

6.004 Fall 2020 Tutorial Problems L02 – RISC-V Assembly

Computational Instructions

R-type: Register-register instructions: opcode = OP = 0110011

Arithmetic	Comparisons	Logical	Shifts
ADD, SUB	SLT, SLTU	AND, OR, XOR	SLL, SRL, SRA

Assembly

instr: `oper rd, rs1, rs2`
Behavior: `reg[rd] <= reg[rs1] oper reg[rs2]`

SLT – Set less than
 SLTU – Set less than unsigned
 SLL – Shift left logical
 SRL – Shift right logical
 SRA – Shift right arithmetic

I-type: Register-immediate instructions: with opcode = OP-IMM = 0010011

Arithmetic	Comparisons	Logical	Shifts
ADDI	SLTI, SLTIU	ANDI, ORI, XORI	SLLI, SRLI, SRAI

Assembly instr: `oper rd, rs1, immI`
Behavior: `imm = signExtend(immI)`
`reg[rd] <= reg[rs1] oper imm`

Same functions as R-type except SUBI is not needed.
 Function is encoded in funct3 bits plus instr[30]. Instr[30] = 1 for SRAI. So SRLI and SRAI use same funct3 encoding.
 immI is a 12 bit constant.

U-type: opcode = LUI or AUIPC = (01|00)10111

LUI – load upper immediate
 AUIPC – add upper immediate to PC

Assembly instr: `lui rd, immU`

Behavior: `imm = {immU,12'b0}`
`Reg[rd] <= imm`

For example `lui x2, 2` would load register x2 with 0x2000.
immU is a 20 bit constant.

Load Store Instructions

I-type: Load: with opcode = LOAD = 0000011

LW – load word

Assembly instr: `lw rd, immI(rs1)`

Behavior: `imm = signExtend(immI)`
`Reg[rd] <= Mem[R[rs1] + imm]`

S-type: Store: opcode = STORE = 0100011

SW – store word

Assembly instr: `sw rs2, immS(rs1)`

Behavior: `imm = signExtend(immS)`
`Mem[R[rs1] + imm] <= R[rs2]`

immS is a 12 bit constant.

Control Instructions

B-type: Conditional Branches: opcode = 1100011

Assembly instr: `oper rs1, rs2, label`

Behavior: `imm = distance to label in bytes =`
`signExtend({immB[12:1],0})`
`pc <= (R[rs1] comp R[rs2]) ? pc + imm : pc + 4`

Compares register rs1 to rs2. If comparison is true then pc is updated with pc + imm, otherwise pc becomes pc + 4. Comparison type is defined by operation.

BEQ – branch if equal (==)

BNE – branch if not equal (!=)

BLT – branch if less than (<)

BGE – branch if greater than or equal (>=)

BLTU – branch if less than using unsigned numbers (< unsigned)

BGEU – branch if greater than or equal using unsigned numbers (>= unsigned)

J-type: Unconditional Jumps: opcode = JAL = 1101111

Assembly instr: JAL rd, label

Behavior: imm = distance to label in bytes =
signExtend({immJ{20:1},0})
pc[rd] <= pc + 4; pc <= pc + imm

I-type: Unconditional Jump: opcode = JALR = 1100111

Assembly instr: JALR rd, rs1, immI

Behavior: imm = signExtend(immI)
pc[rd] <= pc + 4; pc <= (R[rs1]+imm) & ~0x00000001
(zero out the bottom bit of pc)

JAL – jump and link

JALR – jump and link register

immJ is a 20 bit constant (used by JAL)

immI is a 12 bit constant (used by JALR)

Common pseudoinstructions:

j label = jal x0, label (ignore return address)

li x1, 0x1000 = lui x1, 1

li x1, 0x1100 = lui x1, 1; addi x1, x1, 0x100

li x4, 3 = addi x4, x0, 3

mv x3, x2 = addi x3, x2, 0

beqz x1, target = beq x1, x0, target

bneqz x1, target = bneq x1, x0, target

Note: A small subset of essential problems are marked with a red star (*). We especially encourage you to try these out before recitation.

