inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

Lecture 11 – RISC-V Instruction Representation II

Instructors
Dan Garcia and Bora Nikolic

2018-09-17

Developers, Despair: Half Your Time Is Wasted on Bad Code
A study by online payments firm Stripe found 31% of developer time is wasted on routine maintenance work, draining $300 billion each year from global GDP. The survey of 1,000 developers and 1,000 C-level executives found that on average, about half of a developer's work time is devoted to maintenance activities such as debugging, modifying, and fixing code. Developers reported working 41 hours per week on average, including 17.3 hours on maintenance work.
Liam Tung, ZDNet, Sept 11, 2018.

Review

- Simplification works for RISC-V: Instructions are same size as data word (one word) so that they can use the same memory.
- Computer actually stores programs as a series of these 32-bit numbers.
- RISC-V Machine Language Instruction:
  32 bits representing a single instruction
  - We covered R-type, I-type and S-type instructions
  - Learned a few more new instructions
  - `slt rd, rs1, rs2` - set less than: R[rd]=(R[rs1]<R[rs2])?1:0
  - `sltu, slti, sltiu`
  - `lh, lhu, sh`

Levels of Representation/Interpretation

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

Compiler

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

Assembler

Machine Language Program (RISC-V)

Machine Interpretation

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

Architecture Implementation

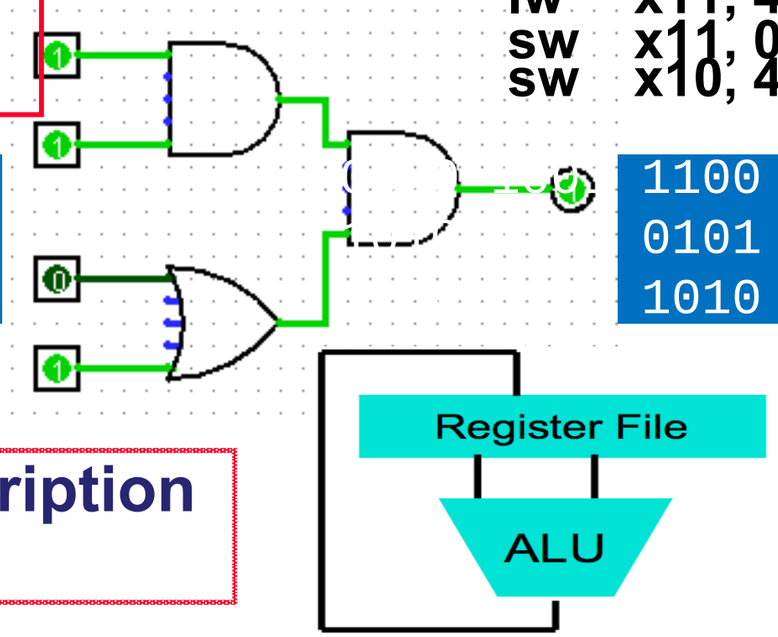
Logic Circuit Description (Circuit Schematic Diagrams)

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

```
lw   x10, 0(x12)
lw   x11, 4(x12)
sw   x11, 0(x12)
sw   x10, 4(x12)
```

Anything can be represented as a *number*, i.e., data or instructions

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



RISC-V Conditional Branches

- E.g., `BEQ x1, x2, Label`
- Branches read two registers but don't write a register (similar to stores)
- How to encode label, i.e., where to branch to?

Branching Instruction Usage

- Branches typically used for loops (`if-else`, `while`, `for`)
  - Loops are generally small (< 50 instructions)
  - Function calls and unconditional jumps handled with jump instructions (J-Format)
- Recall: 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 ± 2^{11} 'unit' addresses from the PC
  - (We will see in a bit that we can encode 12-bit offsets as immediates)
- Why not use byte as a unit of offset from PC?
  - Because instructions are 32-bits (4-bytes)
  - We don't branch into middle of instruction

Scaling Branch Offset

- One idea: To improve the reach of a single branch instruction, multiply the offset by four bytes before adding to PC
- This would allow one branch instruction to reach ± 2^{11} × 32-bit instructions either side of PC
  - Four times greater reach than using byte offset

Branch Calculation

- If we don't take the branch:

  `PC = PC + 4` (i.e., next instruction)

