

6.004 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

SB-type: Conditional Branches: opcode = 1100011

Assembly instr: `oper rs1, rs2, label`

Behavior: `imm = distance to label in bytes = {immS[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)

UJ-type: Unconditional Jump: opcode = JAL = 1101111

Assembly instr: **JAL rd, label**

Behavior: **imm = distance to label in bytes = {immU{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) & ~0x01
(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

MIT 6.004 ISA Reference Card: Instructions

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

MIT 6.004 ISA Reference Card: Pseudoinstructions

Pseudoinstruction	Description	Execution
li <i>rd</i> , constant	Load Immediate	reg[rd] <= constant
mv <i>rd</i> , <i>rs1</i>	Move	reg[rd] <= reg[rs1] + 0
not <i>rd</i> , <i>rs1</i>	Logical Not	reg[rd] <= reg[rs1] ^ -1
neg <i>rd</i> , <i>rs1</i>	Arithmetic Negation	reg[rd] <= 0 - reg[rs1]
j label	Jump	pc <= label
jal label	Jump and Link (with <i>ra</i>)	reg[ra] <= pc + 4 pc <= label
jr <i>rs</i>	Jump Register	pc <= reg[rs1] & ~1
jalr <i>rs</i>	Jump and Link Register (with <i>ra</i>)	reg[ra] <= pc + 4 pc <= reg[rs1] & ~1
ret	Return from Subroutine	pc <= reg[ra]
bgt <i>rs1</i> , <i>rs2</i> , label	Branch > (Signed)	pc <= (reg[rs1] > _s reg[rs2]) ? label : pc + 4
bge <i>rs1</i> , <i>rs2</i> , label	Branch ≥ (Signed)	pc <= (reg[rs1] >= _s reg[rs2]) ? label : pc + 4
bgtu <i>rs1</i> , <i>rs2</i> , label	Branch > (Unsigned)	pc <= (reg[rs1] > _u reg[rs2]) ? label : pc + 4
bgeu <i>rs1</i> , <i>rs2</i> , label	Branch ≥ (Unsigned)	pc <= (reg[rs1] >= _u reg[rs2]) ? label : pc + 4
beqz <i>rs1</i> , label	Branch = 0	pc <= (reg[rs1] == 0) ? label : pc + 4
bnez <i>rs1</i> , label	Branch ≠ 0	pc <= (reg[rs1] != 0) ? label : pc + 4
bltz <i>rs1</i> , label	Branch < 0 (Signed)	pc <= (reg[rs1] < _s 0) ? label : pc + 4
bgez <i>rs1</i> , label	Branch ≥ 0 (Signed)	pc <= (reg[rs1] >= _s 0) ? label : pc + 4
bgtz <i>rs1</i> , label	Branch > 0 (Signed)	pc <= (reg[rs1] > _s 0) ? label : pc + 4
blez <i>rs1</i> , 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][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		JALR
imm[12:10:5]			rs2		000		BEQ
imm[12:10:5]			rs1		001		BNE
imm[12:10:5]			rs2		100		BLT
imm[12:10:5]			rs2		101		BGE
imm[12:10:5]			rs2		110		BLTU
imm[12:10:5]			rs2		111		BGEU
imm[11:0]			rs1		010		LW
imm[11:5]			rs1		010		SW
imm[11:0]			rs1		000		ADDI
imm[11:0]			rs1		010		SLTI
imm[11:0]			rs1		011		SLTIU
imm[11:0]			rs1		100		XORI
imm[11:0]			rs1		110		ORI
imm[11:0]			rs1		111		ANDI
0000000			shamt		001		SLLI
0000000			shamt		101		SRLI
0100000			shamt		101		SRAI
0000000			rs2		000		ADD
0100000			rs2		000		SUB
0000000			rs2		001		SLL
0000000			rs2		010		SLT
0000000			rs2		011		SLTU
0000000			rs2		100		XOR
0000000			rs2		101		SRL
0100000			rs2		101		SRA
0000000			rs2		110		OR
0000000			rs2		111		AND

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.

1. $a = b + 3c$; ★

2. if ($a > b$) $c = 17$; ★

3. $sum = 0$;
for ($i = 0$; $i < 10$; $i = i + 1$) $sum += i$;

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.

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

Problem 3.

Hand assemble the following sequence of instructions into its equivalent binary encoding.

loop:

addi x1, x1, -1 ★

bnez x1, loop

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:** _____ ★
2. ADD x7, x3, x2 **Value of x7:** _____
3. ADDI x8, x1, 2 **Value of x8:** _____
4. SW x2, 4(x4) **Value stored:** _____ **at address:** _____ ★

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? _____

X: .word 0x87654321

Value left in x2? _____

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 -= 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 -= 1  
    return y
```