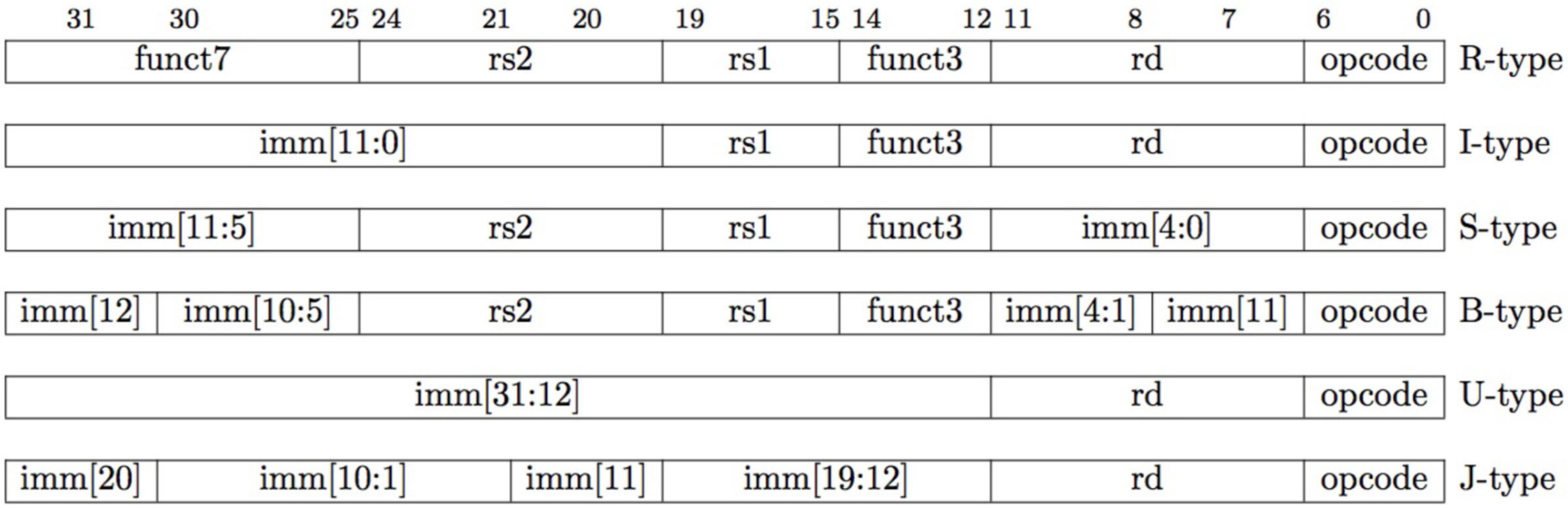
Instruction Set Architecture
(Instruction encoding)

Acknowledgment: Slides are adapted from Harris and Harris textbook instructor’s material

Machine Language and Instruction Encoding

- Binary representation of instructions (computers only understand 1's and 0's)
- 32-bit instructions (for studied, baseline RV32)
  - Instruction bits are divided into fields, each field tells processor something about instruction
- Simplicity favors regularity → Six basic instruction formats (as opposed to, e.g., unique format for each instruction)
  - **R-Type:** for register-register arithmetic operations
  - **I-Type:** for register-immediate arithmetic operations and loads
  - **S-Type:** for stores
  - **B-Type:** for branches (similar to S-type)
  - **U-Type:** for 20-bit upper immediate instructions
  - **J-Type:** for jumping (similar to U-type)