- If we do take the branch:

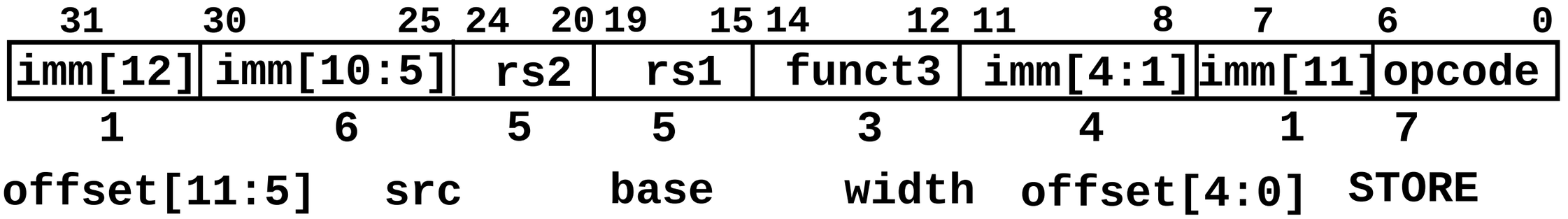
  `PC = PC + immediate*4`

- Observations:
  - `immediate` is number of instructions to jump (remember, specifies words) either forward (+) or backwards (–)

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
- To enable this, RISC-V scales the branch offset by 2 bytes even when there are no 16-bit instructions
- Reduces branch reach by half and means that ½ of possible targets will be errors on RISC-V processors that only support 32-bit instructions (as used in this class)
- RISC-V conditional branches can only reach ± 2^{10} × 32-bit instructions on either side of PC

RISC-V B-Format for Branches

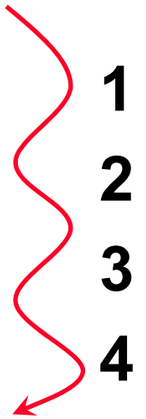


- B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate imm[12:1]
- But now immediate represents values -4096 to +4094 in 2-byte increments
- The 12 immediate bits encode *even* 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)

Branch Example, Determine Offset

- RISC-V Code:

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



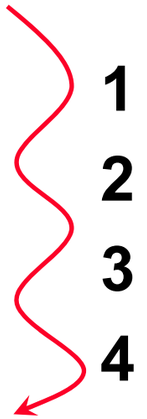
Count instructions from branch

- Branch offset = 4×32-bit instructions = 16 bytes
- (Branch with offset of 0, branches to itself)

Branch Example, Determine Offset

- 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

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

Branch Example, Encode Offset

- RISC-V Code:

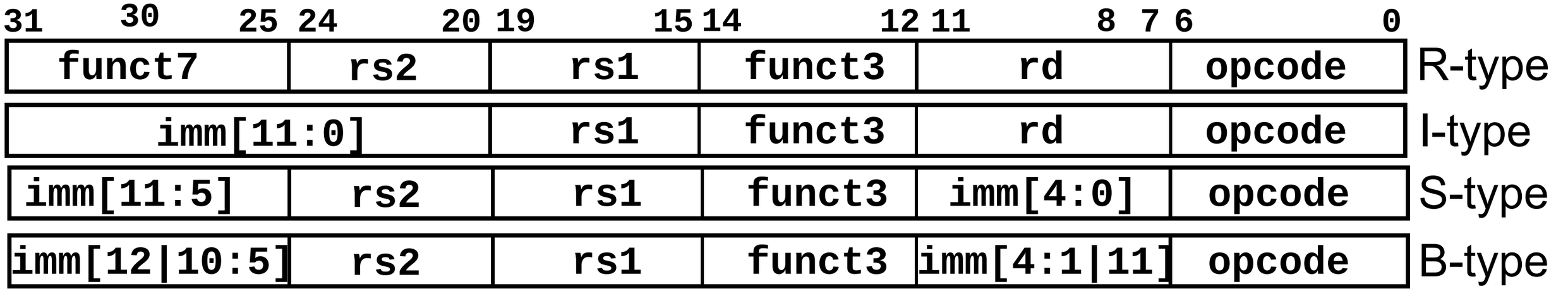
```
Loop: beq   x19,x10,End
      add   x18,x18,x10
      addi  x19,x19,-1
      j     Loop
End: # target instruction
```

