Chapter 6: Architecture

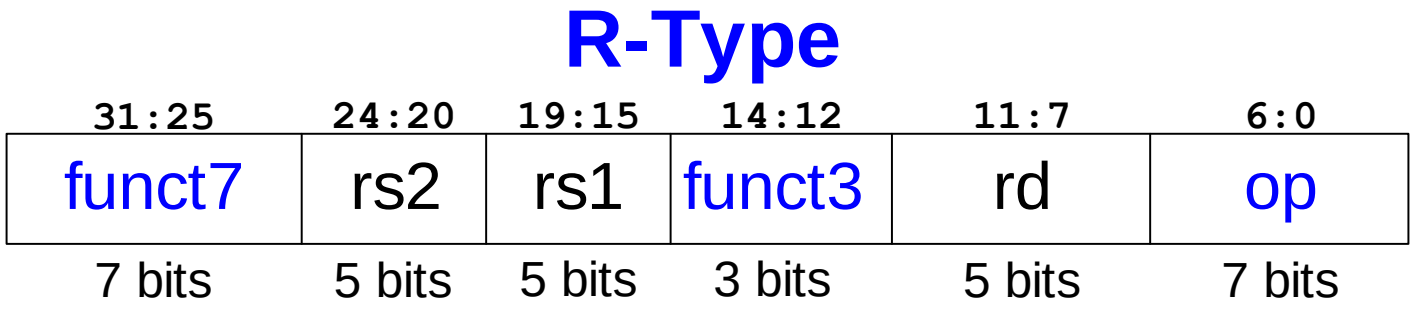
Machine Language

Machine Language

- Binary representation of instructions
- Computers only understand 1's and 0's
- 32-bit instructions
  - Simplicity favors regularity: 32-bit data & instructions
- **4 Types of Instruction Formats:**
  - R-Type
  - I-Type
  - S/B-Type
  - U/J-Type

R-Type

- *Register-type*
- 3 register operands:
  - `rs1`, `rs2`: source registers
  - `rd`: destination register
- Other fields:
  - `op`: the *operation code* or *opcode*
  - `funct7`, `funct3`: the *function* (7 bits and 3-bits, respectively) with opcode, tells computer what operation to perform



R-Type Examples

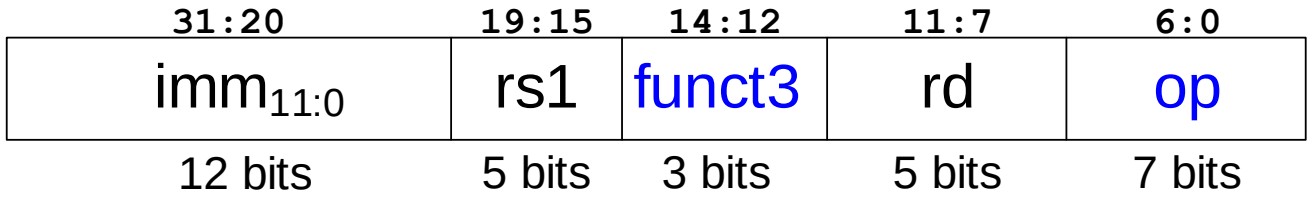
Assembly	Field Values						Machine Code						
	funct7	rs2	rs1	funct3	rd	op	funct7	rs2	rs1	funct3	rd	op	
`add s2, s3, s4` `add x18,x19,x20`	0	20	19	0	18	51	0000 000	10100	10011	000	10010	011 0011	(0x01498933)
`sub t0, t1, t2` `sub x5, x6, x7`	32	7	6	0	5	51	0100 000	00111	00110	000	00101	011 0011	(0x407302B3)
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Machine Language: More Formats

I-Type

- *Immediate-type*
- 3 operands:
  - `rs1`: register source operand
  - `rd`: register destination operand
  - `imm`: 12-bit two's complement immediate
- Other fields:
  - `op`: the opcode
    - Simplicity favors regularity: all instructions have opcode
  - `funct3`: the function (3-bit function code)
    - with opcode, tells computer what operation to perform

