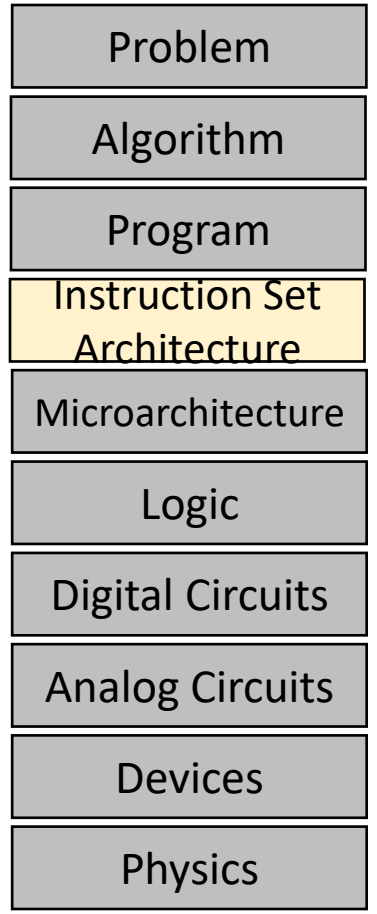
Universität Bremen

# RISC-V Instruction Format

Prof. Dr. Rolf Drechsler
Dr. Muhammad Hassan
M.Sc. Jan Zielasko
M.Sc. Milan Funck



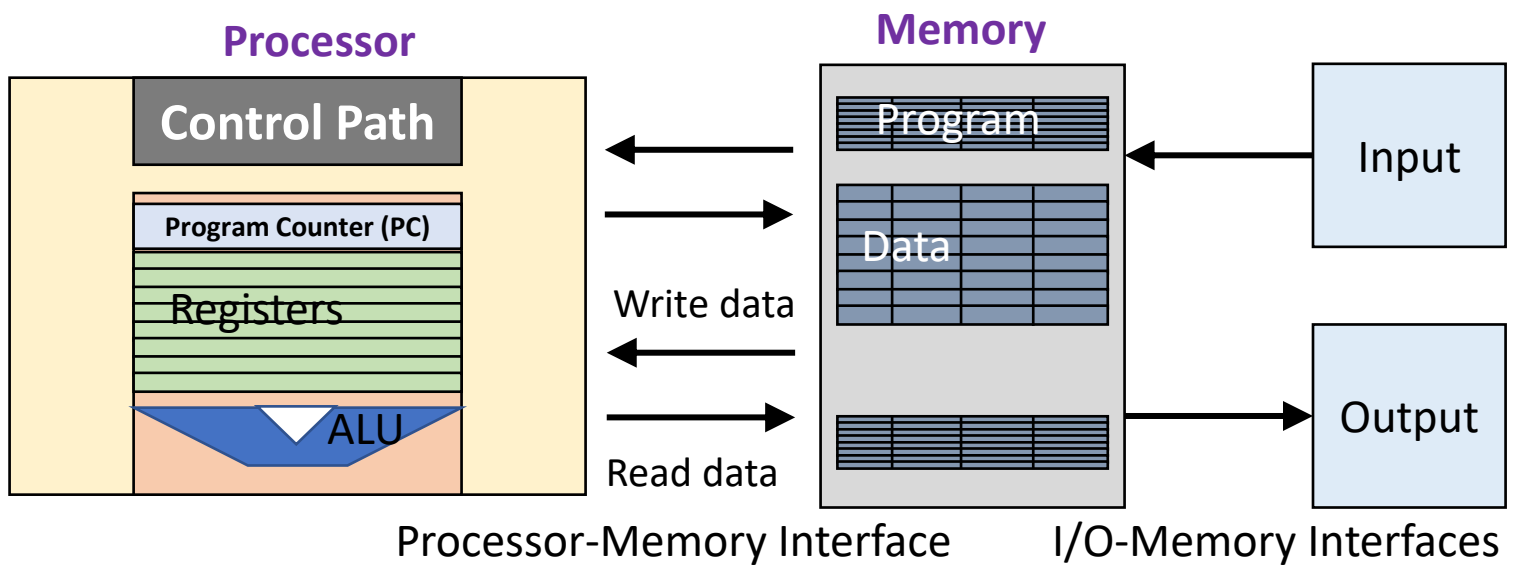
# Studied So Far …

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

```
lw x3, 0(x10)
lw x4, 4(x10)
sw x4, 0(x10)
sw x3, 4(x10)
```

```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

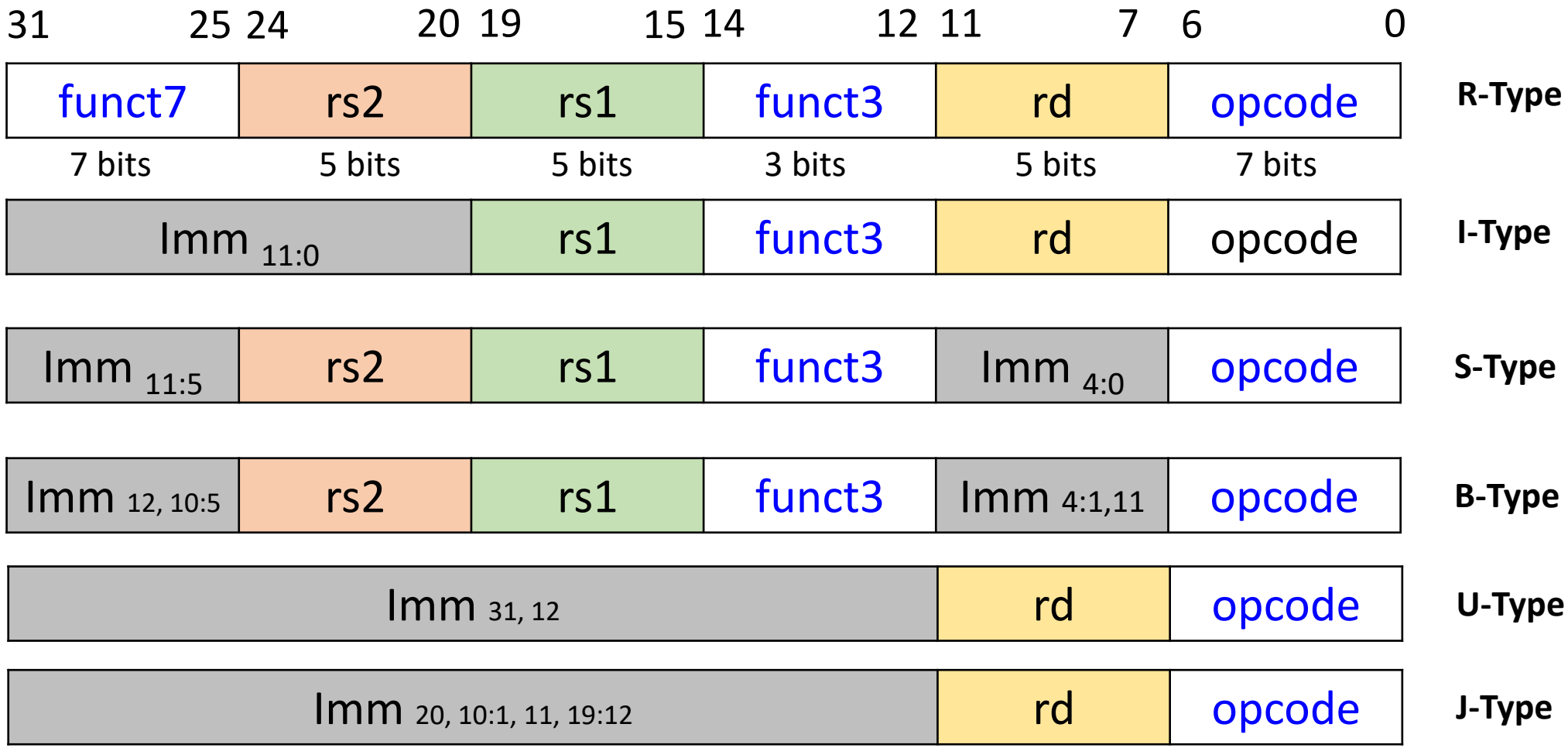
| Name | Register Number | Usage |
|---|---|---|
| **zero** | x0 | Constant value 0 |
| **ra** | x1 | Return address |
| **sp** | x2 | Stack pointer |
| **gp** | x3 | Global pointer |
| **tp** | x4 | Thread pointer |
| **t0-2** | x5-7 | Temporaries |
| **s0/fp** | x8 | Saved register / Frame pointer |
| **s1** | x9 | Saved register |
| **a0-1** | x10-11 | Function arguments / return values |
| **a2-7** | x12-17 | Function arguments |
| **s2-11** | x18-27 | Saved registers |
| **t3-6** | x28-31 | Temporaries |



# Instruction Formats

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

```
lw x3, 0(x10)
lw x4, 4(x10)
sw x4, 0(x10)
sw x3, 4(x10)
```



Universität Bremen

# Today's Agenda

- An Overview of 6 instruction formats
- R-Format
- I-Format
- S-Format
- SB-Format
- U-Format
- UJ-Format

Universität Bremen

# Machine Language

**INSTRUCTIONS ARE DATA**

- Binary representation of instructions
- Computers only understand 1’s and 0’s
- 32-bit instructions
  - Simplicity favors regularity: 32-bit data & instructions
