Digital Design with Verilog

RISC-V : Instruction Set Architecture

Lecture 27 : RISC-V Part 2

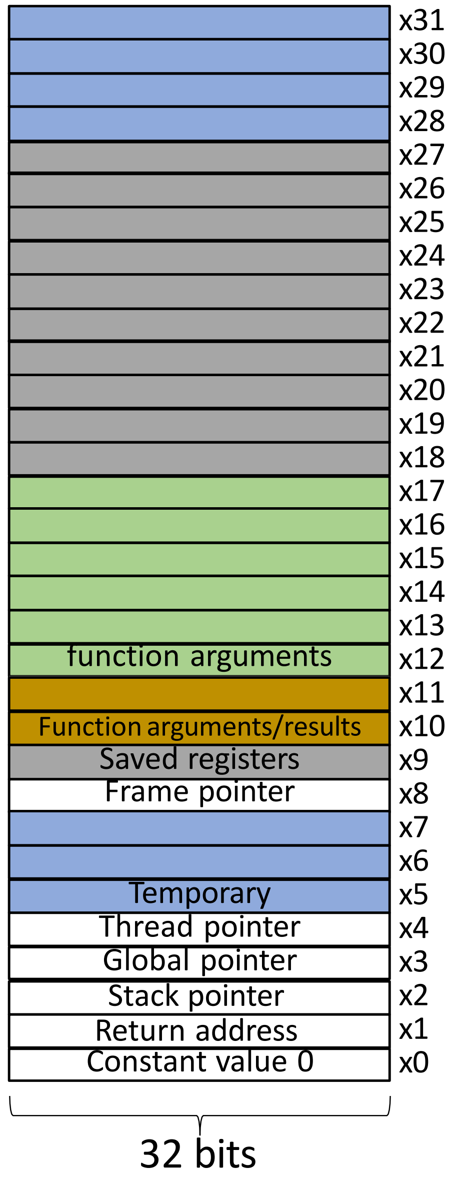
Disclaimer

I do not own all the slides which I am going to present. The content is mostly from

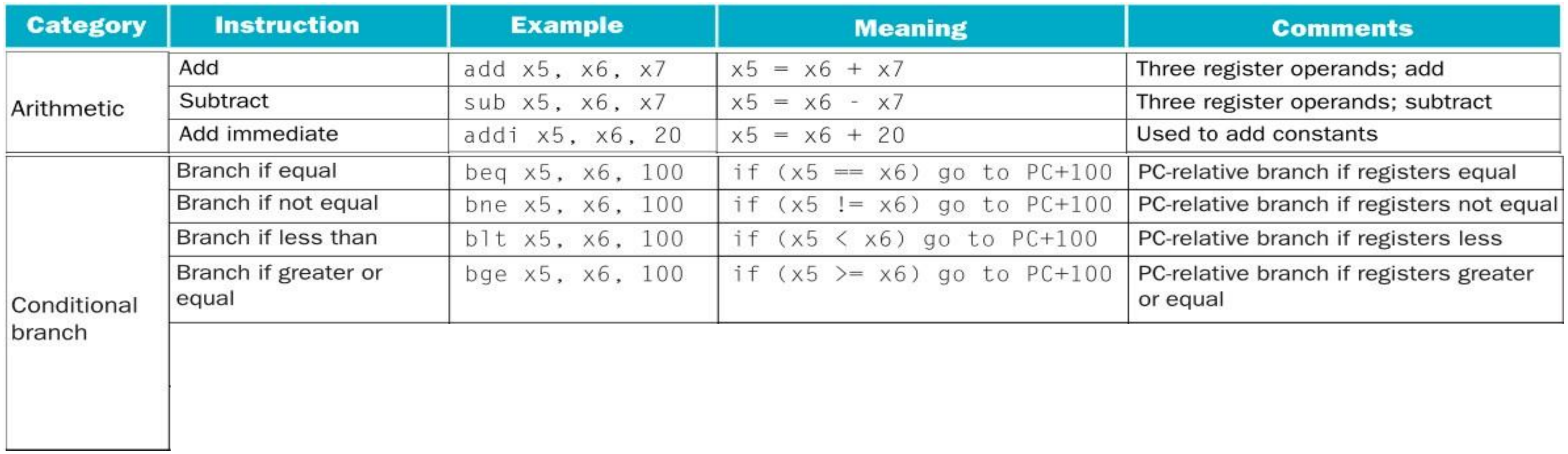
- GIAN course titled "**Next-Generation Semiconductors: RISC-V, AI, TL-Verilog**" by Steeve Hoover. https://github.com/silicon-vlsi/gian-course-2024-IITBBS
- Introduction to Computer Architecture by Hasan Baig, https://www.hasanbaig.com
  - Most of the slides are copied/directly from his course content, please visit his site for reference.
- Digital Design and Computer Architecture, RISC-V Edition by Sarah L. Harris and David Money Harris.
- Other online publicly available sources. I tried my best to acknowledge the source wherever possible. I too might have missed some sources too ☺