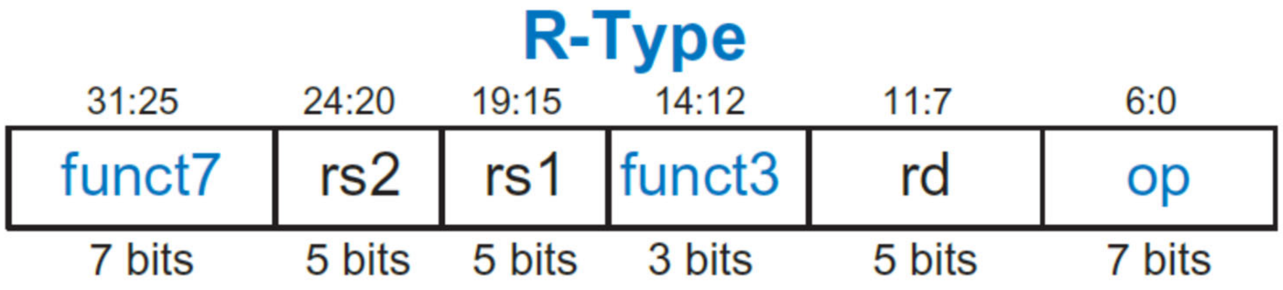
RISC-V Instruction Formats



31	30 25	24 21	20	19 15	14 12	11 8	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]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

R-Type (Register) Instruction Layout

- 3 register operands:
  - `rs1`, `rs2`: specify registers containing first and second operand
  - `rd`: specify register which will receive the results of computation
  - Each register field is 5-bit unsigned number encoding register number
- Other fields:
  - `opcode`: partially specifies what instruction does
  - `funct7+funct3`: combined with opcode, these fields describe the operation



R-Type Examples

Assembly / **Field Values**

Assembly	funct7	rs2	rs1	funct3	rd	op

`add s2, s3, s4`
`add x18,x19,x20`
	0	20	19	0	18	51

`sub t0, t1, t2`
`sub x5, x6, x7`
	32	7	6	0	5	51

	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

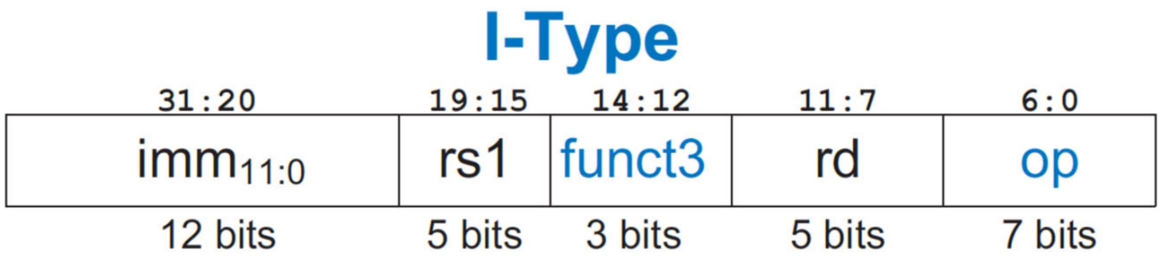
Machine Code

funct7	rs2	rs1	funct3	rd	op	
0000 000	10100	10011	000	10010	011 0011	`(0x01498933)`
0100 000	00111	00110	000	00101	011 0011	`(0x407302B3)`
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Note the different order of registers in the assembly code

I-Type (Immediate) Instruction Layout

- Used for arithmetic / logical operations with immediate and load instructions
- Instruction fields:
  - `rs1`, `rd`: source and destination register operands (note that only two registers!)
  - `imm`: 12-bit two's complement immediate, i.e. any number from [-2048_{ten}, +2047_{ten}], always sign-extended to 32-bits before using in arithmetic operation
  - `op`: the opcode completely determining operation for arithmetic operations; together with `funct3` field determines operation for others
- Load instructions are also I-Type
  - 12-bit immediate is used to create address, not the final result
- The format is similar R-Type to simplify instruction decoding



I-Type Examples

Assembly	Field Values					Machine Code					
	$imm_{11:0}$	rs1	funct3	rd	op	$imm_{11:0}$	rs1	funct3	rd	op	

`addi s0, s1, 12`
`addi x8, x9, 12`
	12	9	0	8	19	0000 0000 1100	01001	000	01000	001 0011	(0x00C48413)

`addi s2, t1, -14`
`addi x18,x6, -14`
	-14	6	0	18	19	1111 1111 0010	00110	000	10010	001 0011	(0xFF230913)

`lw t2, -6(s3)`
`lw x7, -6(x19)`
	-6	19	2	7	3	1111 1111 1010	10011	010	00111	000 0011	(0xFFA9A383)

