Topic V05

Binary Representation
of Instructions

Representing Instructions

Instructions are encoded in a binary representation

Called machine code

RISC-V instructions

Encoded as 32-bit instruction words

Small number of formats encoding operation code (opcode), register number, …

Regularity!

Register Names:

`t0 – t2` $\Leftrightarrow$ `x5 – x7`

`t3 – t6` $\Leftrightarrow$ `x28 – x31`

`s0 – s1` $\Leftrightarrow$ `x8 – x9`

`s2 – s11` $\Leftrightarrow$ `x18 – x27`

Representing Instructions: R-Type

x5 x23 x21

```
add t0, s7, s5
```

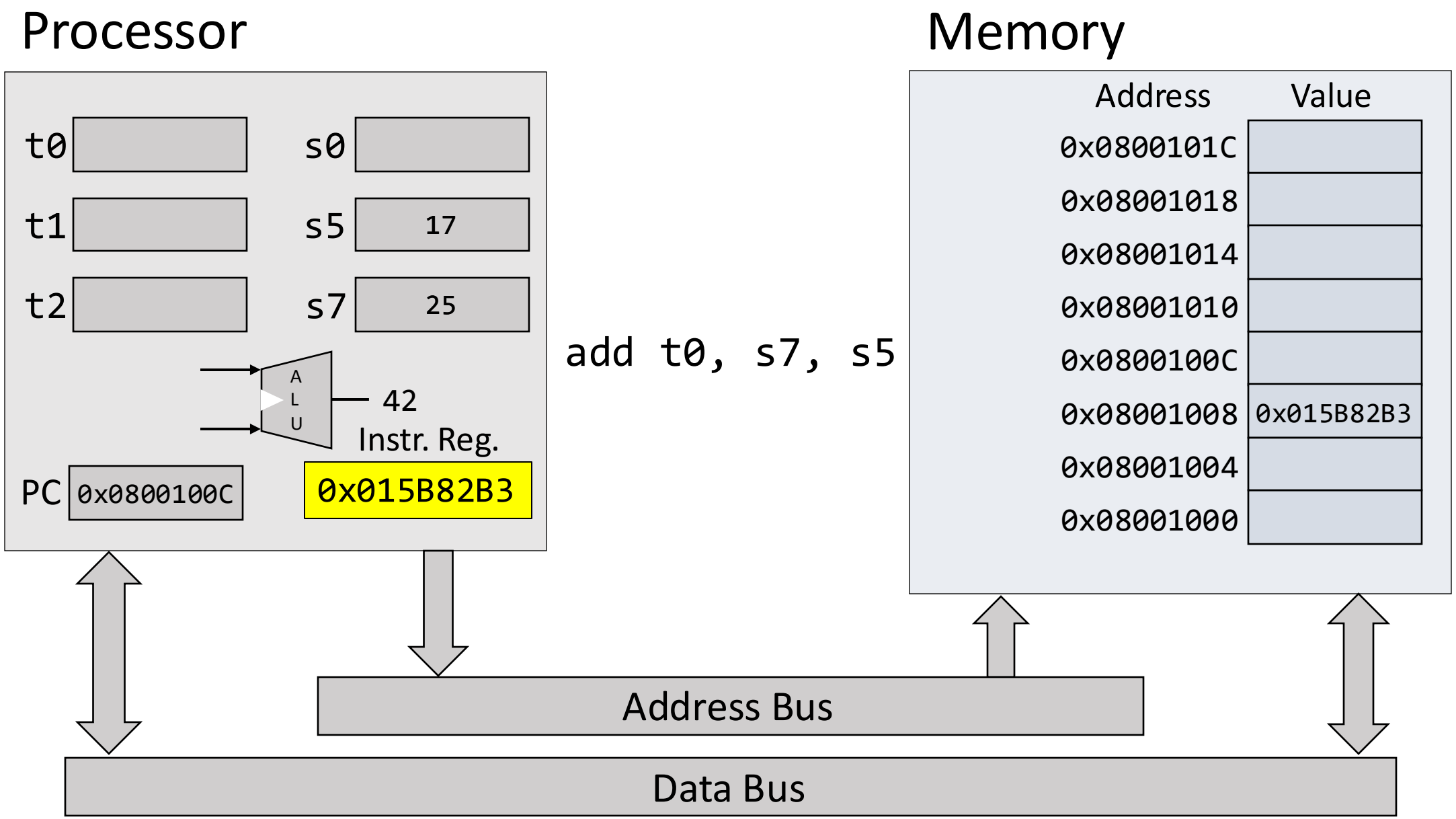
func7	rs2	rs1	func3	rd	op
0x00	21	23	0x0	5	0x33

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
0000000	10101	10111	000	00101	0110011

Hexadecimal Representation : **0x015B82B3**

op: **Opcode** to specify the operation and format of an instruction
funct7: **function code** specifies the variant of the instruction to be executed
funct3: **function code**; partially, specifies the variant of the instruction to be executed
rd: **register destination** operand
rs1: first **register source** operand
rs2: second **register source** operand

Organization of a Computer



Representing Instructions: I-Type

x5 x19

```
lw t0, 23(s3)    # t0 ← Memory[s3 + 23]
```

immediate	rs1	func3	rd	op
23	19	0x2	5	0x03

31 20	19 15	14 12	11 7	6 0
000000010111	10011	010	00101	0000011

Hexadecimal Representation: 0x0179A283

Representing Instructions: S-Type

x5 x6

```
sw t0, 1200(t1) # Memory[t1 + 1200] ← t0
```

imm[11:5]	rs2	rs1	func3	imm[4:0]	op
37	5	6	0x2	16	0x23

31 25	24 20	19 15	14 12	11 7	6 0
0100101	00101	00110	010	10000	0100011

Hexadecimal Representation: 0x4A532823

1200 →

Representing Instructions: U-Type

x5

```
lui t0, 0x0F008        # R[rd] = {imm, 12'b0}
```

immediate[31:12]	rd	op
0x0F008	5	0x37

31 … 12	11 … 7	6 … 0
0000 1111 0000 0000 1000	00101	0110111

Hexadecimal Representation: 0x0F0082B7

This far we learned

Divided into bit fields

x5 x23 x21

```
add t0, s7, s5
```

func7	rs2	rs1	func3	rd	op
0x00	21	23	0x0	5	0x33

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
0000000	10101	10111	000	00101	0110011