Problem 1.

Compile the following expressions to RISC-V assembly. Assume a is stored at address 0x1000, b is stored at 0x1004, and c is stored at 0x1008. Assume that all values are 32-bit signed integers.

1. `a = b + 3*c;`

With a, b, and c being stored at addresses 0x1000, 0x1004, and 0x1008, each of these solutions are loosely structured in the following way:

- 1) Load a,b,c with LW
- 2) Perform operation
- 3) Store result with SW

Note that we do not have a multiplication instruction. We compute $3c$ with $c \ll 1 + c$. A left bit-shift by 1 (`slli`) is equivalent to multiplication by 2. Additionally, when loading, we use the offset field of the LW instruction to read the correct address. $8(x1) = 0x1000+8 = 0x1008$, $4(x1) = 0x1004$

```
// 1. Load values a,b,c
li x1, 0x1000 // actually lui x1, 1
lw x2, 8(x1) // x2 = c, use offset to get 0x1008
lw x3, 4(x1) // x3 = b, use offset to get 0x1004
// 2. Calculate a = b + 3c
slli x4, x2, 1 // x4 = x2 << 1 = 2c
add x4, x4, x2 // x4 = 2c + c = 3c
add x4, x4, x3 // x4 = 3c + b
// 3. Store value into a
sw x4, 0(x1) // store x4 into a
```

2. `if (a > b) { c = 17; }`

We use branching to implement the IF statement, where the load for `c=17` is skipped if the condition `a > b` is not satisfied.

```
li x1, 0x1000 // actually lui x1, 1
lw x2, 0(x1) // x2 = a
lw x3, 4(x1) // x3 = b
// branch to end if a <=b (or b >=a)
bge x3, x2, end
li x4, 17 // actually just addi x4, x0, 17
sw x4, 8(x1) // c = 17
end:
```

```
3. sum = 0;
   for (i = 0; i < 10; i = i+1) { sum += i; }
```

Registers:

- x1: sum – cumulative sum
- x2: i – index
- x3: 10 – condition for FOR loop ($i < 10$).

We loop by checking for the condition ($i < 10$), and branching to the loop body beginning while the condition is met. There are no branch instructions that take an immediate, so we need to first store value 10 into a register, and then do a branch instruction comparing to the register.

```
addi x1, x0, 0    // x1 = 0 (sum)
addi x2, x0, 0    // x2 = 0 (i)
addi x3, x0, 10   // x3 = 10
loop:
add x1, x1, x2    // x1 = x1 + x2 or sum = sum + i
addi x2, x2, 1    // i = i+1
// if i < 10, branch to beginning of loop body
blt x2, x3, loop
```

Problem 2.

Compile the following expression assuming that a is stored at address 0x1100, and b is stored at 0x1200, and c is stored at 0x2000. Assume a, b, and c are arrays whose elements are stored in consecutive memory locations. Assume that all values are 32-bit signed integers.

```
for (i = 0; i < 10; i = i+1) { c[i] = a[i] + b[i]; }
```

Registers:

- x1: address of a[0]
- x2: address of c[0]
- x3: i – index
- x4: 4i – because of the length of a word, we multiply the i by 4 to get the right offset
 - RISC-V memory is indexed by byte and each word is four bytes long
- x5: address of a[i]
- x6: address of c[i]
- x7: 1) value of a[i], 2) a[i] + b[i]
- x8: value of b[i]
- x9: 10 – condition for FOR loop (i < 10)