  - Divided into fields
    - Each field tells processor something about the instruction.
- 6 Types of Instruction Formats:
  - R-Type
  - I-Type
  - S/B-Type
  - U/J-Type

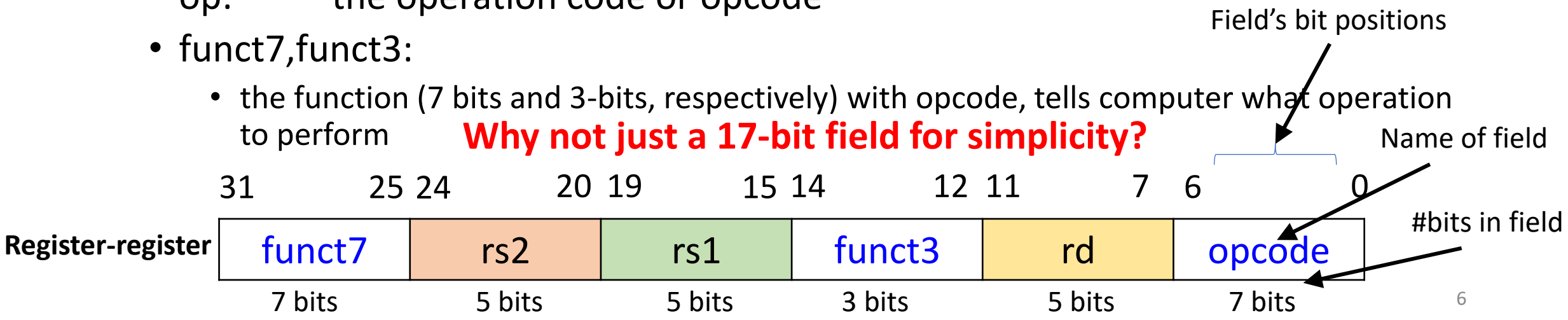
# R-Type

- Register-type **add rd, rs1, rs2**
- 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

Register-register arithmetic instructions
add, xor, sll, etc …

**All R-Format instructions have opcode 0110011 (0x33)**

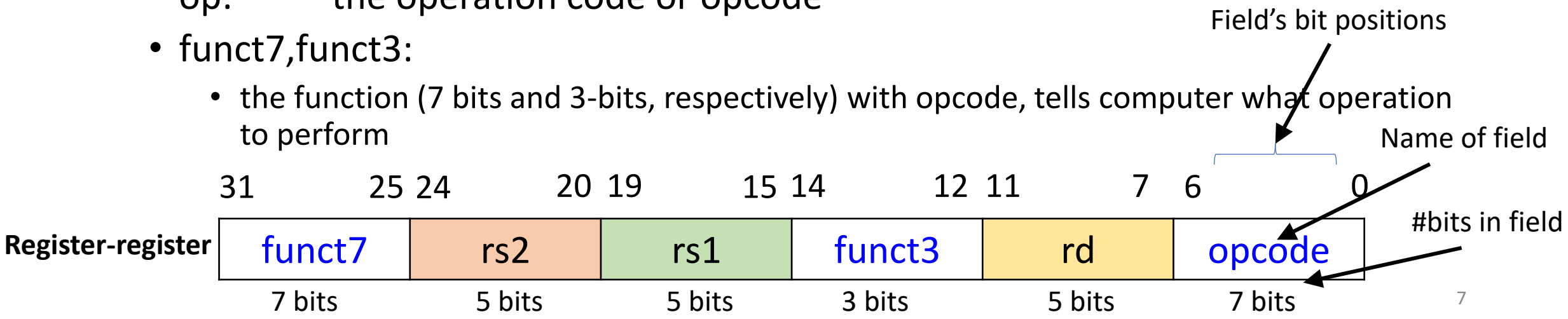
**Why not just a 17-bit field for simplicity?**



# R-Type

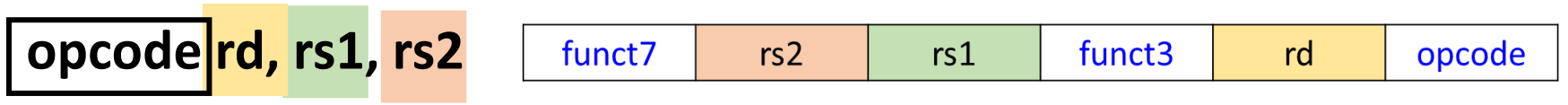
- Register-type **add rd, rs1, rs2**
- 3 register operands:
  - rs1, rs2: source registers
  - rd: destination register
- Other fields:
  - op: the operation code or opcode
  - funct7,funct3:
    - the function (7 bits and 3-bits, respectively) with opcode, tells computer what operation to perform

Register field (rs1, rs2, rd) holds a 5-bit unsigned integer [0-31] corresponding to a register number (x0-x31)



# R-Type Examples

| Name | Register Number | Usage |
|---|---|---|
| zero | x0 | Constant value 0 |
| ra | x1 | Return address |
| sp | x2 | Stack pointer |
| gp | x3 | Global pointer |
| tp | x4 | Thread pointer |
| t0-2 | x5-7 | Temporaries |
| s0/fp | x8 | Saved register / Frame pointer |
| s1 | x9 | Saved register |
| a0-1 | x10-11 | Function arguments / return values |
| a2-7 | x12-17 | Function arguments |
| s2-11 | x18-27 | Saved registers |
| t3-6 | x28-31 | Temporaries |




Any idea how we get this?