Quick Recap

- RISC-V Registers ➔ 32 registers each of them is 32-bits wide
- Learnt How to translate C code to pseudocode and then to assembly language



Quick Recap



Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	`add x5, x6, x7`	`x5 = x6 + x7`	Three register operands; add
	Subtract	`sub x5, x6, x7`	`x5 = x6 - x7`	Three register operands; subtract
	Add immediate	`addi x5, x6, 20`	`x5 = x6 + 20`	Used to add constants
Conditional branch	Branch if equal	`beq x5, x6, 100`	`if (x5 == x6) go to PC+100`	PC-relative branch if registers equal
	Branch if not equal	`bne x5, x6, 100`	`if (x5 != x6) go to PC+100`	PC-relative branch if registers not equal
	Branch if less than	`blt x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
	Branch if greater or equal	`bge x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal

Compiling Loop Statements

```
sum = 0;

for (i = 0; i < 100; i += 1)
  {

      sum += i;

}
```

Convert a for loop to a while loop:

```
sum = 0;

i = 0;

while (i < 100) {

      sum += i;

      i += 1;
}
```

Do not forget to increment the loop control variable.

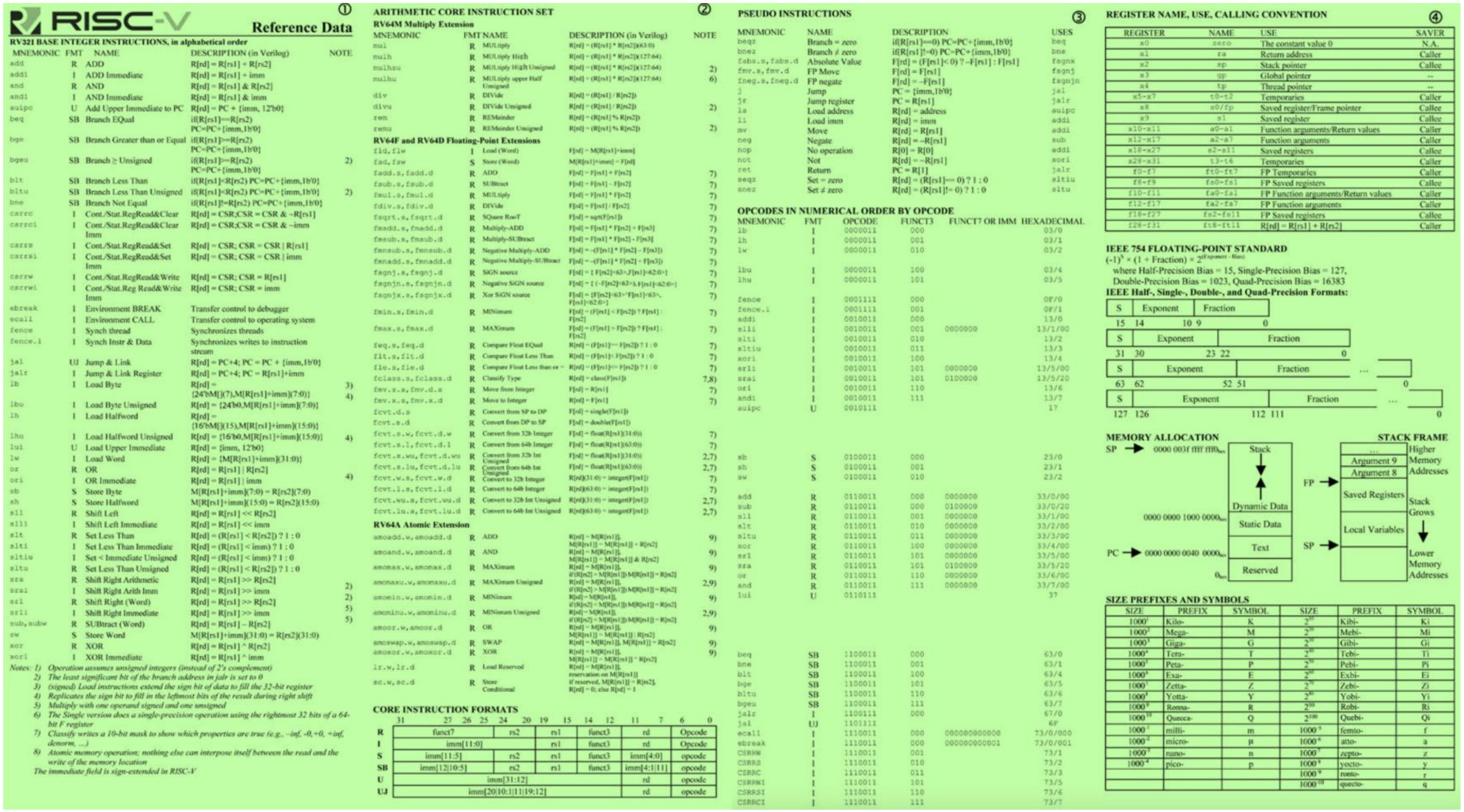
Pseudo Instructions

- Pseudo instructions are fake instructions
- Purpose: Make it easier for programmers to write assembly code
  - Assembler converts pseudo instructions to real instructions

