National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 3: RISC-V Instruction Set, Part 2

CS10014 Computer Organization

Department of Computer Science
Tsung Tai Yeh
Thursday: 1:20 pm– 3:10 pm
Classroom: EC-022

Acknowledgements and Disclaimer

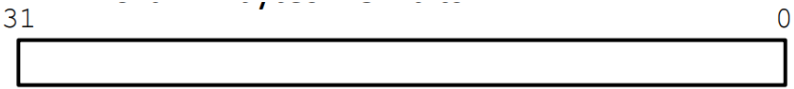
- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - CS 252 at UC Berkeley
    - https://people.eecs.berkeley.edu/~culler/courses/cs252-s05/
  - CSCE 513 at University of South Carolina
    - https://passlab.github.io/CSCE513/

Outline

- R-Format
- I-Format
- S-Format
- SB-Format
- U-Format
- UJ-Format

Instructions as Numbers

- RISC-V instructions are each
  - 1 word = 4 bytes = 32 bits



- Divide the 32-bit instruction into “fields”
  - Regular field sizes -> simpler hardware
  - Will need some variation …
- Define 6 types of instruction formats
  - R-Format, I-Format, S-Format, U-Format
  - SB-Format, UJ-Format

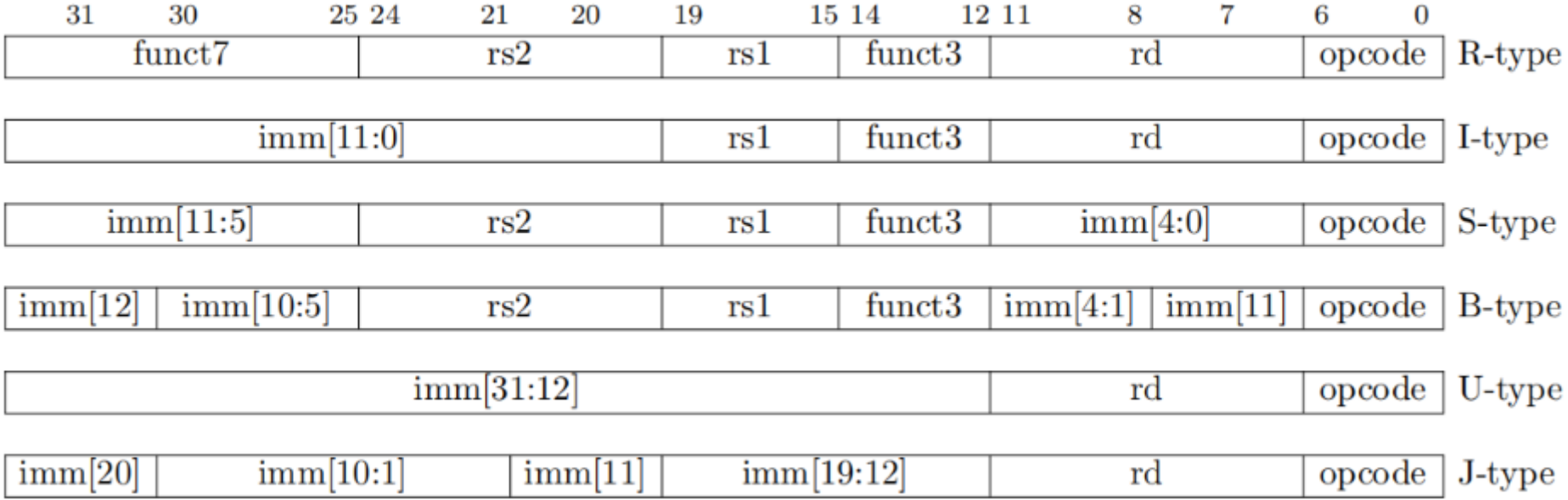
RISC-V Instruction format

- Most data we work with is in words (32-bit chunks)
  - Each register is a word
  - lw and sw both access memory one word at a time
- How do we represent instructions?
  - Instructions are fixed-size 32-bit words
  - Same 32-bit instruction definitions used from RV32, RV64, RV128