**Assembly**

```
add  s2, s3, s4
add  x18,x19,x20
sub  t0, t1, t2
sub  x5, x6, x7
```

**Field Values**

| funct7 | rs2 | rs1 | funct3 | rd | op |
|---|---|---|---|---|---|
| 0 | 20 | 19 | 0 | 18 | 51 |
| 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 | |

**Assembly**

```
sll  s7, t0, s1
sll  x23,x5, x9
xor  s8, s9, s10
xor  x24,x25,x26
srai t1, t2, 29
srai x6, x7, 29
```

**Field Values**

| funct7 | rs2 | rs1 | funct3 | rd | op |
|---|---|---|---|---|---|
| 0 | 9 | 5 | 1 | 23 | 51 |
| 0 | 26 | 25 | 4 | 24 | 51 |
| 32 | 29 | 7 | 5 | 6 | 19 |
| 7 bits | 5 bits | 5 bits | 3 bits | 5 bits | 7 bits |

**Machine Code**

| funct7 | rs2 | rs1 | funct3 | rd | op | |
|---|---|---|---|---|---|---|
| 0000 000 | 01001 | 00101 | 001 | 10111 | 011 0011 | (0x00929BB3) |
| 0000 000 | 11010 | 11001 | 100 | 11000 | 011 0011 | (0x01ACCC33) |
| 0100 000 | 11101 | 00111 | 101 | 00110 | 001 0011 | (0x41D3D313) |
| 7 bits | 5 bits | 5 bits | 3 bits | 5 bits | 7 bits | |

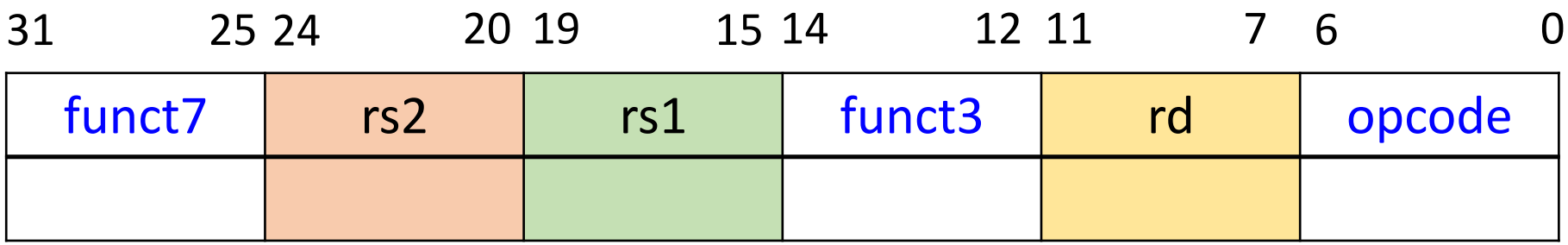
# Take A Guess

- How do we encode

add x4, x3, x2 → **opcode rd, rs1, rs2**

| funct7 | | | funct3 | | opcode | |
|---|---|---|---|---|---|---|
| 0000000 | rs2 | rs1 | 000 | rd | 0110011 | **add** |
| 0100000 | rs2 | rs1 | 000 | rd | 0110011 | **sub** |
| 0000000 | rs2 | rs1 | 100 | rd | 0110011 | **xor** |
| 0000000 | rs2 | rs1 | 111 | rd | 0110011 | **and** |

A. 0x4021 8233
B. 0x0021 82b3
C. 0x4021 82b3
D. 0x0021 8233
E. 0x0021 8234
F. Something else



# Take A Guess

- How do we encode

add x4, x3, x2 → **opcode rd, rs1, rs2**

| funct7 | | | funct3 | | opcode | |
|---|---|---|---|---|---|---|
| 0000000 | rs2 | rs1 | 000 | rd | 0110011 | **add** |
| 0100000 | rs2 | rs1 | 000 | rd | 0110011 | **sub** |
| 0000000 | rs2 | rs1 | 100 | rd | 0110011 | **xor** |
| 0000000 | rs2 | rs1 | 111 | rd | 0110011 | **and** |

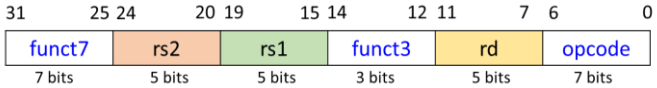
A. 0x4021 8233
B. 0x0021 82b3
C. 0x4021 82b3
D. 0x0021 8233
E. 0x0021 8234
F. Something else

| 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode |
| 0000000 | 00010 | 00011 | 000 | 00100 | 0110011 |
| **0 0** | **2** | **1** | **8** | **2** | **3 3** |

# I-Type

**add rd, rs1, rs2**

**addi rd, rs1, imm**

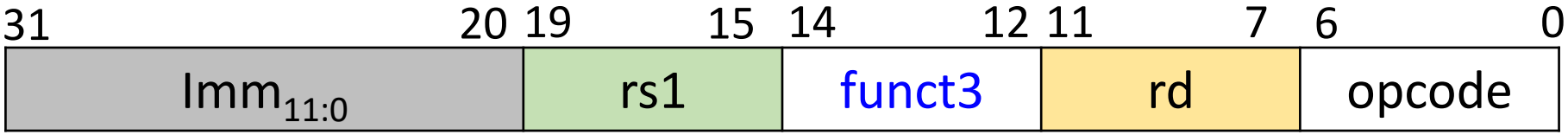



- Immediate-type
- 3 operands:
  - rs1: register source operand
  - rd: register destination operand
  - imm: 12-bit two's complement immediate
- Other fields:
  - op: the opcode
    - Simplicity favors regularity: all instructions have opcode
  - funct3: the function (3-bit function code)
    - with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

- If we used R-Format, we'd only be able to represent 32 values in the 5-bit rs2 field.
  - Immediates may be (and are often) much bigger!
- addi, xori etc.

