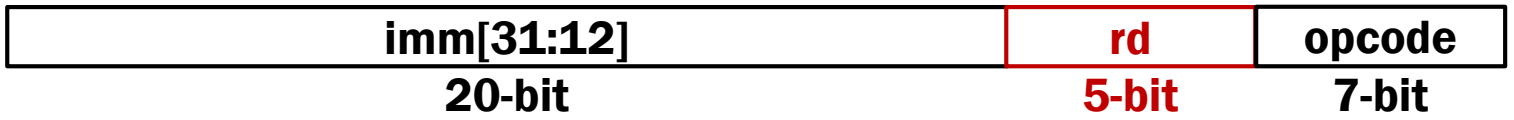
- **I-type**, 2 register operands and 12-bit immediate operand



- **S-type**, 2 register operands and 12-bit immediate operand

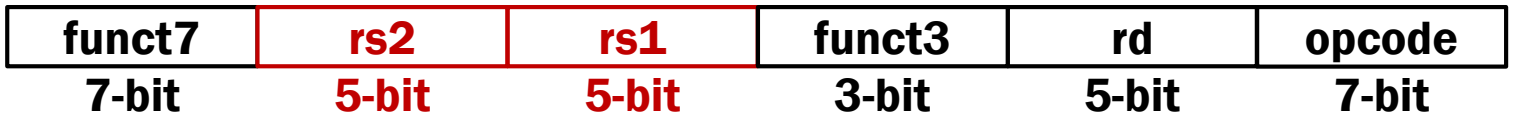


- **U-type**, 1 regitser operand and 20-bit immediate operand

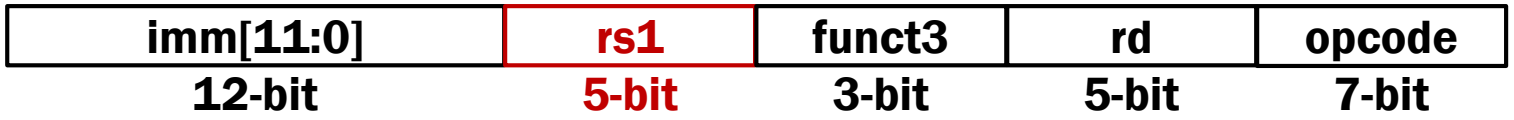


RV32I Intruction Formats

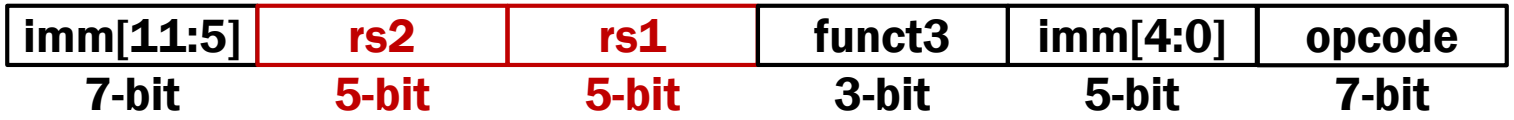
- **R-type**, 3 register operands



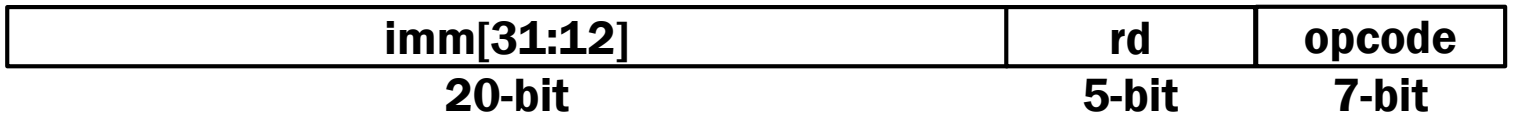
- **I-type**, 2 register operands and 12-bit immediate operand



- **S-type**, 2 register operands and 12-bit immediate operand



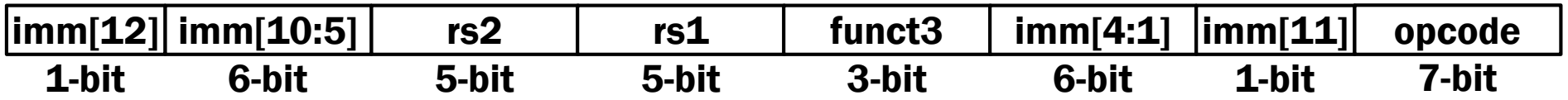
- **U-type**, 1 regitser operand and 20-bit immediate operand



