CS 61C: Great Ideas in Computer Architecture *RISC-V Instruction Formats*

Great Idea #1: Abstraction
Levels of Representation/Interpretation

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., RISC-V)

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

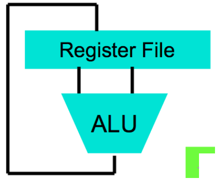
Assembler

Machine Language Program (RISC-V)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

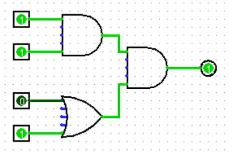
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

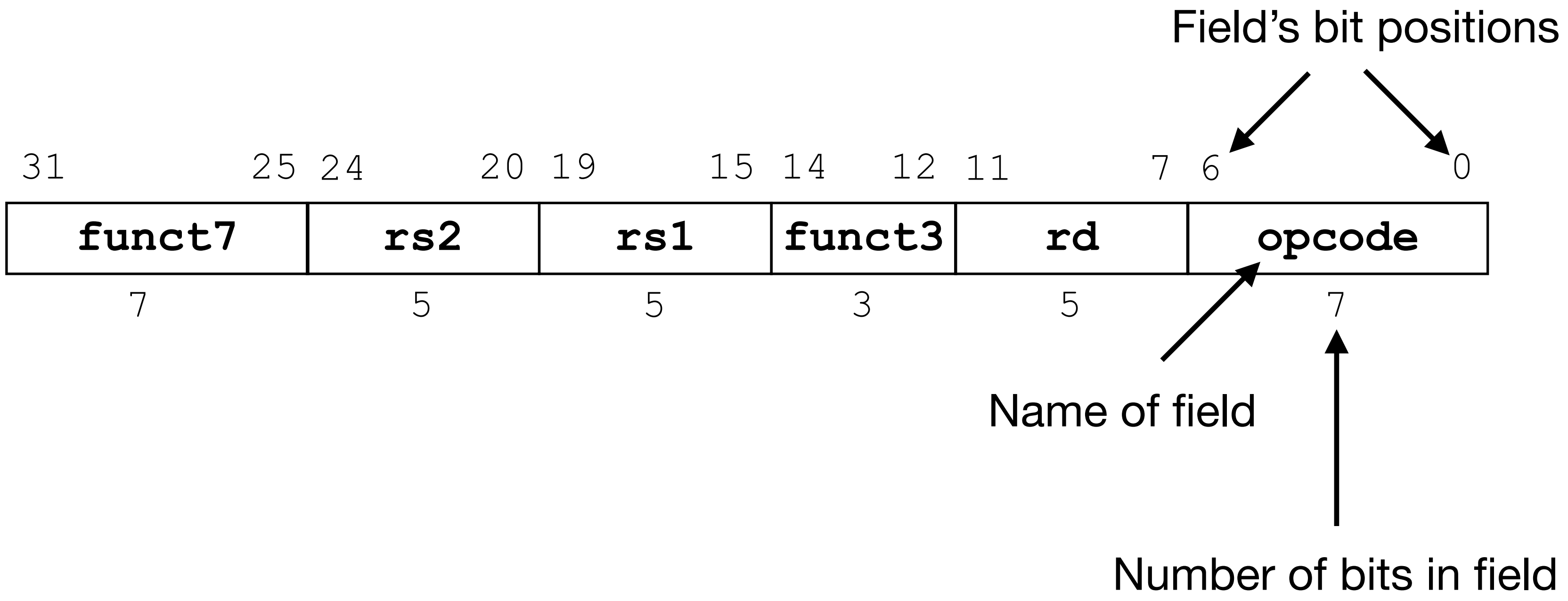


Instruction Formats

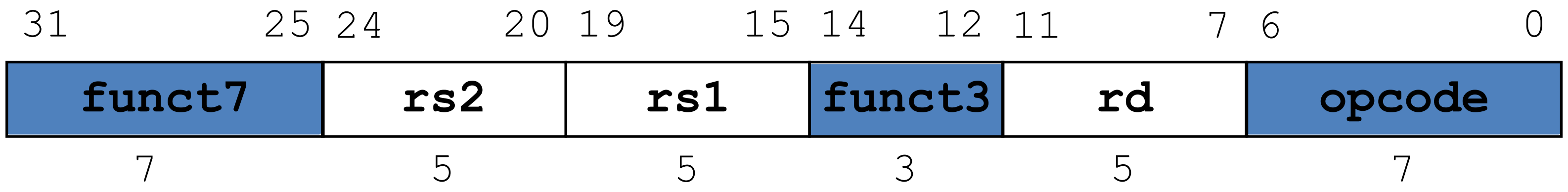
- Each instruction is 32 bits wide
- The instruction is broken down into different fields
- There are several ways that instructions are broken up

R-Format

R-Format Layout

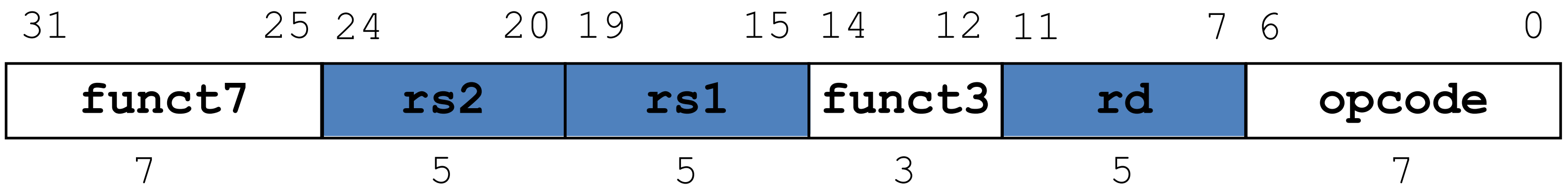


R-Format Instructions opcode/funct fields



- **opcode:** partially specifies which instruction it is
  - opcode = 0b0110011 for all 32-bit R-Format arithmetic/logical instructions
- **funct7+funct3:** combined with opcode, these two fields describe what operation to perform

R-Format Instructions Registers



- Each register field (rs1, rs2, rd) holds a 5-bit unsigned integer [0-31] corresponding to a register number (x0-x31)
- rs1 = source register #1
- rs2 = source register #2
- rd = destination register

R-Format Example

`add x18, x19, x10`

31 25	24 20	19 15	14 12	11 7	6 0
funct7	**rs2**	**rs1**	**funct3**	**rd**	**opcode**
7	5	5	3	5	7

