6.004 Spring 2019 Tutorial Problems
L02 – Model of Computing 2

Two's Complement Representation:

Problem 1.

1. What is the 6-bit two's complement representation of the decimal number -21?

2. What is the hexadecimal representation for decimal -51 encoded as an 8-bit two's complement number?

3. The hexadecimal representation for an 8-bit two's complement number is 0xD6. What is its decimal representation?

4. Using a 5-bit two's complement representation, what is the range of integers that can be represented with a single 5-bit quantity?

5. Can the value of the sum of two 2's complement numbers 0xB3 + 0x47 be represented using an 8-bit 2's complement representation? If so, what is the sum in hex? If not, write NO.

6. Can the value of the sum of two 2's complement numbers 0xB3 + 0xB1 be represented using an 8-bit 2's complement representation? If so, what is the sum in hex? If not, write NO.

7. Please compute the value of the expression 0xBB – 8 using 8-bit two's complement arithmetic and give the result in decimal (base 10).

8. Consider the following subtraction problem where the operands are 5-bit two's complement numbers. Compute the result and give the answer as a decimal (base 10) number.

$$\begin{array}{r} 10101 \\ -\underline{00011} \end{array}$$

Problem 2.

1. Given an unsigned N-bit binary integer $= \boldsymbol{b_{n-1} \ldots b_1 b_0}$, prove that $\boldsymbol{v}$ is a multiple of 4 if and only if $\boldsymbol{b_0 = 0}$ and $\boldsymbol{b_1 = 0}$.

1. Does the same relation hold for two's complement encoding?

Assembly Language:

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 $\neq$	`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 $\geq$ (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 $\geq$ (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	srli *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]`

(*) *luiConstant* is a 20-bit value. *offset* and *constant* are signed 12-bit values that are sign-extended to 32-bit values.

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`
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`

Problem 1.

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 2.

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:

X:  .word 0x87654321
```

Value left in x4? _____

Value left in x2? _____

Problem 3.

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
```