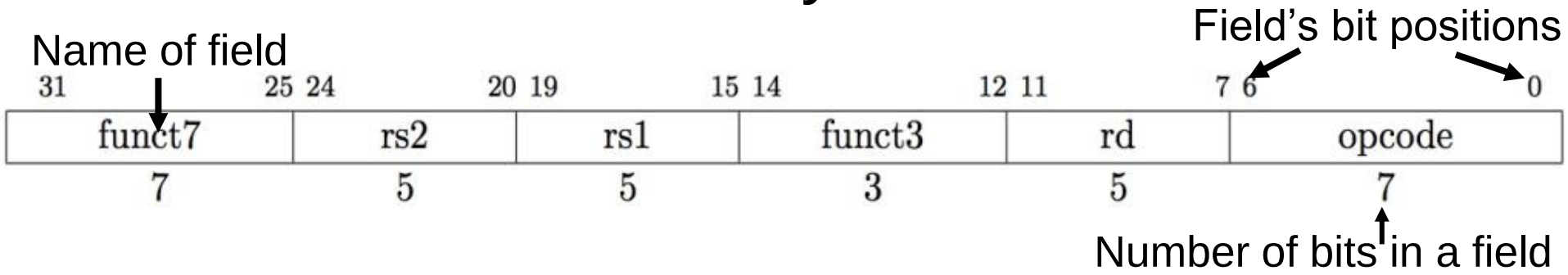
RISC-V 6 Instruction format

- R-format: arithmetic/logical operations (add/xor), instructions using register inputs
- I-format: instructions with immediates, loads (addi/lw/jalr)
- S-format: for stores (sw/sb)
- B-format: for branches (beq, bge)
- U-format: 20-bit upper immediate instructions (lui/auipc)
- J-format: for jumps (jal)

RISC-V Instruction format

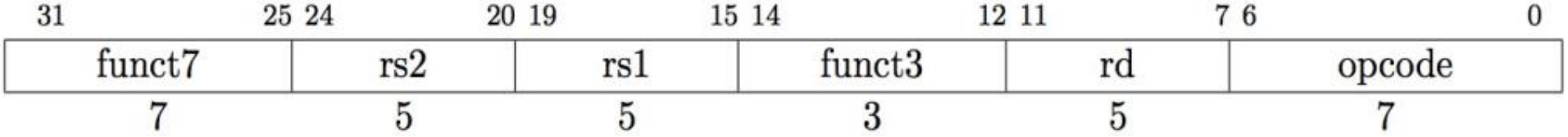


R-Format Instruction Layout



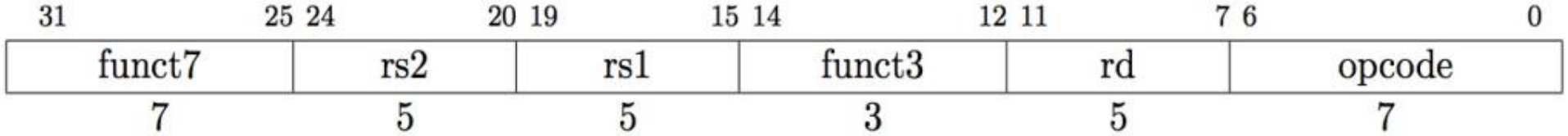
- 32-bit instruction word divided into six fields of varying numbers of bits each = 7+5+5+3+5+7 = 32
  - opcode is a 7-bit field that lives in bits 6-0 of the instruction
  - rs2 is a 5-bit field that lives in bits 24-20 of the instruction
  - Recall: RISC-V has 32 registers
    - A 5-bit field can represent exactly 2^5 = 32 that interpret as the register numbers x0-x31

R-Format Instructions opcode/funct fields



- opcode: partially specifies which instruction it is
  - Note that the field is equal to 0110011_{two} for all R-format
  - Note that the field is equal to 1100011_{two} for all B-format
- funct7+funct3: these two fields describe what operation to perform
- How many R-format instructions can we encode?
  - According to the funct varies: $(2^7) \times (2^3) = 2^{10} = 1024$

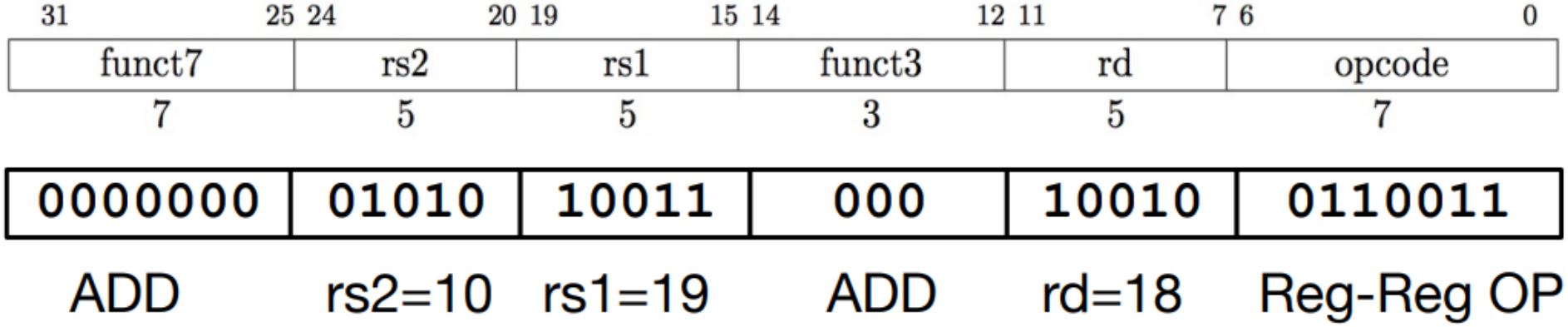
R-Format Instructions register specifiers



- 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)
  - Specifies register containing the first operand
- rs2 (Source Register #2)
  - Specifies the second register operand
- rd (Destination Register)
  - Specifies register which will receive a result of a computation