offset = 16 bytes = 8x2 bytes

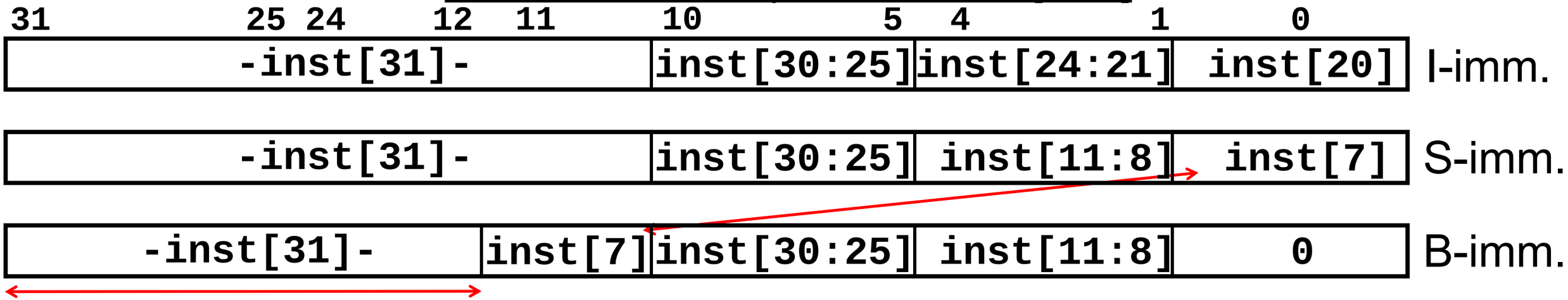
???????	01010	10011	000	?????	1100011
imm	rs2=10	rs1=19	BEQ	imm	BRANCH

RISC-V Immediate Encoding

Instruction encodings, inst[31:0]



32-bit immediates produced, imm[31:0]



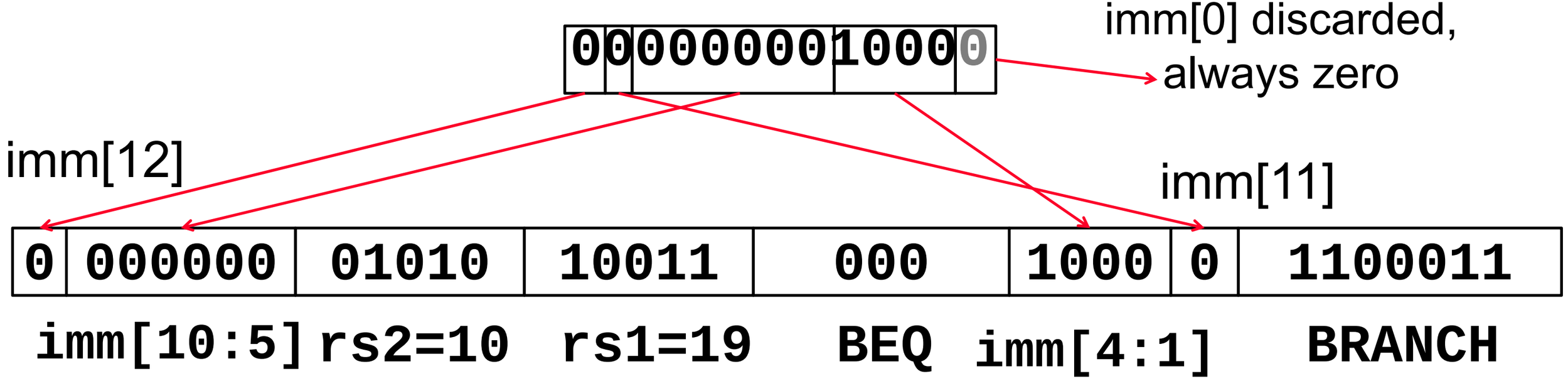
Upper bits sign-extended from inst[31] always

Only bit 7 of instruction changes role in immediate between S and B

Branch Example, complete encoding

```
beq   x19,x10, offset = 16 bytes
```

13-bit immediate, imm[12:0], with value 16



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

Intermission

Project 1-1 status:

a) Done!

b) Almost done.

c) Started. I'm in the mix.

d) Just basically read it. I think I get it.

e) Haven't really started. Due tomorrow, right?