`lb s4, 0x1F(s4)`
`lb x20,0x1F(x20)`
	0x1F	20	0	20	3	0000 0001 1111	10100	000	10100	000 0011	(0x01FA0A03)

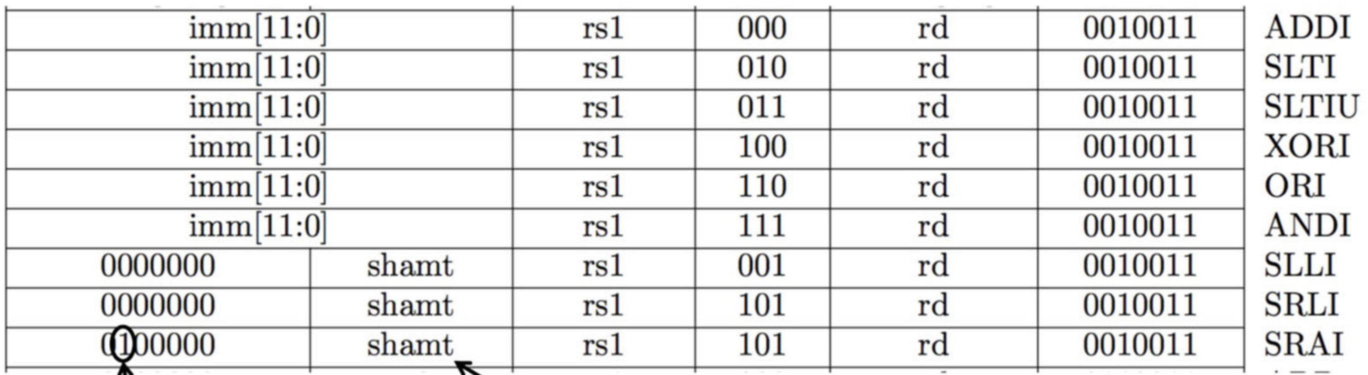
`slli s2, s7, 5`
`slli x18, x23, 5`
	5	23	1	18	19	0000 0000 0101	10111	001	10010	001 0011	(0x005B9913)

`srai t1, t2, 29`
`srai x6, x7, 29`
	(upper 7 bits = 32)
	29
		7	5	6	19	0100 0001 1101	00111	101	00110	001 0011	(0x41D3D313)

	12 bits	5 bits	3 bits	5 bits	7 bits	12 bits	5 bits	3 bits	5 bits	7 bits	

Note again different order of registers in assembly and machine codes!

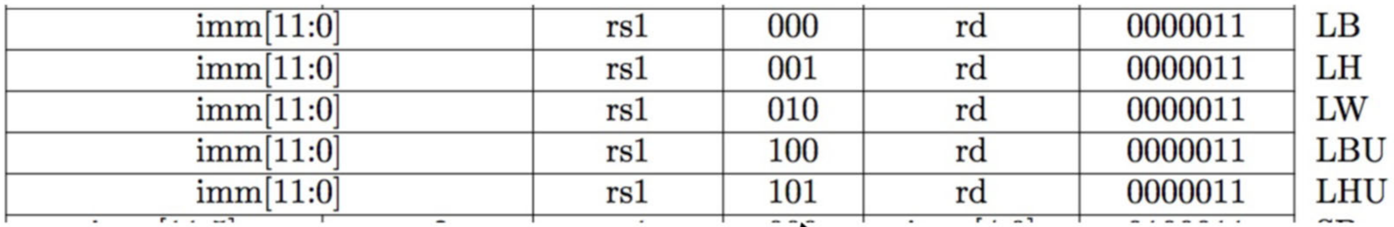
All RV32 I-type Arithmetic/Logical/Load Instructions



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	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

One of the higher-order immediate bits is used to distinguish "shift right logical" (SRLI) from "shift right arithmetic" (SRAI)

