RISC-V User-level ISA Cheat Sheet

Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)

Systems Software & Architecture Lab.

Seoul National University

Spring 2020

The RISC-V Instruction Set

- A completely open ISA that is freely available to academia and industry
- Fifth RISC ISA design developed at UC Berkeley
  - RISC-I (1981), RISC-II (1983), SOAR (1984), SPUR (1989), and RISC-V (2010)
- Now managed by the RISC-V Foundation (http://riscv.org)
- Typical of many modern ISAs
  - See RISC-V Reference Card (or Green Card)
- Similar ISAs have a large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …

RISC-V ISAs

- Three base integer ISAs, one per address width
  - RV32I, RV64I, RV128I
  - RV32I: Only 40 instructions defined
  - RV32E: Reduced version of RV32I with 16 registers for embedded systems
- Standard extensions
  - Standard RISC encoding in a fixed 32-bit instruction format
  - C extension offers shorter 16-bit versions of common 32-bit RISC-V instructions (can be intermixed with 32-bit instructions)

Name	Extension
M	Integer Multiply/Divide
A	Atomic Instructions
F	Single-precision FP
D	Double-precision FP
G	General-purpose (= IMAFD)
Q	Quad-precision FP
C	Compressed Instructions

RISC-V Registers

#	Name	Usage
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporaries (Caller-save registers)
x6	t1	
x7	t2	
x8	s0/fp	Saved register / Frame pointer
x9	s1	Saved register
x10	a0	Function arguments / Return values
x11	a1	
x12	a2	Function arguments
x13	a3	
x14	a4	
x15	a5	

#	Name	Usage
x16	a6	Function arguments
x17	a7	
x18	s2	Saved registers (Callee-save registers)
x19	s3	
x20	s4	
x21	s5	
x22	s6	
x23	s7	
x24	s8	
x25	s9	
x26	s10	
x27	s11	
x28	t3	Temporaries (Caller-save registers)
x29	t4	
x30	t5	
x31	t6	
	pc	Program counter

Arithmetic Operations

Instruction	Type	Example	Meaning
Add	**R**	`add rd, rs1, rs2`	`R[rd] = R[rs1] + R[rs2]`
Subtract	**R**	`sub rd, rs1, rs2`	`R[rd] = R[rs1] – R[rs2]`
Add immediate	**I**	`addi rd, rs1, imm12`	`R[rd] = R[rs1] + SignExt(imm12)`
Set less than	**R**	`slt rd, rs1, rs2`	`R[rd] = (R[rs1] < R[rs2])? 1 : 0`
Set less than immediate	**I**	`slti rd, rs1, imm12`	`R[rd] = (R[rs1] < SignExt(imm12))? 1 : 0`
Set less than unsigned	**R**	`sltu rd, rs1, rs2`	`R[rd] = (R[rs1]` $<_u$ `R[rs2])? 1 : 0`
Set less than immediate unsigned	**I**	`sltiu rd, rs1, imm12`	`R[rd] = (R[rs1]` $<_u$ `SignExt(imm12))? 1 : 0`
Load upper immediate	**U**	`lui rd, imm20`	`R[rd] = SignExt(imm20 << 12)`
Add upper immediate to PC	**U**	`auipc rd, imm20`	`R[rd] = PC + SignExt(imm20 << 12)`

Logical Operations

Instruction	Type	Example	Meaning
AND	**R**	`and rd, rs1, rs2`	`R[rd] = R[rs1] & R[rs2]`
OR	**R**	`or rd, rs1, rs2`	`R[rd] = R[rs1] | R[rs2]`
XOR	**R**	`xor rd, rs1, rs2`	`R[rd] = R[rs1] ^ R[rs2]`
AND immediate	**I**	`andi rd, rs1, imm12`	`R[rd] = R[rs1] & SignExt(imm12)`
OR immediate	**I**	`ori rd, rs1, imm12`	`R[rd] = R[rs1] | SignExt(imm12)`
XOR immediate	**I**	`xori rd, rs1, imm12`	`R[rd] = R[rs1] ^ SignExt(imm12)`
Shift left logical	**R**	`sll rd, rs1, rs2`	`R[rd] = R[rs1] << R[rs2]`
Shift right logical	**R**	`srl rd, rs1, rs2`	`R[rd] = R[rs1] >> R[rs2]` *(logical)*
Shift right arithmetic	**R**	`sra rd, rs1, rs2`	`R[rd] = R[rs1] >> R[rs2]` *(arithmetic)*
Shift left logical immediate	**I**	`slli rd, rs1, shamt`	`R[rd] = R[rs1] << shamt`
Shift right logical imm.	**I**	`srli rd, rs1, shamt`	`R[rd] = R[rs1] >> shamt` *(logical)*
Shift right arithmetic immediate	**I**	`srai rd, rs1, shamt`	`R[rd] = R[rs1] >> shamt` *(arithmetic)*

