Imperial College London

Lecture 6 Instruction Set Architecture (RISC-V ISA)

Peter Cheung
Imperial College London

URL: www.ee.imperial.ac.uk/pcheung/teaching/EE2_CAS/
E-mail: p.cheung@imperial.ac.uk

RISC-V

- Developed by Krste Asanovic, David Patterson and colleagues at UC Berkeley in 2010
- First widely accepted **open-source** computer architecture
- Underlying design principles:
  1. **Simplicity favours regularity**
  2. **Make the common case fast**
  3. **Smaller is faster**
  4. **Good design demands good compromises**

Design Principles

Principle 1: Simplicity favors regularity

- Consistent instruction format
- Same number of operands (two sources and one destination)
- Easier to encode and handle in hardware

Principle 2: Make the common case fast

- RISC-V includes only simple, commonly used instructions
- Hardware to decode and execute instructions can be simple, small, and fast
- More complex instructions (that are less common) performed using multiple simple instructions
- RISC-V is a ***reduced instruction set computer*** **(RISC)**, with a small number of simple instructions
- Other architectures, such as Intel's x86, are ***complex instruction set computers*** **(CISC)**

Principle 3: Smaller is Faster

H&H p301-303

Instructions: Addition & Subtraction

C Code

```
a = b + c;
a = b - c;
```

RISC-V assembly code

```
add a, b, c
sub a, b, c
```

- **`Add/sub:`** mnemonic indicates operation to perform
- **`b, c:`** source operands (on which the operation is performed)
- **`a:`** destination operand (to which the result is written)

More complex code is handled by multiple RISC-V instructions.

C Code

```
a = b + c - d;
```

RISC-V assembly code

```
add t, b, c   # t = b + c
sub a, t, d   # a = t - d
```

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*"
by Sarah Harris and David Harris (H&H),

H&H p303

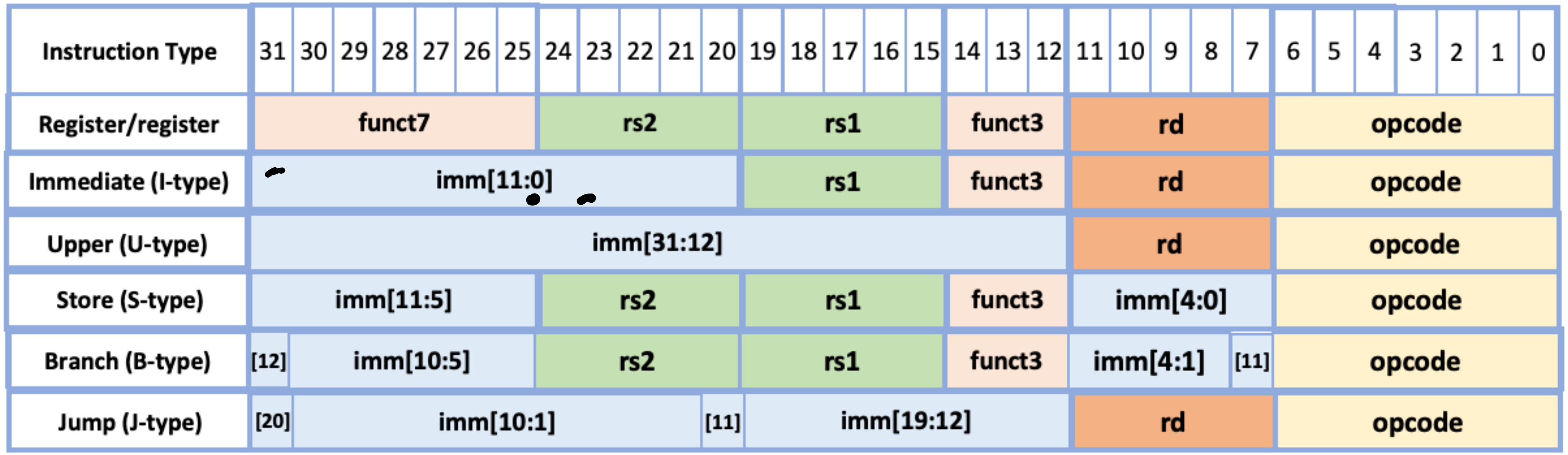
RISC-V Operands

- **Operand location:** physical location in computer
  - Registers
  - Memory
  - Constants (also called *immediates*)

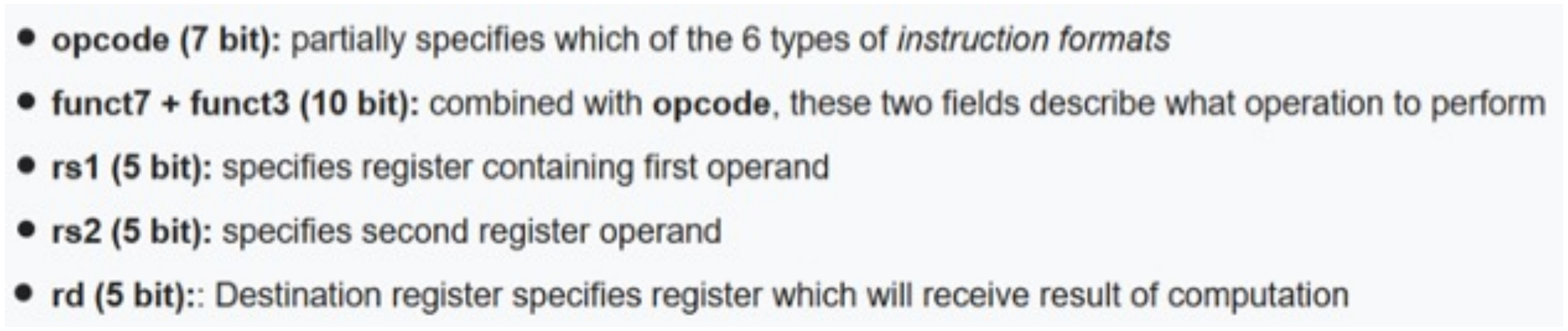
- RISC-V has 32 32-bit registers
- Registers are faster than memory
- RISC-V called “32-bit architecture” because it operates on 32-bit data

H&H p304

32-bit RISC-V Instruction Types



Instruction Type	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate (I-type)	imm[11:0]												rs1					funct3			rd					opcode						
Upper (U-type)	imm[31:12]																				rd					opcode						
Store (S-type)	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch (B-type)	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump (J-type)	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						



- **opcode (7 bit):** partially specifies which of the 6 types of *instruction formats*
- **funct7 + funct3 (10 bit):** combined with **opcode**, these two fields describe what operation to perform
- **rs1 (5 bit):** specifies register containing first operand
- **rs2 (5 bit):** specifies second register operand
- **rd (5 bit):**: Destination register specifies register which will receive result of computation

RISC-V Registers

Name	Register Number	Usage
`zero`	x0	Constant value 0
`ra`	x1	Return address
`sp`	x2	Stack pointer
`gp`	x3	Global pointer
`tp`	x4	Thread pointer
`t0-2`	x5-7	Temporaries
`s0/fp`	x8	Saved register / Frame pointer
`s1`	x9	Saved register
`a0-1`	x10-11	Function arguments / return values
`a2-7`	x12-17	Function arguments
`s2-11`	x18-27	Saved registers
`t3-6`	x28-31	Temporaries

H&H p305

RISC-V operand from Registers

Name	Register Number	Usage
`s0/fp`	x8	Saved register / Frame pointer
`s1`	x9	Saved register
`s2-11`	x18-27	Saved registers

C Code

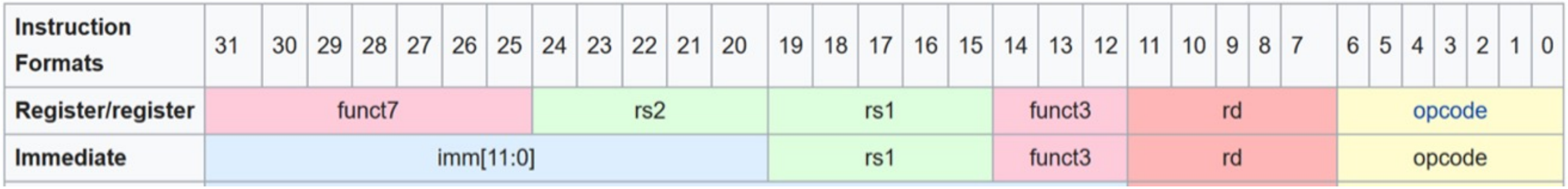
```
a = b + c;


a = b + 6;
```

RISC-V assembly code

```
# s0 = a, s1 = b, s2 = c
add s0, s1, s2

# s0 = a, s1 = b
addi s0, s1, 6
```