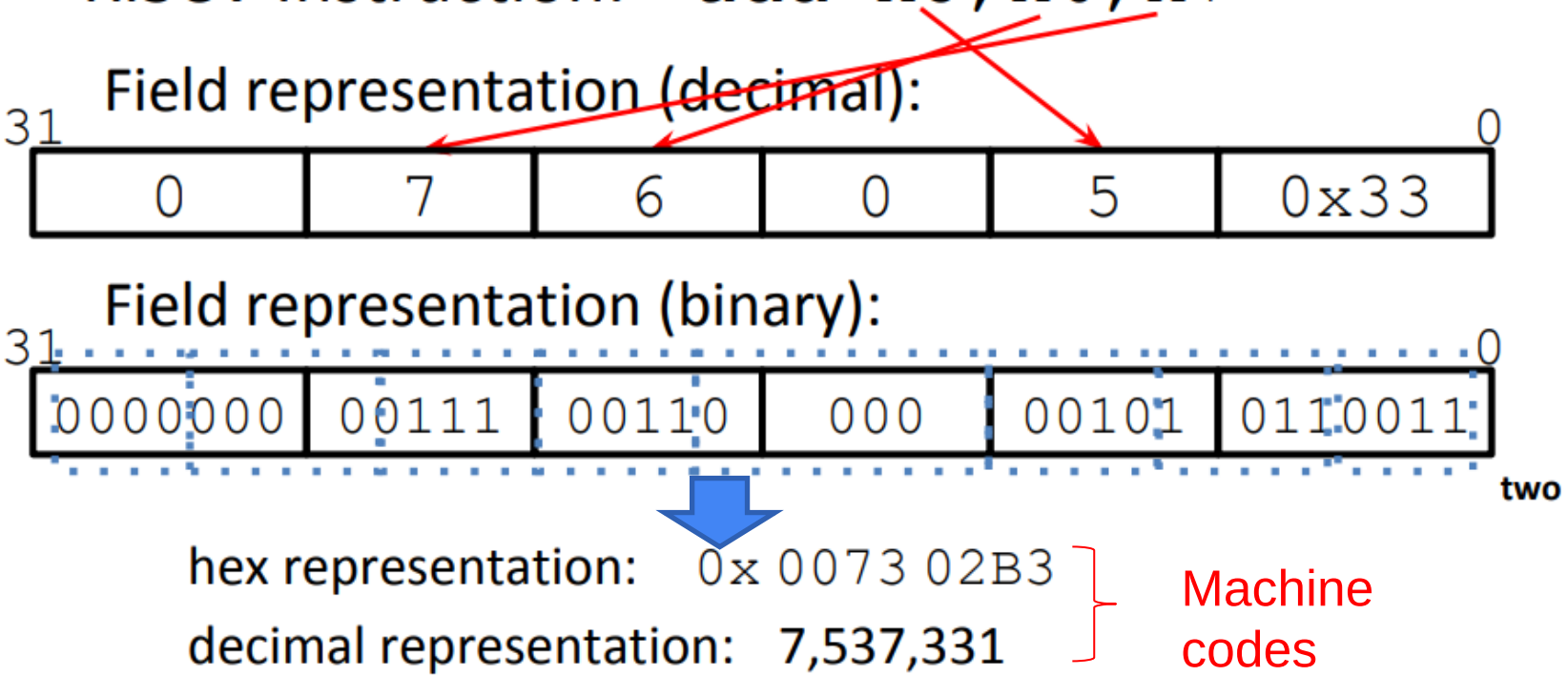
R-Format Example

- RISC-V assembly instruction
  - add x18, x19, x10
  - Why aren't combining funct7 and funct3 as a single 10-bit field?



R-Format Example

- RISCV Instruction: `add x5,x6,x7`



R-Format Example

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

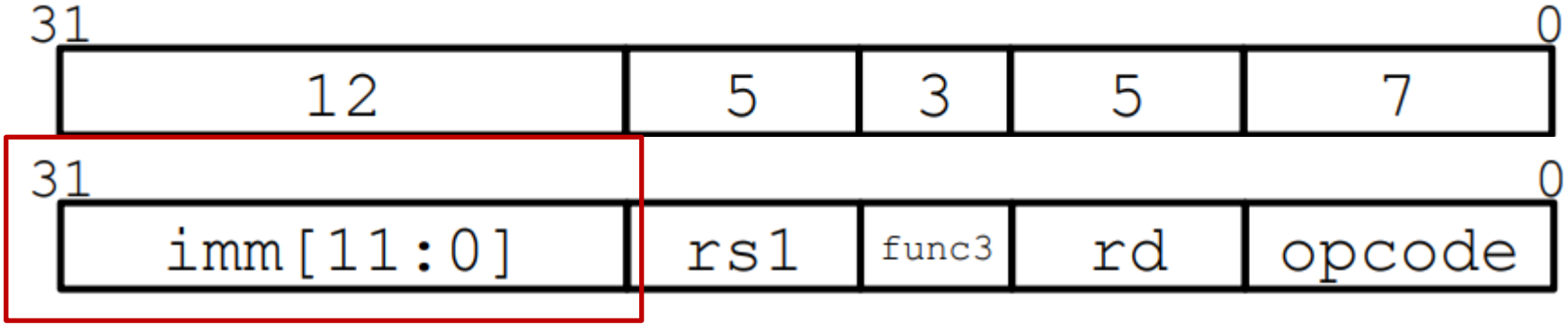
Encoding in funct7 + funct3 selects particular operation