31 25	24 20	19 15	14 12	11 7	6 0
0000000	**rs2**	**rs1**	**000**	**rd**	**0110011**
7	5	5	3	5	7

R-Format Example

`add x18, x19, x10`

31 25	24 20	19 15	14 12	11 7	6 0
funct7	**rs2**	**rs1**	**funct3**	**rd**	**opcode**
7	5	5	3	5	7

31 25	24 20	19 15	14 12	11 7	6 0
0000000	**01010**	**10011**	**000**	**10010**	**0110011**
7	5	5	3	5	7

All RV32 R-format instructions

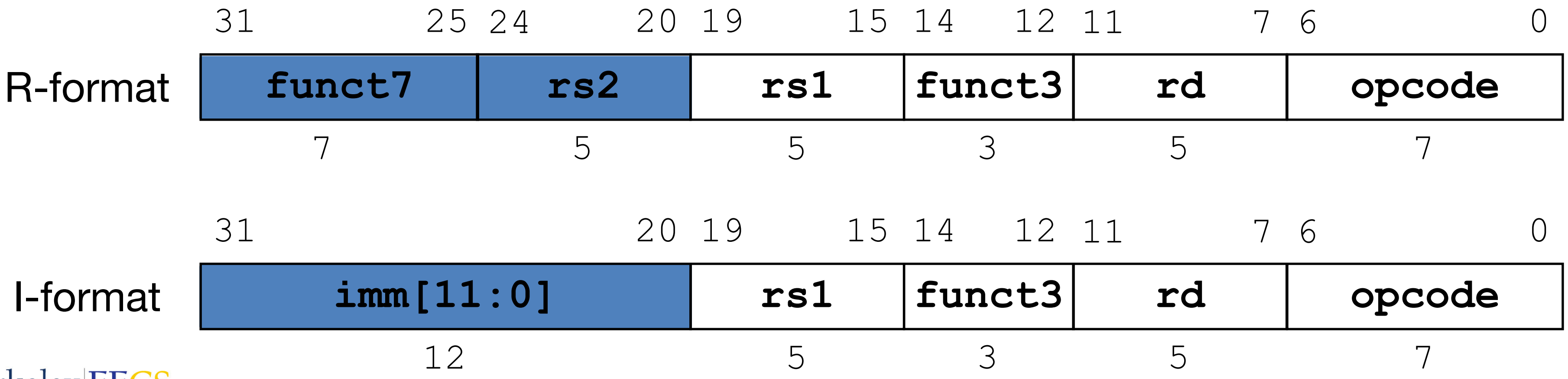
funct7			funct3		opcode	
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

funct7 + funct3 selects particular operation

I-Format

I-Format Layout

```
add x18, x19, x10

addi x18, x19, 2
```



I-Format Instructions

- **`imm[11:0]`** can hold values in range [-2048_{10} , +2047_{10}]
- Immediate is always sign-extended to 32-bits before use in an arithmetic operation
- We'll later see how to handle immediates > 12 bits

31 20	19 15	14 12	11 7	6 0
imm[11:0]	**rs1**	**funct3**	**rd**	**opcode**
12	5	3	5	7

I-Format Example

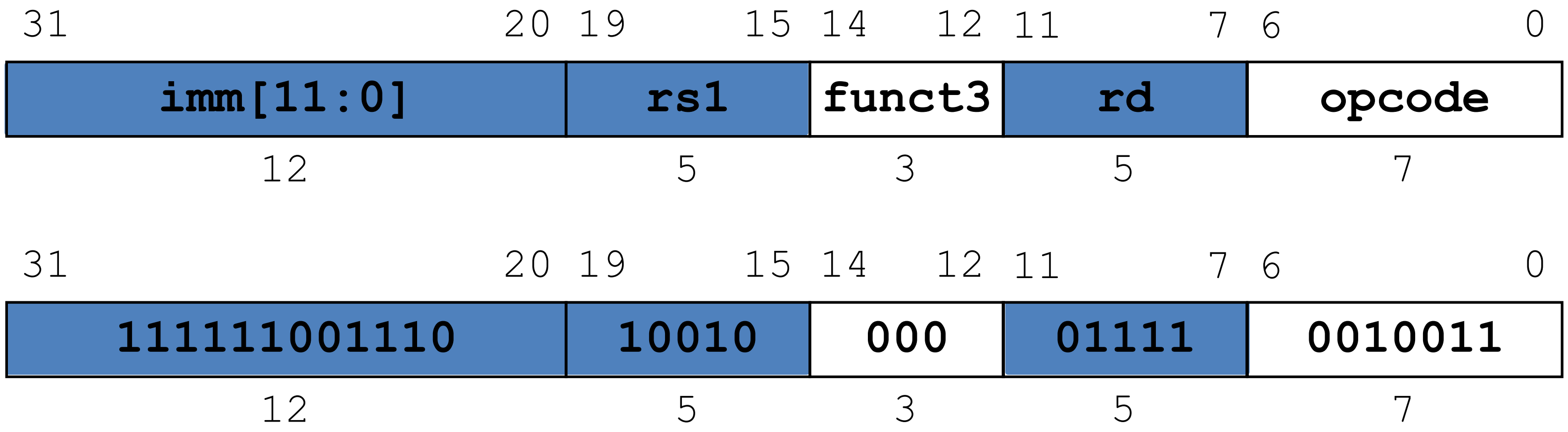
`addi x15, x18, -50`

31 20	19 15	14 12	11 7	6 0
imm[11:0]	**rs1**	**funct3**	**rd**	**opcode**
12	5	3	5	7

31 20	19 15	14 12	11 7	6 0
imm[11:0]	**rs1**	**000**	**rd**	**0010011**
12	5	3	5	7

I-Format Example

`addi x15, x18, -50`



All RV32 I-format Arithmetic/Logical Instructions

Immediate			funct3		opcode	
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

Distinguishes logical and arithmetic right

The max number of bits we need to shift by is 31, so we only need 5 bits to encode the shift amount