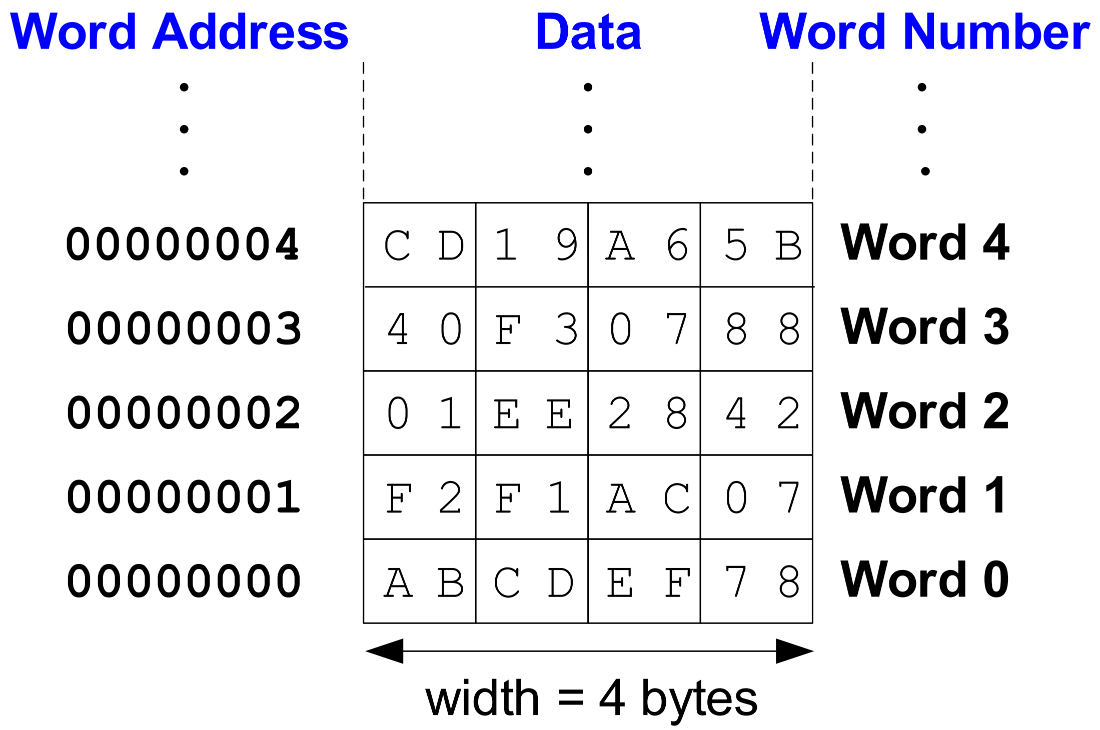


Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						

H&H p305

RISC-V operands from memory

- Each 32-bit data word has a unique address



Word Address	Data	Word Number
⋮	⋮	⋮
00000004	C D 1 9 A 6 5 B	Word 4
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

RISC-V uses **byte-addressable** memory (i.e. byte has a unique address), so each 32-bit word uses 4 byte addresses

H&H p307

RISC-V Byte-addressable Memory

- Each data byte has a unique address
- Load/store words or single bytes: load byte (lb) and store byte (sb)
- 32-bit word = 4 bytes, so word address **increments by 4**

Byte Address				Word Address	Data				Word Number
⋮				⋮	⋮				⋮
13	12	11	10	00000010	CD	19	A6	5B	Word 4
F	E	D	C	0000000C	40	F3	07	88	Word 3
B	A	9	8	00000008	01	EE	28	42	Word 2
7	6	5	4	00000004	F2	F1	AC	07	Word 1
3	2	1	0	00000000	AB	CD	EF	78	Word 0
MSB			LSB						

width = 4 bytes

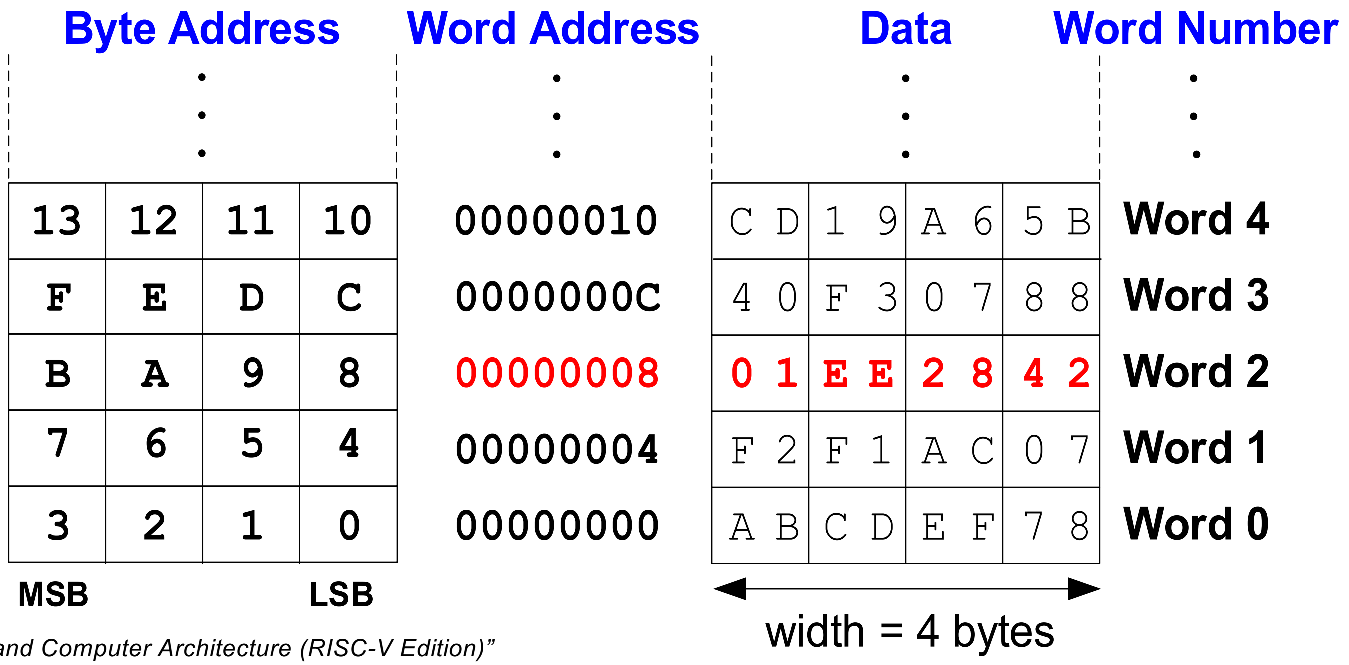
Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*"
by Sarah Harris and David Harris (H&H),

Reading Byte-Addressable Memory

- **Example:** Load a word of data at memory address 8 into s3.
- s3 holds the value 0x1EE2842 after load

RISC-V assembly code

```
lw s3, 8(zero)   # read word at address 8 into s3
```



Byte Address				Word Address	Data	Word Number
⋮				⋮	⋮	⋮
13	12	11	10	00000010	C D 1 9 A 6 5 B	Word 4
F	E	D	C	0000000C	4 0 F 3 0 7 8 8	Word 3
B	A	9	8	00000008	0 1 E E 2 8 4 2	Word 2
7	6	5	4	00000004	F 2 F 1 A C 0 7	Word 1
3	2	1	0	00000000	A B C D E F 7 8	Word 0

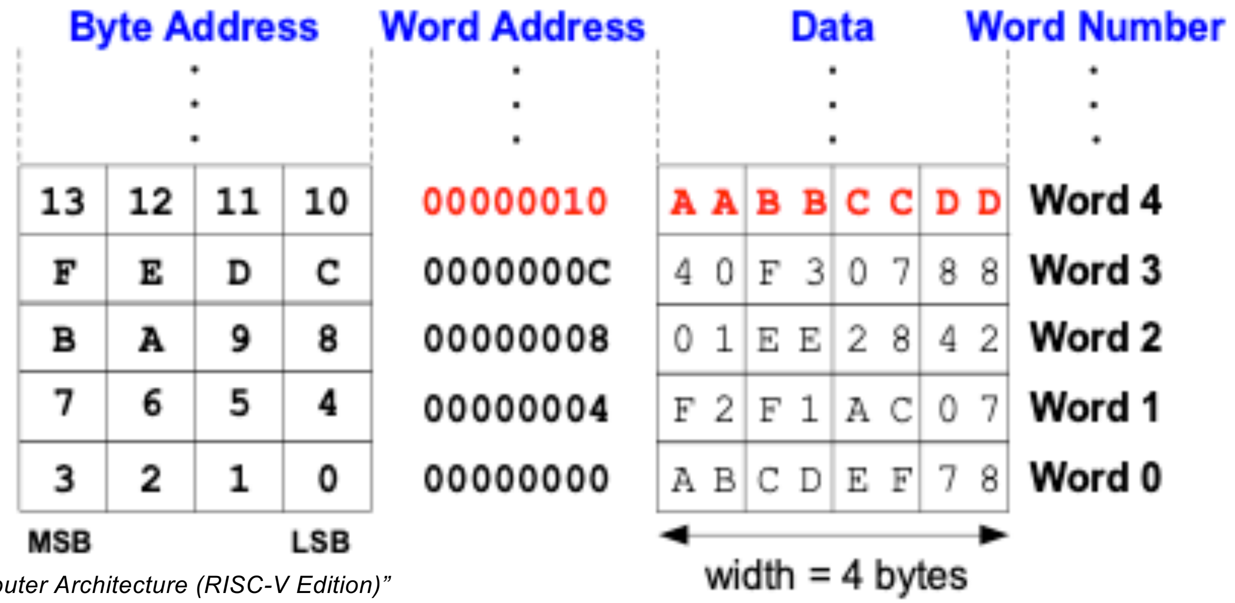
Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