"Shift-by-immediate" instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions)



imm[11:0]	rs1	000	rd	0000011	LB
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

funct3 field encodes size and signedness of load data

S/B-Type Instruction Layout

- Used for stores and branches
- Instruction fields:
  - `rs1`, `rs2`: source1 and source2 register operands (none of them is written)
  - `imm`: 12-bit two's complement immediate, always sign-extended to 32-bits before using in arithmetic operation (i.e., address calculation or comparison)
  - `Op` and `funct3`: completely determines operation
- For performance reasons, rs1 and rs2 field locations (but not imm) are consistent with R-format

31:25	24:20	19:15	14:12	11:7	6:0	
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

S-Type Example

Assembly

```
sw t2, -6(s3)
sw x7, -6(x19)
sh s4, 23(t0)
sh x20,23(x5)
sb t5, 0x2D(zero)
sb x30,0x2D(x0)
```

Field Values

$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op
1111 111	7	19	2	11010	35
0000 000	20	5	1	10111	35
0000 001	30	0	0	01101	35
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Machine Code

$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	
1111 111	00111	10011	010	11010	010 0011	(0xFE79AD23)
0000 000	10100	00101	001	10111	010 0011	(0x01429BA3)
0000 001	11110	00000	000	01101	010 0011	(0x03E006A3)
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

PC-Relative Addressing

- Use the immediate field as a two’s complement offset to PC
    - Branches generally change the PC by small amount
    - To improve the reach of a single branch instruction, we can encode word offset rather than a byte offset (typical for other RISC architectures)
    - However for compatibility reasons with 16-bit compressed instructions extension, encode half word offsets
    - Can jump to +/- 2^{10} 32-bit instructions from the current PC
- Branch calculation scheme:
    - Need to multiply the offset by 2 before adding to PC
    - PC = PC + immediate *2
    - Immediate is a number of instructions to jump either forward or backwards

Branch Example

```
#Address    # RISC-V Assembly
0x70            beq   s0, t5, L1     1
0x74            add   s1, s2, s3     2
0x78            sub   s5, s6, s7     3
0x7C            lw    t0, 0(s1)      4
0x80       L1:  addi  s1, s1, -15
```

L1 is 4 instructions (i.e., **16 bytes**) past beq

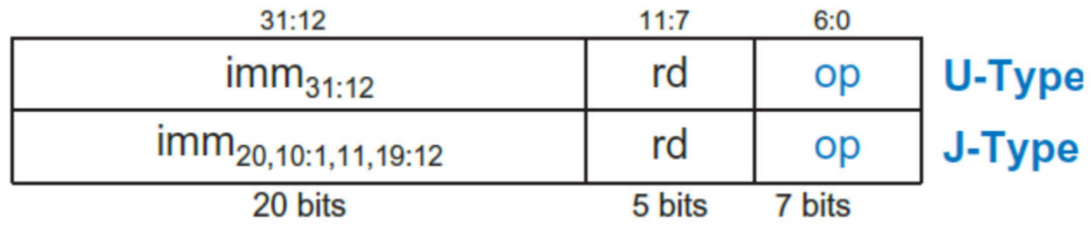
$imm_{12:0}$ = 16	0	0	0	0	0	0	0	0	1	0	0	0	$\cancel{0}$
bit number	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

Assembly | **Field Values** | **Machine Code**

Assembly	$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	
beq s0, t5, L1 beq x8, x30, 16	0000 000	30	8	0	1000 0	99	0000 000	11110	01000	000	1000 0	110 0011	(0x01E40863)
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

U/J-Type Instruction Layout

- Used for loading 20-bit immediate and unconditional branches
- Only one destination register
- `opcode` completely determines operation



31:12	11:7	6:0	
$imm_{31:12}$	rd	op	U-Type
$imm_{20,10:1,11,19:12}$	rd	op	J-Type
20 bits	5 bits	7 bits	

- J-Type: JAL instruction
  - JAL saves PC+4 in register rd (the return address)
  - Set PC to PC + immediate * 2
    - PC-relative addressing with +/- 2^{18} 32-bit instruction range from the current value of PC (using similar trick to B-type format)
- Assembler “J” jump is pseudo-instruction, uses JAL with rd=x0 to discard return address

Jump Example

```
# Address               RISC-V Assembly
0x0000540C              jal ra, func1
0x00005410              add s1, s2, s3
...                     ...

0x000ABC04    func1:    add s4, s5, s8
...                     ...
```