These funct3 fields are the same as the corresponding r-type operation

Opcode is the same for all arithmetic and logical immediate instructions

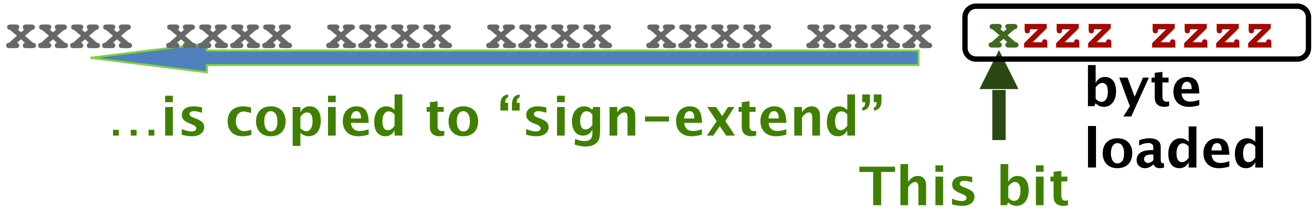
Loads

Review: Load Instructions

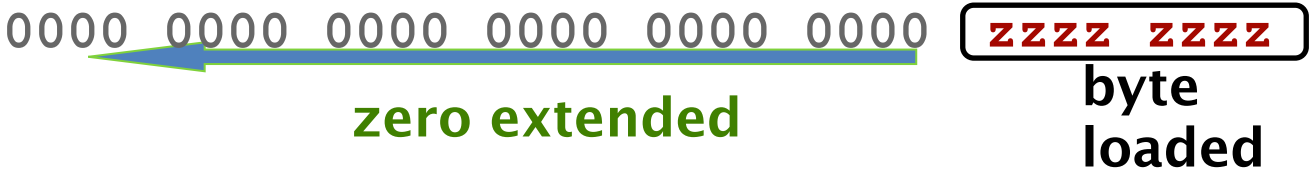
- Load word
  - Loads one word from memory into the destination register
- Load Byte
  - Loads one byte from memory into the lowest byte of the destination register
  - Value is sign extended
- Load Byte Unsigned
  - Loads one byte from memory into the lowest byte of the destination register
  - Value is not sign extended