Writing Byte-Addressable Memory

- **Example:** store the value held in t7 into memory address 0x10 (16)
  – if t7 holds the value 0xAABBCCDD, then after the sw completes, word 4 (at address 0x10) in memory will contain that value

RISC-V assembly code

```
sw t7, 0x10(zero)   # write t7 into address 16
```



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

RISC-V: Operands from Constants

- 12-bit signed constants (immediates) using addi:

C Code

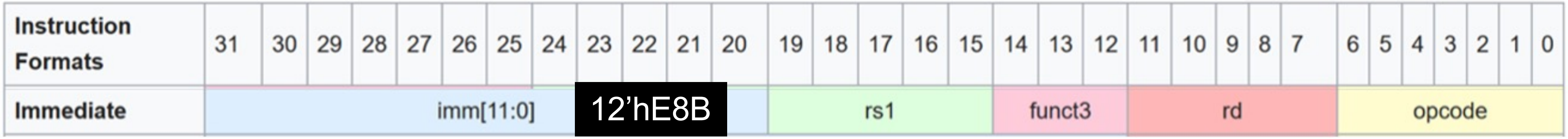
```
// int is a 32-bit signed word
int a = -372;
int b = a + 6;
```

RISC-V assembly code

```
# s0 = a, s1 = b
addi s0, zero, -372
addi s1, s0, 6
```

372 = 12'h174 = 12'b0001_0111_0100
-372 = 12'b1110_1000_1100 = 12'hE8B

- Form 32-bit constant using **sign extension**



Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	imm[11:0] 12'hE8B												rs1					funct3			rd					opcode						

Any immediate that needs **more than 12 bits** cannot use this method.

H&H p306

RISC-V: Operand with 32-bit Constants

- Use load upper immediate (lui) and addi
- lui: puts an immediate in the upper 20 bits of destination register and 0's in lower 12 bits

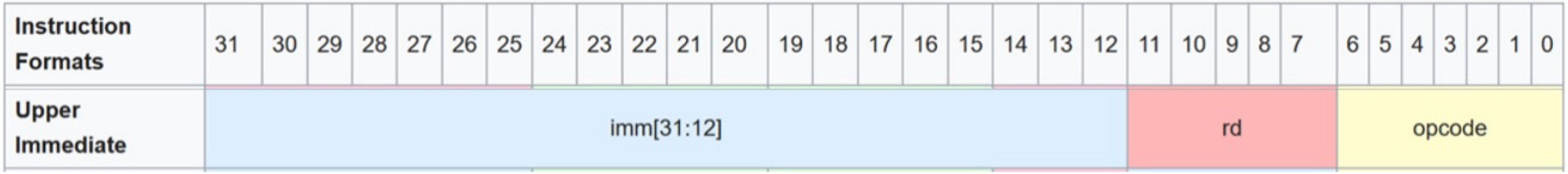
C Code

```
int a = 0xFEDC8765;
```

RISC-V assembly code

```
# s0 = a
lui  s0, 0xFEDC8
addi s0, s0, 0x765
```

Remember that addi **sign-extends** its 12-bit immediate constant



Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Upper Immediate	imm[31:12]																				rd					opcode						

H&H p306

RISC-V: 32-bit Constants (bit 11 is 1)

- If **bit 11** of the constant is **1**, increment upper 20 bits by **1** in `lui`

C Code

```
int a = 0xFEDC8EAB;
```

Note: -341 = 0xEAB

RISC-V assembly code

```
# s0 = a
lui  s0, 0xFEDC9     # s0 = 0xFEDC9000
addi s0, s0, -341    # s0 = 0xFEDC9000 + 0xFFFFFEAB

                     #    = 0xFEDC8EAB
```

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*"
by Sarah Harris and David Harris (H&H),

RISC-V: Psuedoinstruction

- Load immediate 32-bit word is tedious.
- Pseudoinstruction – Assembler program translate “Load Immediate” instruction “li” to two real RISC-V instructions: ”lui” and ”addi”

C Code

```
int a = 0xFEDC8EAB;
```

Note: -341 = 0xEAB

RISC-V pseudoinstructions

```
# s0 = a
li   s0, 0xFEDC8EAB
```

RISC-V real instructions

```
# s0 = a
lui  s0, 0xFEDC9
addi s0, s0, 0xEAB
```

- RISC-V has many pseudoinstructions (see later lectures)

RISC-V: Addressing Modes

How do we address the operands?

- Register Only
- Immediate
- Base Addressing
- PC-Relative

Register Only

- Operands found in registers
  - **Example:** add s0, t2, t3
  - **Example:** sub t6, s1, 0

Immediate

- 12-bit signed immediate used as an operand
  - **Example:** addi s4, t5, -73
  - **Example:** ori t3, t7, 0xFF

H&H p340

RISC-V: Base + Offset Addressing

Base Addressing

- Loads and Stores
- Address of operand is:

  base address + immediate

  - **Example:** lw s4, 72(zero)
    - address = 0 + 72

  - **Example:** sw t2, -25(t1)
    - address =   t1 - 25

RISC-V: PC-relative Addressing

PC-Relative Addressing: branches and jal

Example:

```
Address              Instruction
0x354       L1:      addi s1, s1, 1
0x358                sub  t0, t1, s7
...                  ...
0xEB0                bne  s8, s9, L1
```

The label is (0xEB0-0x354) = 0xB5C (**2908**) instructions **before** bne

RISC-V: Instruction coding for Branch offset

Assembly

```
 beq s8,   s9,   L1
(beq x24, x25,   L1)
```

Relative offset = -2908

$imm_{12:0}$ = -2908	1	0	1	0	0	1	0	1	0	0	1	0	$\cancel{0}$
bit number	12	11	10	9	8	7	6	5	4	3	2	1	0

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						

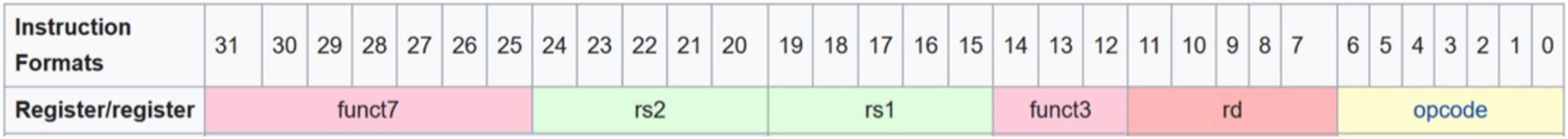
Field Values

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
1100 101	24	25	1	0010 0	99
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

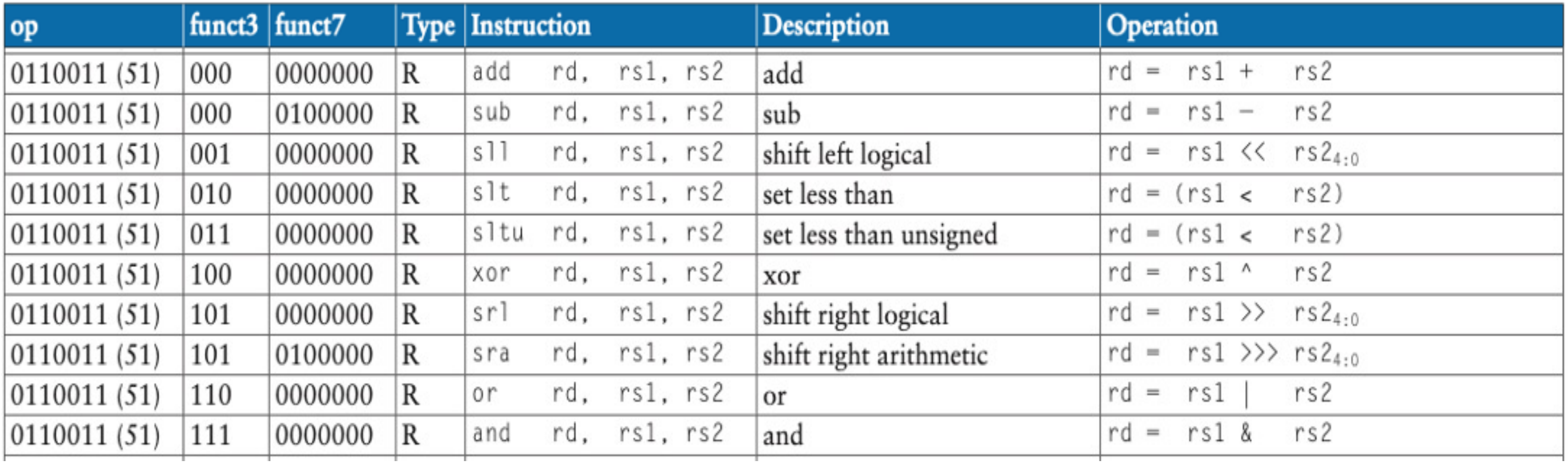
Machine Code

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	
1100 101	11000	11001	001	0010 0	110 0011	(0xCB8C9263)
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