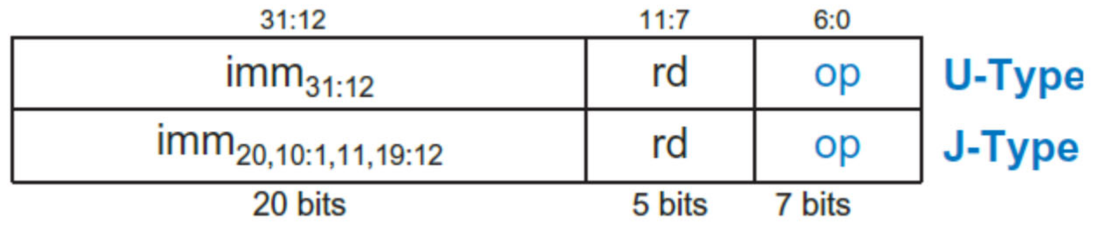
func1 is **0xA67F8 bytes** past jal

imm = 0xA67F8	0	1	0	1	0	0	1	1	0	0	1	1	1	1	1	1	1	1	0	0	~~0~~
bit number	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Assembly	Field Values			Machine Code			
	$imm_{20,10:1,11,19:12}$	rd	op	$imm_{20,10:1,11,19:12}$	rd	op	
jal ra, func1 jal x1, 0xA67F8	0111 1111 1000 1010 0110	1	111	0111 1111 1000 1010 0110	00001	110 1111	**(0x7F8A60EF)**
	20 bits	5 bits	7 bits	20 bits	5 bits	7 bits	

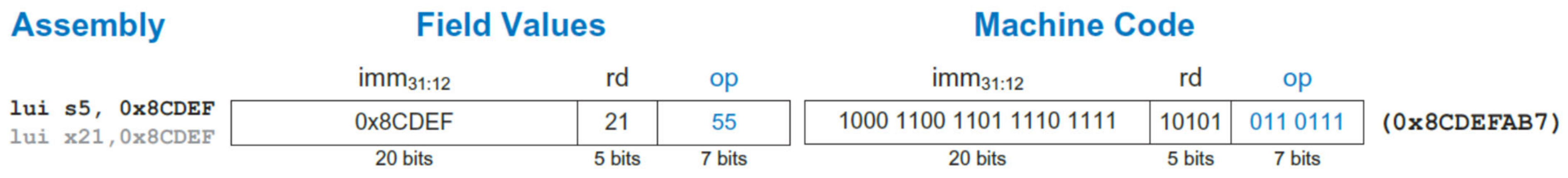
U-Type Instructions

- U-type instructions
  - LUI – load upper immediate (writes the upper 20 bits of the destination with the immediate values, and clears the lower 12 bits)
  - AUIPC – to be discussed shortly



31:12	11:7	6:0	
$imm_{31:12}$	rd	op	U-Type
$imm_{20,10:1,11,19:12}$	rd	op	J-Type
20 bits	5 bits	7 bits	

U-type example:



Assembly	Field Values			Machine Code			
	$imm_{31:12}$	rd	op	$imm_{31:12}$	rd	op	

`lui s5, 0x8CDEF`
 `lui x21,0x8CDEF`
	0x8CDEF	21	55	1000 1100 1101 1110 1111	10101	011 0111	`(0x8CDEFAB7)`

	20 bits	5 bits	7 bits	20 bits	5 bits	7 bits	

Loading Large Constants Revisited (Corner Case)

- Recollect how 32 bit constants are loaded (e.g., 0xDEADBEEF → x10)
  - LUI x10, 0xDEADB
  - ADDI x10, x10, 0xEEF
- However, ADDI 12-bit immediate is always sign-extended, so if the top bit is 1, it will add "-1" to upper 20 bits
  - Instead of 0xDEADBEEF, it will be 0xDEADAEEF
- Solution: Pre-increment value placed in upper 20 bits, if the top bit is set in the immediate lower 12 bits
- Assembler pseudo-instruction LI takes care of it
  - LI x10, 0xDEADBEEF → LUI x10, OxDEADC followed by ADDI x10,x10, 0xEEF