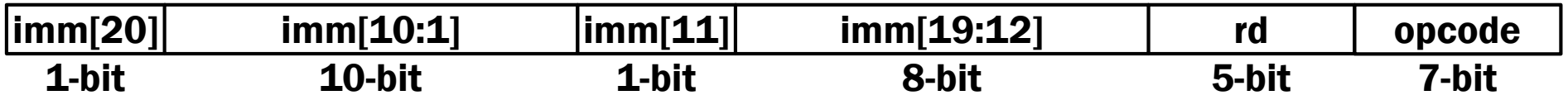
RV32I Intruction Formats - Variants

- Two types of variants (B/J)
  - Same position for opcode, rd, rs1, rs2
  - Immediate is shifted 1 bit

- **B-type**, 2 register operands and 12-bit immediate operand



- **J-type**, 1 register operands and 20-bit immediate operand



RV32I Intruction Formats - Immediate Operands

- The **LUI** instruction is used to build 32-bit constants using immediate value
  - Takes in 20-bit immediate operand, generates 32-bit value

1. Assembly

lui x1, 0x12345

2. Encoded ISA

imm[31:12]	rd	opcode
20-bit	5-bit	7-bit
0x12345 (=0b 0001 0010 0011 0100 0101)	0x1	0110111

2. Decoded Immediate

0001 0010 0011 0100 0101	0000 0000 0000
imm[31:12]	imm[11:0]

What are the Semantics?

- Integer Computational Instructions
- Control/Transfer Instructions
- Load/Store Instructions

Integer Computational Instructions - R-type

- Assembly

ADD rd, rs1, rs2

- Encoded ISA

funct7	rs2	rs1	funct3	rd	opcode
7-bit	5-bit	5-bit	3-bit	5-bit	7-bit
0000000			000		0110011

- Semantics
  - GPR[rd] ← GPR[rs1] + GPR[rs2]
  - PC ← PC + 4
  - Overflow is ignored (no exceptions raised)
- Variations: SUB, AND, OR, XOR, SLL, etc.
  - Variations are determined using funct3 and funct7 fields

Integer Computational Instructions - R-type

- Assembly

AND rd, rs1, rs2

- Encoded ISA

funct7	rs2	rs1	funct3	rd	opcode
7-bit	5-bit	5-bit	3-bit	5-bit	7-bit
0000000			111		0110011

- Semantics
  - GPR[rd] ← GPR[rs1] & GPR[rs2]
  - PC ← PC + 4
  - No exception

Integer Computational Instructions - I-type

- Assembly

ADDI rd, rs1, immediate

- Encoded ISA

imm[11:0]	rs1	funct3	rd	opcode
12-bit	5-bit	3-bit	5-bit	7-bit
		000		0010011

- Semantics
  - GPR[rd] ← GPR[rs1] + *sign-extend*(immediate)
  - PC ← PC + 4
  - Overflow is ignored (no exceptions raised)
- Variations: SUBI, ANDI, ORI, XORI, etc.
  - Variations are determined using funct3 field

Integer Computational Instructions - I-type

- Assembly

ANDI rd, rs1, immediate

- Encoded ISA

imm[11:0]	rs1	funct3	rd	opcode
12-bit	5-bit	3-bit	5-bit	7-bit
		111		0010011

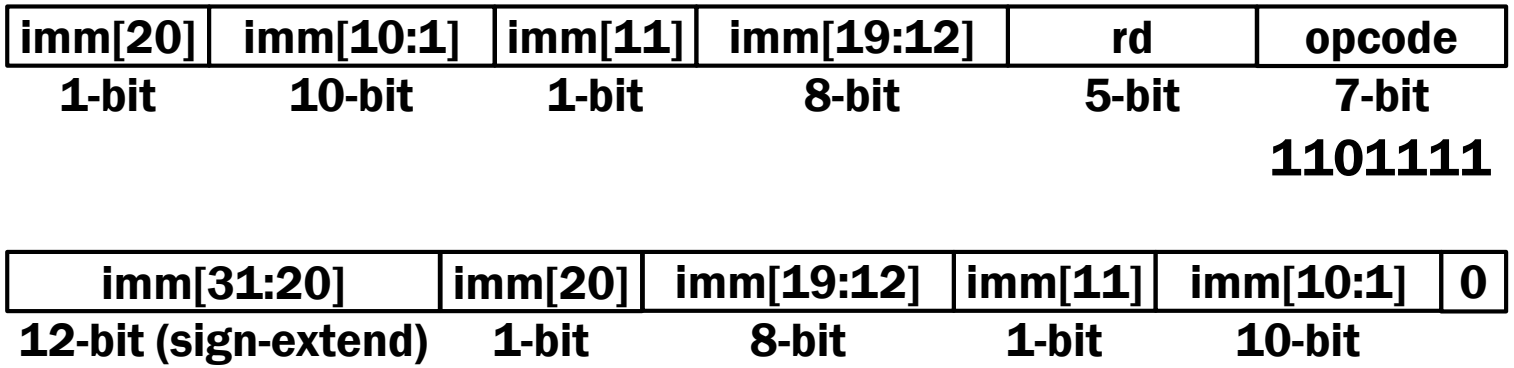
- Semantics
  - GPR[rd] ← GPR[rs1] & *sign-extend*(immediate)
  - PC ← PC + 4
  - No exceptions