**All arithmetic immediate instructions have opcode 0010011 (0x23)**

**Load**



# I-Type Example

**addi rd, rs1, imm**


**Assembly**

```
addi s0, s1, 12
addi x8, x9, 12
addi s2, t1, -14
addi x18,x6, -14
lw   t2, -6(s3)
lw   x7, -6(x19)
lh   s1, 27(zero)
lh   x9, 27(x0)
lb   s4, 0x1F(s4)
lb   x20,0x1F(x20)
```

**Field Values**

| $imm_{11:0}$ | rs1 | funct3 | rd | op |
|---|---|---|---|---|
| 12 | 9 | 0 | 8 | 19 |
| -14 | 6 | 0 | 18 | 19 |
| -6 | 19 | 2 | 7 | 3 |
| 27 | 0 | 1 | 9 | 3 |
| 0x1F | 20 | 0 | 20 | 3 |
| 12 bits | 5 bits | 3 bits | 5 bits | 7 bits |

**Machine Code**

| $imm_{11:0}$ | rs1 | funct3 | rd | op | |
|---|---|---|---|---|---|
| 0000 0000 1100 | 01001 | 000 | 01000 | 001 0011 | (0x00C48413) |
| 1111 1111 0010 | 00110 | 000 | 10010 | 001 0011 | (0xFF230913) |
| 1111 1111 1010 | 10011 | 010 | 00111 | 000 0011 | (0xFFA9A383) |
| 0000 0001 1011 | 00000 | 001 | 01001 | 000 0011 | (0x01B01483) |
| 0000 0001 1111 | 10100 | 000 | 10100 | 000 0011 | (0x01FA0A03) |
| 12 bits | 5 bits | 3 bits | 5 bits | 7 bits | |

offset base Load word Load opcode

All load instructions have opcode 0000011.

**Load (lw) instructions are also I-Format**

Load address = (Base Register) + (Immediate Offset)

# I-Type Example

## addi rd, rs1, imm


**Assembly**

```
addi s0, s1, 12
addi x8, x9, 12
addi s2, t1, -14
addi x18,x6, -14
lw   t2, -6(s3)
lw   x7, -6(x19)
lh   s1, 27(zero)
lh   x9, 27(x0)
lb   s4, 0x1F(s4)
lb   x20,0x1F(x20)
```

**Field Values**

| $imm_{11:0}$ | rs1 | funct3 | rd | op |
|---|---|---|---|---|
| 12 | 9 | 0 | 8 | 19 |
| -14 | 6 | 0 | 18 | 19 |
| -6 | 19 | 2 | 7 | 3 |
| 27 | 0 | 1 | 9 | 3 |
| 0x1F | 20 | 0 | 20 | 3 |
| 12 bits | 5 bits | 3 bits | 5 bits | 7 bits |

**Machine Code**

| $imm_{11:0}$ | rs1 | funct3 | rd | op | |
|---|---|---|---|---|---|
| 0000 0000 1100 | 01001 | 000 | 01000 | 001 0011 | (0x00C48413) |
| 1111 1111 0010 | 00110 | 000 | 10010 | 001 0011 | (0xFF230913) |
| 1111 1111 1010 | 10011 | 010 | 00111 | 000 0011 | (0xFFA9A383) |
| 0000 0001 1011 | 00000 | 001 | 01001 | 000 0011 | (0x01B01483) |
| 0000 0001 1111 | 10100 | 000 | 10100 | 000 0011 | (0x01FA0A03) |
| 12 bits | 5 bits | 3 bits | 5 bits | 7 bits | |

offset base Load word Load opcode

**Load (lw) instructions are also I-Format**

xxxx … xxxx xzzz zzzz

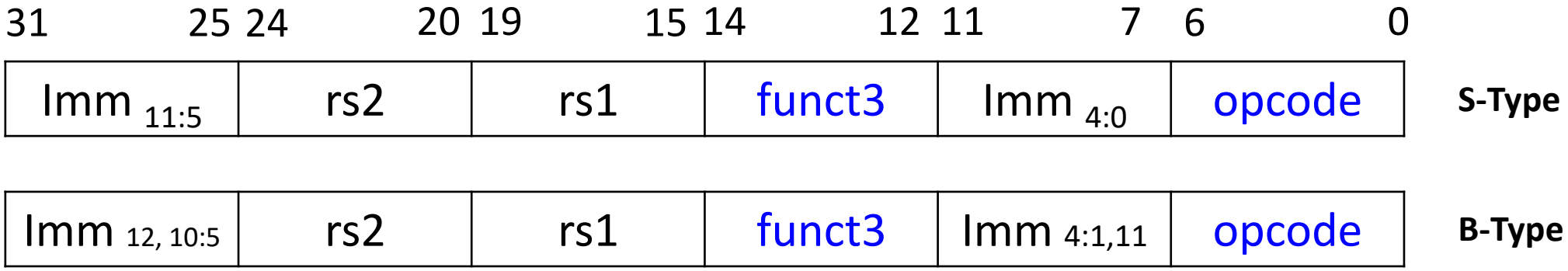
Extend x to upper bits

Load data in lower bits

- lb: load byte, lh: load halfword (16 bits)
  - Sign extend to fill upper bits of destination 32-bit register.
- lbu: load unsigned byte, lhu: load unsigned halfword
  - Zero extend to fill upper bits of destination 32-bit register.
- Note: No lwu instruction in RISC-V
  - Simplicity: No need to sign/zero extend when copying 32 bits into 32-bit register.

# S/B-Type

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



# S-Type

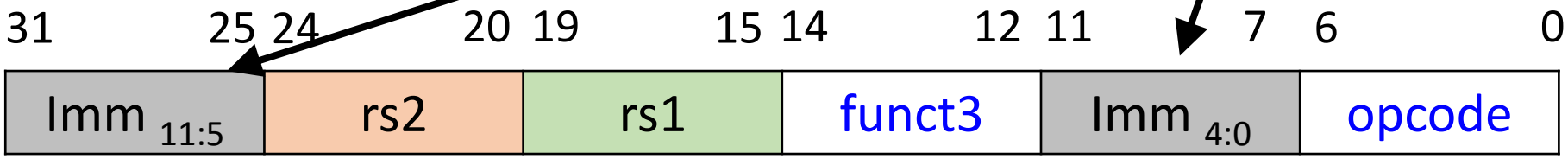
- Store-Type **storeop rs2, imm(rs1)**
- 3 operands:
  - rs1: base register
  - rs2: value to be stored to memory
  - imm: 12-bit two's complement immediate
- Other fields:
  - op: the opcode
    - Simplicity favors regularity: all instructions have opcode
  - funct3: the function (3-bit function code)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

Immediate "offset" added to base address to form memory address.

Store address = (Base Register) + (Immediate Offset)

The immediate's higher 7 bits and lower 5 bits are in separate fields!

All store instructions have opcode 0100011.



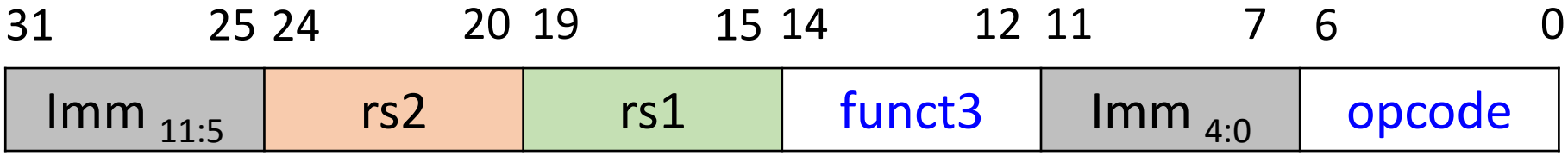

**S-Type**

# S-Type cont …

- Store-Type **storeop rs2, imm(rs1)**
- 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

**Why split up the immediate field?**

RISC-V design prioritizes keeping register fields in the same places. Immediates are less critical to hardware.