Far-Away Jumps

- JAL (J) and branches jump relative to PC-relative. The range is limited (up to +/- 2^{18} instructions for JALs)
- For far away branches (i.e. call to O/S functions), use combination of AUIPC and JALR instructions

- AUIPC (U-type instruction): Add 20-bit immediate to upper PC bits and places the result in the destination register
  - AUIPC rd, upimm rd = {upimm, 12’b0} + PC
  - Label: AUIPC x10, 0 # puts address of label in x10

- JALR (I-type instruction): JALR rd, rs, immediate
  - writes PC+4 to rd
  - Sets PC = rs + immediate (note: no multiplication by 2)

Far-away jump example:

Address of instruction specified by label L5

`call L5`	`auipc ra,` $imm_{31:12}$ `jalr ra, ra,` $imm_{11:0}$	call far away function	`PC = L5, ra = PC + 4`

Immediate Encoding Summary

Instruction encodings (31:7 bits only):

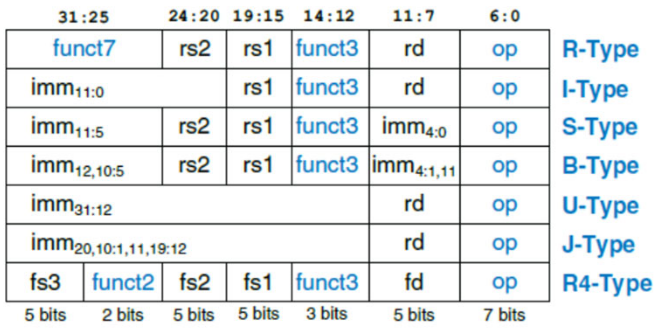
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	
11	10	9	8	7	6	5	4	3	2	1	0	rs1					funct3			rd					I
11	10	9	8	7	6	5	rs2					rs1					funct3			4	3	2	1	0	S
12	10	9	8	7	6	5	rs2					rs1					funct3			4	3	2	1	11	B
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	rd					U
20	10	9	8	7	6	5	4	3	2	1	11	19	18	17	16	15	14	13	12	rd					J

32-bit immediate produced:

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	
imm_{11}																				$imm_{11:1}$											imm_0	I, S
imm_{12}																				$imm_{11:1}$											0	B
$imm_{31:21}$											$imm_{20:12}$									0												U
imm_{20}											$imm_{20:12}$									$imm_{11:1}$											0	J