Control/Transfer Instructions - Unconditional (J-type)

- Assembly

JAL rd, immediate

- Encoded ISA



- Semantics
  - GPR[rd] ← PC + 4
  - PC ← PC + *sign-extended*(immediate)
  - Can jump to addresses ± 1MiB

Control/Transfer Instructions - Unconditional (I-type)

- Assembly

JALR rd, rs1, immediate

- Encoded ISA

imm[11:0]	rs1	funct3	rd	opcode
12-bit	5-bit	3-bit	5-bit	7-bit
		000		1100111

- Semantics
  - GPR[rd] ← PC + 4
  - PC ← (GPR[rs1] + *sign-extended*(immediate)) & 0xfffffffe
  - Exception if PC is not aligned to 4-bytes

Control/Transfer Instructions - Conditional (B-type)

- Assembly

BEQ rs1, rs2, immediate

- Encoded ISA

imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1-bit	6-bit	5-bit	5-bit	3-bit	6-bit	1-bit	7-bit
				000			1100011

- Semantics
  - if (GPR[rs1] == GPR[rs2])
    PC ← PC + *sign-extended*(immediate)
    else
    PC ← PC + 4
- Variations: BNE, BLT, etc.
  - Variations are determined using funct3 field

Control/Transfer Instructions - Conditional (B-type)

- Assembly

BLT rs1, rs2, immediate

- Encoded ISA

imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1-bit	6-bit	5-bit	5-bit	3-bit	6-bit	1-bit	7-bit
				100			1100011

- Semantics
  - if (GPR[rs1] < GPR[rs2])
    PC ← PC + *sign-extended*(immediate)
    else
    PC ← PC + 4

Load/Store Instructions - Load (I-type)

- Assembly

LD rd, rs1, immediate

- Encoded ISA

imm[11:0]	rs1	funct3	rd	opcode
12-bit	5-bit	3-bit	5-bit	7-bit
		width		0000011

- Semantics
  - GPR[rd] ← MEM[GPR[rs1] + *sign-extended*(immediate)]
  - PC ← PC + 4
- How much to load is determined using funct3 field
  - LW (load 32 bits): 010
  - LH (load 16 bits): 001
  - LB (load 8 bits): 000

Load/Store Instructions - Store (S-type)

- Assembly

SW rs2, immediate(rs1)

- Encoded ISA

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7-bit	5-bit	5-bit	3-bit	5-bit	7-bit
			width		0100011

- Semantics
  - MEM[GPR[rs1] + *sign-extended*(immediate)] ← GPR[rs2]
  - PC ← PC + 4
- How much to load is determined using funct3 field
  - SW (load 32 bits): 010
  - SH (load 16 bits): 001
  - SB (load 8 bits): 000

How to find opcodes? funct fields? sementics?

- Opcodes/funct fields for all operations are on Chapter 19 ("RV32/64G Instruction Set Listings", pg 103)
- Sementics of all RV32 operations are on Chaper 2.1-2.6 ("RV32I Base Integer Instruction Set", pg 9-19)

- Note that all labs does **not** cover fencing nor control status operations

0000	pred	su	00000	000	000	0001111	FENCE
0000	0000	0000	00000	001	000	0001111	FENCE.I
000000000000			000	000	00000	1110011	ECALL
000000000001			000	000	00000	1110011	EBREAK
csr			rs1	001	rd	1110011	CSRRW
csr			rs1	010	rd	1110011	CSRRS
csr			rs1	011	rd	1110011	CSRRC
csr			zimm	101	rd	1110011	CSRRWI
csr			zimm	110	rd	1110011	CSRRSI
csr			zimm	111	rd	1110011	CSRRCI

Thank you