Outline

- R-Format
- I-Format
- S-Format
- SB-Format
- U-Format
- UJ-Format

I-Format Instruction

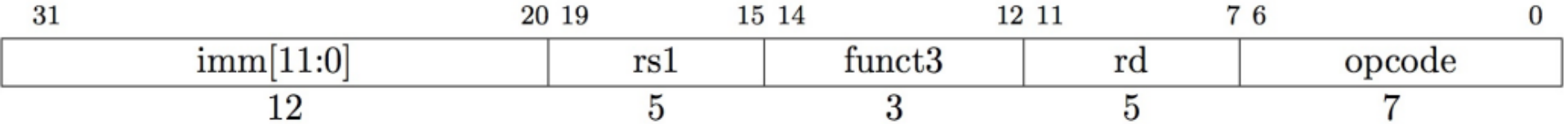
- What about instructions with immediates?
  - addi x15, x1, -50
  - The 5-bit field is too small for most immediates



  - Only the imm field is different from the R-format
    - rs2 and funct7 are replaced by 12-bit signed immediate

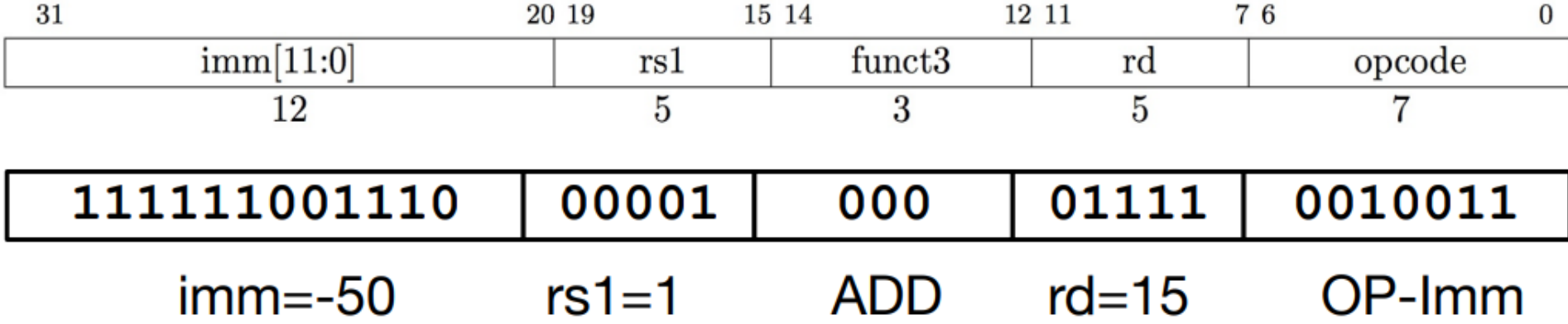
I-Format Instruction

- Immediate (12): 12-bit number?
  - All computations are done in words, so 12-bit immediate must be extended to 32-bits
  - Always sign-extended to 32-bits before used in an arithmetic operation
  - imm[11:0] can hold values in range [-2048_{ten}, $+2047_{ten}$]
  - How does the immediate handle > 12 bits values?



I-Format Example

- addi x15, x1, -50



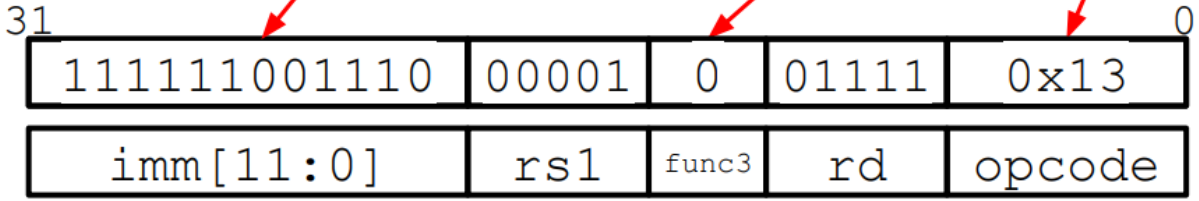
I-Format Example

```
addi x15,x1,-50
```

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
fence.i	I	0001111	001		0F/1
addi	I	0010011	000		13/0
slli	I	0010011	001	0000000	13/1/00

```
rd  = x15
rs1 = x1
```