Data Transfer Operations

Instruction	Type	Example	Meaning
Load doubleword	**I**	`ld rd, imm12(rs1)`	`R[rd] = Mem8[R[rs1] + SignExt(imm12)]`
Load word	**I**	`lw rd, imm12(rs1)`	`R[rd] = SignExt(Mem4[R[rs1] + SignExt(imm12)])`
Load halfword	**I**	`lh rd, imm12(rs1)`	`R[rd] = SignExt(Mem2[R[rs1] + SignExt(imm12)])`
Load byte	**I**	`lb rd, imm12(rs1)`	`R[rd] = SignExt(Mem1[R[rs1] + SignExt(imm12)])`
Load word unsigned	**I**	`lwu rd, imm12(rs1)`	`R[rd] = ZeroExt(Mem4[R[rs1] + SignExt(imm12)])`
Load halfword unsigned	**I**	`lhu rd, imm12(rs1)`	`R[rd] = ZeroExt(Mem2[R[rs1] + SignExt(imm12)])`
Load byte unsigned	**I**	`lbu rd, imm12(rs1)`	`R[rd] = ZeroExt(Mem1[R[rs1] + SignExt(imm12)])`
Store doubleword	**S**	`sd rs2, imm12(rs1)`	`Mem8[R[rs1] + SignExt(imm12)] = R[rs2]`
Store word	**S**	`sw rs2, imm12(rs1)`	`Mem4[R[rs1] + SignExt(imm12)] = R[rs2](31:0)`
Store halfword	**S**	`sh rs2, imm12(rs1)`	`Mem2[R[rs1] + SignExt(imm12)] = R[rs2](15:0)`
Store byte	**S**	`sb rs2, imm12(rs1)`	`Mem1[R[rs1] + SignExt(imm12)] = R[rs2](7:0)`

Control Transfer Instructions

Instruction	Type	Example	Meaning
Branch equal	**SB**	`beq rs1, rs2, imm12`	`if (R[rs1] == R[rs2])` `  pc = pc + SignExt(imm12 << 1)`
Branch not equal	**SB**	`bne rs1, rs2, imm12`	`if (R[rs1] != R[rs2])` `  pc = pc + SignExt(imm12 << 1)`
Branch greater than or equal	**SB**	`bge rs1, rs2, imm12`	`if (R[rs1] >= R[rs2])` `  pc = pc + SignExt(imm12 << 1)`
Branch greater than or equal unsigned	**SB**	`bgeu rs1, rs2, imm12`	`if (R[rs1] >=u R[rs2])` `  pc = pc + SignExt(imm12 << 1)`
Branch less than	**SB**	`blt rs1, rs2, imm12`	`if (R[rs1] < R[rs2])` `  pc = pc + SignExt(imm12 << 1)`
Branch less than unsigned	**SB**	`bltu rs1, rs2, imm12`	`if (R[rs1] <u R[rs2])` `  pc = pc + SignExt(imm12 << 1)`
Jump and link	**UJ**	`jal rd, imm20`	`R[rd] = PC + 4` `PC = PC + SignExt(imm20 << 1)`
Jump and link register	**I**	`jalr rd, imm12(rs1)`	`R[rd] = PC + 4` `PC = (R[rs1] + SignExt(imm12)) & (~1)`

Assembler Pseudo-Instructions

Pseudo-instruction	Base instruction(s)	Meaning
`li rd, imm`	`addi rd, x0, imm`	Load immediate
`la rd, symbol`	`auipc rd, D[31:12]+D[11]` `addi rd, rd, D[11:0]`	Load absolute address where D = `symbol` – `pc`
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`bgt{u} rs, rt, offset`	`blt{u} rt, rs, offset`	Branch if > (u: unsigned)
`ble{u} rs, rt, offset`	`bge{u} rt, rs, offset`	Branch if ≥ (u: unsigned)
`b{eq|ne}z rs, offset`	`b{eq|ne} rs, x0, offset`	Branch if { = |≠ }
`b{ge|lt}z rs, offset`	`b{ge|lt} rs, x0, offset`	Branch if { ≥ | < }
`b{le|gt}z rs, offset`	`b{ge|lt} x0, rs, offset`	Branch if { ≤ | > }
`j offset`	`jal x0, offset`	Unconditional jump
`call offset`	`jal ra, offset`	Call subroutine (near)
`ret`	`jalr x0, 0(ra)`	Return from subroutine
`nop`	`addi x0, x0, 0`	No operation