R-type Instructions: 3 register instructions

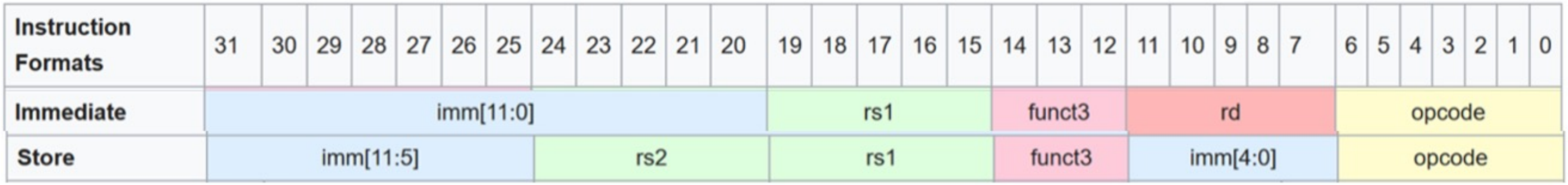


Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						

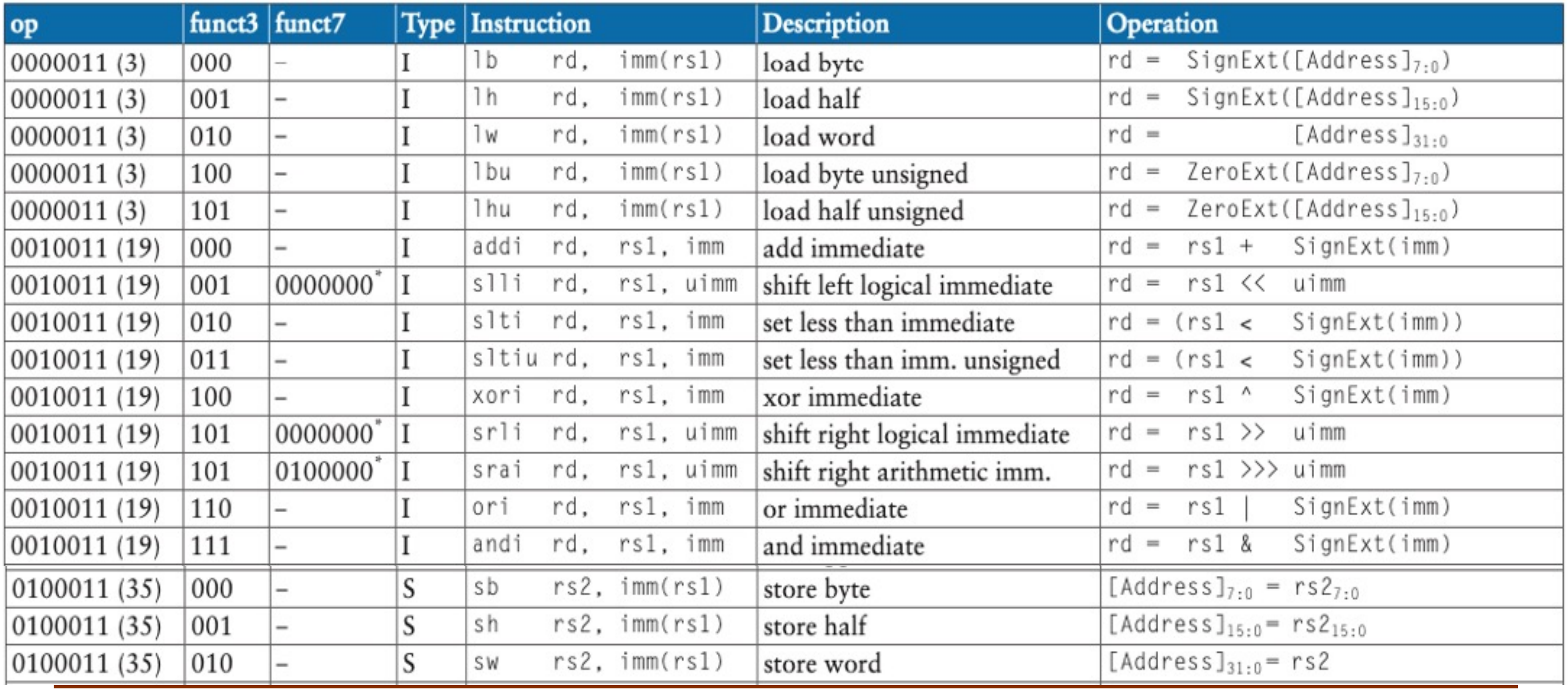


op	funct3	funct7	Type	Instruction	Description	Operation
0110011 (51)	000	0000000	R	`add rd, rs1, rs2`	add	rd = rs1 + rs2
0110011 (51)	000	0100000	R	`sub rd, rs1, rs2`	sub	rd = rs1 − rs2
0110011 (51)	001	0000000	R	`sll rd, rs1, rs2`	shift left logical	rd = rs1 << rs2$_{4:0}$
0110011 (51)	010	0000000	R	`slt rd, rs1, rs2`	set less than	rd = (rs1 < rs2)
0110011 (51)	011	0000000	R	`sltu rd, rs1, rs2`	set less than unsigned	rd = (rs1 < rs2)
0110011 (51)	100	0000000	R	`xor rd, rs1, rs2`	xor	rd = rs1 ^ rs2
0110011 (51)	101	0000000	R	`srl rd, rs1, rs2`	shift right logical	rd = rs1 >> rs2$_{4:0}$
0110011 (51)	101	0100000	R	`sra rd, rs1, rs2`	shift right arithmetic	rd = rs1 >>> rs2$_{4:0}$
0110011 (51)	110	0000000	R	`or rd, rs1, rs2`	or	rd = rs1 | rs2
0110011 (51)	111	0000000	R	`and rd, rs1, rs2`	and	rd = rs1 & rs2