All I-Type Arithmetic Instructions

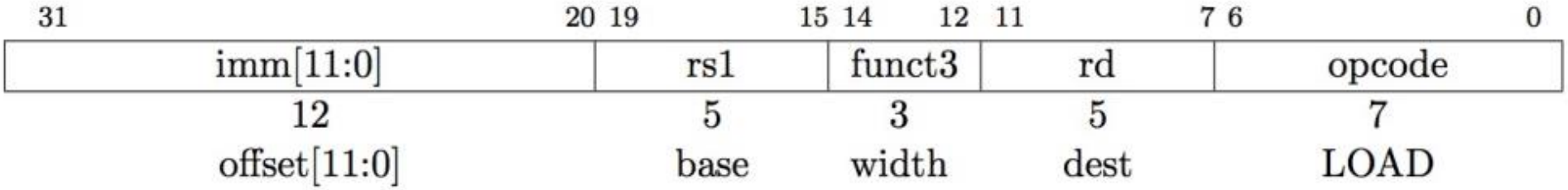
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	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)

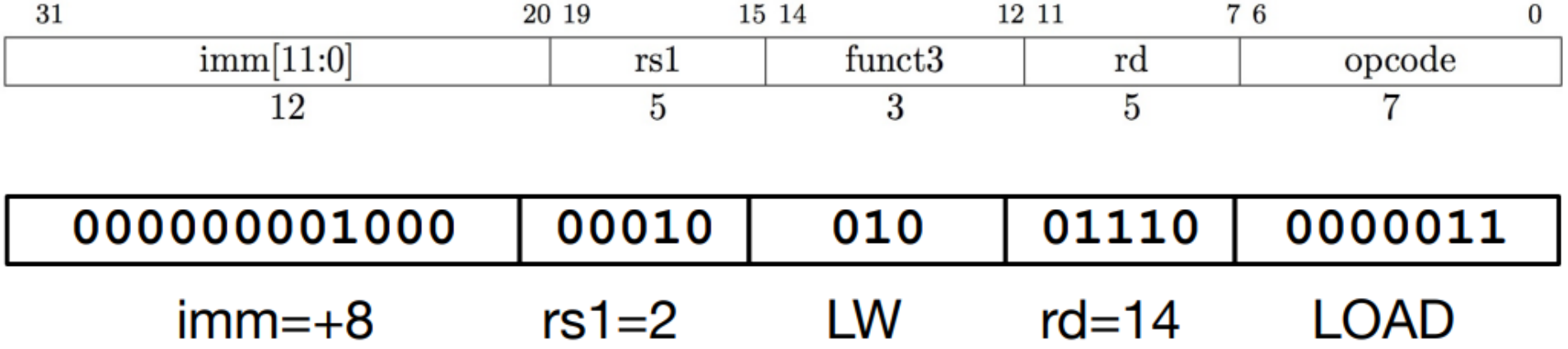
Memory Load Instruction

- Memory load instructions are also I-type
  - 12-bit signed immediate is added to the base address in register rs1 to form the memory address
  - The value loaded from memory is stored in register rd



I-Format Load Example

- lw x14, 8(x2)



Takeaway Questions

- In I-type, if the number of registers were halved, which statement is true?
  - (A) There must be fewer I-type instructions
  - (B) There must be more R-type instructions
  - (C) I-type instructions could have 2 more immediate bits

Outline

- R-Format
- I-Format
- S-Format
- SB-Format
- U-Format
- UJ-Format

All RV32 Load Instructions

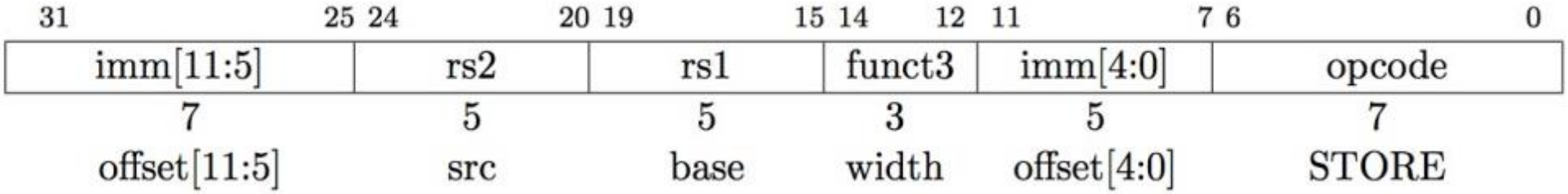
imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU

funct3 field encodes size and signedness of load data

- LBU: load unsigned byte
- LH: load halfword, which loads 16 bits (2 bytes) and sign-extends to fill the destination 32-bit register
- LHU is a load unsigned halfword, which zero-extends 16 bits to fill destination 32-bit register