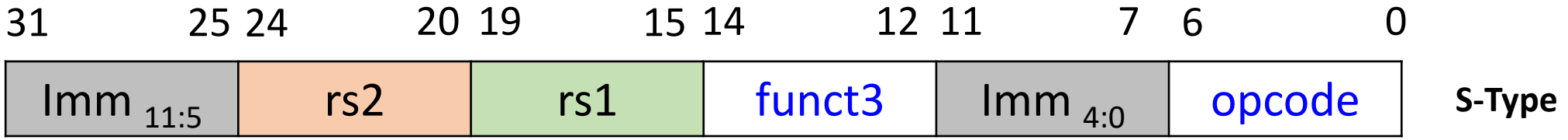

**S-Type**

# S-Type cont …

**storeop rs2, imm(rs1)**

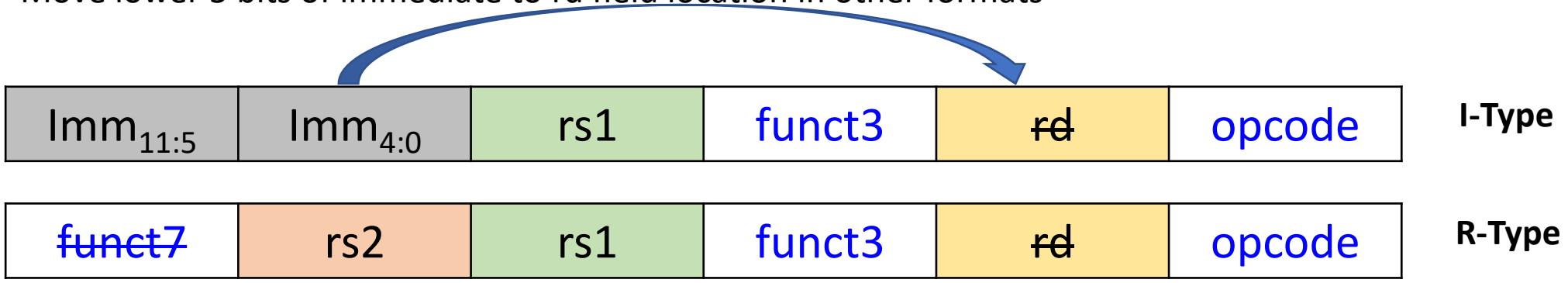
**Why split up the immediate field?**

RISC-V design prioritizes keeping register fields in the same places. Immediates are less critical to hardware.




Store needs immediate, 2 read registers, and no destination register

Move lower 5 bits of immediate to rd field location in other formats



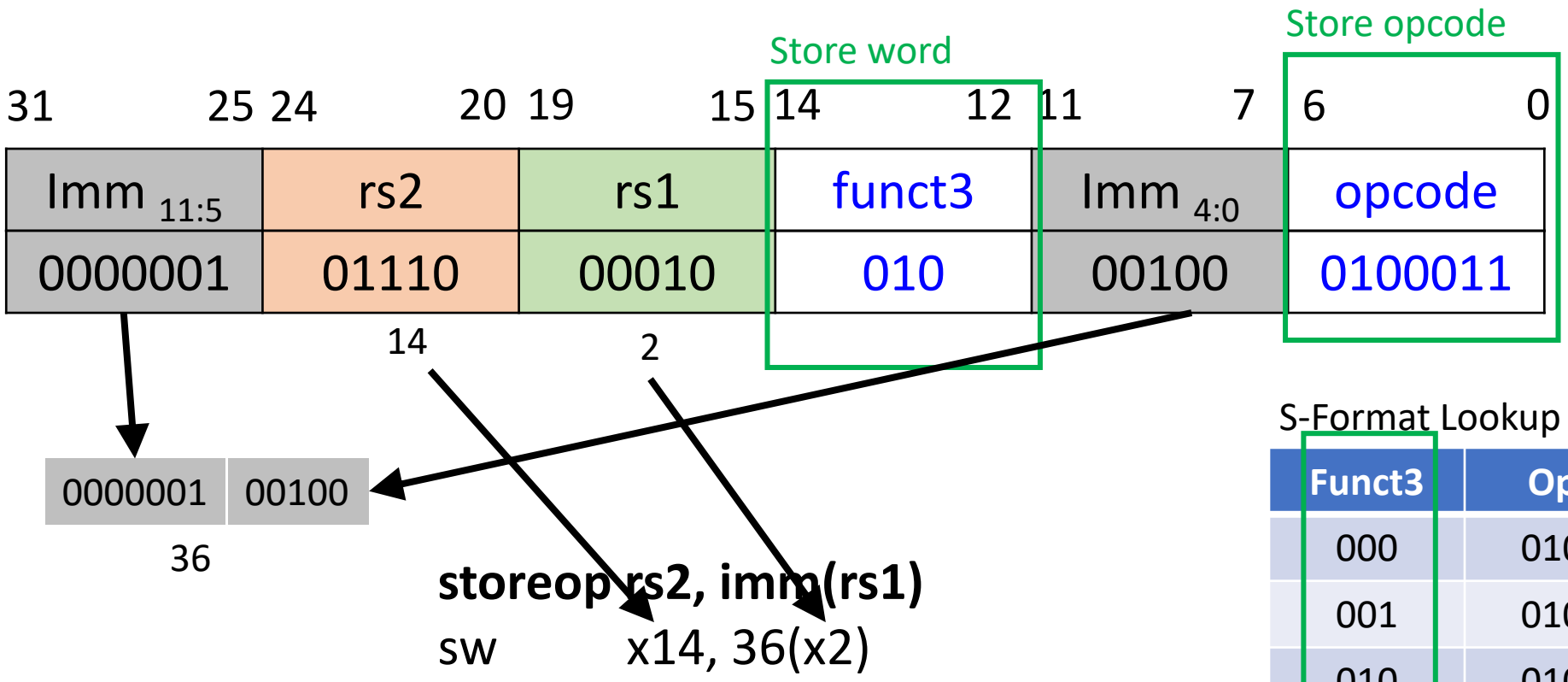
# S-Type Example

**storeop rs2, imm(rs1)**

| 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)`<br>`sw x7, -6(x19)` | 1111 111 | 7 | 19 | 2 | 11010 | 35 | 1111 111 | 00111 | 10011 | 010 | 11010 | 010 0011 | **(0xFE79AD23)** |
| `sh s4, 23(t0)`<br>`sh x20,23(x5)` | 0000 000 | 20 | 5 | 1 | 10111 | 35 | 0000 000 | 10100 | 00101 | 001 | 10111 | 010 0011 | **(0x01429BA3)** |
| `sb t5, 0x2D(zero)`<br>`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 | |

Store opcode

# Disassembling the S-Format




S-Format Lookup

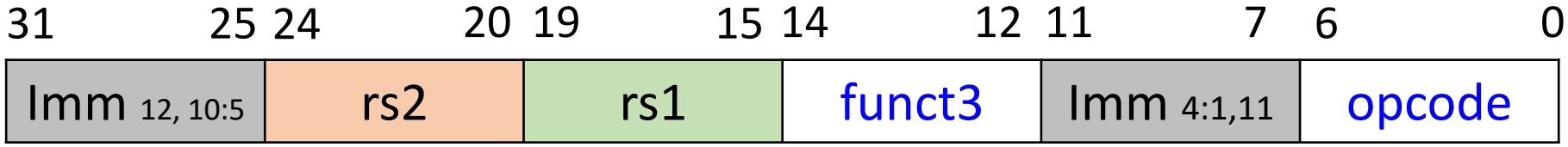
| Funct3 | Opcode | name |
|---|---|---|
| 000 | 0100011 | Sb |
| 001 | 0100011 | Sh |
| 010 | 0100011 | sw |