The loop is implemented identically to above in Problem 1-3. We must first obtain the address given index i, which is 0x1100 + 4i for a[i], 0x1200 + 4i for b[i], and 0x2000 for c[i]

```
li x1, 0x1100    // x1 = address of a[0] (lui x1, 1; addi x1,
                // x1, 0x100)
li x2, 0x2000    // x2 = address of c[0] (lui x2, 2)
li x3, 0         // x3 = 0 (i)           (addi x3, x0, 0)
li x9, 10
loop:
sll x4, x3, 2    // x4 = 4 * i
add x5, x1, x4   // x5 = address of a[i]
add x6, x2, x4   // x6 = address of c[i]
lw x7, 0(x5)    // x7 = a[i]
lw x8, 0x100(x5) // x8 = b[i]; b is offset from a by 0x100
add x7, x7, x8   // x7 = a[i] + b[i]
sw x7, 0(x6)    // c[i] = a[i] + b[i]
addi x3, x3, 1   // i = i + 1
blt x3, x9, loop // branch back to loop if i < 10
```

Problem 3.

Hand assemble the following sequence of instructions into its equivalent binary encoding. (Hint: use the ISA Reference Card at the end of this worksheet to parse and encode the instruction)

```
addi x1, x1, -1
```

addi x1, x1, -1

-1 encoded as 12 bits is 0xfff

x1 in 5 bits is 0b00001

func3 for addi = 000

op = 0010011 (since addi is a register-immediate instruction)

addi: imm[11:0],rs1,func3,rd,op = 0xff08093 =

0b1111111111_00001_000_00001_0010011

Problem 4.

A) Assume that the registers are initialized to: $x1=8$, $x2=10$, $x3=12$, $x4=0x1234$, $x5=24$ before execution of each of the following assembly instructions. For each instruction, provide the value of the specified register or memory location. **If your answers are in hexadecimal, make sure to prepend them with the prefix 0x.**

1. SLL $x6$, $x4$, $x5$ Value of $x6$: 0x34000000

We shift left $0x1234$ ($x4$) by 24 ($x5$) into $x6$:

$0x1234 \ll 24 = 0x1234000000$

However, since we are working in 32bits, we truncate correspondingly to get: $0x34000000$

2. ADD $x7$, $x3$, $x2$ Value of $x7$: 22

We add 12 ($x3$) by $x2$ (10) into $x7$: $12 + 10 = 22$

3. ADDI $x8$, $x1$, 2 Value of $x8$: 10

We add 8 ($x1$) by constant 2 into $x8$: $8 + 2 = 10$

4. SW $x2$, 4($x4$) Value stored: 10 at address: 0x1238

$x2$ is the value we are writing into the address at $x4 + 4$

$x2 = 10$ (value stored)

$x4 + 4 = 0x1234 + 4 = 0x1238$

B) Assume X is at address $0x1CE8$

```
li x1, 0x1CE8
lw x4, 0(x1)
blt x4, x0, L1
addi x2, x0, 17
beq x0, x0, L2
L1: srai x2, x4, 4
L2:
```

Value left in $x4$? 0x_87654321

Value left in $x2$? 0x_F8765432

X: .word $0x87654321$

Line by line decomposition:

1. $x1 = 0x1CE8$ – load value $0x1CE8$ into $x1$
2. $x4 = 0x87654321$ – load word at address $x1 + 0 = 0x1CE8$ into $x4$
3. Branch into L1 – if ($0x87654321 < 0$), then jump to L1
4. $x2 = 0xF876432$ – $0x87654321 \gg 4$ into $x2$ (right shift arithmetic)

Problem 5.

Compile the following Fibonacci implementation to RISC-V assembly.

```
# Reference Fibonacci implementation in Python
def fibonacci_iterative(n):
    if n == 0:
        return 0
    n = n - 1
    x, y = 0, 1
    while n > 0:
        # Parallel assignment of x and y
        # The new values for x and y are computed at the same time, and
        # then the values of x and y are updated afterwards
        x, y = y, x + y
        n = n - 1
    return y
```

Registers:

- x1: n
- x2: y (final result)
- x3: x
- x5: x + y

```
// x1 = n
// x2 = final result
bne x1, x0, start // branch if n!=0
li x2, 0
j end // pseudo instruction for jal x0, end
start:
addi x1, x1, -1 // n = n - 1
li x3, 0 // x = 0
li x2, 1 // y = 1 (you're returning y at the end, so use
// x2 to hold y)
loop:
bge x0, x1, end // stop loop if 0 >= n
addi x5, x3, x2 // tmp = x + y
mv x3, x2 // x = y (pseudo instruction for addi x3, x2, 0)
mv x2, x5 // y = tmp (pseudo instruction for addi x2, x5, 0)
addi x1, x1, -1 // n = n - 1
j loop // pseudo instruction for jal x0, loop
end:
```

MIT 6.004 ISA Reference Card: Instructions

Instruction	Syntax	Description	Execution
LUI	lui rd, luiConstant	Load Upper Immediate	reg[rd] <= luiConstant « 12
JAL	jal rd, label	Jump and Link	reg[rd] <= pc + 4 pc <= label
JALR	jalr rd, offset(rs1)	Jump and Link Register	reg[rd] <= pc + 4 pc <= {(reg[rs1] + offset)[31:1], 1'b0}
BEQ	beq rs1, rs2, label	Branch if =	pc <= (reg[rs1] == reg[rs2]) ? label : pc + 4
BNE	bne rs1, rs2, label	Branch if ≠	pc <= (reg[rs1] != reg[rs2]) ? label : pc + 4
BLT	blt rs1, rs2, label	Branch if < (Signed)	pc <= (reg[rs1] < _s reg[rs2]) ? label : pc + 4
BGE	bge rs1, rs2, label	Branch if ≥ (Signed)	pc <= (reg[rs1] >= _s reg[rs2]) ? label : pc + 4
BLTU	bltu rs1, rs2, label	Branch if < (Unsigned)	pc <= (reg[rs1] < _u reg[rs2]) ? label : pc + 4
BGEU	bgeu rs1, rs2, label	Branch if ≥ (Unsigned)	pc <= (reg[rs1] >= _u reg[rs2]) ? label : pc + 4
LW	lw rd, offset(rs1)	Load Word	reg[rd] <= mem[reg[rs1] + offset]
SW	sw rs2, offset(rs1)	Store Word	mem[reg[rs1] + offset] <= reg[rs2]
ADDI	addi rd, rs1, constant	Add Immediate	reg[rd] <= reg[rs1] + constant
SLTI	slti rd, rs1, constant	Compare < Immediate (Signed)	reg[rd] <= (reg[rs1] < _s constant) ? 1 : 0
SLTIU	sltiu rd, rs1, constant	Compare < Immediate (Unsigned)	reg[rd] <= (reg[rs1] < _u constant) ? 1 : 0
XORI	xori rd, rs1, constant	Xor Immediate	reg[rd] <= reg[rs1] ^ constant
ORI	ori rd, rs1, constant	Or Immediate	reg[rd] <= reg[rs1] constant
ANDI	andi rd, rs1, constant	And Immediate	reg[rd] <= reg[rs1] & constant
SLLI	slli rd, rs1, constant	Shift Left Logical Immediate	reg[rd] <= reg[rs1] « constant
SRLI	srl rd, rs1, constant	Shift Right Logical Immediate	reg[rd] <= reg[rs1] » _u constant
SRAI	srai rd, rs1, constant	Shift Right Arithmetic Immediate	reg[rd] <= reg[rs1] » _s constant
ADD	add rd, rs1, rs2	Add	reg[rd] <= reg[rs1] + reg[rs2]
SUB	sub rd, rs1, rs2	Subtract	reg[rd] <= reg[rs1] - reg[rs2]
SLL	sll rd, rs1, rs2	Shift Left Logical	reg[rd] <= reg[rs1] « reg[rs2]
SLT	slt rd, rs1, rs2	Compare < (Signed)	reg[rd] <= (reg[rs1] < _s reg[rs2]) ? 1 : 0
SLTU	sltu rd, rs1, rs2	Compare < (Unsigned)	reg[rd] <= (reg[rs1] < _u reg[rs2]) ? 1 : 0
XOR	xor rd, rs1, rs2	Xor	reg[rd] <= reg[rs1] ^ reg[rs2]
SRL	srl rd, rs1, rs2	Shift Right Logical	reg[rd] <= reg[rs1] » _u reg[rs2]
SRA	sra rd, rs1, rs2	Shift Right Arithmetic	reg[rd] <= reg[rs1] » _s reg[rs2]
OR	or rd, rs1, rs2	Or	reg[rd] <= reg[rs1] reg[rs2]
AND	and rd, rs1, rs2	And	reg[rd] <= reg[rs1] & reg[rs2]