I & S-type Instructions: All involve imm constants

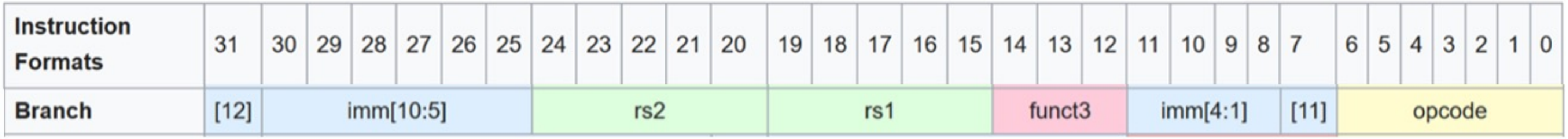


Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						

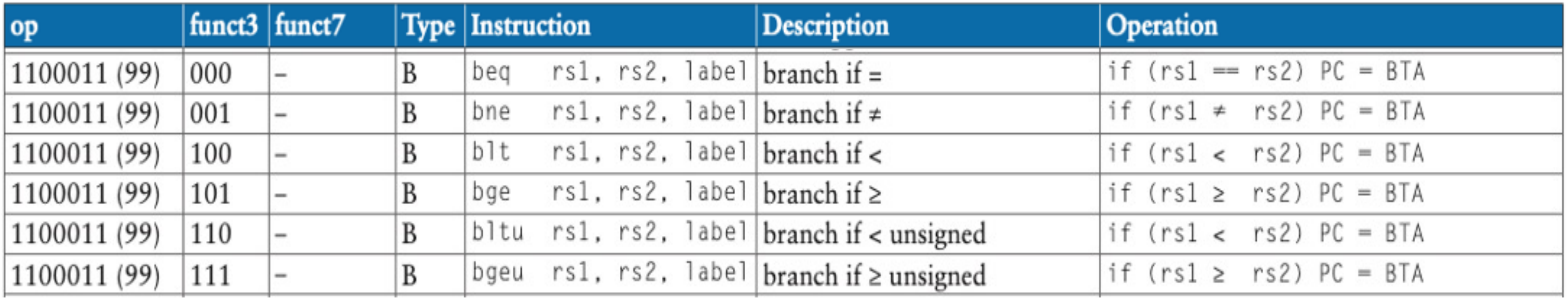


op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	`lb rd, imm(rs1)`	load byte	rd = SignExt([Address]$_{7:0}$)
0000011 (3)	001	–	I	`lh rd, imm(rs1)`	load half	rd = SignExt([Address]$_{15:0}$)
0000011 (3)	010	–	I	`lw rd, imm(rs1)`	load word	rd = [Address]$_{31:0}$
0000011 (3)	100	–	I	`lbu rd, imm(rs1)`	load byte unsigned	rd = ZeroExt([Address]$_{7:0}$)
0000011 (3)	101	–	I	`lhu rd, imm(rs1)`	load half unsigned	rd = ZeroExt([Address]$_{15:0}$)
0010011 (19)	000	–	I	`addi rd, rs1, imm`	add immediate	rd = rs1 + SignExt(imm)
0010011 (19)	001	0000000*	I	`slli rd, rs1, uimm`	shift left logical immediate	rd = rs1 << uimm
0010011 (19)	010	–	I	`slti rd, rs1, imm`	set less than immediate	rd = (rs1 < SignExt(imm))
0010011 (19)	011	–	I	`sltiu rd, rs1, imm`	set less than imm. unsigned	rd = (rs1 < SignExt(imm))
0010011 (19)	100	–	I	`xori rd, rs1, imm`	xor immediate	rd = rs1 ^ SignExt(imm)
0010011 (19)	101	0000000*	I	`srli rd, rs1, uimm`	shift right logical immediate	rd = rs1 >> uimm
0010011 (19)	101	0100000*	I	`srai rd, rs1, uimm`	shift right arithmetic imm.	rd = rs1 >>> uimm
0010011 (19)	110	–	I	`ori rd, rs1, imm`	or immediate	rd = rs1 | SignExt(imm)
0010011 (19)	111	–	I	`andi rd, rs1, imm`	and immediate	rd = rs1 & SignExt(imm)
0100011 (35)	000	–	S	`sb rs2, imm(rs1)`	store byte	[Address]$_{7:0}$ = rs2$_{7:0}$
0100011 (35)	001	–	S	`sh rs2, imm(rs1)`	store half	[Address]$_{15:0}$ = rs2$_{15:0}$
0100011 (35)	010	–	S	`sw rs2, imm(rs1)`	store word	[Address]$_{31:0}$ = rs2

B-type Instructions: PC-relative Branches



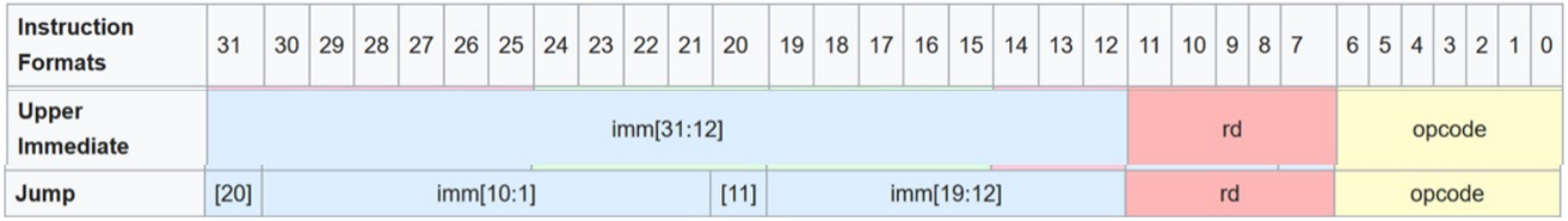
Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						



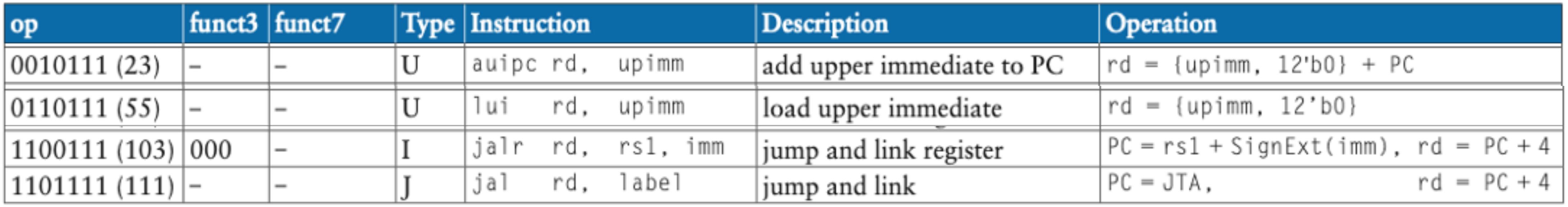
op	funct3	funct7	Type	Instruction	Description	Operation
1100011 (99)	000	–	B	`beq  rs1, rs2, label`	branch if =	`if (rs1 == rs2) PC = BTA`
1100011 (99)	001	–	B	`bne  rs1, rs2, label`	branch if ≠	`if (rs1 ≠ rs2) PC = BTA`
1100011 (99)	100	–	B	`blt  rs1, rs2, label`	branch if <	`if (rs1 < rs2) PC = BTA`
1100011 (99)	101	–	B	`bge  rs1, rs2, label`	branch if ≥	`if (rs1 ≥ rs2) PC = BTA`
1100011 (99)	110	–	B	`bltu rs1, rs2, label`	branch if < unsigned	`if (rs1 < rs2) PC = BTA`
1100011 (99)	111	–	B	`bgeu rs1, rs2, label`	branch if ≥ unsigned	`if (rs1 ≥ rs2) PC = BTA`

H&H p311

U & I -type Instructions: Upper & Jump/Link



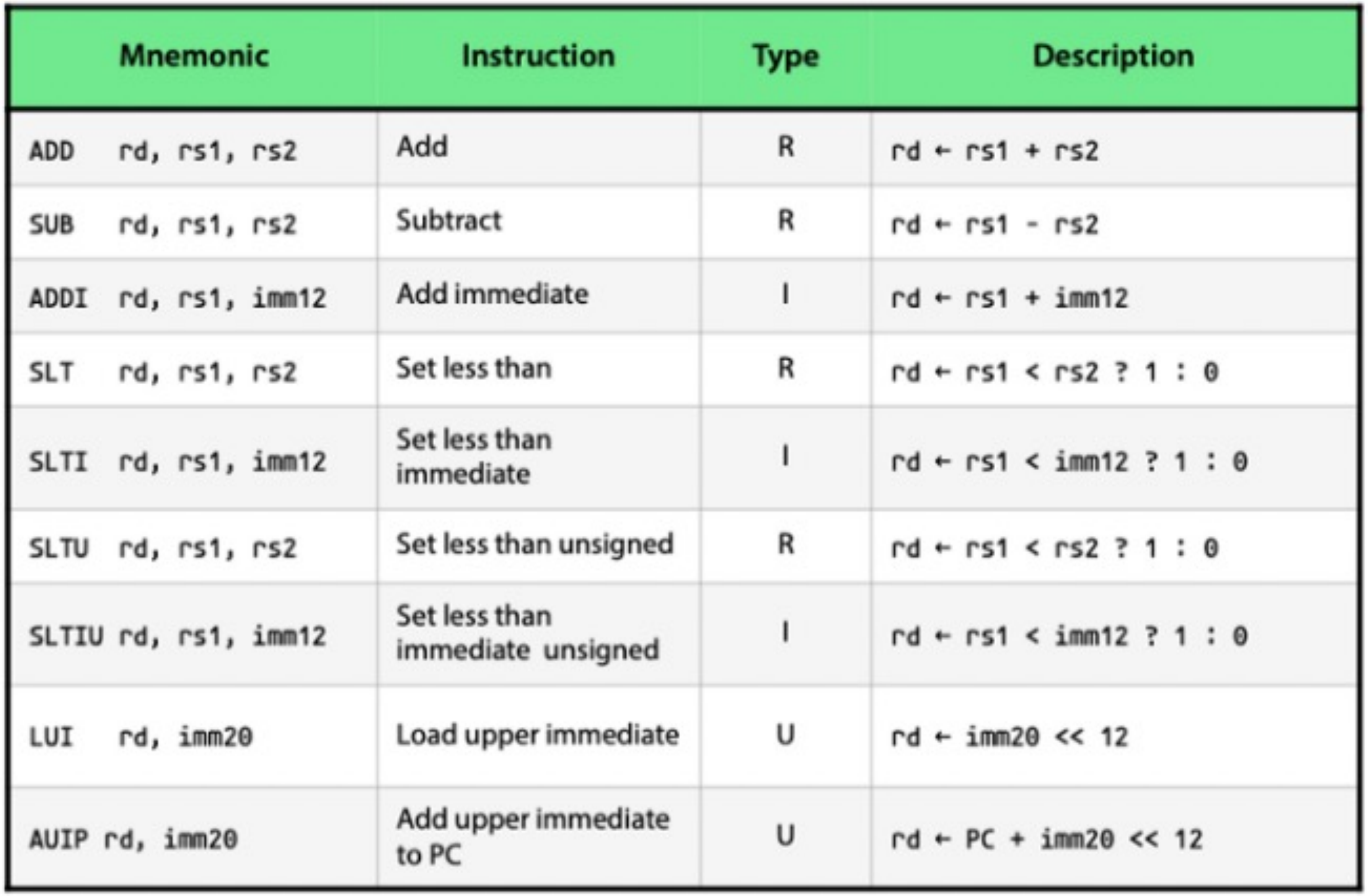
Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Upper Immediate	imm[31:12]																				rd					opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						