# B-Type

- Branch-Type (similar format to S-Type) **opname rs1,rs2,Label**
- 3 operands:
  - rs1: register source 1
  - rs2: register source 2
  - imm12:1: 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

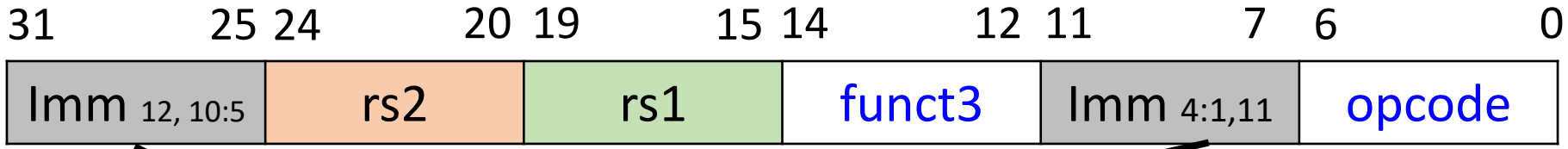
RISC-V uses PC-relative addressing for labels
- If we don't branch: PC = PC + 4 bytes
- If we do branch: PC = PC + (relative byte offset to Label)

Signed two's complement means we can represent ± $2^{11}$ "units" from the Program Counter.



# B-Type

- Branch-Type (similar format to S-Type) **opname rs1,rs2,Label**





- Instruction alignment – multiple of 4
- Branch target alignment – LSB of target address always 0
- Offset encoding efficiency – double branch targets

Lowest bit of offset is always zero, so no need to store in instruction.

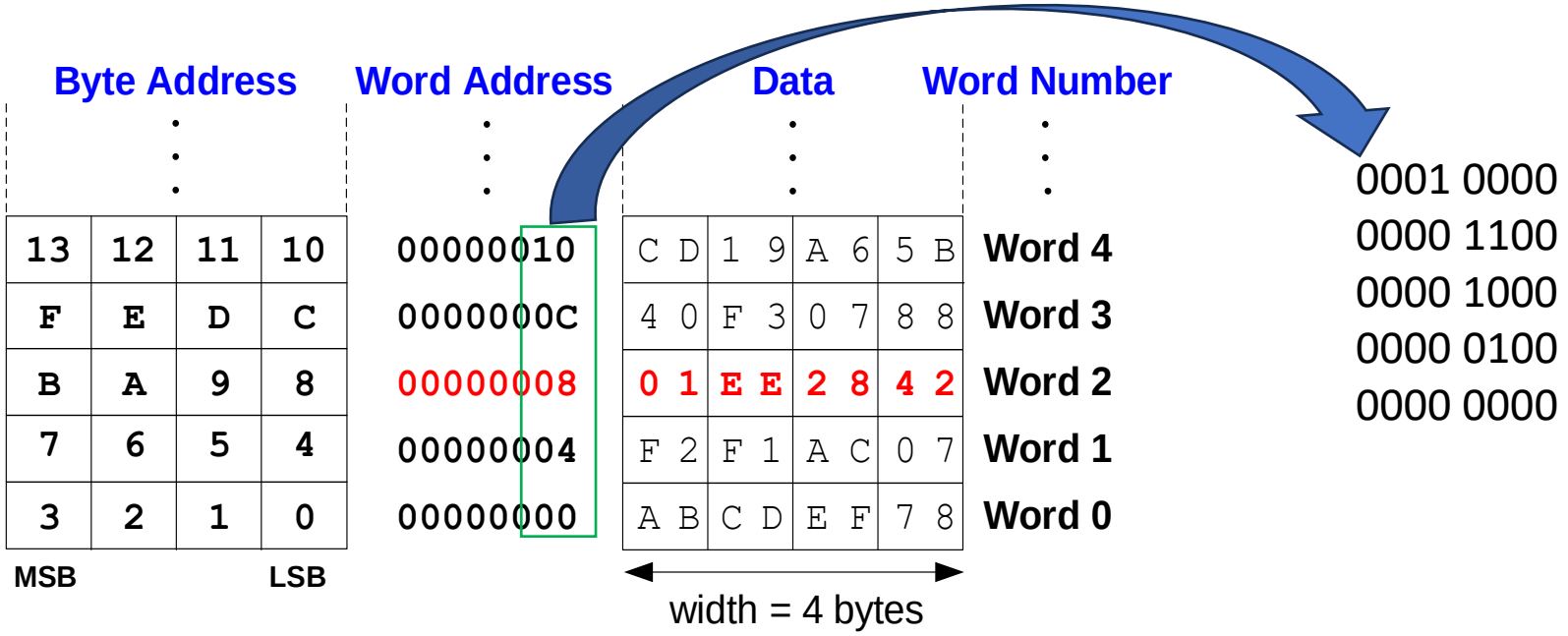
| 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Z | Y | X | X | X | X | X | X | W | W | W | W | 0 |

# B-Type cont...

- Instruction alignment – multiple of 4
- Branch target alignment – LSB of target address always 0
- Offset encoding efficiency – double branch targets

Lowest bit of offset is always zero, so no need to store in instruction.