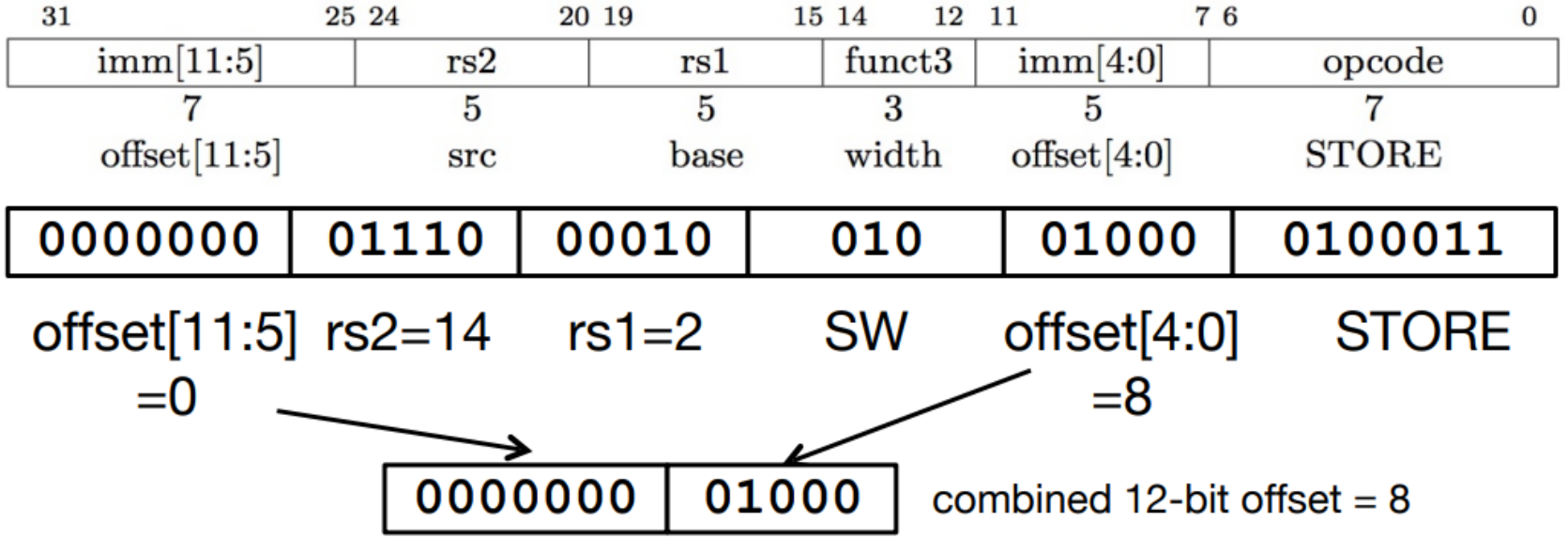
S-Format for Stores

- Store needs to read two registers
  - rs1 for base memory address
  - rs2 for data to be stored, as well as need immediate offset
  - The store instruction doesn't write a value to the register file, no rd !



S-Format Example

- sw x14, 8(x2)



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

Outline

- R-Format
- I-Format
- S-Format
- SB-Format
- U-Format
- UJ-Format

RISC-V Conditional Branches

- beq, bne, bge, blt
  - BEQ x1, x2, Label
  - Need to specify an address to go to
  - Also take **two** registers to compare
  - Don’t write into a register (similar to stores)
  - How do you encode the label, i.e., where do you branch to?

Branching Instruction Usage

- Branches are typically used by “for loops” (if-else, while, for)
  - Loops are generally small (< 50 instructions)
  - Function calls and unconditional jumps handled with jump instructions (J-Format)
- Instructions stored in a localized area of memory (Code/Text)
  - Largest branch distance limited by size of code
  - Address of current instruction stored in the program counter (PC)

PC-Relative Addressing

- PC-Relative Addressing
  - Use the immediate field as a two's-complement offset to PC
  - Branches generally change the PC by a small amount
  - Can specify $\pm 2^{11}$ addresses from the PC
- Why not use byte address offset from PC as the *immediate*?

Branching Reach

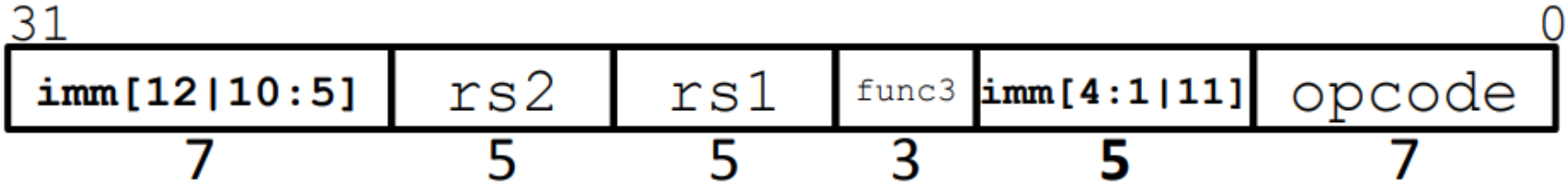
- RISC-V uses 32-bit addresses, and memory is **byte-addressed**
  - Instructions are “word-aligned”
    - Address is always a multiple of 4 (in bytes)
  - PC always points to an instruction
    - PC is typed as a pointer to a word
    - Can do C-like pointer arithmetic
  - Let immediate specify # of words instead of # of bytes
    - Instead of specifying $\pm 2^{11}$ bytes from the PC
    - Specifying $\pm 2^{11}$ words = $\pm 2^{13}$ bytes addresses around PC

