

CSSE 232 – Computer Architecture I
Rose-Hulman Institute of Technology
Computer Science and Software Engineering Department

Quiz 1

ANSWER KEY

This quiz is **closed book**. You are allowed to use the reference card from the book (attached at the back of this quiz) and one 8.5" × 11" single sided page of hand written notes. You may not use a computer, phone, etc. during the examination.

Write all answers on these pages. Be sure to **show all work** and document your code. Do not use instructions that we have not covered (e.g. no `mul` or `div` but you can use instructions like `slli`, `srl`, `xori`, etc).

RISC-V code is judged both by its correctness and its efficiency. Unless otherwise stated, you may not use RISC-V pseudoinstructions when writing RISC-V code.

Question	Points	Score
Problem 1	35	
Problem 2	25	
Total:	60	

Solution: Immediate: 0 0 000001 0000 = 0000 0001 0000
`bne x16, x8, 32`
 $newPC = oldPC + imm * 2$
 $= 0x00400C28 + 2 * (16) = 0x00400C28 + 0x020 = 0x00400C48$

- (c) (5 points) Because of a non-optimized compiler, this little snippet of code was replicated as is starting from address `0xDABBAD00`. Can you write the machine language translation of the instruction `blt x4, x5, LP` in the replicated code snippet? Explain your answer in either way.

Solution: It will be exactly the same since everything is PC-relative, so we'll have the same offset and thus the same immediate.

Problem 2. (25 points) Pretend you are an assembler. For each pseudo-instruction in the following table, give a **minimal** sequence of actual RISC-V instructions to accomplish the same thing. You may need to use `x31` for some of the sequences. `BIG` indicates an immediate value that is 32 bits and `SML` indicates an immediate value that fits in 12 bits. You may need to refer to specific bits of the immediate by index, e.g. `SML[11]`.

Pseudo-instruction	Description	
<code>lli t0, MED</code>	Loads the 20-bit value <code>MED</code> into the lowest 20 bits of <code>t0</code> , placing zeroes in the upper 12 bits of the register.	<div style="border: 1px solid black; padding: 5px;"> Solution: <code>lui t0, MED</code> <code>srlt t0, t0, 12</code> </div>
<code>ogre t0, t1, BIG</code>	Overwrites the register <code>t0</code> with 32-bit value <code>BIG</code> if <code>t0</code> is greater than or equal to <code>t1</code> .	<div style="border: 1px solid black; padding: 5px;"> Solution: <code>blt t0, t1, NOPE</code> <code>lui t0, BIG[31:12] +</code> <code>BIG[11]</code> <code>addi t0, t0, BIG[11:0]</code> <code>NOPE:</code> </div>

RISC-V REFERENCE

Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description	Note
add	ADD	R	0110011	000	000 0000	$R[rd] = R[rs1] + R[rs2]$	
sub	SUB	R	0110011	000	010 0000	$R[rd] = R[rs1] - R[rs2]$	
xor	XOR	R	0110011	100	000 0000	$R[rd] = R[rs1] \wedge R[rs2]$	
or	OR	R	0110011	110	000 0000	$R[rd] = R[rs1] \vee R[rs2]$	
and	AND	R	0110011	111	000 0000	$R[rd] = R[rs1] \& R[rs2]$	
sll	Shift Left Logical	R	0110011	001	000 0000	$R[rd] = R[rs1] \ll R[rs2]$	
srl	Shift Right Logical	R	0110011	101	000 0000	$R[rd] = R[rs1] \gg R[rs2]$	
sra	Shift Right Arith*	R	0110011	101	010 0000	$R[rd] = R[rs1] \ggg R[rs2]$	sign-extends
slt	Set Less Than	R	0110011	010	000 0000	$R[rd] = (rs1 < rs2)?1:0$	
addi	ADD Immediate	I	0010011	000		$R[rd] = R[rs1] + SE(imm)$	
xori	XOR Immediate	I	0010011	100		$R[rd] = R[rs1] \wedge SE(imm)$	
ori	OR Immediate	I	0010011	110		$R[rd] = R[rs1] \vee SE(imm)$	
andi	AND Immediate	I	0010011	111		$R[rd] = R[rs1] \& SE(imm)$	
slli	Shift Left Logical Imm	I	0010011	001	imm[11:5] = 0x00	$R[rd] = R[rs1] \ll imm[4:0]$	
srl	Shift Right Logical Imm	I	0010011	101	imm[11:5] = 0x00	$R[rd] = R[rs1] \gg imm[4:0]$	
sra	Shift Right Arith Imm	I	0010011	101	imm[11:5] = 0x20	$R[rd] = R[rs1] \ggg imm[4:0]$	sign-extends
lw	Load Word	I	0000011	010		$R[rd] = M[R[rs1] + SE(imm)]$	
sw	Store Word	S	0100011	010		$M[R[rs1] + SE(imm)] = R[rs2]$	
beq	Branch ==	SB	1100011	000		if(rs1 == rs2) PC += SE(imm) << 1	
bne	Branch !=	SB	1100011	001		if(rs1 != rs2) PC += SE(imm) << 1	
blt	Branch <	SB	1100011	100		if(rs1 < rs2) PC += SE(imm) << 1	
bge	Branch >=	SB	1100011	101		if(rs1 >= rs2) PC += SE(imm) << 1	
jal	Jump And Link	UJ	1101111			$R[rd] = PC + 4;$ PC += SE(imm) << 1	
jalr	Jump And Link Reg	I	1100111	000		$R[rd] = PC + 4;$ PC = $R[rs1] + SE(imm)$	
lui	Load Upper Imm	U	0110111			$R[rd] = SE(imm) \ll 12$	
auipc	Add Upper Imm to PC	U	0010111			$R[rd] = PC + (SE(imm) \ll 12)$	
csrrw	CSR read & write	I	1110011	001		$R[rd] = CSRs[csr];$ $CSRs[csr] = R[rs1]$	
csrrs	CSR read & set	I	1110011	010		$R[rd] = CSRs[csr];$ $CSRs[csr] = CSRs[csr] \vee R[rs1]$	
csrrc	CSR read & clear	I	1110011	011		$R[rd] = CSRs[csr];$ $CSRs[csr] = CSRs[csr] \& \sim R[rs1]$	
ecall	Environment Call	I	1110011	000	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	000	imm=0x1	Transfer control to debugger	