| 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Z | Y | X | X | X | X | X | X | W | W | W | W | 0 |



# B-Type Example

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

- Fill in fields other than imm
- Construct branch offset (in bytes), then encode immediate

**opname rs1, rs2, Label**

```
# 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
```

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

Each instruction is 32-bits long

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $imm_{12:0}$ = 16 | 0 | | 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 |

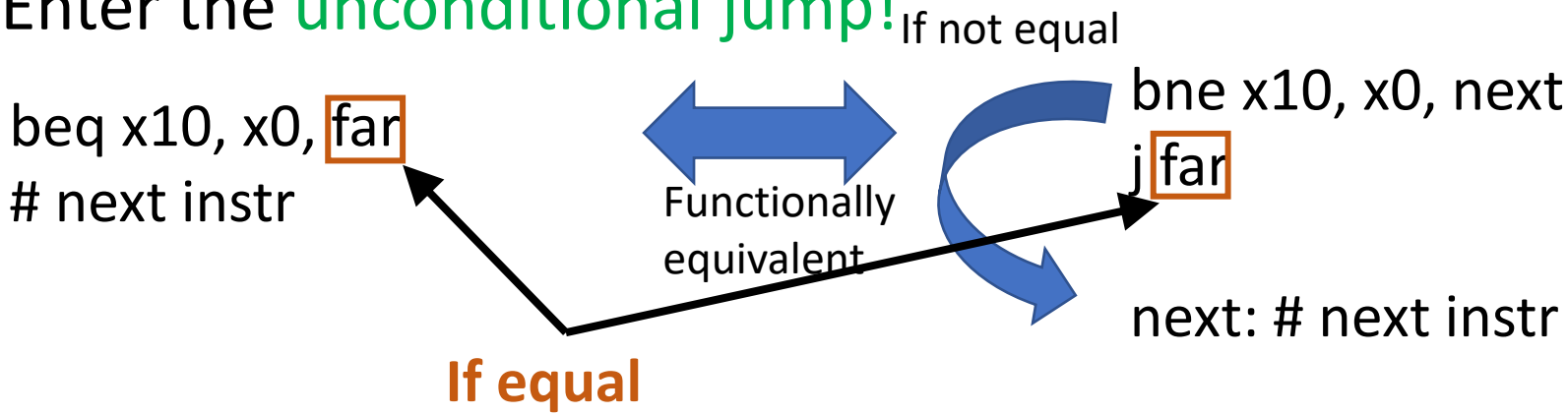
**Machine Code**

| $imm_{12,10:5}$ | 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)**

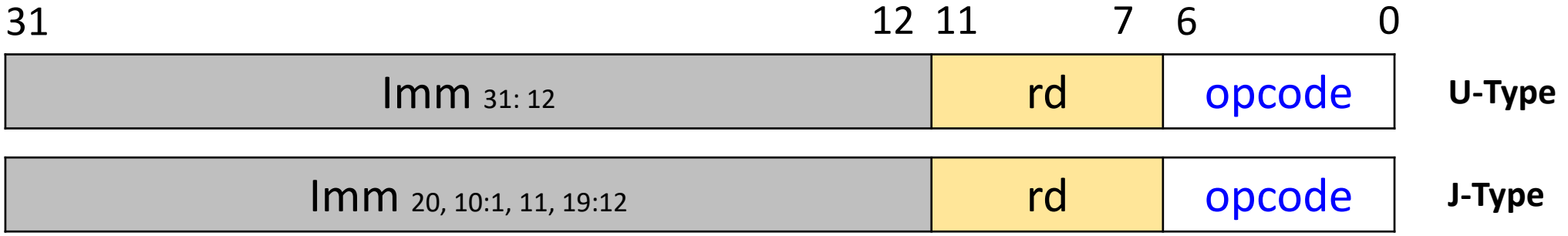
# Conditional Branching Even Further

- Conditional branches are typically used for if-else statements and for/while loops
  - Usually pretty small (<50 lines of C code)
- B-Format has limited range
  - ± $2^{10}$ 32-bit instructions from current instruction
- What if destination is further away?
- Enter the unconditional jump!



# U/J-Type

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

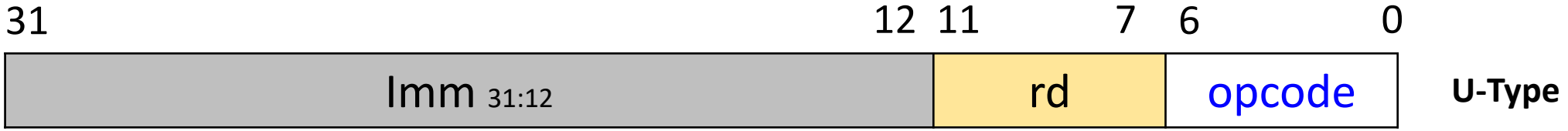


# U-Type

- Upper-immediate-Type **opname rd, imm**
- Used for load upper immediate (lui)

**U-Format opcode:**
**lui 0110111**

- 2 operands:
  - rd: destination register (saves result of operation)
  - $imm_{31:12}$: upper 20 bits of a 32-bit immediate
- opcode: the operation code or opcode – tells computer what operation to perform




| Assembly | Field Values | | | Machine Code | | | |
|---|---|---|---|---|---|---|---|
| | $imm_{31:12}$ | rd | op | $imm_{31:12}$ | rd | op | |
| `lui s5, 0x8CDEF` | 0x8CDEF | 21 | 55 | 1000 1100 1101 1110 1111 | 10101 | 011 0111 | **(0x8CDEFAB7)** |
| `lui x21,0x8CDEF` | 20 bits | 5 bits | 7 bits | 20 bits | 5 bits | 7 bits | |

# J-Type

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

**Write address of the following instruction to rd.**

# J-Type

- Jump-Type **jal rd, Label**
- Used for jump-and-link instruction (jal)
- Two use cases
  - Call a function and simultaneously save return address in named register

    **jal rd, FooLabel**
  - Unconditional branch
    - j pseudoinstruction: jump and discard return address

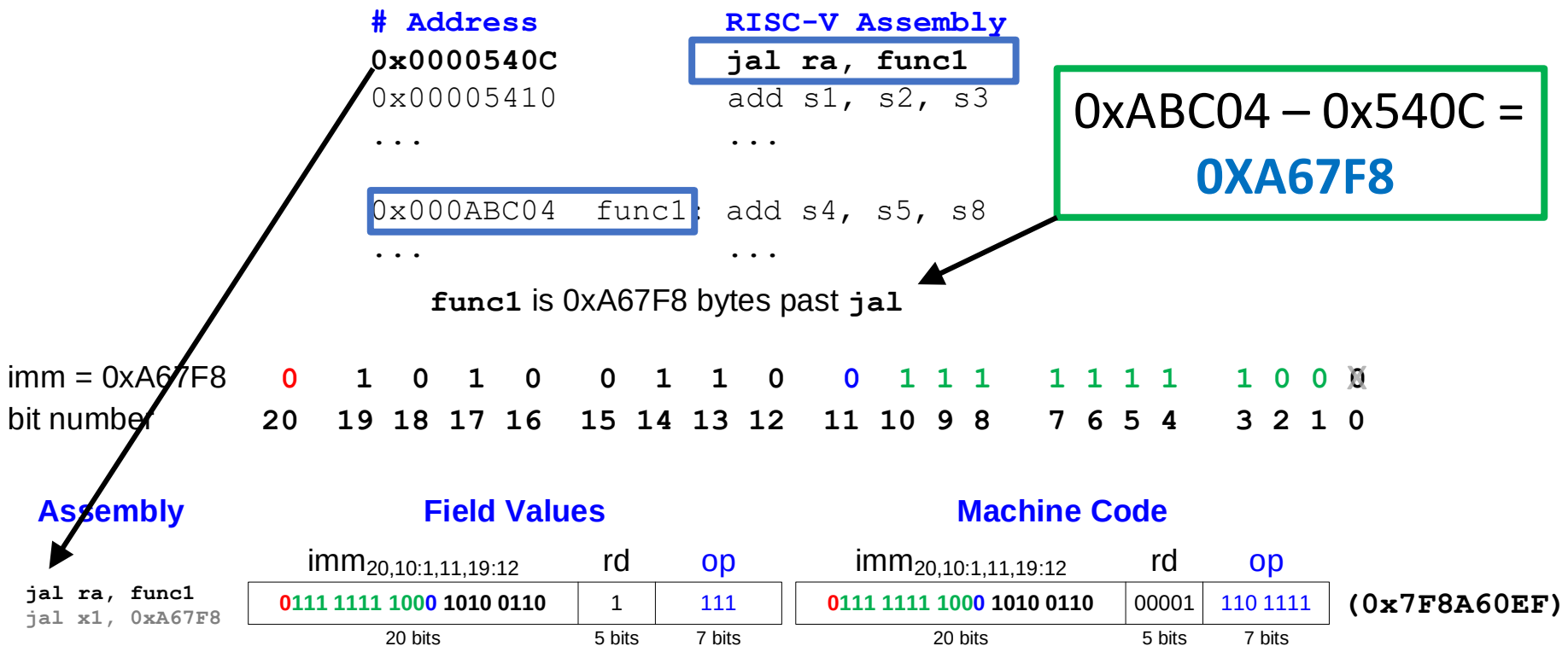
      **j FooLabel** ⟺ **jal x0, FooLabel**

Jump to FooLabel (i.e., add relative offset to PC).
