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?

   21 = 16+4+1 = 0b010101, (using 6 bit binary)
   -21 = 0b101010+1 = **0b101011**

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

   51 = 32+16+2+1 = 0b0011_0011, (using 8-bit binary)
   -51 = 0b1100_1101 = **0xCD**

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

   0xD6 = 0b1101_0110 = -128+64+16+4+2 = **-42**

   Alternative (*may* be easier):
   0xD6 = 0b1101_0110 which is negative, so use -0xD6 = 0b0010_1010 = 42, which gives +0xD6 = **-42**

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?

   -2^4 to (2^4)-1
   -16 to 15

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.

   Yes: negative + positive is always within range.
   0xB3+0x47 =
   0b1011_0011 +
   0b0100_0111 =
   0b1111_1010 = **0xFA** (in decimal: -6, can you see why it's a very small negative?)

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.

No: negative + negative gave us positive, not okay.
0xB3+0xB1 =
0b1011_0011 +
0b1011_0001 =
0b0110_0100 :((("9th bit" dropped: we're in 8-bit notation)

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).

0xBB - 8 = 0b1011_1011 + -0b0000_1000 =
0b1011_1011 +
0b1111_1000 =
0b1011_0011 =
-77 (negative, so positie is 0b0100_1101 = 77) =

This is okay: negative - positive is always okay. (Same as positive-negative)

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.

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

```
  0b1_0101       0b1_0101          0b1_0101
- 0b0_0011 => + 0b1_1100 + 1 => + 0b1_1101
                                 = 0b1_0010 = -(0b0_1101 + 1) = -14
```

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 $\boldsymbol{b_1 = 0}$.

   - Powers of 2 greater than or equal to 4 are multiples of 4 (for all n>=2, 2^n = 4*2^n and 2^n is an integer)
   - The sum of numbers divisible by 4 is divisible by 4.
     Therefore any number of the form $b_{n-1} \ldots 00$ is a multiple of 4

   If a number ends in one of 01, 10, or 11 we are adding 1, 2, or 3 respectively to a multiple of 4. Therefore it a number is a multiple 4 only if it ends in 00.

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

   Yes. The above proof is unmodified: the highest order bit is now -2^n instead of +2^n.

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 ≠	`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	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 ≤ (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 ≤ (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 ≠ 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];

```
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]
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 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:** __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_____

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? 0x87654321_____

Value left in x2? 0xF8765432_____

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
```

```
// x1 = n
// x2 = final result
bne x1, x0, start
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
end:
```