In memory we would see: **0x015B82B3**

32-bit words

5 bits to specify each register

In memory as a 32-bit hexadecimal

Instructions for Making Decisions

Instructions for Making Decisions

C code

```
if (i == j)
     f = g + h;
else
     f = g – h;
```

Assumption

```
f ↔ s0
g ↔ s1
h ↔ s2
i ↔ s3
j ↔ s4
```

RISC-V assembly

```
            bne  s3, s4, Subtract   # if i ≠ j goto Subtract
            add  s0, s1, s2         # f ← g + h
            jal  zero, Exit         # goto Exit
Subtract:   sub  s0, s1, s2         # f ← g – h
Exit:            …
```

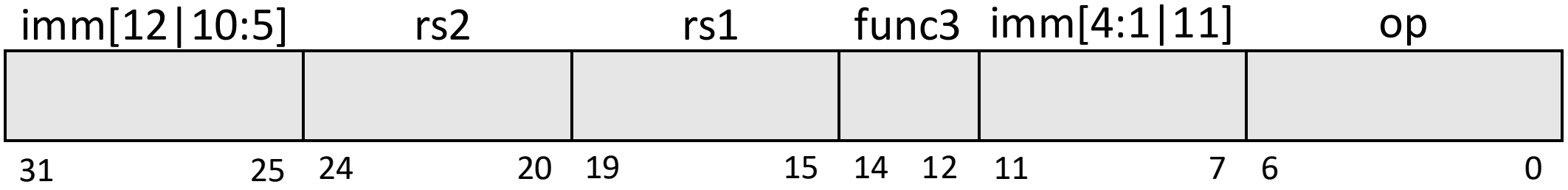
Branch Instructions: SB-Type

Memory Address

```
0x1000 0000          bne    s3, s4, Subtr      # if i ≠ j goto Subtr
0x1000 0004          add    s0, s1, s2         # f ← g + h
0x1000 0008          jal    zero, Exit         # goto Exit
0x1000 000C Subtr:   sub    s0, s1, s2         # f ← g – h
0x1000 0010 Exit:           …
```

We want to jump to a label 12 bytes away

```
bne s3, s4, 12    ↔    if(R[rs1] ≠ R[rs2]) PC ← PC + {imm, 1b'0}
                       else PC ← PC + 4
```



Branch Instructions: SB-Type

Memory Address

```
0x1000 0000          bne    s3, s4, Subtr      # if i ≠ j goto Subtr
0x1000 0004          add    s0, s1, s2         # f ← g + h
0x1000 0008          jal    zero, Exit         # goto Exit
0x1000 000C Subtr:   sub    s0, s1, s2         # f ← g – h
0x1000 0010 Exit:           …
```

We want to jump to a label **12** bytes away

```
bne s3, s4, 12    ↔    if(R[rs1] ≠ R[rs2]) PC ← PC + {imm, 1b'0}
                       else PC ← PC + 4
```

imm[12|10:5]	rs2	rs1	func3	imm[4:1|11]	op

31 25 24 20 19 15 14 12 11 7 6 0

→ **0 0000 0000 1100**

What is 12 expressed in 13 bit signed binary?

12 11 10:5 4:1 0

0 | 0 | 000000 | 0110 | 0

Bit 0 is always zero and does not need representation

```
 0000 0000 0000 0000 0000 0000 0000 1100
+0001 0000 0000 0000 0000 0000 0000 0000
-----------------------------------------
 0001 0000 0000 0000 0000 0000 0000 1100
```

Immediate is sign-extended to 32 bits

→ New PC

Branch Instructions: SB-Type

```
Memory Address                    x19  x20
                                   |    |
0x1000 0000          bne    s3, s4, Subtr        # if i ≠ j goto Subtr
0x1000 0004          add    s0, s1, s2           # f ← g + h
0x1000 0008          jal    zero, Exit           # goto Exit
0x1000 000C Subtr:   sub    s0, s1, s2           # f ← g – h
0x1000 0010 Exit:           …
```

```
bne s3, s4, 12    ↔    if(R[rs1] ≠ R[rs2]) PC ← PC + {imm, 1b'0}
                       else PC ← PC + 4
```

imm[12|10:5]	rs2	rs1	func3	imm[4:1|11]	op
0|0x00	20	19	0x1	0x6|0	0x63
31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
0000000	10100	10011	001	01100	1100011

Hexadecimal Representation: 0x01499663

Branch recap

• 0x1000 0000	bne	s3, s4, Subtr	# if i ≠ j goto Subtr
• 0x1000 0004	add	s0, s1, s2	# f ← g + h
• 0x1000 0008	jal	zero, Exit	# goto Exit
• 0x1000 000C Subtr:	sub	s0, s1, s2	# f ← g – h
• 0x1000 0010 Exit:		…	

Labels appear in the assembly program,
but not in the binary representation

Branch recap

Bit 0 of immediate is implicitly zero.

Immediate is sign-extended to 32-bits.

Immediate is negative for backward branches.

- 0x1000 0000 bne s3, s4, Subtr # if i ≠ j goto Subtr
- 0x1000 0004 add s0, s1, s2 # f ← g + h
- 0x1000 0008 jal zero, Exit # goto Exit
- 0x1000 000C Subtr: sub s0, s1, s2 # f ← g – h
- 0x1000 0010 Exit: …

imm[12|10:5]	rs2	rs1	func3	imm[4:1|11]	op
0|0x00	20	19	0x1	0x6|0	0x63
31 25	24 20	19 15	14 12	11 7	6 0
0000000	10100	10011	001	01100	1100011

Immediate field has the distance between the branch and its target

Example

A 32-bit RISC-V processor fetched an instruction whose hexadecimal representation is:

0xFE0008E3

From the address **0x00400010**

What is the address of the next instruction that will be executed?