Note: *luiConstant* is a 20-bit value. *offset* and *constant* are signed 12-bit values that are sign-extended to 32-bit values. *label* is a 32-bit memory address or its alias name.

MIT 6.004 ISA Reference Card: Pseudoinstructions

Pseudoinstruction	Description	Execution
li rd, constant	Load Immediate	reg[rd] <= constant
mv rd, rs1	Move	reg[rd] <= reg[rs1] + 0
not rd, rs1	Logical Not	reg[rd] <= reg[rs1] ^ -1
neg rd, rs1	Arithmetic Negation	reg[rd] <= 0 - reg[rs1]
j label	Jump	pc <= label
jal label	Jump and Link (with ra)	reg[ra] <= pc + 4 pc <= label
call label		
jr rs	Jump Register	pc <= reg[rs1] & ~1
jalr rs	Jump and Link Register (with ra)	reg[ra] <= pc + 4 pc <= reg[rs1] & ~1
ret	Return from Subroutine	pc <= reg[ra]
bgt rs1, rs2, label	Branch > (Signed)	pc <= (reg[rs1] > _s reg[rs2]) ? label : pc + 4
ble rs1, rs2, label	Branch ≤ (Signed)	pc <= (reg[rs1] <= _s reg[rs2]) ? label : pc + 4
bgtu rs1, rs2, label	Branch > (Unsigned)	pc <= (reg[rs1] > _u reg[rs2]) ? label : pc + 4
bleu rs1, rs2, label	Branch ≤ (Unsigned)	pc <= (reg[rs1] <= _u reg[rs2]) ? label : pc + 4
beqz rs1, label	Branch = 0	pc <= (reg[rs1] == 0) ? label : pc + 4
bnez rs1, label	Branch ≠ 0	pc <= (reg[rs1] != 0) ? label : pc + 4
bltz rs1, label	Branch < 0 (Signed)	pc <= (reg[rs1] < _s 0) ? label : pc + 4
bgez rs1, label	Branch ≥ 0 (Signed)	pc <= (reg[rs1] >= _s 0) ? label : pc + 4
bgtz rs1, label	Branch > 0 (Signed)	pc <= (reg[rs1] > _s 0) ? label : pc + 4
blez rs1, label	Branch ≤ 0 (Signed)	pc <= (reg[rs1] <= _s 0) ? label : pc + 4

MIT 6.004 ISA Reference Card: Calling Convention

Registers	Symbolic names	Description	Saver
x0	zero	Hardwired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporary registers	Caller
x8-x9	s0-s1	Saved registers	Callee
x10-x11	a0-a1	Function arguments and return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporary registers	Caller

MIT 6.004 ISA Reference Card: Instruction Encodings

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[12 10:5]			rs2	rs1	funct3		imm[4:1 11]		opcode			B-type
imm[31:12]								rd	opcode			U-type
imm[20 10:1 11 19:12]								rd	opcode			J-type

RV32I Base Instruction Set (MIT 6.004 subset)

imm[31:12]								rd	0110111			LUI
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	010		rd	0000011			LW	
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	

- For JAL and branch instructions (BEQ, BNE, BLT, BGE, BLTU, BGEU), the immediate encodes the target address as an offset from the current pc (i.e., $pc + imm = label$).
- Not all immediate bits are encoded. Missing lower bits are filled with zeros and missing upper bits are sign-extended.