Pseudoinstruction	Real Instructions
`nop`	`addi x0, x0, 0`
`mv rd, rs`	`addi rd, rs, 0`
`neg rd, rs`	`sub rd, x0, rs`
`li rd, immd`	`addi rd, x0, immd` `Or more than one instruction` *

* Depending on immd. If immd is small, use only one instruction.

RISC-V Instruction Set



RISC-V Reference Data ①

RV32I 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

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
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

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
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 … 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

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:

S	Exponent	Fraction
15	14 … 10	9 … 0

S	Exponent	Fraction
31	30 … 23	22 … 0

S	Exponent	Fraction …
63	62 … 52	51 … 0

S	Exponent	Fraction …
127	126 … 112	111 … 0

MEMORY ALLOCATION

SP → 0000 003f ffff fff0hex: Stack ↓ ↑ Dynamic Data
0000 0000 1000 0000hex: Static Data
PC → 0000 0000 0040 0000hex: Text
0hex: Reserved

STACK FRAME

Higher Memory Addresses: ... | Argument 9 | Argument 8
FP → Saved Registers
Local Variables (Stack Grows ↓)
SP →
Lower Memory Addresses

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

Warmup Activity

Which register is larger? Is the condition true?

```
if (t0 < t1) goto L
```

```
t0   1111 1111 1111 1111 1111 1111 1111 1111

t1   0000 0000 0000 0000 0000 0000 0000 0001
```

signed: t0 < t1 because -1 < 1

unsigned: t0 > t1 because $(2^{32} – 1) > 1$

Branch with unsigned comparisons

- Conditional branches
  - If a condition is true, go to the instruction indicated by the label
  - Otherwise, continue sequentially

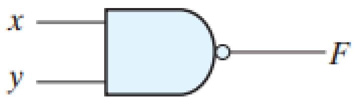
```
beq   rs1, rs2, L1    # if (rs1 == rs2) goto L1
bne   rs1, rs2, L2    # if (rs1 != rs2) goto L2

# compare signed numbers
blt   rs1, rs2, L3    # if (rs1 <  rs2) goto L3
bge   rs1, rs2, L4    # if (rs1 >= rs2) goto L4
```

```
# compare unsigned numbers
bltu  rs1, rs2, L     # if (rs1 <  rs2) goto L
bgeu  rs1, rs2, L     # if (rs1 >= rs2) goto L
```

Recalling Logical Operations

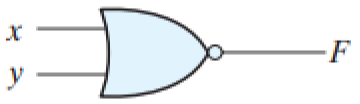
NAND



$F = (xy)'$

x	y	F
0	0	1
0	1	1
1	0	1
1	1	0

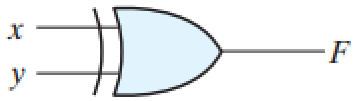
NOR



$F = (x + y)'$

x	y	F
0	0	1
0	1	0
1	0	0
1	1	0

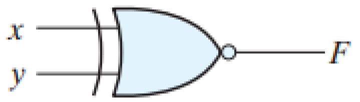
Exclusive-OR (XOR)



$F = xy' + x'y$
$= x \oplus y$

x	y	F
0	0	0
0	1	1
1	0	1
1	1	0

Exclusive-NOR or equivalence



$F = xy + x'y'$
$= (x \oplus y)'$

x	y	F
0	0	1
0	1	0
1	0	0
1	1	1

Recalling Logical Operations

A	1001 1011
B	1100 1101
A AND B	1000 1001

A	1001 0011
B	1101 1001
A OR B	1101 1011

A	1101 1011
B	1001 1111
A XOR B	0100 0100

Z	0xA6
	1010 0110
NOT Z	0101 1001

Recalling Logical Operations

Shift left	1001 1011
By 3 (<< 3)	1101 1000

Shift right logical	1001 1011
By 4 (>> 4)	0000 1001

Shift right arith.	1001 1011
By 4 (>> 4)	1111 1001

There are two versions of shift right.

The sign bit is padded from the left for shift right arithmetic.