Write address of the following instruction to rd.

PC = PC + offset
rd = PC + 4

Often used to conditionally branch further than B-Types

# J-Type Example



# J-Type Example

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

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

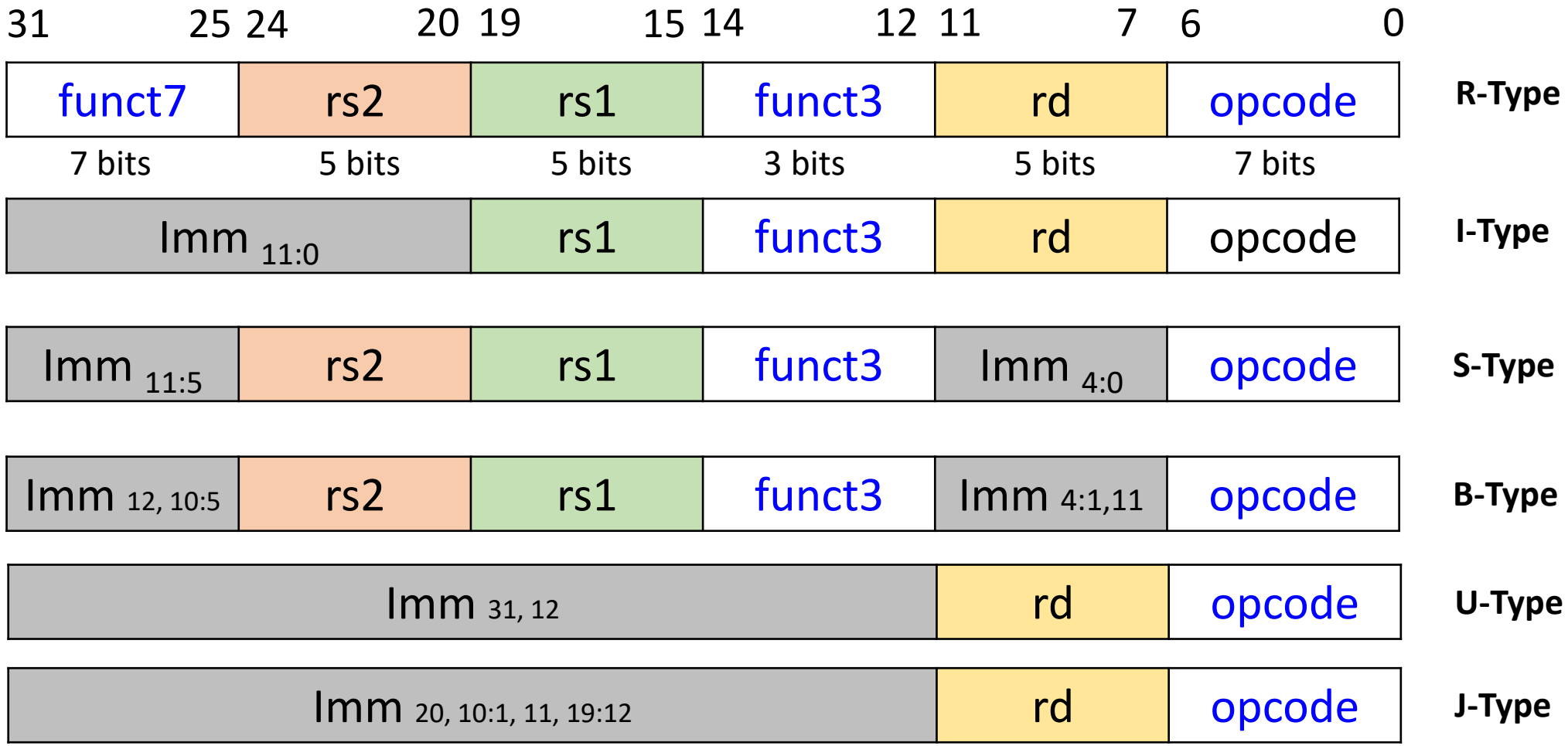
0xABC04 – 0x540C = **0XA67F8**

**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 | 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`<br>`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 | |

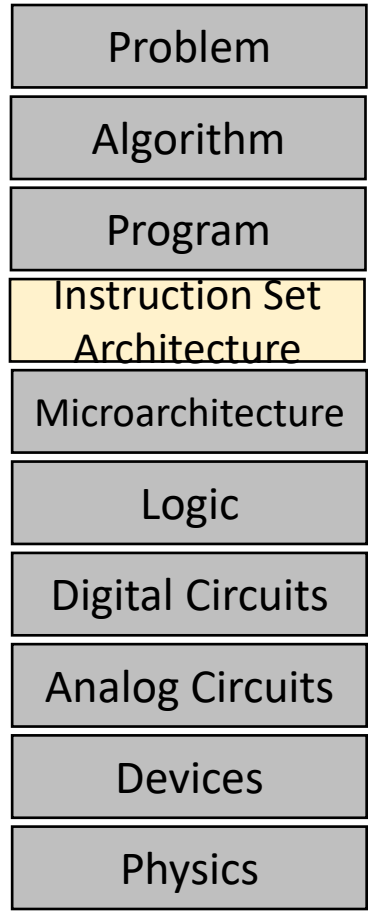
# Summarizing the Formats



Universität Bremen

# RISC-V Instruction Format

Prof. Dr. Rolf Drechsler
Dr. Muhammad Hassan
M.Sc. Jan Zielasko
M.Sc. Milan Funck



Universität Bremen

# RC32 B-Format Instructions

| Funct3 | Opcode | name |
|---|---|---|
| 000 | 1100011 | beq |
| 001 | 1100011 | bne |
| 100 | 1100011 | blt |
| 101 | 1100011 | bge |
| 110 | 1100011 | bltu |
| 111 | 1100011 | bgeu |