Instruction binary in hexadecimal:

0xFE0008E3

1111 1110 0000 0000 0000 1000 1110 0011

OpCode

PSEUDO INSTRUCTIONS ③

MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	if(R[rs1]==0) PC=PC+{imm,1b'0}	beq
bnez	Branch ≠ zero	if(R[rs1]!=0) PC=PC+{imm,1b'0}	bne
fabs.s,fabs.d	Absolute Value	F[rd] = (F[rs1]< 0) ? –F[rs1] : F[rs1]	fsgnx
fmv.s,fmv.d	FP Move	F[rd] = F[rs1]	fsgnj
fneg.s,fneg.d	FP negate	F[rd] = –F[rs1]	fsgnjn
j	Jump	PC = {imm,1b'0}	jal
jr	Jump register	PC = R[rs1]	jalr
la	Load address	R[rd] = address	auipc
li	Load imm	R[rd] = imm	addi
mv	Move	R[rd] = R[rs1]	addi
neg	Negate	R[rd] = –R[rs1]	sub
nop	No operation	R[0] = R[0]	addi
not	Not	R[rd] = ~R[rs1]	xori
ret	Return	PC = R[1]	jalr
seqz	Set = zero	R[rd] = (R[rs1]== 0) ? 1 : 0	sltiu
snez	Set ≠ zero	R[rd] = (R[rs1]!= 0) ? 1 : 0	sltu

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
lb	I	0000011	000		03/0
lh	I	0000011	001		03/1
lw	I	0000011	010		03/2
lbu	I	0000011	100		03/4
lhu	I	0000011	101		03/5
fence	I	0001111	000		0F/0
fence.i	I	0001111	001		0F/1
addi	I	0010011	000		13/0
slli	I	0010011	001	0000000	13/1/00
slti	I	0010011	010		13/2
sltiu	I	0010011	011		13/3
xori	I	0010011	100		13/4
srli	I	0010011	101	0000000	13/5/00
srai	I	0010011	101	0100000	13/5/20
ori	I	0010011	110		13/6
andi	I	0010011	111		13/7
auipc	U	0010111			17
sb	S	0100011	000		23/0
sh	S	0100011	001		23/1
sw	S	0100011	010		23/2
add	R	0110011	000	0000000	33/0/00
sub	R	0110011	000	0100000	33/0/20
sll	R	0110011	001	0000000	33/1/00
slt	R	0110011	010	0000000	33/2/00
sltu	R	0110011	011	0000000	33/3/00
xor	R	0110011	100	0000000	33/4/00
srl	R	0110011	101	0000000	33/5/00
sra	R	0110011	101	0100000	33/5/20
or	R	0110011	110	0000000	33/6/00
and	R	0110011	111	0000000	33/7/00
lui	U	0110111			37
beq	SB	1100011	000		63/0
bne	SB	1100011	001		63/1
blt	SB	1100011	100		63/4
bge	SB	1100011	101		63/5
bltu	SB	1100011	110		63/6
bgeu	SB	1100011	111		63/7
jalr	I	1100111	000		67/0
jal	UJ	1101111			6F
ecall	I	1110011	000	000000000000	73/0/000
ebreak	I	1110011	000	000000000001	73/0/001
CSRRW	I	1110011	001		73/1
CSRRS	I	1110011	010		73/2
CSRRC	I	1110011	011		73/3
CSRRWI	I	1110011	101		73/5
CSRRSI	I	1110011	110		73/6
CSRRCI	I	1110011	111		73/7

REGISTER NAME, USE, CALLING CONVENTION ④

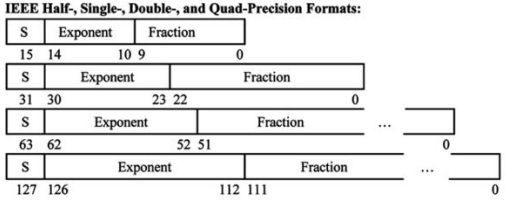
REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP Temporaries	Caller
f8-f9	fs0-fs1	FP Saved registers	Callee
f10-f11	fa0-fa1	FP Function arguments/Return values	Caller
f12-f17	fa2-fa7	FP Function arguments	Caller
f18-f27	fs2-fs11	FP Saved registers	Callee
f28-f31	ft8-ft11	R[rd] = R[rs1] + R[rs2]	Caller

IEEE 754 FLOATING-POINT STANDARD

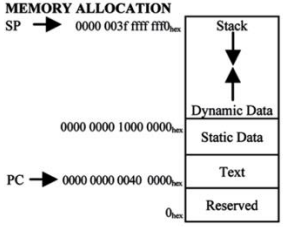
$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

where Half-Precision Bias = 15, Single-Precision Bias = 127, Double-Precision Bias = 1023, Quad-Precision Bias = 16383

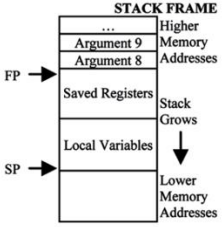
IEEE Half-, Single-, Double-, and Quad-Precision Formats:



MEMORY ALLOCATION



STACK FRAME



SIZE PREFIXES AND SYMBOLS

SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
1000^1	Kilo-	K	2^{10}	Kibi-	Ki
1000^2	Mega-	M	2^{20}	Mebi-	Mi
1000^3	Giga-	G	2^{30}	Gibi-	Gi
1000^4	Tera-	T	2^{40}	Tebi-	Ti
1000^5	Peta-	P	2^{50}	Pebi-	Pi
1000^6	Exa-	E	2^{60}	Exbi-	Ei
1000^7	Zetta-	Z	2^{70}	Zebi-	Zi
1000^8	Yotta-	Y	2^{80}	Yobi-	Yi
1000^9	Ronna-	R	2^{90}	Robi-	Ri
1000^{10}	Quecca-	Q	2^{100}	Quebi-	Qi
1000^{-1}	milli-	m	1000^{-5}	femto-	f
1000^{-2}	micro-	µ	1000^{-6}	atto-	a
1000^{-3}	nano-	n	1000^{-7}	zepto-	z
1000^{-4}	pico-	p	1000^{-8}	yocto-	y
			1000^{-9}	ronto-	r
			1000^{-10}	quecto-	q

RISC-V Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

OpCode

1100011

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
lb	I	0000011	000		03/0
lh	I	0000011	001		03/1
lw	I	0000011	010		03/2
lbu	I	0000011	100		03/4
lhu	I	0000011	101		03/5
fence	I	0001111	000		0F/0
fence.i	I	0001111	001		0F/1
addi	I	0010011	000		13/0
slli	I	0010011	001	0000000	13/1/00
slti	I	0010011	010		13/2
sltiu	I	0010011	011		13/3
xori	I	0010011	100		13/4
srli	I	0010011	101	0000000	13/5/00
srai	I	0010011	101	0100000	13/5/20
ori	I	0010011	110		13/6
andi	I	0010011	111		13/7
auipc	U	0010111			17
sb	S	0100011	000		23/0
sh	S	0100011	001		23/1
sw	S	0100011	010		23/2
add	R	0110011	000	0000000	33/0/00
sub	R	0110011	000	0100000	33/0/20
sll	R	0110011	001	0000000	33/1/00
slt	R	0110011	010	0000000	33/2/00
sltu	R	0110011	011	0000000	33/3/00
xor	R	0110011	100	0000000	33/4/00
srl	R	0110011	101	0000000	33/5/00
sra	R	0110011	101	0100000	33/5/20
or	R	0110011	110	0000000	33/6/00
and	R	0110011	111	0000000	33/7/00
lui	U	0110111			37
beq	SB	1100011	000		63/0
bne	SB	1100011	001		63/1
blt	SB	1100011	100		63/4
bge	SB	1100011	101		63/5
bltu	SB	1100011	110		63/6
bgeu	SB	1100011	111		63/7
jalr	I	1100111	000		67/0
jal	UJ	1101111			6F
ecall	I	1110011	000	000000000000	73/0/000
ebreak	I	1110011	000	000000000001	73/0/001
CSRRW	I	1110011	001		73/1
CSRRS	I	1110011	010		73/2
CSRRC	I	1110011	011		73/3
CSRRWI	I	1110011	101		73/5
CSRRSI	I	1110011	110		73/6
CSRRCI	I	1110011	111		73/7

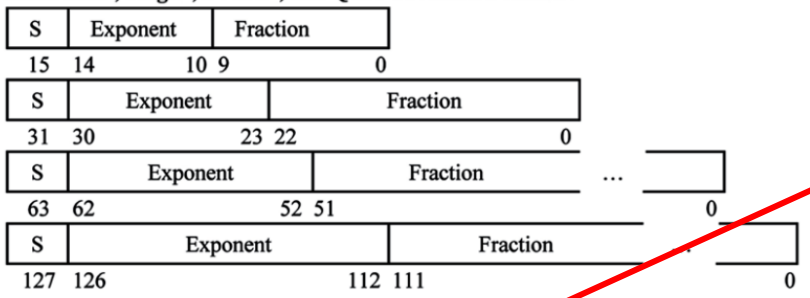
f12-f17	fa2-fa7	FP Function arguments	Caller
f18-f27	fs2-fs11	FP Saved registers	Callee
f28-f31	ft8-ft11	R[rd] = R[rs1] + R[rs2]	Caller

IEEE 754 FLOATING-POINT STANDARD

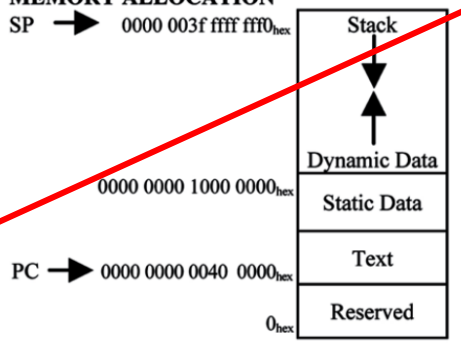
$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

where Half-Precision Bias = 15, Single-Precision Bias = 127, Double-Precision Bias = 1023, Quad-Precision Bias = 16383

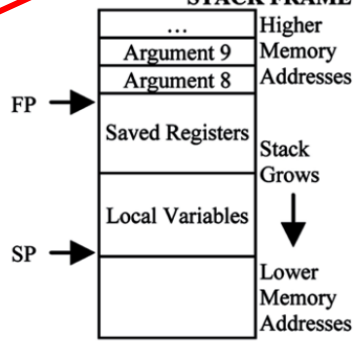
IEEE Half-, Single-, Double-, and Quad-Precision Formats:



MEMORY ALLOCATION



STACK FRAME



SIZE PREFIXES AND SYMBOLS

SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
1000^1	Kilo-	K	2^{10}	Kibi-	Ki
1000^2	Mega-	M	2^{20}	Mebi-	Mi
1000^3	Giga-	G	2^{30}	Gibi-	Gi
1000^4	Tera-	T	2^{40}	Tebi-	Ti
1000^5	Peta-	P	2^{50}	Pebi-	Pi
1000^6	Exa-	E	2^{60}	Exbi-	Ei
1000^7	Zetta-	Z	2^{70}	Zebi-	Zi
1000^8	Yotta-	Y	2^{80}	Yobi-	Yi
1000^9	Ronna-	R	2^{90}	Robi-	Ri
1000^{10}	Quecca-	Q	2^{100}	Quebi-	Qi
1000^{-1}	milli-	m	1000^{-5}	femto-	f
1000^{-2}	micro-	µ	1000^{-6}	atto-	a
1000^{-3}	nano-	n	1000^{-7}	zepto-	z
1000^{-4}	pico-	p	1000^{-8}	yocto-	y
			1000^{-9}	ronto-	r
			1000^{-10}	quecto-	q