RISC-V Instructions for Logical Operations

Operation	C/Python	RISC-V
Shift left	<<	sll, slli
Shift right logic	>>	srl, srli
Shift right arith.	>>	sra, srai
Bitwise AND	&	and, andi
Bitwise OR	|	or, ori
Bitwise NOT	~	xori
XOR	^	xor, xori

*i instructions take an immediate as the second operand

Immediates are 12-bit long and sign extended

RISC-V: **and** and **andi operations**

```
and    t0, t1, t2
andi   t0, t1, 0x5CC
```

The 12-bit immediate in ANDI is sign extended

```
         t1   0000 0000 0000 0000 0011 0100 0101 0111

t2 or immd    0000 0000 0000 0000 0000 0101 1100 1100

         t0   0000 0000 0000 0000 0000 0100 0100 0100
```

RISC-V: or and ori operations

```
or   t0, t1, t2
ori  t0, t1, 0xDC0
```

The 12-bit immediate is sign extended

```
         t1   1000 0100 0000 0000 0101 0010 0101 1010
t2 or immd    1111 1111 1111 1111 1111 1101 1100 0000
         t0   1111 1111 1111 1111 1111 1111 1101 1010
```

RISC-V: **xor** and **xori operations**

```
xor   t0, t1, t2
xori  t0, t1, 0x5CF
```

The 12-bit immediate is sign extended

t1	0101 1110 1111 1111 1100 1100 1001 1110
t2 or immd	0000 0000 0000 0000 0000 0101 1100 1111
t0	0101 1110 1111 1111 1100 1001 0101 0001

RISC-V: **xor** and **xori operations**

```
xori  t0, t1, -1
```

The 12-bit immediate is sign extended

t1	`0101 1110 1111 1111 1100 1100 1001 1110`
immd	`1111 1111 1111 1111 1111 1111 1111 1111`
t0	

What is this operation?

RISC-V: NOT Operation

- Invert bits in a word (32 bits)
  - Change 0 to 1, and 1 to 0
- RISC-V does not have NOT. NOT is done with an XOR
  - NOT is a pseudoinstruction

```
not t0, t1    # xori t0, t1, -1
```

```
t1    1111 0000 0000 0000 0011 1100 0000 0000

immd  1111 1111 1111 1111 1111 1111 1111 1111

t0    0000 1111 1111 1111 1100 0011 1111 1111
```

Quick Question

What are the bits in t0 after the following instruction is executed?

```
ori   t0, t1, -1
```

A. All bits in t0 are 1
B. All bits in t0 are 0
C. 32 bits from t1
D. Higher 20 bits are from t1. Lower 12 bits are set to 0
E. Higher 20 bits are from t1. Lower 12 bits are set to 1

Logical Operations

- Logical operators in Python: `and, or, not`
- Logical operators in C: `&&, ||, !`

In logical expressions, non-zero values are True

Logical Operations

- How do we do logical operators AND and OR in C/Python?

```
// Python: and, or
// C: &&, ||

if (cond1 && cond2) {
        // if_branch
} else {
        // else_branch
}
```

Short-circuit evaluation!

If cond1 is not true,
cond2 is not evaluated.

```
Example:
if ((p != NULL) && (p[0] < 0)) { …      // C

if obj is not None and obj.v :          # Python
```

Logical Operations in IF Statements

```
if (cond1 and cond2) then
        if_branch
Else
        else_branch
```

Pseudocode

```
if ! cond1 goto Else
if ! cond2 goto Else
        if_branch
        goto EndIf
Else:
        else_branch
EndIf:
```

```
if (cond1 or cond2) then
        if_branch
Else
        else_branch
```

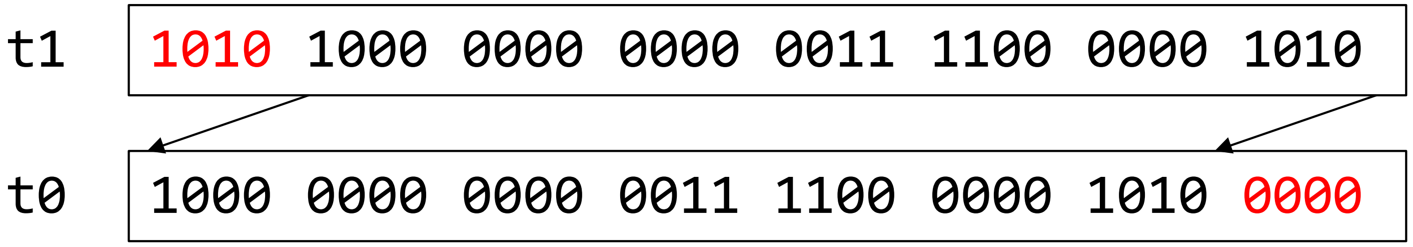
Pseudocode

```
if cond1 goto If
if ! cond2 goto else
If:
        if_branch
        goto EndIf
Else:
        else_branch
EndIf:
```

RISC-V: slli and sll operations

```
slli  t0,  t1,  4
sll   t0,  t1,  t2        # assume   t2  is  4
```

- Shift the bits in **t1** left by 4 positions
- The highest 4 bits in t1 are lost. 0 is shifted in from the right



RISC-V: srli and srai operations

```
srli  t2,  t1,  4
srai  t3,  t1,  4
```

- Shift the bits right by 4 positions
- SRLI pads with 0 and SRAI pads with the sign bit (not always 1!)

	t1	1010 1000 0000 0000 0011 1100 0000 1010
srli	t2	0000 1010 1000 0000 0000 0011 1100 0000
srai	t3	1111 1010 1000 0000 0000 0011 1100 0000

How to load 32-bit constant into a Register ?