Questions on PC-addressing

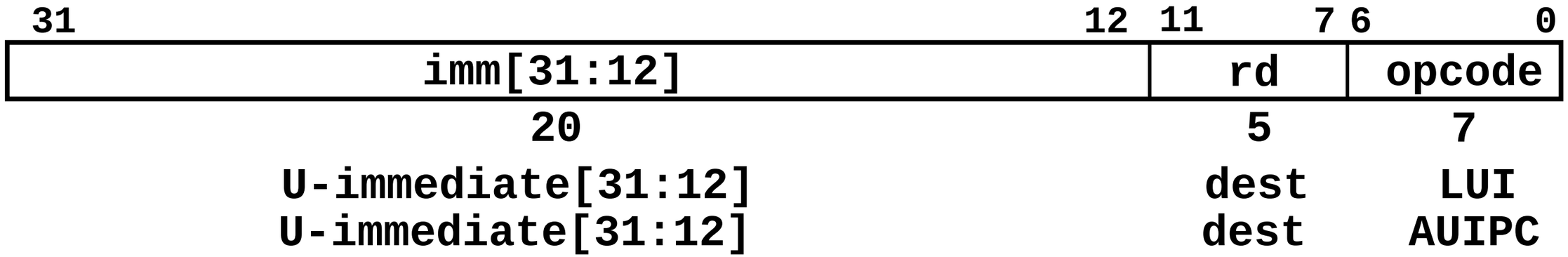
- Does the value in branch immediate field change if we move the code?
  - If moving individual lines of code, then yes
  - If moving all of code, then no ('position-independent code')
- What do we do if destination is > 2^{10} instructions away from branch?
  - Other instructions save us

Questions on PC-addressing

- Does the value in branch immediate field change if we move the code?
  - If moving individual lines of code, then yes
  - If moving all of code, then no (because PC-relative offsets)
- What do we do if destination is > 2^{10} instructions away from branch?
  - Other instructions save us

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

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 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
```

One Corner Case

How to set 0xDEADBEEF?

```
LUI x10, 0xDEADB        # x10 = 0xDEADB000
ADDI x10, x10, 0xEEF    # x10 = 0xDEADAEEF
```

ADDI 12-bit immediate is always sign-extended, if top bit is set, will subtract -1 from upper 20 bits

Solution

How to set 0xDEADBEEF?

```
LUI x10, 0xDEADC        # x10 = 0xDEADC000
ADDI x10, x10, 0xEEF    # x10 = 0xDEADBEEF
```

Pre-increment value placed in upper 20 bits, if sign bit will be set on immediate in lower 12 bits.

Assembler pseudo-op handles all of this:

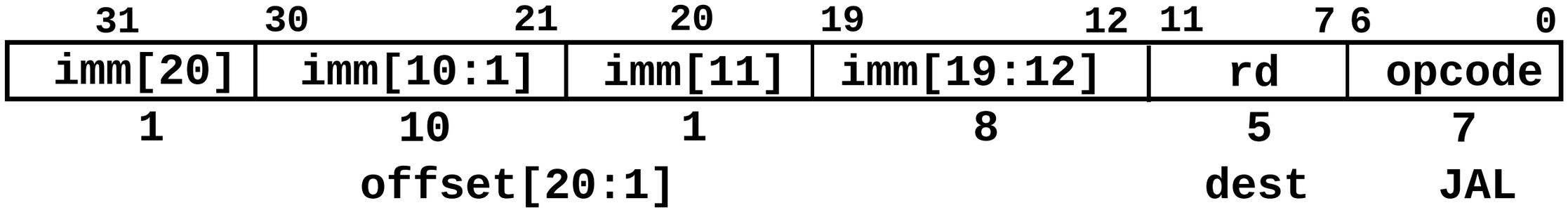
```
li x10, 0xDEADBEEF # Creates two instructions
```

AUIPC

- Adds upper immediate value to PC and places result in destination register
- Used for PC-relative addressing

