6.004 Tutorial Problems
L02 – RISC-V Assembly

Computational Instructions

操作码. operating code

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

word – 4bytes – 32bits

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 bye 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 rd, immU`	Load Upper Immediate	`reg[rd] <= immU << 12`
JAL	`jal rd, immJ`	Jump and Link	`reg[rd] <= pc + 4` `pc <= pc + immJ`
JALR	`jalr rd, rs1, immI`	Jump and Link Register	`reg[rd] <= pc + 4` `pc <= {(reg[rs1] + immI)[31:1], 1'b0}`
BEQ	`beq rs1, rs2, immB`	Branch if $=$	`pc <= (reg[rs1] == reg[rs2]) ? pc + immB` `: pc + 4`
BNE	`bne rs1, rs2, immB`	Branch if $\neq$	`pc <= (reg[rs1] != reg[rs2]) ? pc + immB` `: pc + 4`
BLT	`blt rs1, rs2, immB`	Branch if $<$ (Signed)	pc <= (reg[rs1] <$_s$ reg[rs2]) ? pc + immB : pc + 4
BGE	`bge rs1, rs2, immB`	Branch if $\geq$ (Signed)	pc <= (reg[rs1] >=$_s$ reg[rs2]) ? pc + immB : pc + 4
BLTU	`bltu rs1, rs2, immB`	Branch if $<$ (Unsigned)	pc <= (reg[rs1] <$_u$ reg[rs2]) ? pc + immB : pc + 4
BGEU	`bgeu rs1, rs2, immB`	Branch if $\geq$ (Unsigned)	pc <= (reg[rs1] >=$_u$ reg[rs2]) ? pc + immB : pc + 4
LW	`lw rd, immI(rs1)`	Load Word	`reg[rd] <= mem[reg[rs1] + immI]`
SW	`sw rs2, immS(rs1)`	Store Word	`mem[reg[rs1] + immS] <= reg[rs2]`
ADDI	`addi rd, rs1, immI`	Add Immediate	`reg[rd] <= reg[rs1] + immI`
SLTI	`slti rd, rs1, immI`	Compare $<$ Immediate (Signed)	reg[rd] <= (reg[rs1] <$_s$ immI) ? 1 : 0
SLTIU	`sltiu rd, rs1, immI`	Compare $<$ Immediate (Unsigned)	reg[rd] <= (reg[rs1] <$_u$ immI) ? 1 : 0
XORI	`xori rd, rs1, immI`	Xor Immediate	`reg[rd] <= reg[rs1] ^ immI`
ORI	`ori rd, rs1, immI`	Or Immediate	reg[rd] <= reg[rs1] | immI
ANDI	`andi rd, rs1, immI`	And Immediate	`reg[rd] <= reg[rs1] & immI`
SLLI	`slli rd, rs1, immI`	Shift Left Logical Immediate	`reg[rd] <= reg[rs1] << immI`
SRLI	`srli rd, rs1, immI`	Shift Right Logical Immediate	reg[rd] <= reg[rs1] >>$_u$ immI
SRAI	`srai rd, rs1, immI`	Shift Right Arithmetic Immediate	reg[rd] <= reg[rs1] >>$_s$ immI
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: All immediate values (*immU*, *immJ*, *immI*, *immB*, and *immS*) are sign-extended to 32-bits.

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` `call label`	Jump and Link (with ra)	`reg[ra] <= pc + 4` `pc <= 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 $\leq$ (Signed)	pc <= (reg[rs1] <=$_s$ reg[rs2]) ? label : pc + 4
`bgtu rs1, rs2, label`	Branch $>$ (Unsigned)	pc <= (reg[rs1] >$_s$ reg[rs2]) ? label : pc + 4
`bleu rs1, rs2, label`	Branch $\leq$ (Unsigned)	pc <= (reg[rs1] <=$_s$ reg[rs2]) ? label : pc + 4
`beqz rs1, label`	Branch $= 0$	`pc <= (reg[rs1] == 0) ? label : pc + 4`
`bnez rs1, label`	Branch $\neq 0$	`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

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 RISCV assembly. Assume a is stored at address 0x1000, b is stored at 0x1004, and c is stored at 0x1008.

Load value abc.

1. a = b + 3c; ★

a=0x1000
b=0x1004
c=0x1008

```
lui x1, 1 // 将0x1000加载到x1
lw x2, 8(x1) // x2=c
lw x3, 4(x1) // x3=b
slli x4, x2, 1 // x4=2c 计算
add x4, x4, x2 // x4=2c+c=3c
add x4, x4, x3 // x4=3c+b
add x4, x4, x1 // a=b+3c
```

3. Store

```
sw x4 0(x1) // 将x4存到a.
```

2. if (a > b) c = 17; ★

```
li x1, ~~0x1000~~ // a ×
lw x2, 0(x1) // b
lw x3, 0(x1) // ⊙值不确定.
bge, x2, x1
```

```
li x1, 0x1000
lw x2, 0(x1) // a
lw x3, 4(x1) // b
lw x4
bge x3, x2, end
li x4, 17 // c=17
sw x4, 8(x1) // c存
```

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

```
li x1, 0 // x1=sum
li x2, 0 // x2=i
li x3, 10 // x3=10
loop: addi x3, x3(x1) // sum+=i
addi x2, 1(x2) // i=i+1
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.

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

a 0x1100
b 0x1200
c 0x2000

lw x1, 0x1100, // base a.
lw x2 0x1200. // base b
lw x3 0x2000 // base c
lw x4 0 // i=0.
lw x5 10 // x5=10

loop:

add: x3 x1, x2.
sw: x3 0
add: x1, x1, 4
add: x2, x2, 4
lw, x6 : x3
add: x3: x2: 4 有bug. C也加
~~addi: x4 1~~
sw x6 0(x6)
blt x3, x5, loop.

Problem 3.

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

```
loop:
addi x1, x1, -1
bnez x1, loop
```

1 2 4 8 16
0 0 0 0 1
1 0 0 0 0

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:** 0X 12340000000
   32bit 有overflow
   0x34000000
2. ADD x7, x3, x2 **Value of x7:** 22
3. ADDI x8, x1, 2 **Value of x8:** 10
4. SW x2, 4(x4) **Value stored:** 10 **at address:** 0x1238
   store
   x2存到 4(x4)中

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:

X:  .word 0x87654321
```

x1= 0x1CE8
x4=0x1CE8
x4<x0
x2=17

load word x into x1
x: word 0x87654321

0x87654321

Value left in x4? 0x1CE8

Value left in x2? 0x1CE8

Problem 5.

Compile the following Fibonacci implementation to RISCV 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
```