- We are good at 12-bit immediate most of the time, but sometimes need larger numbers
- How do we load a 32-bit constant in a register?

Example:

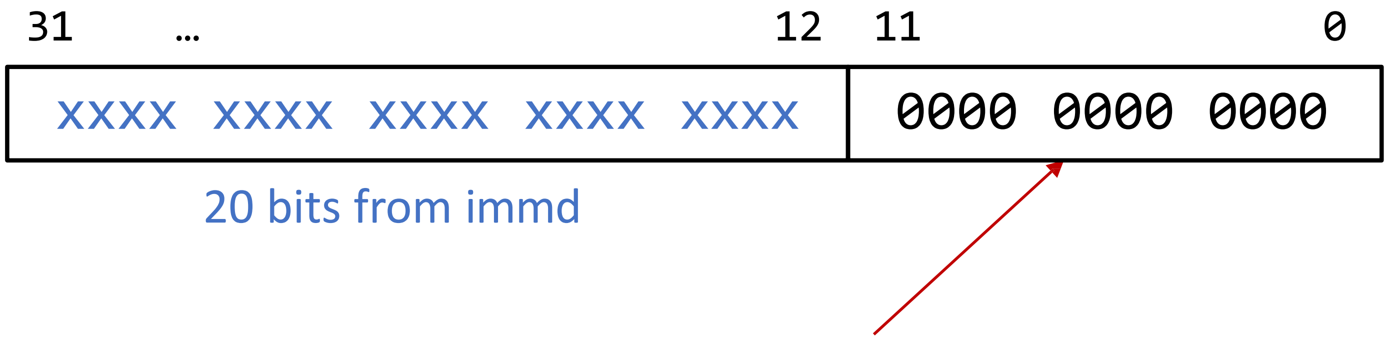
0x12345678

```
31    …                                        0
0001 0010 0011 0100 0101 0110 0111 1000
```

RISV-V: lui instruction

```
LUI rd, immd
```

- LUI allows 20-bit immediate
  - Assembler supports %hi(C) to get the higher 20 bits of C
- The 20 bits are placed into bits 12 to bits 31
  - Lower 12 bits are cleared



How do we set the lower 12 bits to other values?

RISC-V: lui Loading Large Constants

- Load 0x12345678 into register x18

```
lui    x18, 0x12345
addi   x18, x18, 0x678
```

```
    31   …                                12   11
    0001 0010 0011 0100 0101     0110 0111 1000
```

Quick Question

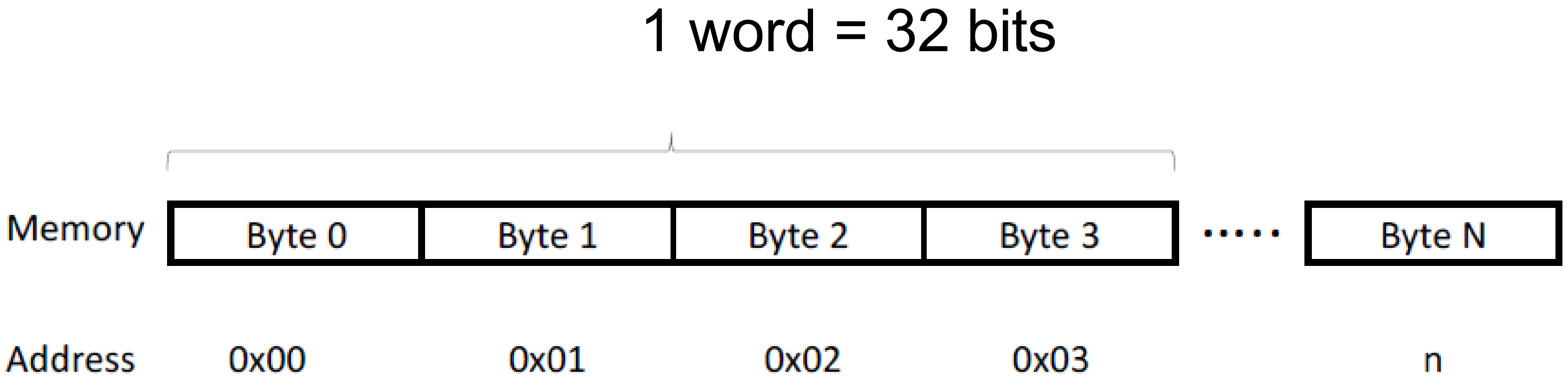
Write the instructions to load **0x789ABCDE** into register x18, so the values should appear in x18 register like below:

31 … 12	11 0
0111 1000 1001 1010 1011	1100 1101 1110

Memory Addressing

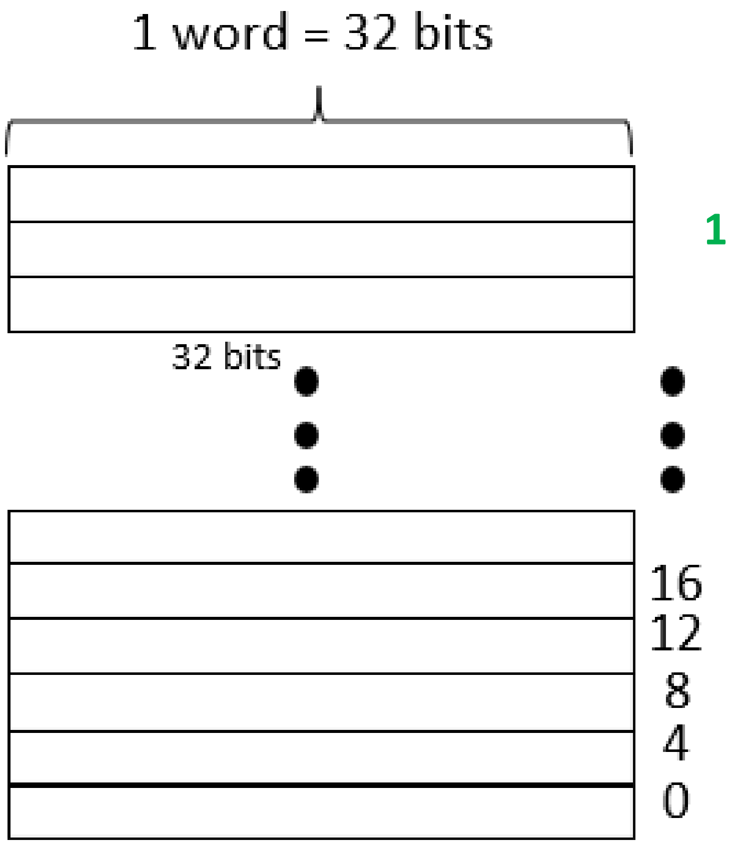
Memory is just a large, single-dimensional array of bytes

- 1 Byte = 8 bits ; 1 Word = 4 bytes = 32 bits
- Most architectures address individual bytes in memory.
- Thus, the memory address of a word must be a multiple of 4.



Memory Addressing

- Visual representation of a memory



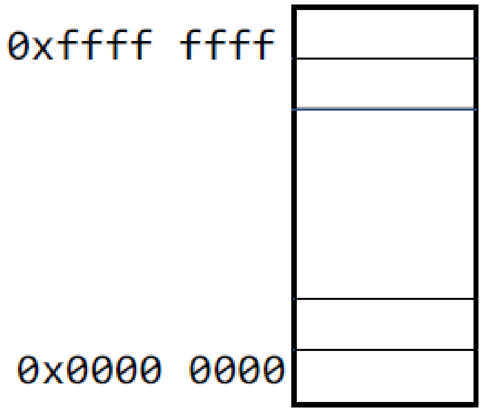
1 Byte

1 Word

Byte Address	Word Address	Memory
0x 000	0x000	
0x 001		
0x 010		
0x 011		
0x 100	0x100	
0x 101		
0x 110		
0x 111		

Memory Operands

- Main memory used for composite data
  - Arrays, structures, etc.
  - Arithmetic operations occur only on registers
- To apply arithmetic operations on memory contents (data transfer instructions)
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- A 32-bit address space supports 4 GB



Memory Addresses

Suppose word array A starts from 0x9000 (stored in s1). A[0], A[1], A[2], …, A[4000], … A[i]

We need to specify an address in load/store instructions. What are the addresses of these words?

C	Address	Values (Decimal)
	…	
A[7]	0x901C	700
A[6]	0x9018	600
A[5]	0x9014	500
A[4]	0x9010	400
A[3]	0x900C	300
A[2]	0x9008	200
A[1]	0x9004	100
A[0]	0x9000	0

4 bytes in A[1]

Address	Value
0x9007	0x00
0x9006	0x00
0x9005	0x00
0x9004	0x64

Memory Operands Example

Format of Load and Store instructions

- Load a word form memory into destination register
  - Load: **lw** R1, Offset (R2)
    - Reg [R1] = Mem[Reg[R2] + Offset]

    ➔ Load the contents from memory address **R2 + Offset** into register **R1**
- Store a word from register to memory
  - Store: **sw** R3, Offset (R2)
    - Mem[Reg[R2] + Offset] = Reg[R3]

    ➔ Store the contents of register **R3** to memory address **R2 + Offset**

Effective Address: Offset + base address in register

Offset → is a constant value. Not a register

offset range [-2048, 2047]

Memory Operands Example

Example C code:

g = h + A[7]; //A is array/list of 100 W

- g in x20, h in x21, base address of A in x22

Compiled RISC-V code:

```
lw      x9, 7(x22)
add     x20, x21, x9
```

- Index 7 requires the offset of 28
  - 4 bytes per Word

Therefore,

```
lw      x9, 28(x22)
add     x20, x21, x9
```

Byte Order

How is a word stored in memory?

```
#x1 is 0x01020304
sw     x1, 0x100(x0)
```

Which byte goes to address 0x100?