I-Type



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)**
`lh s1, 27(zero)` `lh x9, 27(x0)`	27	0	1	9	3	0000 0001 1011	00000	001	01001	000 0011	**(0x01B01483)**
`lb s4, 0x1F(s4)` `lb x20,0x1F(x20)`	0x1F	20	0	20	3	0000 0001 1111	10100	000	10100	000 0011	**(0x01FA0A03)**
	12 bits	5 bits	3 bits	5 bits	7 bits	12 bits	5 bits	3 bits	5 bits	7 bits	

S/B-Type

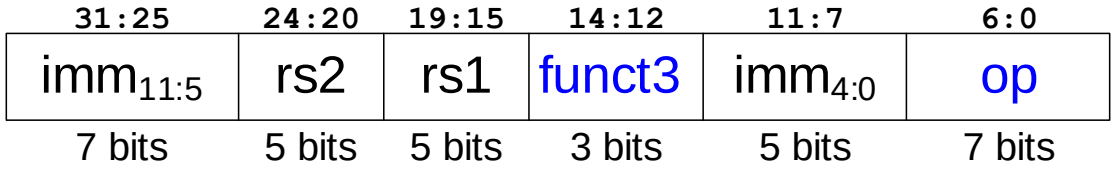
- *Store-Type*
- *Branch-Type*
- Differ only in immediate encoding

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

- *Store-Type*
- 3 operands:
  - `rs1`: base register
  - `rs2`: value to be stored to memory
  - `imm`: 12-bit two's complement immediate
- Other fields:
  - `op`: the opcode
    - Simplicity favors regularity: all instructions have opcode
  - `funct3`: the function (3-bit function code)
    - with opcode, tells computer what operation to perform

S-Type



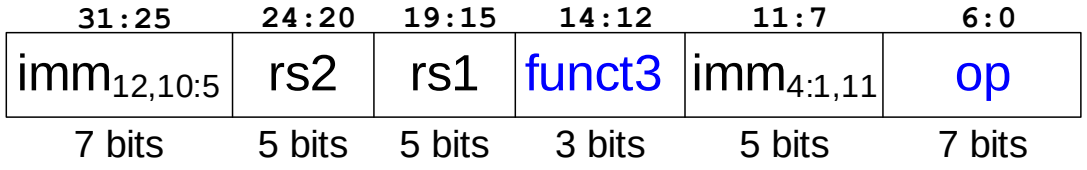
S-Type Examples

Assembly	$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	
	Field Values						Machine Code						
`sw t2, -6(s3)` `sw x7, -6(x19)`	1111 111	7	19	2	11010	35	1111 111	00111	10011	010	11010	010 0011	`(0xFE79AD23)`
`sh s4, 23(t0)` `sh x20,23(x5)`	0000 000	20	5	1	10111	35	0000 000	10100	00101	001	10111	010 0011	`(0x01429BA3)`
`sb t5, 0x2D(zero)` `sb x30,0x2D(x0)`	0000 001	30	0	0	01101	35	0000 001	11110	00000	000	01101	010 0011	`(0x03E006A3)`
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

B-Type

- *Branch-Type* (similar format to S-Type)
- 3 operands:
  - `rs1`: register source 1
  - `rs2`: register source 2
  - $\texttt{imm}_{12:1}$: 12-bit two's complement immediate – address offset
- Other fields:
  - `op`: the opcode
    - Simplicity favors regularity: all instructions have opcode
  - `funct3`: the function (3-bit function code)
    - with opcode, tells computer what operation to perform

B-Type



B-Type Example

- The 13-bit immediate encodes where to branch (relative to the branch instruction)
- Immediate encoding is strange
- Example:

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

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

Assembly

```
beq s0, t5,  L1
beq x8, x30, 16
```

Field Values

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
0000 000	30	8	0	1000 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
0000 000	11110	01000	000	1000 0	110 0011
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

(0x01E40863)

U/J-Type

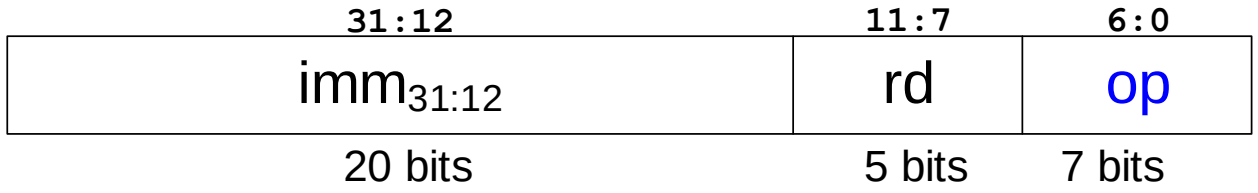
- *Upper-Immediate-Type*
- *Jump-Type*
- Differ only in immediate encoding