op	funct3	funct7	Type	Instruction	Description	Operation
0010111 (23)	–	–	U	`auipc rd, upimm`	add upper immediate to PC	`rd = {upimm, 12'b0} + PC`
0110111 (55)	–	–	U	`lui rd, upimm`	load upper immediate	`rd = {upimm, 12'b0}`
1100111 (103)	000	–	I	`jalr rd, rs1, imm`	jump and link register	`PC = rs1 + SignExt(imm), rd = PC + 4`
1101111 (111)	–	–	J	`jal rd, label`	jump and link	`PC = JTA, rd = PC + 4`

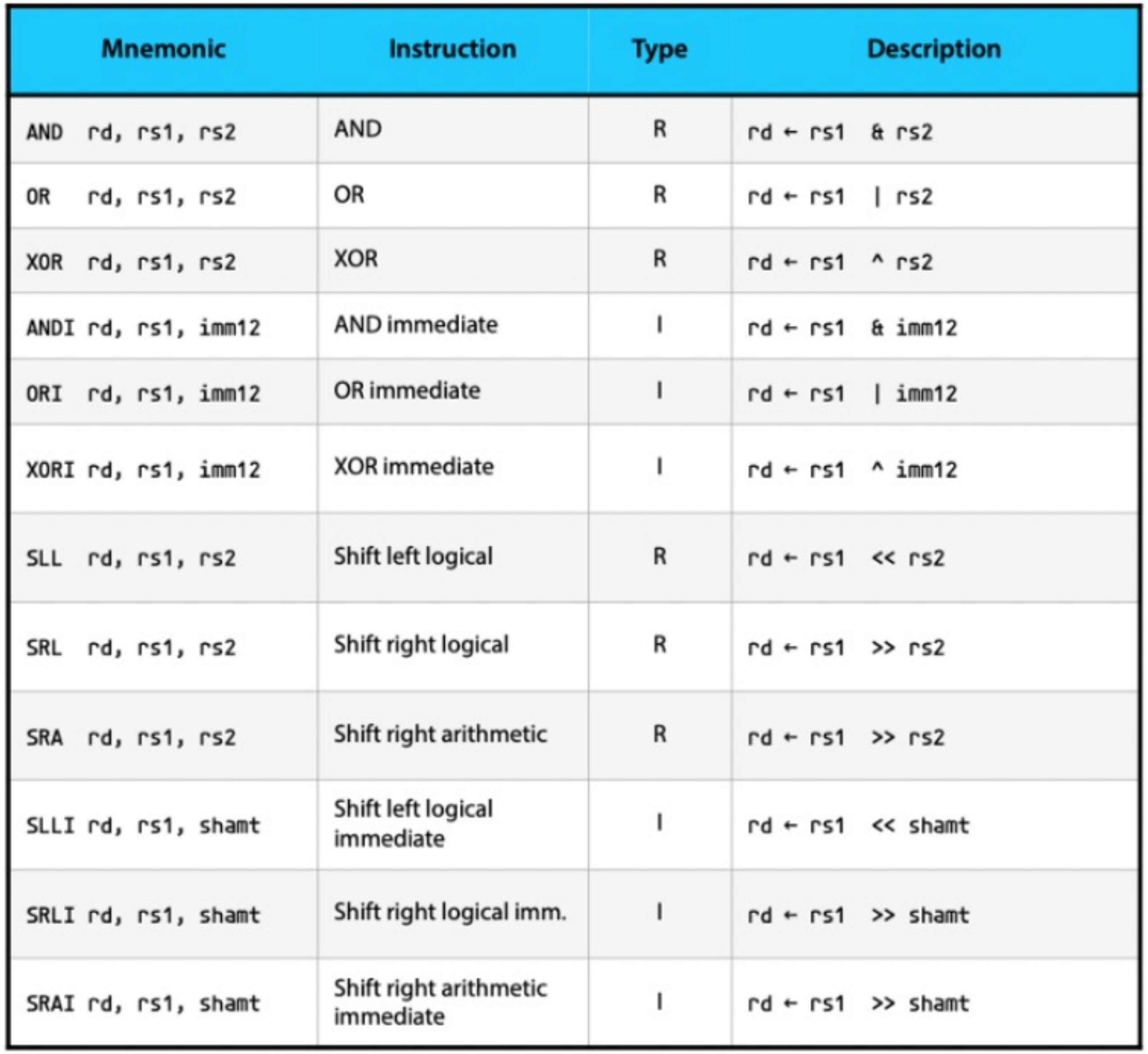
- We will discuss auipc, jalr and jal instructions in another lecture

RISC-V Arithmetic instructions



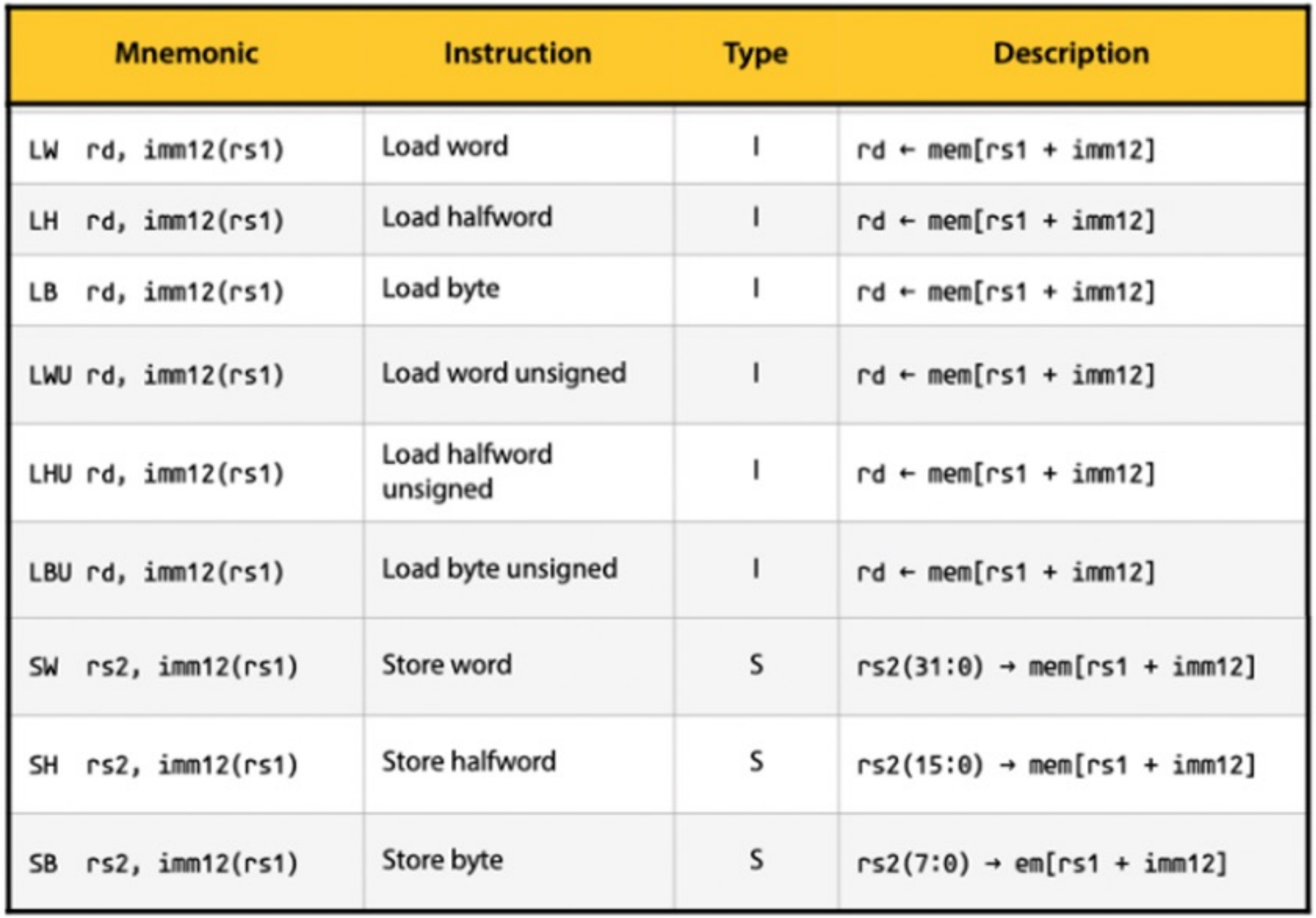
Mnemonic	Instruction	Type	Description
ADD rd, rs1, rs2	Add	R	rd ← rs1 + rs2
SUB rd, rs1, rs2	Subtract	R	rd ← rs1 - rs2
ADDI rd, rs1, imm12	Add immediate	I	rd ← rs1 + imm12
SLT rd, rs1, rs2	Set less than	R	rd ← rs1 < rs2 ? 1 : 0
SLTI rd, rs1, imm12	Set less than immediate	I	rd ← rs1 < imm12 ? 1 : 0
SLTU rd, rs1, rs2	Set less than unsigned	R	rd ← rs1 < rs2 ? 1 : 0
SLTIU rd, rs1, imm12	Set less than immediate unsigned	I	rd ← rs1 < imm12 ? 1 : 0
LUI rd, imm20	Load upper immediate	U	rd ← imm20 << 12
AUIP rd, imm20	Add upper immediate to PC	U	rd ← PC + imm20 << 12