Review: Load Instructions

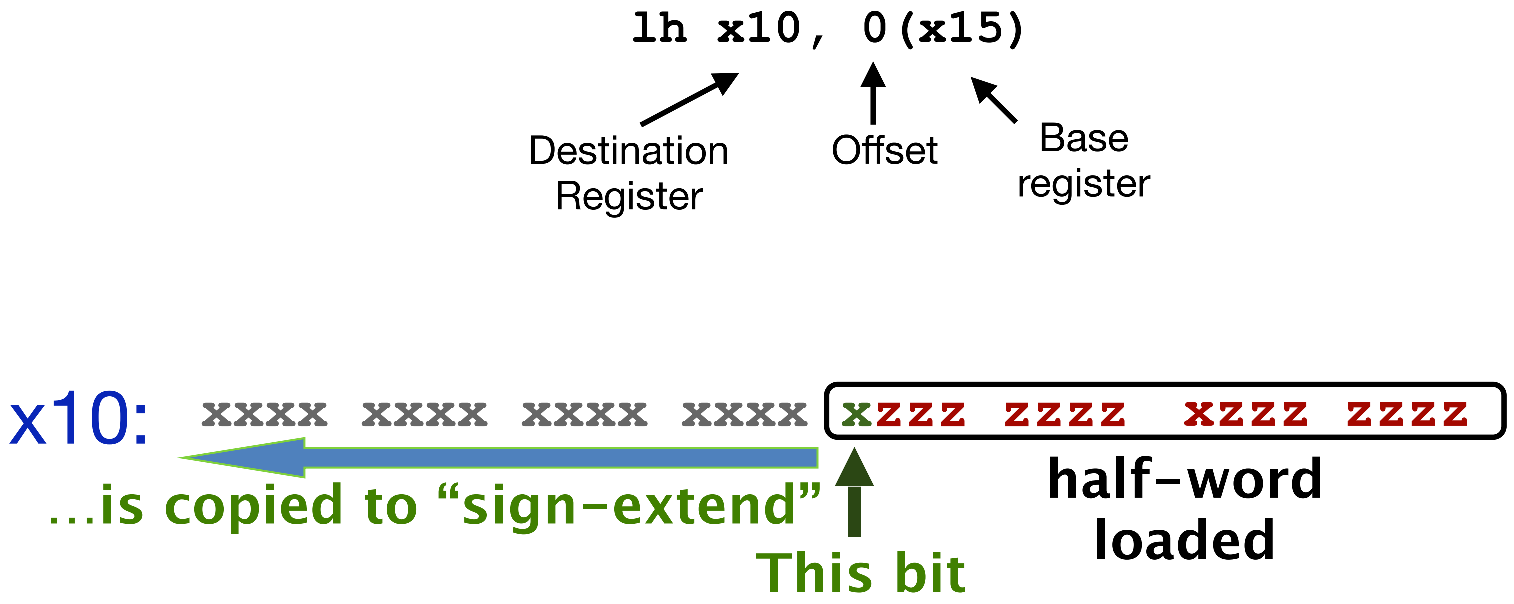
Load Byte



Load Byte Unsigned

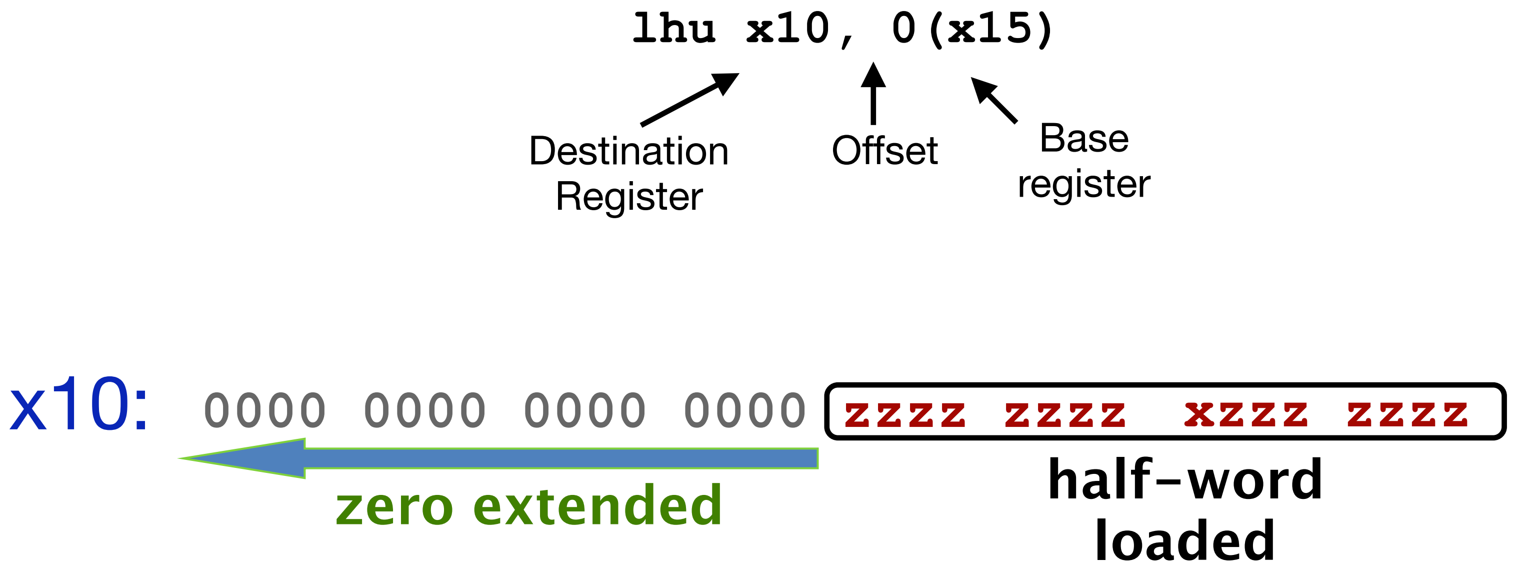


Load Halfword

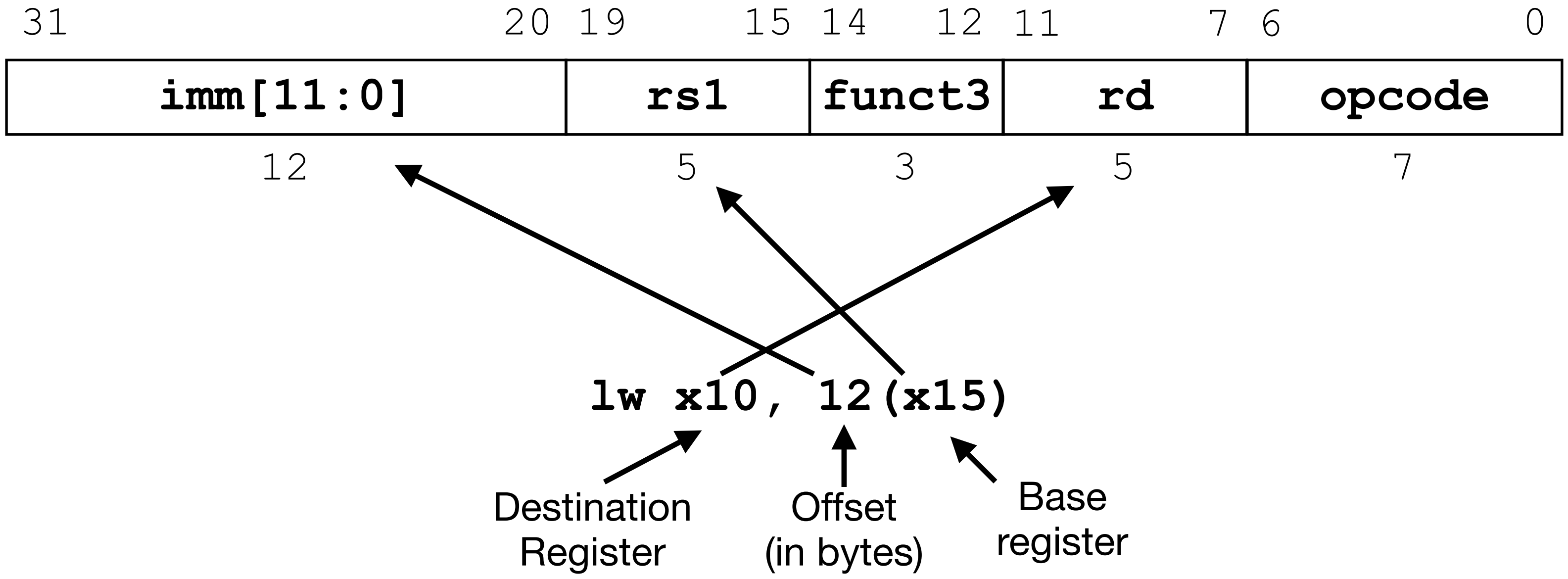


```
lh x10, 0(x15)
```

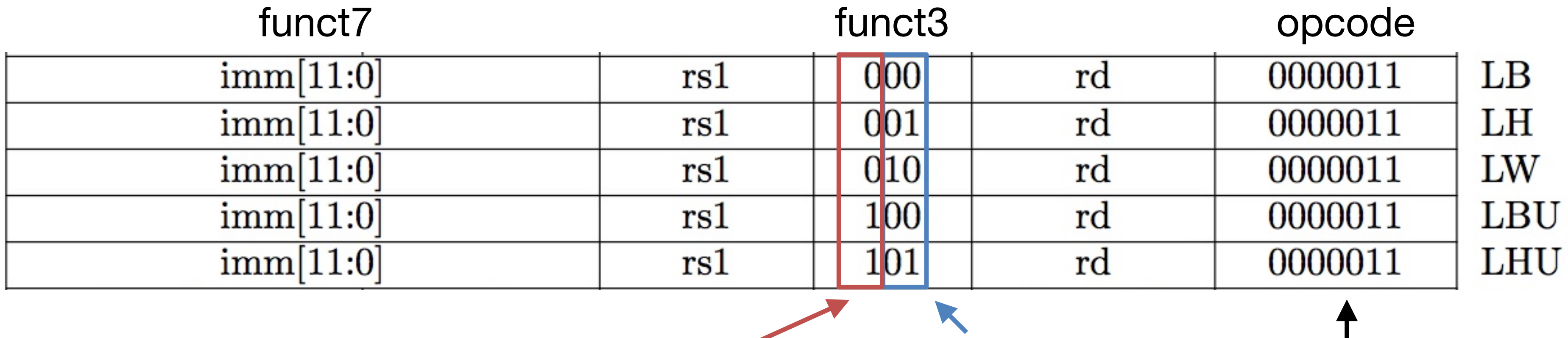
Load Halfword Unsigned



Load Instructions are also I-Type



All RV32 Load Instructions



funct7		funct3		opcode	
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

There is no lwu because a register is 32 bits, so its never sign extended

MSB tells us if it’s unsigned

0 = signed

1 = unsigned

Bottom 2 bits tell us how much to load

00: 1 byte

01: 2 bytes

10: 4 bytes

Same opcode

Load Example

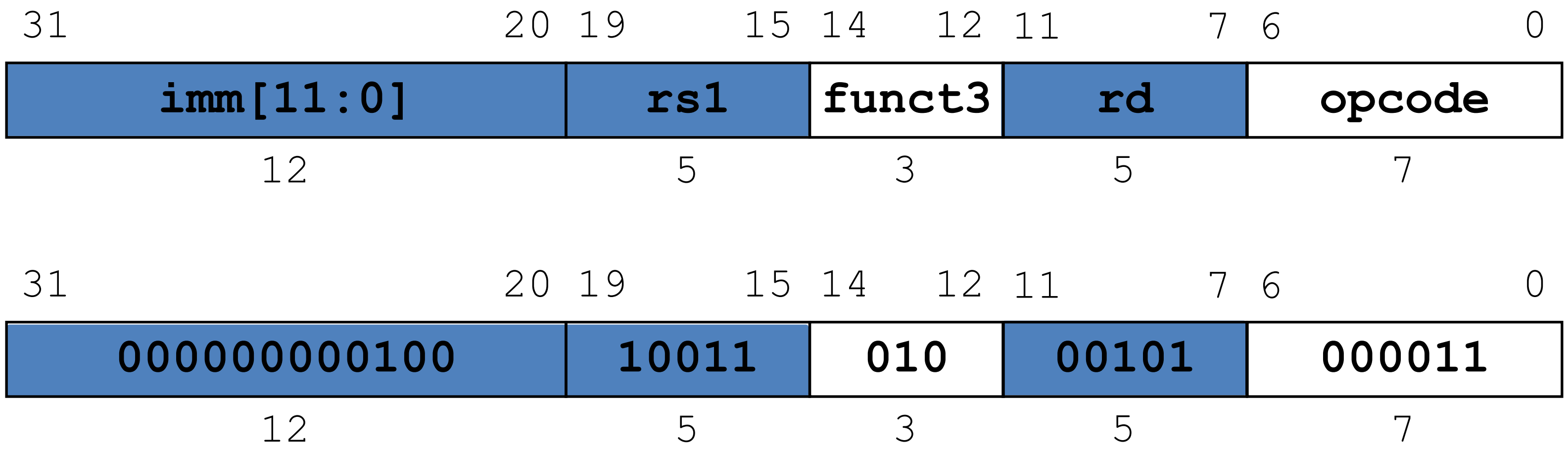
`lw x18, 4(x19)`

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	010	rd	000011
12	5	3	5	7

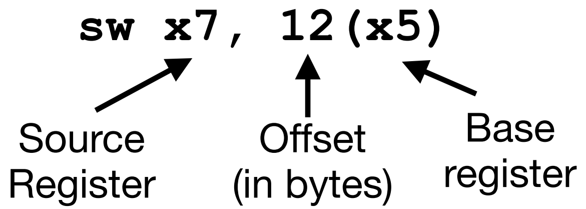
Load Example

`lw x5, 4(x19)`



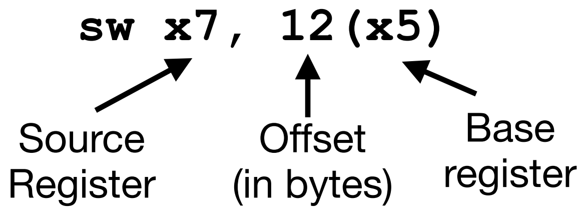
S-Format

Store Instructions



- Stores have 2 source registers, no destination
- I-type instructions have 1 source and 1 destination
- We want to prioritize keeping registers in the same place

Store Instructions



R-format

31 25	24 20	19 15	14 12	11 7	6 0
funct7	**rs2**	**rs1**	**funct3**	**rd**	**opcode**
7	5	5	3	5	7

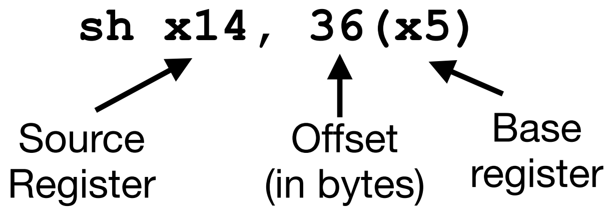
S-format

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	**rs2**	**rs1**	**funct3**	**imm[4:0]**	**opcode**
7	5	5	3	5	7

Review: Store Instructions

- Store Word
  - Stores the entire contents of the source register into memory
- Store Byte
  - Stores the least significant byte of the source register into memory