Branch Calculation

- If we don’t take the branch
  - PC = PC + 4 = next instruction
- If we do take the branch
  - PC = PC + (immediate * 4)
- “Immediate” indicates the number of instructions to move (remember, specifies words) either forward (+) or backwards (-)

RISC-V B-Format for Branches

- B-Format is mostly the same as S-Format
  - With two register sources (rs1/rs2) and a 12-bit immediate
- But now immediate represents values -2^{12} to $+2^{12} - 2$ in 2-byte increments
  - The 12 immediate bits encode even a 13-bit signed byte offset (the lowest bit of offset is always zero, so there is no need to store it)

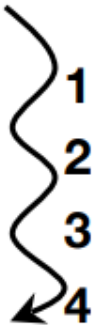


Branch Example (1/2)

- Determine offset
  - Branch offset = 4 x 32-bit instructions = 16 bytes
  - Branch with offset of 0, branches to itself

RISC-V Code:

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



Count instructions from branch

Branch Example (2/2)

- Encode offset

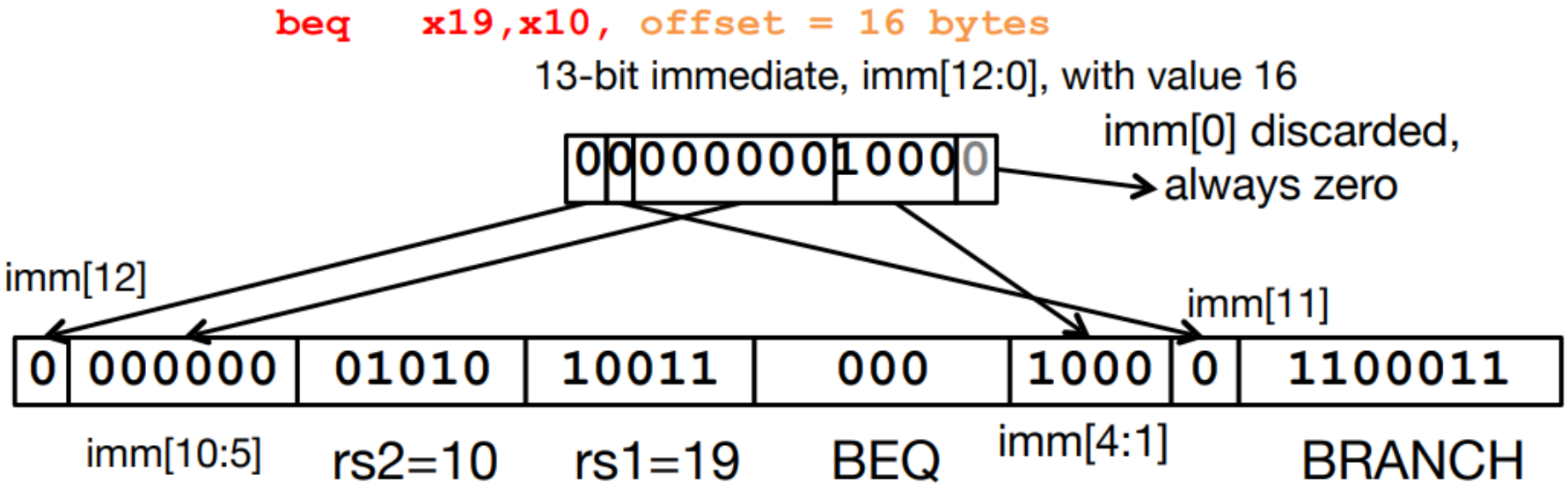
RISC-V Code:

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

???????	01010	10011	000	?????	1100011
imm	rs2=10	rs1=19	BEQ	imm	BRANCH

Branch Example (2/2)

- Complete encoding



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

Questions on PC-addressing

- Encode offset

RISC-V Code:

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

offset = 16 bytes = 8x2

???????	01010	10011	000	?????	1100011
imm	rs2=10	rs1=19	BEQ	imm	BRANCH

Questions on PC-addressing

- What do we do if the destination is > 2^{10} instructions away from the branch?
  - Other instructions save us