RISC-V Logic instructions



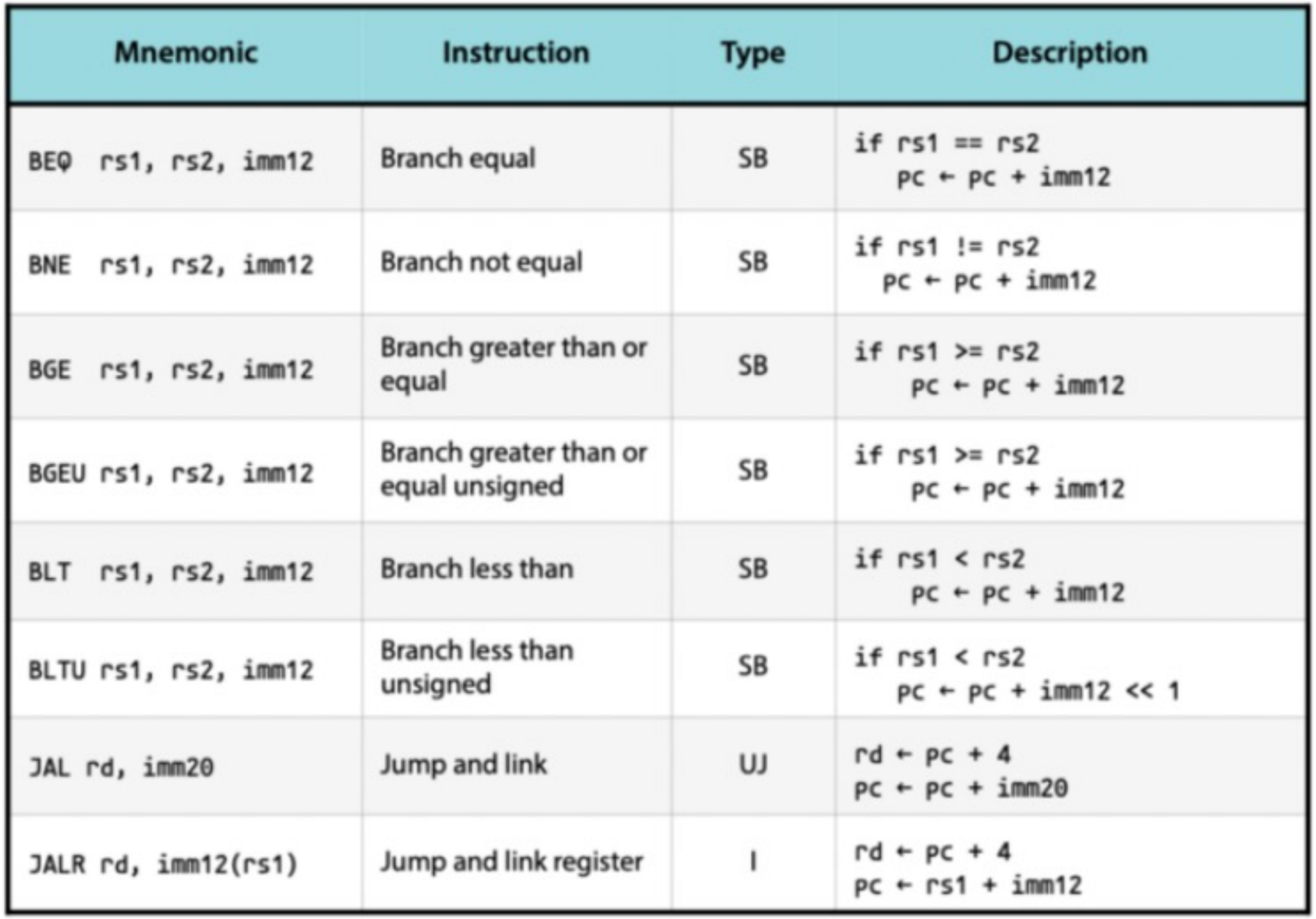
Mnemonic	Instruction	Type	Description
`AND  rd, rs1, rs2`	AND	R	`rd ← rs1  & rs2`
`OR   rd, rs1, rs2`	OR	R	`rd ← rs1  | rs2`
`XOR  rd, rs1, rs2`	XOR	R	`rd ← rs1  ^ rs2`
`ANDI rd, rs1, imm12`	AND immediate	I	`rd ← rs1  & imm12`
`ORI  rd, rs1, imm12`	OR immediate	I	`rd ← rs1  | imm12`
`XORI rd, rs1, imm12`	XOR immediate	I	`rd ← rs1  ^ imm12`
`SLL  rd, rs1, rs2`	Shift left logical	R	`rd ← rs1  << rs2`
`SRL  rd, rs1, rs2`	Shift right logical	R	`rd ← rs1  >> rs2`
`SRA  rd, rs1, rs2`	Shift right arithmetic	R	`rd ← rs1  >> rs2`
`SLLI rd, rs1, shamt`	Shift left logical immediate	I	`rd ← rs1  << shamt`
`SRLI rd, rs1, shamt`	Shift right logical imm.	I	`rd ← rs1  >> shamt`
`SRAI rd, rs1, shamt`	Shift right arithmetic immediate	I	`rd ← rs1  >> shamt`

RISC-V Load/Store instructions



Mnemonic	Instruction	Type	Description
`LW  rd, imm12(rs1)`	Load word	I	`rd ← mem[rs1 + imm12]`
`LH  rd, imm12(rs1)`	Load halfword	I	`rd ← mem[rs1 + imm12]`
`LB  rd, imm12(rs1)`	Load byte	I	`rd ← mem[rs1 + imm12]`
`LWU rd, imm12(rs1)`	Load word unsigned	I	`rd ← mem[rs1 + imm12]`
`LHU rd, imm12(rs1)`	Load halfword unsigned	I	`rd ← mem[rs1 + imm12]`
`LBU rd, imm12(rs1)`	Load byte unsigned	I	`rd ← mem[rs1 + imm12]`
`SW  rs2, imm12(rs1)`	Store word	S	`rs2(31:0) → mem[rs1 + imm12]`
`SH  rs2, imm12(rs1)`	Store halfword	S	`rs2(15:0) → mem[rs1 + imm12]`
`SB  rs2, imm12(rs1)`	Store byte	S	`rs2(7:0) → em[rs1 + imm12]`

RISC-V Branch & Jump instructions



Mnemonic	Instruction	Type	Description

`BEQ  rs1, rs2, imm12`	Branch equal	SB	`if rs1 == rs2`
			`pc ← pc + imm12`

`BNE  rs1, rs2, imm12`	Branch not equal	SB	`if rs1 != rs2`
			`pc ← pc + imm12`

`BGE  rs1, rs2, imm12`	Branch greater than or equal	SB	`if rs1 >= rs2`
			`pc ← pc + imm12`

`BGEU rs1, rs2, imm12`	Branch greater than or equal unsigned	SB	`if rs1 >= rs2`
			`pc ← pc + imm12`