- No Sign extension

Store Halfword



```
sh x14, 36(x5)
```

- Stores the lower 16 bits of register x14 into memory at address [x5] + 36
- No sign extension

All RV32 Store Instructions

imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

Bottom 2 bits tell us how much to store

00: 1 byte
01: 2 bytes
10: 4 bytes

Same opcode

Store Format Example

`sw x14, 36(x5)`

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	**rs2**	**rs1**	**funct3**	**imm[4:0]**	**opcode**
7	5	5	3	5	7

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	**rs2**	**rs1**	**010**	**imm[4:0]**	**0100011**
7	5	5	3	5	7

Store Format Example

`sw x14, 36(x5)`

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	**rs2**	**rs1**	**funct3**	**imm[4:0]**	**opcode**
7	5	5	3	5	7

31 25	24 20	19 15	14 12	11 7	6 0
0000001	**01110**	**00101**	**010**	**00100**	**0100011**
7	5	5	3	5	7

$36 = 0b\boxed{01}\boxed{00100}$

Branches

Recall: Conditional Branches

- Used for ifs, loops, etc...
- Format: {comparison} {reg1} {reg2} {label}
  - beq
    bne
    blt, bltu
    bge, bgeu

Recall: Incrementing PC

- RV32 instructions are 32 bits = 4 bytes
- When we want to move to the next instruction, the processor increments PC by 4 bytes

```
if (a == b)
     e = c + d;
else
    e = c - d;
```

x10 = a
x11 = b
x12 = c
x13 = d
x14 = e

```
PC ──▶ bne x10,x11,else
       add x14,x12,x13
       j done
else:  sub x14,x12,x13
done:
```

Recall: Conditional Branches

- Format: {comparison} {reg1} {reg2} {label}
- If we don’t branch
  - PC = PC + 4
- If we do branch
  - PC = PC + immediate

What range of instructions can we branch to?