1. Pull along perforation to separate card 2. Fold bottom sie

e Data Card ("Green Card")

OpCode

1100011

OpCode

1100011

Format: SB

beq	SB	1100011	000		63/0
bne	SB	1100011	001		63/1
blt	SB	1100011	100		63/4
bge	SB	1100011	101		63/5
bltu	SB	1100011	110		63/6
bgeu	SB	1100011	111		63/7
jalr	I	1100111	000		67/0
jal	UJ	1101111			6F
ecall	I	1110011	000	000000000000	73/0/000
ebreak	I	1110011	000	000000000001	73/0/001
CSRRW	I	1110011	001		73/1
CSRRS	I	1110011	010		73/2
CSRRC	I	1110011	011		73/3
CSRRWI	I	1110011	101		73/5
CSRRSI	I	1110011	110		73/6
CSRRCI	I	1110011	111		73/7

RISC-V Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

RISC-V Reference Data ①

RV321 BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
add	R	ADD	R[rd] = R[rs1] + R[rs2]	
addi	I	ADD Immediate	R[rd] = R[rs1] + imm	
and	R	AND	R[rd] = R[rs1] & R[rs2]	
andi	I	AND Immediate	R[rd] = R[rs1] & imm	
auipc	U	Add Upper Immediate to PC	R[rd] = PC + {imm, 12'b0}	
beq	SB	Branch EQual	if(R[rs1]==R[rs2] PC=PC+{imm,1b'0}	
bge	SB	Branch Greater than or Equal	if(R[rs1]>=R[rs2] PC=PC+{imm,1b'0}	
bgeu	SB	Branch ≥ Unsigned	if(R[rs1]>=R[rs2] PC=PC+{imm,1b'0}	2)
blt	SB	Branch Less Than	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	
bltu	SB	Branch Less Than Unsigned	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	2)
bne	SB	Branch Not Equal	if(R[rs1]!=R[rs2) PC=PC+{imm,1b'0}	
csrrc	I	Cont./Stat.RegRead&Clear	R[rd] = CSR;CSR = CSR & ~R[rs1]	
csrrci	I	Cont./Stat.RegRead&Clear Imm	R[rd] = CSR;CSR = CSR & ~imm	
csrrs	I	Cont./Stat.RegRead&Set	R[rd] = CSR; CSR = CSR | R[rs1]	
csrrsi	I	Cont./Stat.RegRead&Set Imm	R[rd] = CSR; CSR = CSR | imm	
csrrw	I	Cont./Stat.RegRead&Write	R[rd] = CSR; CSR = R[rs1]	
csrrwi	I	Cont./Stat.Reg Read&Write Imm	R[rd] = CSR; CSR = imm	
ebreak	I	Environment BREAK	Transfer control to debugger	
ecall	I	Environment CALL	Transfer control to operating system	
fence	I	Synch thread	Synchronizes threads	
fence.i	I	Synch Instr & Data	Synchronizes writes to instruction stream	
jal	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1b'0}	
jalr	I	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm	
lb	I	Load Byte	R[rd] = {24'bM[](7),M[R[rs1]+imm](7:0)}	3) 4)
lbu	I	Load Byte Unsigned	R[rd] = {24'b0,M[R[rs1]+imm](7:0)}	
lh	I	Load Halfword	R[rd] = {16'bM[](15),M[R[rs1]+imm](15:0)}	
lhu	I	Load Halfword Unsigned	R[rd] = {16'b0,M[R[rs1]+imm](15:0)}	4)
lui	U	Load Upper Immediate	R[rd] = {imm, 12'b0}	
lw	I	Load Word	R[rd] = {M[R[rs1]+imm](31:0)}	
or	R	OR	R[rd] = R[rs1] | R[rs2]	
ori	I	OR Immediate	R[rd] = R[rs1] | imm	4)
sb	S	Store Byte	M[R[rs1]+imm](7:0) = R[rs2](7:0)	
sh	S	Store Halfword	M[R[rs1]+imm](15:0) = R[rs2](15:0)	
sll	R	Shift Left	R[rd] = R[rs1] << R[rs2]	
slli	I	Shift Left Immediate	R[rd] = R[rs1] << imm	
slt	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
slti	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0	
sltiu	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0	
sltu	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
sra	R	Shift Right Arithmetic	R[rd] = R[rs1] >> R[rs2]	
srai	I	Shift Right Arith Imm	R[rd] = R[rs1] >> imm	2)
srl	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	2)
srli	I	Shift Right Immediate	R[rd] = R[rs1] >> imm	5)
sub,subw	R	SUBtract (Word)	R[rd] = R[rs1] – R[rs2]	5)
sw	S	Store Word	M[R[rs1]+imm](31:0) = R[rs2](31:0)	
xor	R	XOR	R[rd] = R[rs1] ^ R[rs2]	
xori	I	XOR Immediate	R[rd] = R[rs1] ^ imm	

Notes: 1) Operation assumes unsigned integers (instead of 2's complement)
2) The least significant bit of the branch address in jalr is set to 0
3) (signed) Load instructions extend the sign bit of data to fill the 32-bit register
4) Replicates the sign bit to fill in the leftmost bits of the result during right shift
5) Multiply with one operand signed and one unsigned
6) The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F register
7) Classify writes a 10-bit mask to show which properties are true (e.g., –inf, -0, +0, +inf, denorm, ...)
8) Atomic memory operation; nothing else can interpose itself between the read and the write of the memory location
The immediate field is sign-extended in RISC-V