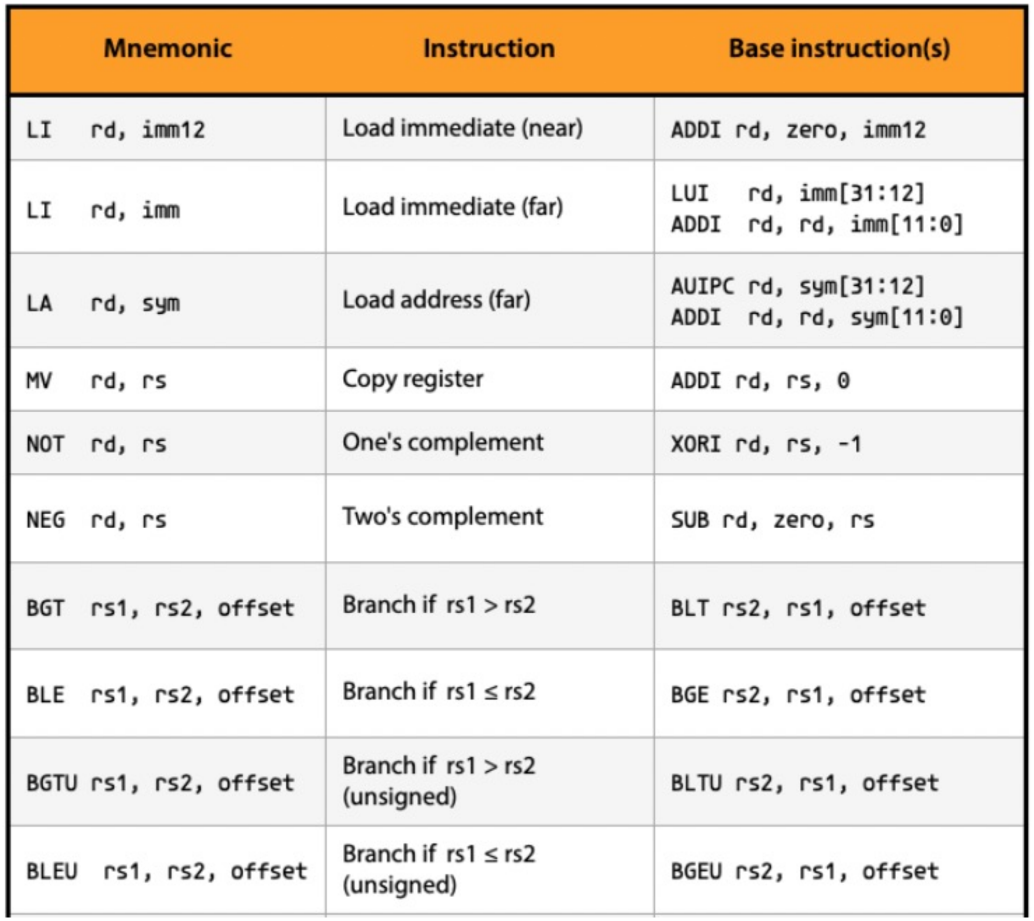
`BLT  rs1, rs2, imm12`	Branch less than	SB	`if rs1 < rs2`
			`pc ← pc + imm12`

`BLTU rs1, rs2, imm12`	Branch less than unsigned	SB	`if rs1 < rs2`
			`pc ← pc + imm12 << 1`

`JAL rd, imm20`	Jump and link	UJ	`rd ← pc + 4`
			`pc ← pc + imm20`

`JALR rd, imm12(rs1)`	Jump and link register	I	`rd ← pc + 4`
			`pc ← rs1 + imm12`

RISC-V Psuedoinstructions

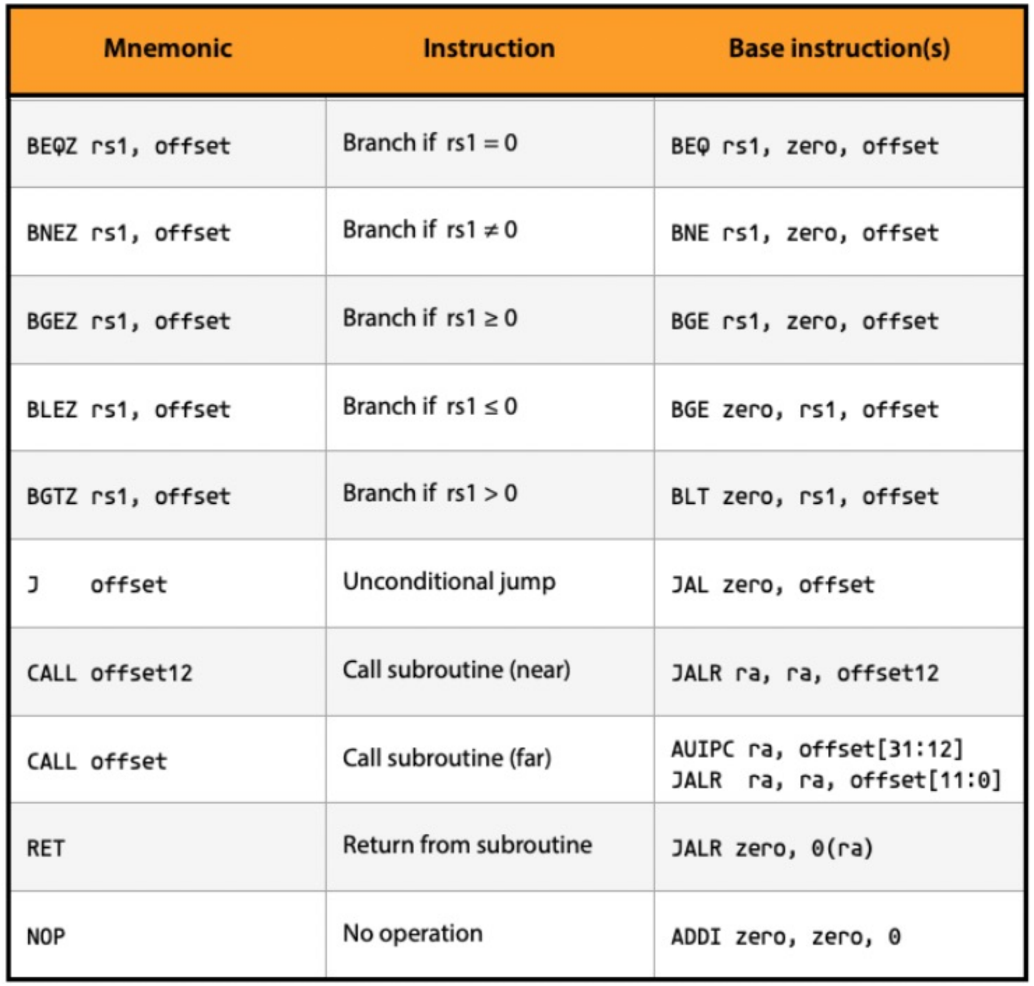


Mnemonic	Instruction	Base instruction(s)
`LI rd, imm12`	Load immediate (near)	`ADDI rd, zero, imm12`

`LI rd, imm`	Load immediate (far)	`LUI rd, imm[31:12]`
		`ADDI rd, rd, imm[11:0]`

`LA rd, sym`	Load address (far)	`AUIPC rd, sym[31:12]`
		`ADDI rd, rd, sym[11:0]`

`MV rd, rs`	Copy register	`ADDI rd, rs, 0`
`NOT rd, rs`	One's complement	`XORI rd, rs, -1`
`NEG rd, rs`	Two's complement	`SUB rd, zero, rs`
`BGT rs1, rs2, offset`	Branch if rs1 > rs2	`BLT rs2, rs1, offset`
`BLE rs1, rs2, offset`	Branch if rs1 ≤ rs2	`BGE rs2, rs1, offset`
`BGTU rs1, rs2, offset`	Branch if rs1 > rs2 (unsigned)	`BLTU rs2, rs1, offset`
`BLEU rs1, rs2, offset`	Branch if rs1 ≤ rs2 (unsigned)	`BGEU rs2, rs1, offset`



Mnemonic	Instruction	Base instruction(s)
`BEQZ rs1, offset`	Branch if rs1 = 0	`BEQ rs1, zero, offset`
`BNEZ rs1, offset`	Branch if rs1 ≠ 0	`BNE rs1, zero, offset`
`BGEZ rs1, offset`	Branch if rs1 ≥ 0	`BGE rs1, zero, offset`
`BLEZ rs1, offset`	Branch if rs1 ≤ 0	`BGE zero, rs1, offset`
`BGTZ rs1, offset`	Branch if rs1 > 0	`BLT zero, rs1, offset`
`J offset`	Unconditional jump	`JAL zero, offset`
`CALL offset12`	Call subroutine (near)	`JALR ra, ra, offset12`

`CALL offset`	Call subroutine (far)	`AUIPC ra, offset[31:12]`
		`JALR ra, ra, offset[11:0]`

`RET`	Return from subroutine	`JALR zero, 0(ra)`
`NOP`	No operation	`ADDI zero, zero, 0`