- 2's complement range: $[-2^{n-1}, 2^{n-1}-1]$
- With 12 bits: $\pm\ 2^{11}$ bytes away from the PC
- Instructions are 4-bytes, so we can jump $\pm\ 2^{9}$ instructions away from the current instruction

Instruction Addressing

0x10000000	Instruction 0
0x10000004	Instruction 1
0x10000008	Instruction 2
0x1000000C	Instruction 3
0x10000010	Instruction 4
0x10000014	Instruction 5
0x10000018	Instruction 6
0x1000001C	Instruction 7

The last nibble is always `0x(0, 4, 8, or C)`

```
0x0 = 0b0000
0x4 = 0b0100
0x8 = 0b1000
0xC = 0b1100
```

The last two bits are always `0b00`

Branch Instruction Addressing

- The last two bits are always 0, so we can increase our range by not storing those bits

encoded immediate = 0b0000 0000 0011

actual immediate offset = 0b00 0000 0000 1100

- (PC + immediate) will go 3 instructions (or 12 bytes) away
- Now, we can jump ± 2^{11} instructions (or ± 2^{13} bytes) away from the current PC

RISC-V Feature, n×16-bit instructions

- Extensions to RISC-V base ISA support 16-bit compressed instructions and also variable-length instructions that are multiples of 16 bits in length

Instruction Addressing

0x10000000	Instruction 0
0x10000002	Instruction 1
0x10000004	Instruction 2
0x10000006	Instruction 3
0x10000008	Instruction 4
0x1000000A	Instruction 5
0x1000000C	Instruction 6
0x1000000E	Instruction 7

The last nibble always ends in an even number

```
0x0  =  0b0000
0x2  =  0b0010
0x4  =  0b0100
0x6  =  0b0110
0x8  =  0b1000
0xA  =  0b1010
0xC  =  0b1100
0xE  =  0b1110
```

The last bit is always `0b0`

RISC-V Feature, nx16-bit instructions

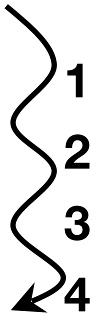
- With the 16-bit encoding, we can only save room by not storing the last bit
- Don't want to have two different branch encodings, so we will choose to only discard 1 bit instead of discarding 2 bits
- Range of bytes we can jump to
  - $\pm 2^{12}$
- Range of 32-bit instructions we can jump to
  - $\pm 2^{10}$

Branch Format

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	**imm[10:5]**	**rs2**	**rs1**	**funct3**	**imm[4:1]**	**imm[11]**	**opcode**
1	6	5	5	3	4	1	7

Branch Example

```
Loop: beq  x19,x10,End
       add  x18,x18,x10
       addi x19,x19,-1
       j    Loop
 End:   # target instruction
```



Count instructions from branch

- Branch offset = 4 instructions
  - (4 instructions x 4 bytes) = 16 bytes
  - 0b 0000 0001 0000

Branch Layout Example

`beq  x19, x10, End`

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	**imm[10:5]**	**rs2**	**rs1**	**funct3**	**imm[4:1]**	**imm[11]**	**opcode**
1	6	5	5	3	4	1	7

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	**imm[10:5]**	**rs2**	**rs1**	**000**	**imm[4:1]**	**imm[11]**	**1100011**
1	6	5	5	3	4	1	7

Branch Layout Example

`beq  x19, x10, End`

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	01010	10011	000	imm[4:1]	imm[11]	1100011
1	6	5	5	3	4	1	7

Branch Layout Example

`beq  x19, x10, End`

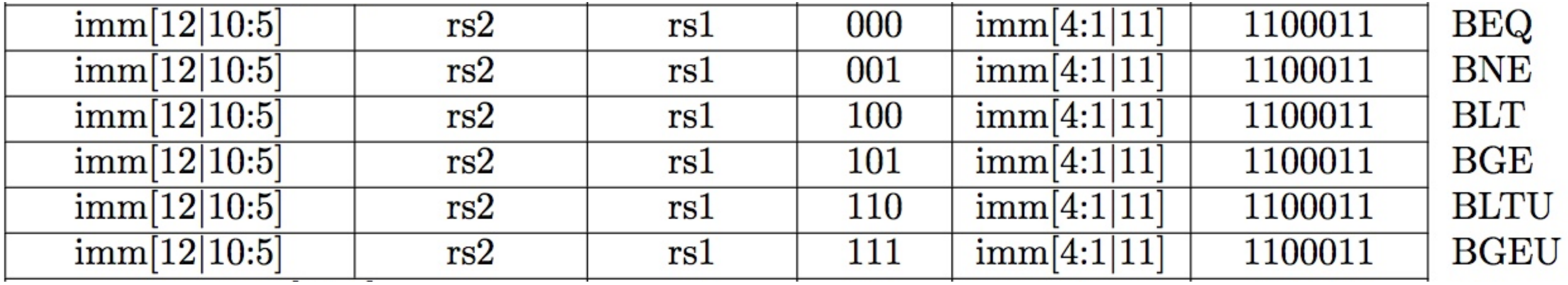
31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7

31	30 25	24 20	19 15	14 12	11 8	7	6 0
0	000000	01010	10011	000	1000	0	1100011
1	6	5	5	3	4	1	7

Branch offset = 16 bytes = 0b 0000 0001 0000

0	0	000000	1000	0

All RISC-V Branch Instructions



imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

funct3 field specifies with type

Same opcode

Conditional Branches

- Used for if-else statements and loops
  - These are usually pretty small (< 50 lines of code)
- Branch instructions have a limited range
  - $\pm 2^{10}$ 32-bit instructions
- What do we do if the location where we want to jump is farther away?

```
beq x10,x0,far
# next instr
```

```
      bne x10,x0,next
      j   far
next: # next instr
```

J-Format

Recall: JAL

`jal rd, Label` $\leftarrow$ The label that we want to jump to

$\uparrow$ rd = register where the return address will be stored

rd = return address

PC = PC + offset

- The label that we want to jump to gets translated by the assembler to a 20-bit offset

J-Format

- 20 bits for immediate field
- Just like with branch instructions, we can leave off the last bit
- So we can jump by
  - ± 2^{20} bytes
  - ± 2^{18} 32-bit instructions