```
Label: AUIPC x10, 0 # Puts address of label in x10
```

J-Format for Jump Instructions



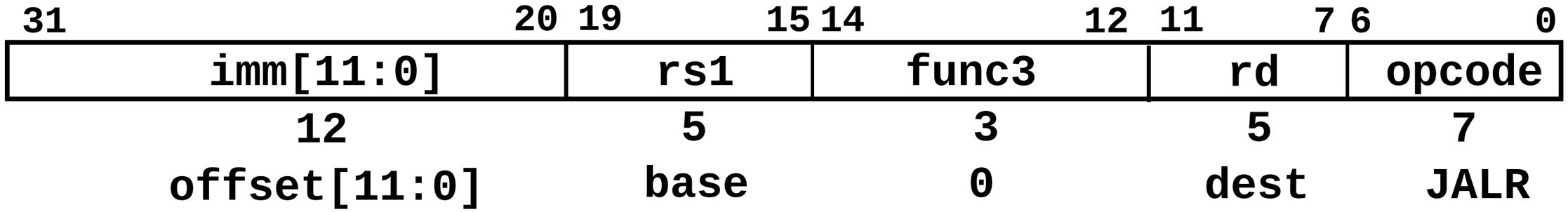
- JAL saves PC+4 in register rd (the return address)
  - Assembler "j" jump is pseudo-instruction, uses JAL but sets rd=x0 to discard return address
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within ±2^{19} locations, 2 bytes apart
  - ±2^{18} 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

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)

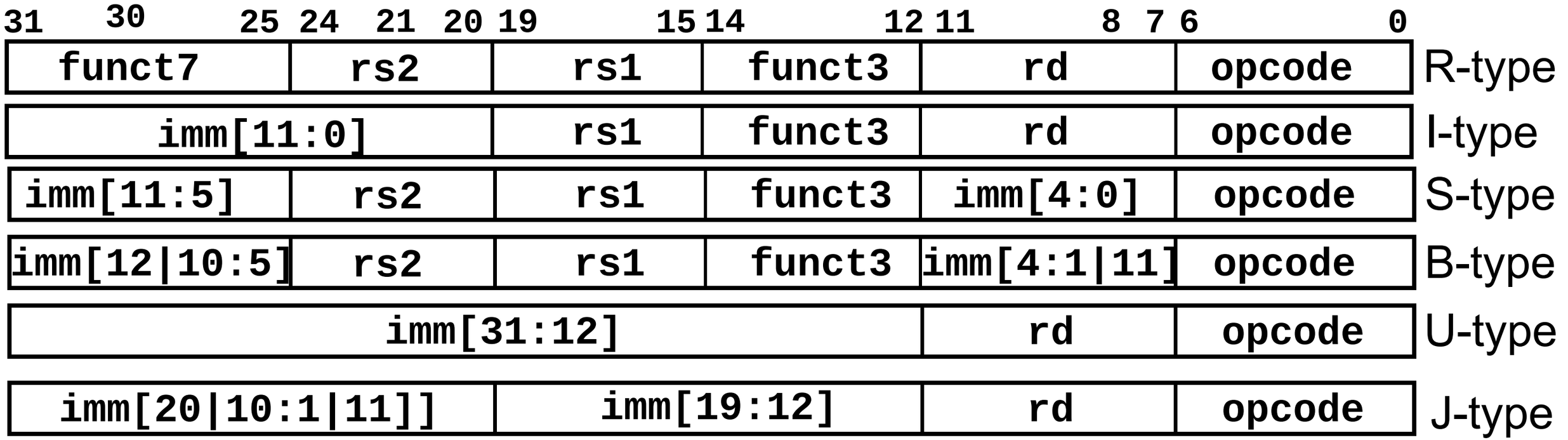


- JALR rd, rs, immediate
  - Writes PC+4 to rd (return address)
  - Sets PC = rs + immediate
  - Uses same immediates as arithmetic and loads
    - *no* multiplication by 2 bytes
    - In contrast to branches and JAL

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, <hi20bits>
jalr ra, x1, <lo12bits>
# Jump PC-relative with 32-bit offset
auipc x1, <hi20bits>
jalr x0, x1, <lo12bits>
```

Summary of RISC-V Instruction Formats



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

“And in Conclusion…”

- **We have covered all RISC-V instructions and registers**
  - **R-type, I-type, S-type, B-type, U-type and J-type instructions**
  - **Practice assembling and disassembling**