ARITHMETIC CORE INSTRUCTION SET ②

RV64M Multiply Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
mul	R	MULtiply	R[rd] = (R[rs1] * R[rs2])(63:0)	
mulh	R	MULtiply High	R[rd] = (R[rs1] * R[rs2])(127:64)	
mulhsu	R	MULtiply High Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	2)
mulhu	R	MULtiply upper Half Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	6)
div	R	DIVide	R[rd] = (R[rs1] / R[rs2])	
divu	R	DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])	2)
rem	R	REMainder	R[rd] = (R[rs1] % R[rs2])	
remu	R	REMainder Unsigned	R[rd] = (R[rs1] % R[rs2])	2)

RV64F and RV64D Floating-Point Extensions

fld,flw	I	Load (Word)	F[rd] = M[R[rs1]+imm]	
fsd,fsw	S	Store (Word)	M[R[rs1]+imm] = F[rd]	
fadd.s,fadd.d	R	ADD	F[rd] = F[rs1] + F[rs2]	7)
fsub.s,fsub.d	R	SUBtract	F[rd] = F[rs1] – F[rs2]	7)
fmul.s,fmul.d	R	MULtiply	F[rd] = F[rs1] * F[rs2]	7)
fdiv.s,fdiv.d	R	DIVide	F[rd] = F[rs1] / F[rs2]	7)
fsqrt.s,fsqrt.d	R	SQuare RooT	F[rd] = sqrt(F[rs1])	7)
fmadd.s,fmadd.d	R	Multiply-ADD	F[rd] = F[rs1] * F[rs2] + F[rs3]	7)
fmsub.s,fmsub.d	R	Multiply-SUBtract	F[rd] = F[rs1] * F[rs2] - F[rs3]	7)
fmnsub.s,fmnsub.d	R	Negative Multiply-ADD	F[rd] = –(F[rs1] * F[rs2] – F[rs3])	7)
fmnadd.s,fmnadd.d	R	Negative Multiply-SUBtract	F[rd] = –(F[rs1] * F[rs2] + F[rs3])	7)
fsgnj.s,fsgnj.d	R	SiGN source	F[rd] = { F[rs2]<63>,F[rs1]<62:0>}	7)
fsgnjn.s,fsgnjn.d	R	Negative SiGN source	F[rd] = { (~F[rs2]<63>),F[rs1]<62:0>}	7)
fsgnjx.s,fsgnjx.d	R	Xor SiGN source	F[rd] = {F[rs2]<63>^F[rs1]<63>, F[rs1]<62:0>}	7)
fmin.s,fmin.d	R	MINimum	F[rd] = (F[rs1] < F[rs2]) ? F[rs1] : F[rs2]	7)
fmax.s,fmax.d	R	MAXimum	F[rd] = (F[rs1] > F[rs2]) ? F[rs1] : F[rs2]	7)
feq.s,feq.d	R	Compare Float EQual	R[rd] = (F[rs1]== F[rs2]) ? 1 : 0	7)
flt.s,flt.d	R	Compare Float Less Than	R[rd] = (F[rs1]< F[rs2]) ? 1 : 0	7)
fle.s,fle.d	R	Compare Float Less than or =	R[rd] = (F[rs1]<= F[rs2]) ? 1 : 0	7)
fclass.s,fclass.d	R	Classify Type	R[rd] = class(F[rs1])	7,8)
fmv.s.x,fmv.d.x	R	Move from Integer	F[rd] = R[rs1]	7)
fmv.x.s,fmv.x.d	R	Move to Integer	R[rd] = F[rs1]	7)
fcvt.d.s	R	Convert from SP to DP	F[rd] = single(F[rs1])	
fcvt.s.d	R	Convert from DP to SP	F[rd] = double(F[rs1])	
fcvt.s.w,fcvt.d.w	R	Convert from 32b Integer	F[rd] = float(R[rs1](31:0))	7)
fcvt.s.l,fcvt.d.l	R	Convert from 64b Integer	F[rd] = float(R[rs1](63:0))	7)
fcvt.s.wu,fcvt.d.wu	R	Convert from 32b Int Unsigned	F[rd] = float(R[rs1](31:0))	2,7)
fcvt.s.lu,fcvt.d.lu	R	Convert from 64b Int Unsigned	F[rd] = float(R[rs1](63:0))	2,7)
fcvt.w.s,fcvt.w.d	R	Convert to 32b Integer	R[rd](31:0) = integer(F[rs1])	7)
fcvt.l.s,fcvt.l.d	R	Convert to 64b Integer	R[rd](63:0) = integer(F[rs1])	7)
fcvt.wu.s,fcvt.wu.d	R	Convert to 32b Int Unsigned	R[rd](31:0) = integer(F[rs1])	2,7)
fcvt.lu.s,fcvt.lu.d	R	Convert to 64b Int Unsigned	R[rd](63:0) = integer(F[rs1])	2,7)

RV64A Atomic Extension

amoadd.w,amoadd.d	R	ADD	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] + R[rs2]	9)
amoand.w,amoand.d	R	AND	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] & R[rs2]	9)
amomax.w,amomax.d	R	MAXimum	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amomaxu.w,amomaxu.d	R	MAXimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amomin.w,amomin.d	R	MINimum	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amominu.w,amominu.d	R	MINimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amoor.w,amoor.d	R	OR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] | R[rs2]	9)
amoswap.w,amoswap.d	R	SWAP	R[rd] = M[R[rs1]], M[R[rs1]] = R[rs2]	9)
amoxor.w,amoxor.d	R	XOR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] ^ R[rs2]	9)
lr.w,lr.d	R	Load Reserved	R[rd] = M[R[rs1]], reservation on M[R[rs1]]	
sc.w,sc.d	R	Store Conditional	if reserved, M[R[rs1]] = R[rs2], R[rd] = 0; else R[rd] = 1	

CORE INSTRUCTION FORMATS

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

OpCode

1100011

Format: SB

OpCode

1100011

Format: SB

Instruction binary in hexadecimal:

0xFE0008E3

funct3

OpCode

1111 1110 0000 0000 0000 1000 1110 0011

CORE INSTRUCTION FORMATS

	31 27 26 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	Opcode
I	imm[11:0]		rs1	funct3	rd	Opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode

Funct3: 000

OpCode

1100011

Format: SB

beq	SB	1100011	000		63/0
bne	SB	1100011	001		63/1
blt	SB	1100011	100		63/4
bge	SB	1100011	101		63/5
bltu	SB	1100011	110		63/6
bgeu	SB	1100011	111		63/7
jalr	I	1100111	000		67/0
jal	UJ	1101111			6F
ecall	I	1110011	000	000000000000	73/0/000
ebreak	I	1110011	000	000000000001	73/0/001
CSRRW	I	1110011	001		73/1
CSRRS	I	1110011	010		73/2
CSRRC	I	1110011	011		73/3
CSRRWI	I	1110011	101		73/5
CSRRSI	I	1110011	110		73/6
CSRRCI	I	1110011	111		73/7

OpCode

1100011

Format: SB

Instruction binary in hexadecimal:

0xFE0008E3

	rs2	rs1	funct3	OpCode
1111 1110	0000 0000	0000 1000	1110	0011

Immediate: 1111 1111 1111 1111 1111 1111 1111 0000

CORE INSTRUCTION FORMATS

	31 27 26 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	Opcode
I	imm[11:0]		rs1	funct3	rd	Opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode

Example

A 32-bit RISC-V processor fetched an instruction whose hexadecimal representation is:

0xFE0008E3

From the address **0x00400010**

What is the address of the next instruction that will be executed?

PC = 0x00400010

PC = 0000 0000 0100 0000 0000 0000 0001 0000

+ 1111 1111 1111 1111 1111 1111 1111 0000

New PC = 0000 0000 0100 0000 0000 0000 0000 0000

New PC = 0x00400000

$$
\begin{array}{r}
PC = 0x00400010 \\
+\ 0xFFFFFFF0 \\
\hline
\text{New } PC = 0x00400000
\end{array}
$$

Example

A 32-bit RISC-V processor fetched an instruction whose hexadecimal representation is:

0xFE0008E3

From the address **0x00400010**

What is the address of the next instruction that will be executed?

0x00400000

RISC-V does not have a separate jump instruction

Jump

Unconditional jump to an address

Longer range than a branch instruction

Jump-and-Link (**jal**)

It uses a jal for a common jump

Unconditional jump to first instruction of another function

Must record, in a register, the return address

But it ``records” the ``return address” in x0

```
void foo()
{
 …
 r = bar(a, b);
 t = r + 2;
}
```

When making this call

```
int bar(int x, int y)
{
 …
 return p;
}
```

```
…
jal   ra, bar
addi   s1, a0, 2
…
```

Must remember this address

The jump-and-link instruction

The Constant Zero

RISC-V register zero (`x0`) is the constant `0`

It is hard-wired and cannot be overwritten

Useful for common operations

E.g. moving between registers

```
add   t2, s1, zero
```

E.g. setting a variable to `0`

```
add s4, zero, zero
```

E.g. two ways to perform an unconditional jump

```
beq   zero, zero, LABEL
jal   zero, LABEL
```

Jump Instructions: UJ-Type

Memory Address

```
0x1000 0000          bne    s3, s4, Subtr       # if i ≠ j goto Subtr
0x1000 0004          add    s0, s1, s2          # f ← g + h
0x1000 0008          jal    zero, Exit          # goto Exit
0x1000 000C Subtr:   sub    s0, s1, s2          # f ← g – h
0x1000 0010 Exit:           …
```

`jal zero, Exit` ↔ `R[rd] = PC + 4 ; PC = PC + {imm, 1b'0}`

immediate[20|10:1|11|19:12]	rd	op
0|0x004|0|0x00	0	0x6F

31 … 12	11 … 7	6 … 0
0|0000000100|0|00000000	00000	1101111

Hexadecimal Representation: 0x0080006F

Jump Instructions: UJ-Type

Memory Address

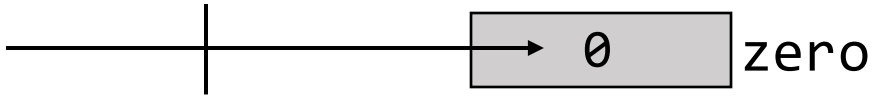
```
0x1000 0000          bne    s3, s4, Subtr       # if i ≠ j goto Subtr
0x1000 0004          add    s0, s1, s2          # f ← g + h
0x1000 0008          jal    zero, Exit          # goto Exit
0x1000 000C Subtr:   sub    s0, s1, s2          # f ← g – h
0x1000 0010 Exit:           …
```

```
jal zero, Exit ↔ R[rd] = PC + 4 ; PC = PC + {imm, 1b'0}
```

31 immediate[20|10:1|11|19:12] 12	11 rd 7	6 op 0
0|0000000100|0|00000000	00000	1101111

Saving return address:

```
PC      →    0x1000 0008
4       →  + 0x0000 0004
PC + 4  ←    0x1000 000C
```



Register zero cannot be overwritten so PC + 4 is discarded

Jump Instructions: UJ-Type

Memory Address

```
0x1000 0000          bne    s3, s4, Subtr        # if i ≠ j goto Subtr
0x1000 0004          add    s0, s1, s2           # f ← g + h
0x1000 0008          jal    zero, Exit           # goto Exit
0x1000 000C Subtr:   sub    s0, s1, s2           # f ← g – h
0x1000 0010 Exit:           …
```

```
jal zero, Exit ↔ R[rd] = PC + 4 ; PC = PC + {imm, 1b'0}
```

31 immediate[20|10:1|11|19:12] 12	11 rd 7	6 op 0
0|0000000100|0|00000000	00000	1101111