J-Format

`jal rd, Label` ← The label that we want to jump to

↑ rd = register where the return address will be stored

31	30 21	20	19 12	11 7	6 0
`imm[20]`	**`imm[10:1]`**	**`imm[11]`**	**`imm[19:12]`**	**`rd`**	**`opcode`**
1	10	1	8	5	7

JALR

Recall: JALR

Register containing the base address (source register)

```
jalr rd, rs, imm
```

rd = register where the return address will be stored

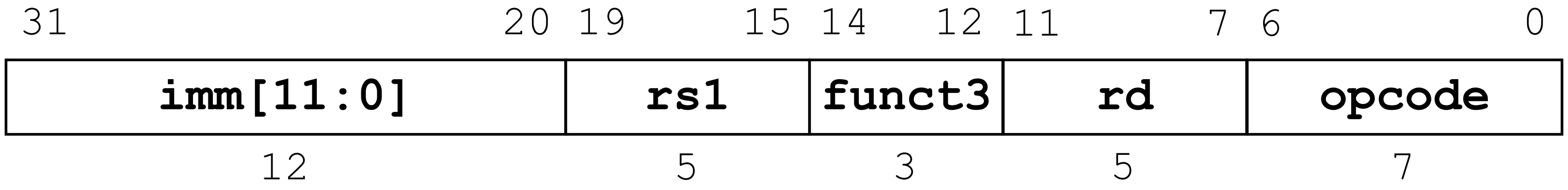
Immediate value to be added to the base register

rd = return address

PC = [rs] + imm

JALR is an I-type Instruction

- Usually used to return from a function (ret)
- **`imm[11:0]`** can hold values in range [-2048$_{10}$, +2047$_{10}$]
- Immediate is always sign-extended to 32-bits before use
- Unlike JAL, we must include the last 0 because we are using the I-format which specifies that we include the 0th bit



PC-Relative Address vs Absolute Address

- PC Relative Address
  - Jump to a location based on the current location of the PC
  - PC + offset
- Absolute Address
  - Jump to a location using that location’s full address

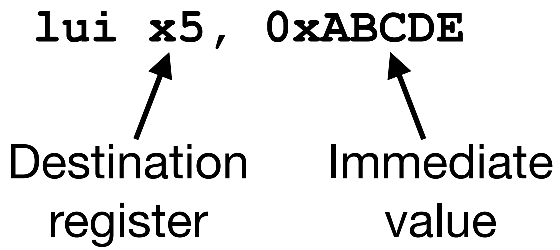
Questions on PC-addressing

- Does the value in branch immediate field change if we change the location of the code in memory?
  - If moving all of code, then no (because PC-relative offsets)

U-Format

Load Upper Immediate (LUI)

- destination register = immediate << 12



```
lui x5, 0xABCDE
```

```
x5 = 0xABCDE000
```

LUI to create long immediates

- LUI writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits.
- Together with an ADDI to set low 12 bits, can create any 32-bit value in a register using two instructions (LUI/ADDI).

```
LUI x10, 0x87654         # x10 = 0x87654000
ADDI x10, x10, 0x321     # x10 = 0x87654321
```

Load Immediate

- Pseudo-instruction that performs lui and addi

```
LUI x10, 0x87654          # x10 = 0x87654000
ADDI x10, x10, 0x321      # x10 = 0x87654321
```

$$\downarrow$$

```
LI x10, 0x87654321        # x10 = 0x87654321
```

One Corner Case

- How to set 0xABCDEEEE?

```
lui x10, ABCDE  # x10 = 0xABCDE000
addi x10, 0xEEE # x10 = 0xABCDDEEE
```

```
  0xABCDE000
+ 0xFFFFFEEE
  ----------
  0xABCDDEEE
```

```
  0xABCDE      0x000
+ 0xFFFFF    + 0xEEE
  -------      -----
  0xABCDD      0xEEE
```

Solution

- If the value being added is negative, add 1 to the upper 20 bits before adding the 12-bit value
- How to set 0xABCDEEEE?

```
lui x10, ABCDF  # x10 = 0xABCDF000
addi x10, 0xEEE # x10 = 0xABCDEEEE
```

- `li` instruction will automatically handle this corner case

Add Upper Immediate PC (AUIPC)

- rd = PC + (immediate << 12)