Memory Address	Value
0x0000 0103	
0x0000 0102	
0x0000 0101	
0x0000 0100	

Byte Order

```
# x1 is 0x01020304
sw      x1, 0x100(x0)
```

Big-endian: The highest byte goes to the lowest memory address.

Memory Address	Value
0x0000 0103	04
0x0000 0102	03
0x0000 0101	02
0x0000 0100	01

Little-endian: The lowest byte goes to the lowest memory address.

Memory Address	Value
0x0000 0103	01
0x0000 0102	02
0x0000 0101	03
0x0000 0100	04

RISC-V uses little endian.

Knowledge Check

What are the bits in t0 after executing the following RISC V instruction?

```
lw t0, 0x200(x0)
```

A. 0x3265 81AC

B. 0xAC81 6532

C. 0xCA18 5623

D. 0x6532 AC81

E. None of the above

Memory Address	Value
0x0000 0203	0x32
0x0000 0202	0x65
0x0000 0201	0x81
0x0000 0200	0xAC

Data of other Sizes

- RISC-V supports data of other sizes
  - Each type can be signed or unsigned

Number of bits	Name	C types (typical)
8 bits	byte	char
16 bits	half word	short int
32 bits	word	int, long int

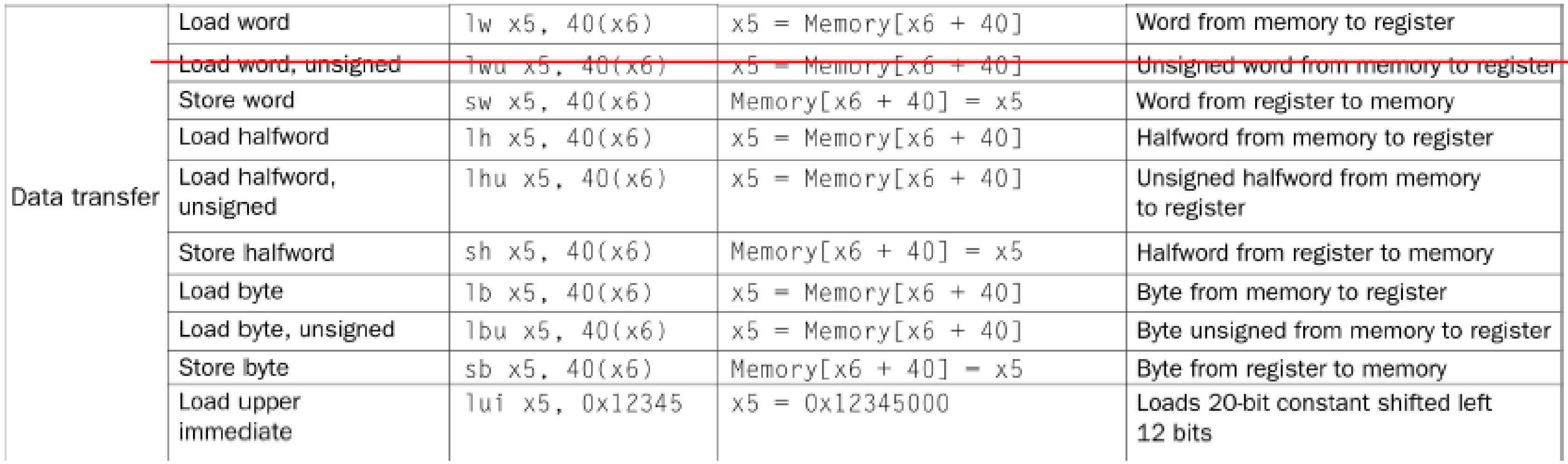
```
# load signed (sign extended) byte/halfword
lb/lh     rd, offset(rs1)

# load unsigned (0 extended) byte/halfword
lbu/lhu   rd, offset(rs1)

# Store the lowest byte/halfword
sb/sh     rs2, offset(rs1)
```