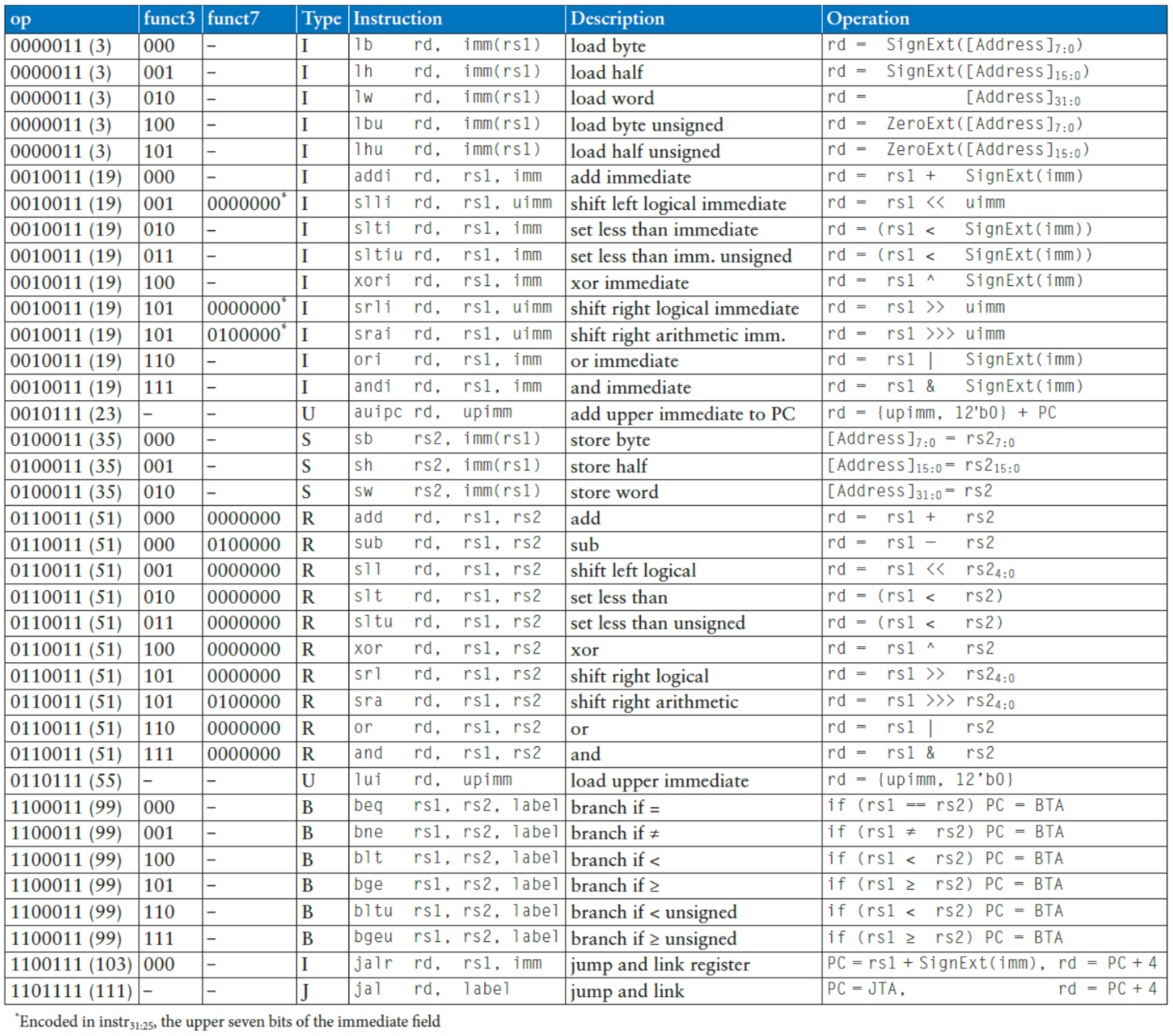
RISC-V Base Instruction Reference Sheet

- 2nd page of Harris & Harris book



31:25		24:20	19:15	14:12	11:7	6:0	
funct7		rs2	rs1	funct3	rd	op	R-Type
$imm_{11:0}$			rs1	funct3	rd	op	I-Type
$imm_{11:5}$		rs2	rs1	funct3	$imm_{4:0}$	op	S-Type
$imm_{12,10:5}$		rs2	rs1	funct3	$imm_{4:1,11}$	op	B-Type
$imm_{31:12}$					rd	op	U-Type
$imm_{20,10:1,11,19:12}$					rd	op	J-Type
fs3	funct2	fs2	fs1	funct3	fd	op	R4-Type
5 bits	2 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

• imm:	signed immediate in $imm_{11:0}$
• uimm:	5-bit unsigned immediate in $imm_{4:0}$
• upimm:	20 upper bits of a 32-bit immediate, in $imm_{31:12}$
• Address:	memory address: rs1 + SignExt($imm_{11:0}$)
• [Address]:	data at memory location Address
• BTA:	branch target address: PC + SignExt({$imm_{12:1}$, 1'b0})
• JTA:	jump target address: PC + SignExt({$imm_{20:1}$, 1'b0})
• label:	text indicating instruction address
• SignExt:	value sign-extended to 32 bits
• ZeroExt:	value zero-extended to 32 bits
• csr:	control and status register



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)
0010111 (23)	–	–	U	auipc rd, upimm	add upper immediate to PC	rd = {upimm, 12'b0} + PC
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
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
0110111 (55)	–	–	U	lui rd, upimm	load upper immediate	rd = {upimm, 12'b0}
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
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

*Encoded in instr$_{31:25}$, the upper seven bits of the immediate field