0 0000 0000 0000 0000 1000

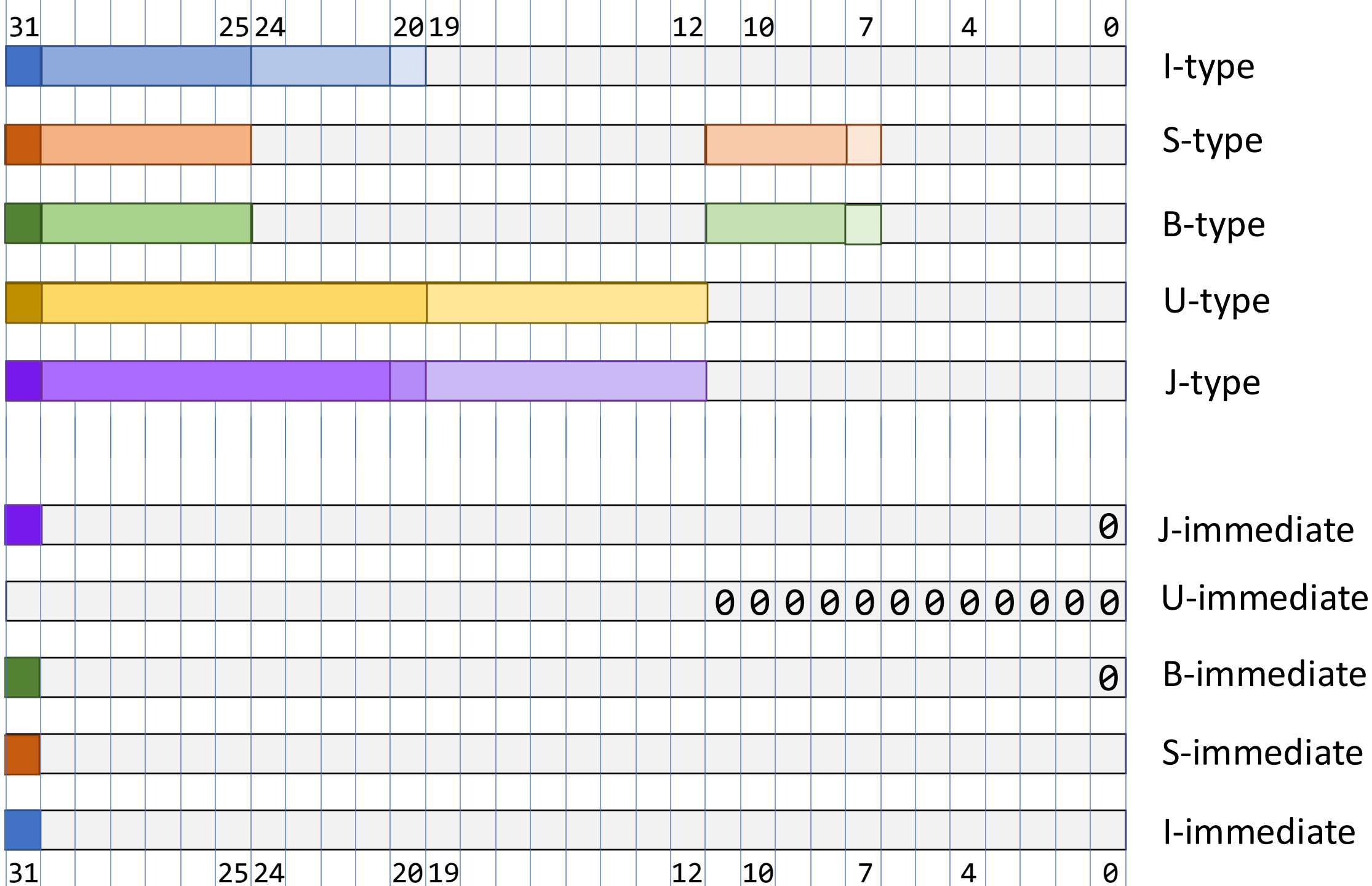
implicit bit 0

Calculating immediate for jump:

8 bytes to jump → immediate is 8

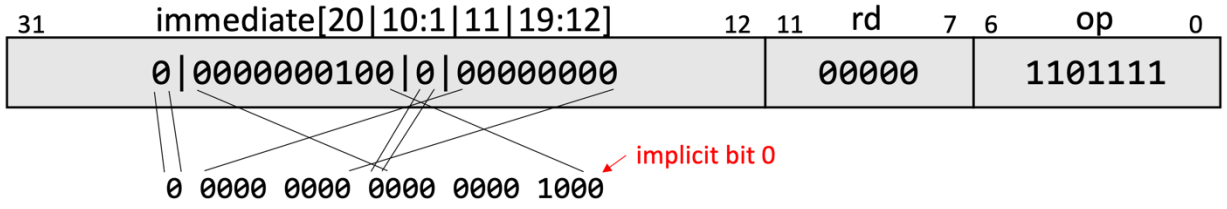
What is 8 expressed in 21 bit signed binary?

```
0000 0000 0000 0000 0000 0000x0000 0008   ← sign-extended (written as hex)
                               + 0x1000 0008   ← PC
                                 0x1000 0010   ← New PC
```


31
25
24
20
19
12
10
7
4
0
I-type
S-type
B-type
U-type
J-type
J-immediate
0
U-immediate
0 0 0 0 0 0 0 0 0 0 0 0
B-immediate
0
S-immediate
I-immediate
31
25
24
20
19
12
10
7
4
0

```
0x1000 0000          bne    s3, s4, Subtr
0x1000 0004          add    s0, s1, s2
0x1000 0008          jal    zero, Exit
0x1000 000C Subtr:   sub    s0, s1, s2
0x1000 0010 Exit:           …
```

Labels appear in the assembly program, but not in the binary representation



21-bit immediate

Immediate is sign extended

Can jump back

Jump recap

Hardware-oriented binary format