Data Transfer Instructions

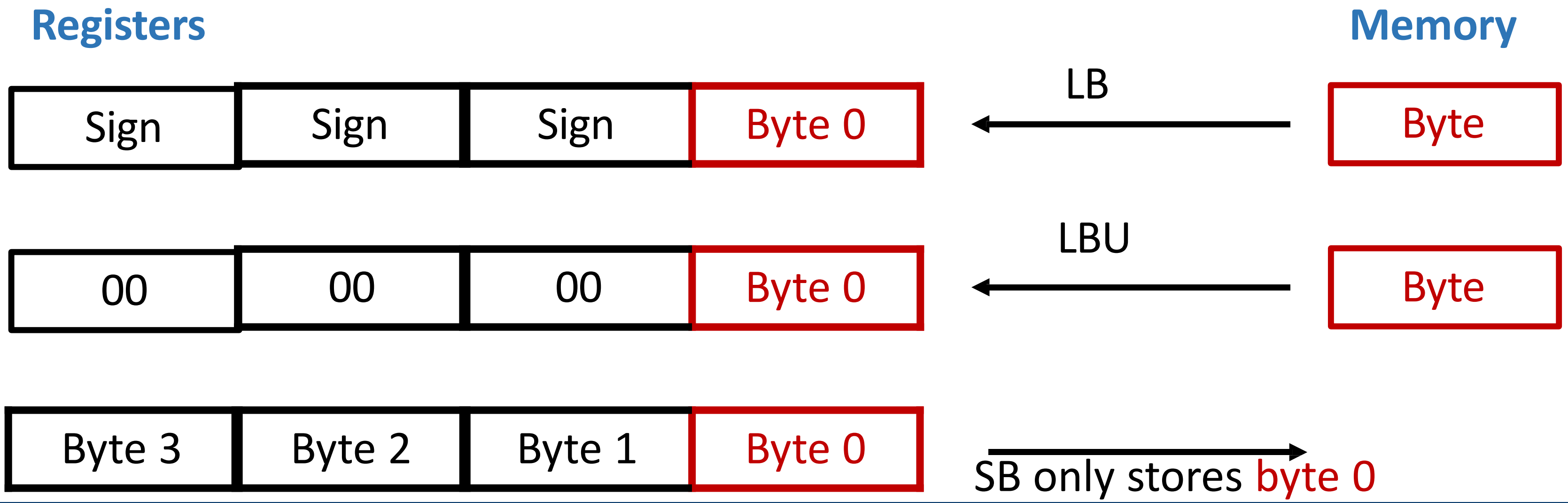
Not needed for 32-bit processor



Data transfer	Load word	`lw x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Word from memory to register
	~~Load word, unsigned~~	~~`lwu x5, 40(x6)`~~	~~`x5 = Memory[x6 + 40]`~~	~~Unsigned word from memory to register~~
	Store word	`sw x5, 40(x6)`	`Memory[x6 + 40] = x5`	Word from register to memory
	Load halfword	`lh x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Halfword from memory to register
	Load halfword, unsigned	`lhu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned halfword from memory to register
	Store halfword	`sh x5, 40(x6)`	`Memory[x6 + 40] = x5`	Halfword from register to memory
	Load byte	`lb x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte from memory to register
	Load byte, unsigned	`lbu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte unsigned from memory to register
	Store byte	`sb x5, 40(x6)`	`Memory[x6 + 40] = x5`	Byte from register to memory
	Load upper immediate	`lui x5, 0x12345`	`x5 = 0x12345000`	Loads 20-bit constant shifted left 12 bits

Data Transfer Instructions

- LB or LBU instruction loads a **byte** from memory to a register
    - LB: the byte is sign extended to a word
    - LBU: the byte is zero extended to a word
- SB instruction stores a **byte** in a register to a memory
    - Only the lowest 8 bits of the register are stored



Knowledge Check

What is the value in t0 after executing the following RISC V instruction?

```
lb   t0, 0x201(x0)
```

A. 0x0000 00AC

B. 0x0000 0081

C. 0xFFFF FFAC

D. 0xFFFF FF81

E. None of the above

Memory Address	Value
0x0000 0203	0x32
0x0000 0202	0x65
0x0000 0201	0x81
0x0000 0200	0xAC

Knowledge Check

What is the value in t0 after executing the following RISC V instruction?

```
lh   t0, 0x200(x0)
```

A. 0x0000 81AC

B. 0x0000 AC81

C. 0xFFFF 81AC

D. 0xFFFF AC81

E. None of the above

Memory Address	Value
0x0000 0203	0x32
0x0000 0202	0x65
0x0000 0201	0x81
0x0000 0200	0xAC

Registers vs Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
    - More instructions to be executed
- Compiler must use registers for variables as much as possible
    - Only spill to memory for less frequently used variables
    - Register optimization is important!

Memory Addressing

- RISC-V has only one memory addressing mode

```
// the offset must be an immediate
      lw    x1, x2(x3)    # this is wrong

// calculate the address with
      ADD add    t0, x2, x3
      lw    x1, 0(t0)
```

Thank you