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

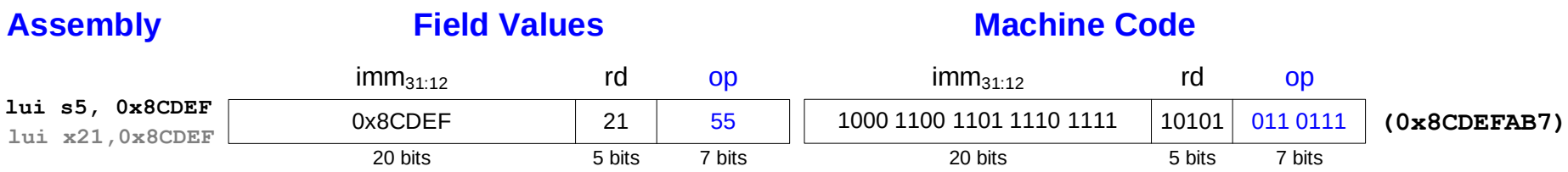
- *Upper-immediate-Type*
- Used for load upper immediate (`lui`)
- 2 operands:
  - `rd`: destination register
  - $\texttt{imm}_{31:12}$:upper 20 bits of a 32-bit immediate
- Other fields:
  - `op`: the *operation code* or *opcode* – tells computer what operation to perform

U-Type



U-Type Example

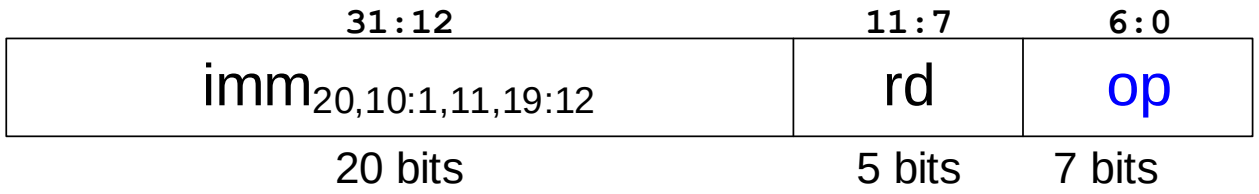
- *Upper-immediate-Type*
- Used for load upper immediate (`lui`)
- 2 operands:
  - `rd`: destination register
  - $\texttt{imm}_{31:12}$:upper 20 bits of a 32-bit immediate
- Other fields:
  - `op`: the *operation code* or *opcode* – tells computer what operation to perform



J-Type

- *Jump-Type*
- Used for jump-and-link instruction (`jal`)
- 2 operands:
  - `rd`: destination register
  - $\texttt{imm}_{20,10:1,11,19:12}$: 20 bits (20:1) of a 21-bit immediate
- Other fields:
  - `op`: the operation code or opcode – tells computer what operation to perform

J-Type



- Note: `jalr` is I-type, not j-type, to specify rs1

J-Type Example

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

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

0xABC04 – 0x540C = **0XA67F8**

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

```
jal ra, func1
jal x1, 0xA67F8
```

Field Values

$imm_{20,10:1,11,19:12}$	rd	op
0111 1111 1000 1010 0110	1	111
20 bits	5 bits	7 bits

Machine Code

$imm_{20,10:1,11,19:12}$	rd	op
0111 1111 1000 1010 0110	00001	110 1111
20 bits	5 bits	7 bits

(0x7F8A60EF)

Review: Instruction Formats

7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
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**
20 bits				5 bits	7 bits	

Design Principle 4

Good design demands good compromises

- Multiple instruction formats allow flexibility
  - `add`, `sub`: use 3 register operands
  - `lw`, `sw`, `addi`: use 2 register operands and a constant
- Number of instruction formats kept small
  - to adhere to design principles 1 and 3 (simplicity favors regularity and smaller is faster).