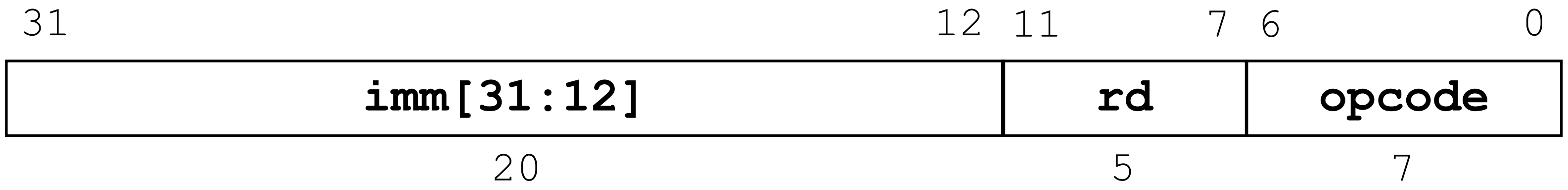
```
auipc x5, 0xABCDE
```

Destination register

Immediate value

```
x5 = PC + 0xABCDE000
```

U-Format for Upper Immediate Instructions



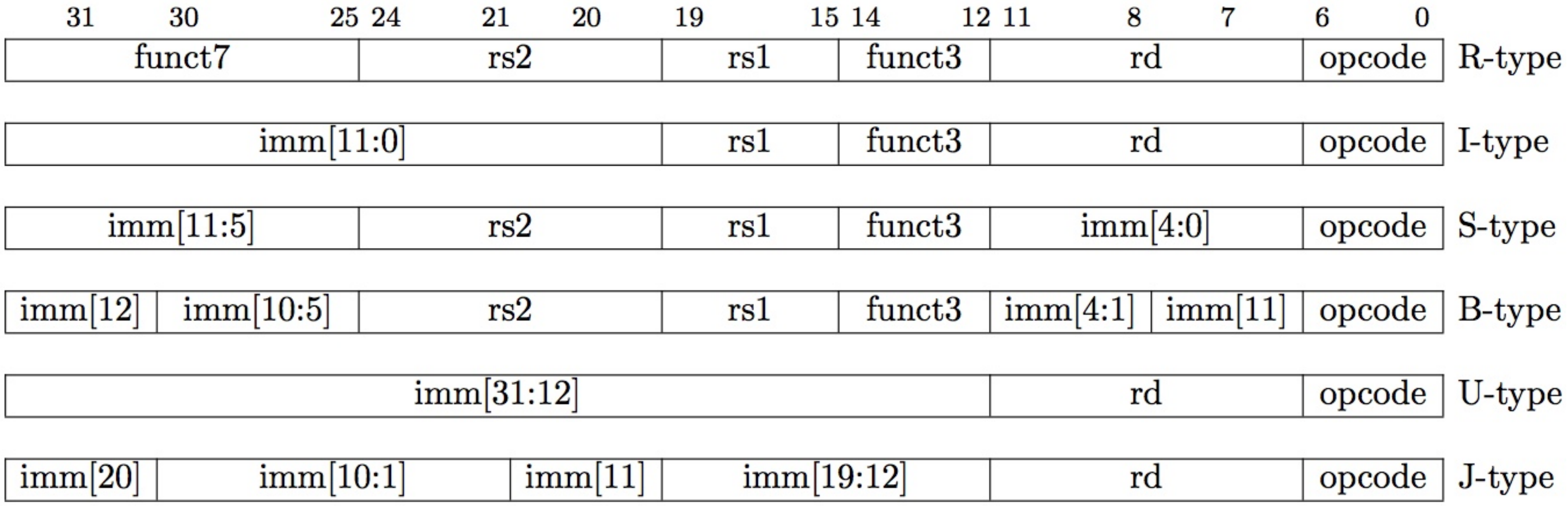
lui opcode = 0b0110111

auipc opcode = 0b0010111

LUI and AUIPC with JALR

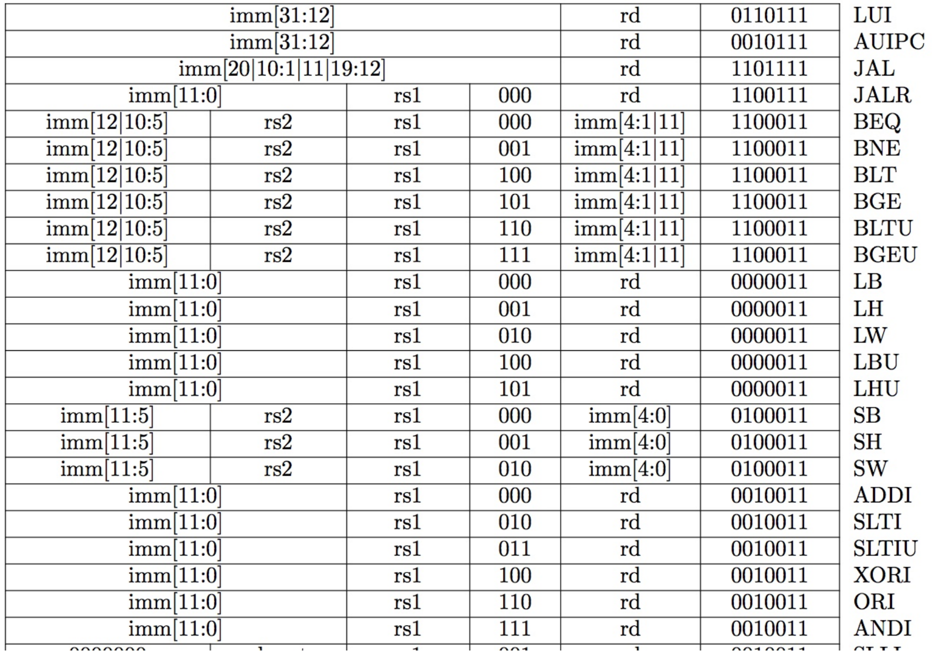
```
# Call function at any 32-bit absolute address
lui  x5, <hi20bits>
jalr ra, x5, <lo12bits>


# Jump PC-relative with 32-bit offset
auipc x5, <hi20bits>
jalr ra, x5, <lo12bits>
```

Summary of RISC-V Instruction Formats

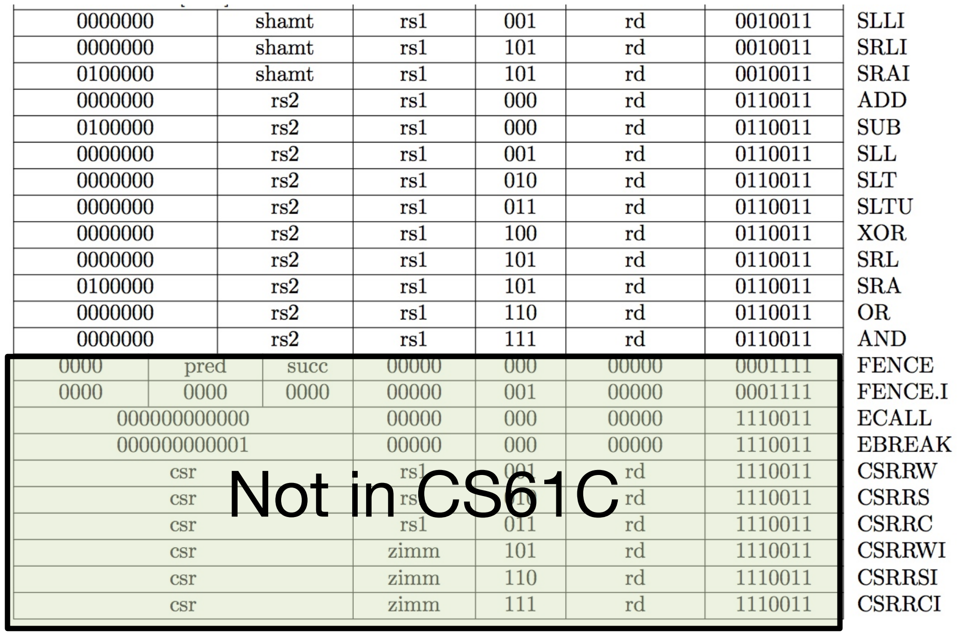


31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

Complete RV32I ISA



imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
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
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
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
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000 pred	succ	00000	000	00000	0001111	FENCE
0000 0000	0000	00000	001	00000	0001111	FENCE.I
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	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

Not in CS61C