R = Register file access CSR = Coprocessor Register
SE = Sign extend (e.g., scause, sepc)

Core Instruction Formats

31	25	24	20	19	15	14	12	11	7	6	0	
funct7		rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]				rs1		funct3		rd		opcode		I-type
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[11 9:4]		rs2		rs1		funct3		imm[3:0 10]		opcode		SB-type
imm[19:0]								rd		opcode		U-type
imm[19 9:0 10 18:11]								rd		opcode		UJ-type

Opcodes, Base conversion

Binary	Hex	Opcode	Binary	Hex	Opcode	Binary	Hex	Opcode	Binary	Hex	Opcode
000 0000	00	lw	010 0000	20	sw	100 0000	40		110 0000	60	SB-type
000 0001	01		010 0001	21		100 0001	41		110 0001	61	
000 0010	02		010 0010	22		100 0010	42		110 0010	62	
000 0011	03		010 0011	23		100 0011	43		110 0011	63	
000 0100	04		010 0100	24		100 0100	44		110 0100	64	
000 0101	05		010 0101	25		100 0101	45		110 0101	65	
000 0110	06		010 0110	26		100 0110	46		110 0110	66	
000 0111	07		010 0111	27		100 0111	47		110 0111	67	
000 1000	08		010 1000	28		100 1000	48		110 1000	68	
000 1001	09		010 1001	29		100 1001	49		110 1001	69	
000 1010	0A		010 1010	2A		100 1010	4A		110 1010	6A	
000 1011	0B		010 1011	2B		100 1011	4B		110 1011	6B	
000 1100	0C		010 1100	2C		100 1100	4C		110 1100	6C	
000 1101	0D		010 1101	2D		100 1101	4D		110 1101	6D	
000 1110	0E		010 1110	2E		100 1110	4E		110 1110	6E	
000 1111	0F		010 1111	2F		100 1111	4F		110 1111	6F	
001 0000	10	I-type	011 0000	30	R-type	101 0000	50		111 0000	70	exceptions
001 0001	11		011 0001	31		101 0001	51		111 0001	71	
001 0010	12		011 0010	32		101 0010	52		111 0010	72	
001 0011	13		011 0011	33		101 0011	53		111 0011	73	
001 0100	14		011 0100	34		101 0100	54		111 0100	74	
001 0101	15		011 0101	35		101 0101	55		111 0101	75	
001 0110	16		011 0110	36		101 0110	56		111 0110	76	
001 0111	17		011 0111	37		101 0111	57		111 0111	77	
001 1000	18		011 1000	38		101 1000	58		111 1000	78	
001 1001	19		011 1001	39		101 1001	59		111 1001	79	
001 1010	1A		011 1010	3A		101 1010	5A		111 1010	7A	
001 1011	1B		011 1011	3B		101 1011	5B		111 1011	7B	
001 1100	1C		011 1100	3C		101 1100	5C		111 1100	7C	
001 1101	1D		011 1101	3D		101 1101	5D		111 1101	7D	
001 1110	1E		011 1110	3E		101 1110	5E		111 1110	7E	
001 1111	1F		011 1111	3F		101 1111	5F				

Registers

Register	Name	Description	Saver
x0	zero	Zero constant	—
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 / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x30	t3-t5	Temporaries	Caller
x31	at	Assembler Temporary	Caller

Memory Allocation