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

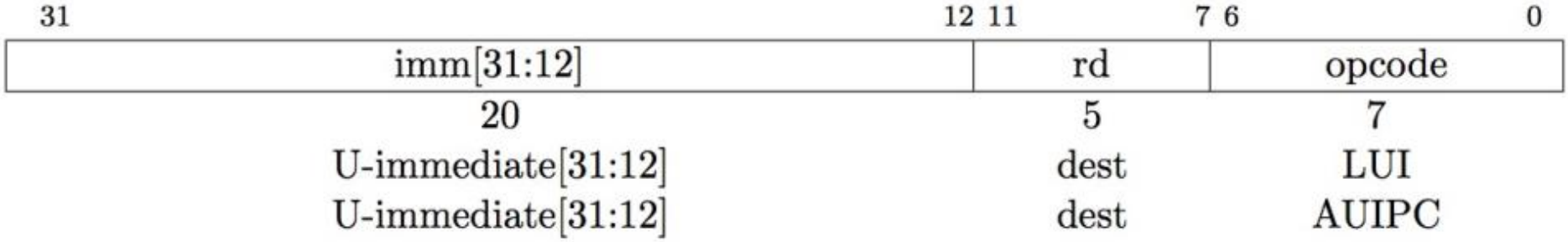
Dealing with Large Immediates

- How do we deal with 32-bit immediates?
  - The I-type instructions only give us 12 bits
- Solution
  - Need a new instruction format for dealing with the rest of the 20 bits
  - This instruction should deal with
    - A destination register to put the 20 bits into
    - The immediate of 20 bits
    - The instruction opcode

Outline

- R-Format
- I-Format
- S-Format
- SB-Format
- U-Format
- UJ-Format

U-Format for “Upper Immediate” Instructions



- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, rd
- Used for two instructions
  - LUI – Load Upper Immediate
  - AUIPC – Add Upper Immediate to PC

LUI to create long immediates

- LUI instruction
  - Write the upper 20 bits of the destination with the immediate value, and clear 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
```

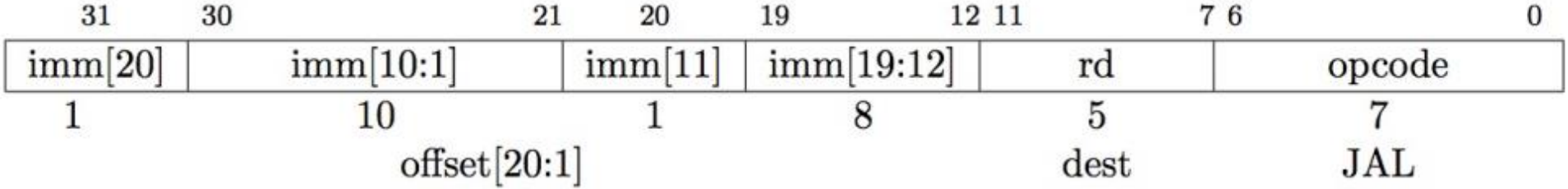
Outline

- R-Format
- I-Format
- S-Format
- SB-Format
- U-Format
- UJ-Format

UJ-Format Instructions (1/3)

- For branches, we assumed that we won't want to branch too far, so we can specify a change in the PC
- For general jumps (jal), we may jump to anywhere in code memory
  - Ideally, we would specify a 32-bit memory address to jump to
  - We cannot fit both a 7-bit opcode and a 32-bit address into a single 32-bit word
  - We must write to an rd register when linking

UJ-Format Instructions (2/3)



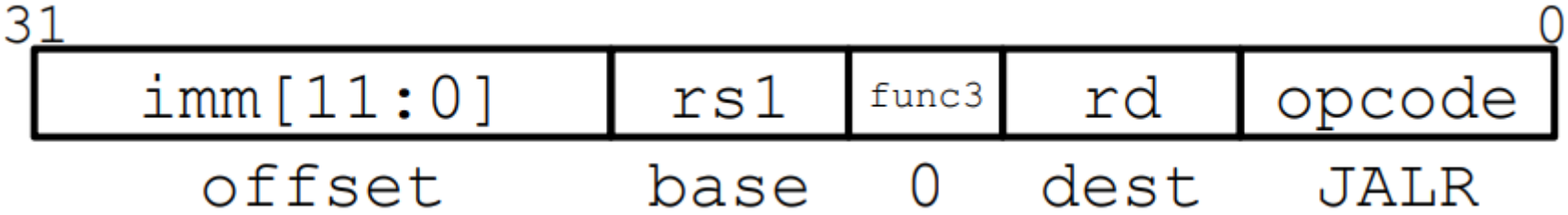
- jal saves PC+4 in register rd (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
- $\pm 2^{18}$ 32-bit instructions
- “j” jump is a pseudo-instruction – the assembler will instead use jal but sets rd=x0 to discard the return address

Uses of JAL

```
# j pseudo-instruction
j Label = jal x0, Label # Discard return address

# Call function within 2^18 instructions of PC
jal ra, FuncName
```

JALR Instruction (I-Format)



- jalr rd, rs1, offset
- Writes PC + 4 to rd (return address)
- Sets PC = rs1 + offset

Uses of jalr

```
# ret and jr psuedo-instructions
ret = jr ra = jalr x0, ra, 0
# Call function at any 32-bit absolute address
lui x1, <hi 20 bits>
jalr ra, x1, <lo 12 bits>
# Jump PC-relative with 32-bit offset
auipc x1, <hi 20 bits>
jalr x0, x1, <lo 12 bits>
```

auipc (Add Upper Immediate to Program Counter): this sets rd to the sum of the current PC and a 32-bit value with the low 12 bits as 0 and the high 20 bits coming from the U-type immediate.

Summary of RISC-V Instruction Formats