5.5 Tutorial 5

5.5.1 Exercise 1

1. addi x5, x7, -5
   add x5, x6, x5

2. f = g + h + i;
3. sub x30, x28, x29 //compute i−j
   slli x30, x30, 2 // multiply by 4 to convert the word offset to a byte offset
   add x30, x10, x30 // compute the&A[i-j] lw x30, 0(x30) // load A[i-j]
   sw x30, 32(x11)

4. slli x30, x5, 3 // x30 = f*8
   add x30, x10, x30 // x30 = &A[f×2]
   slli x31, x6, 3 // x31 = g*8
   add x31, x11, x31 // x31 = &B[g×2]
   lw x5, 0(x30) // f = A[f×2]
   addi x12, x30, 8 // x12 = &A[(f×2)+2]
   lw x30, 0(x12) // x30 = A[(f×2)+2]
   add x30, x30, x5 // x30 = A[(f×2)+2] + A[(f×2)]
   sw x30, 0(x31) // B[g×2] = A[(f×2)+2] + A[(f×2)]

5. abcdef12

Little-Endian		Big-Endian	
Address	**Data**	**Address**	**Data**
3	ab	**3**	12
2	cd	**2**	ef
1	ef	**1**	cd
0	12	0	ab

6. slli x28, x28, 3 // x28 = i × 8
   add x28, x28, x10 // x28 = &A[i]
   ld x28, 0(x28) // x28 = A[i]
   slli x29, x29, 3 // x29 = j × 8
   add x29, x29, x11 // x29 = &A[j]
   ld x29, 0(x29) // x29 = A[j]
   add x30, x28, x29 // x30 = A[i] + A[j]
   sd x30, 64(x11) // B[8] = A[i] + A[j]

7. addi x30, x10, 8 // x30 = &A[2]
   addi x31, x10, 0 // x31 = &A[0]
   sw x31, 0(x30) // A[2] = &A[0]
   lw x30, 0(x30) // x30 = &A[0]
   add x5, x30, x31 // f = &A[0] + &A[0]

5.5.2 Exercise 2

1. addi s0, x0, 4
   addi s1, x0, 5
   addi s2, x0, 6
   add s3, s0, s1
   add s3, s3, s2
   addi s3, s3, 10

2. sw x0, 0(s0)
   addi s1, x0, 2
   sw s1, 4(s0)
   slli t0, s1, 2
   add t0, t0, s0
   sw s1, 0(t0)

5.5.3 Exercise 3

1. Sets t0 equal to arr[3]
2. Stores t0 into arr[4]
3. Increments arr[t0] by 1
4. Sets t0 to -1 * arr[0]

5.5.4 Exercise 4

1. t0 will hold 0x00FF0004, adding 4 to the initial address. t1 will hold 36, loading the word from the address 0x00FF0000. t2 will hold 0xC, loading the upper half of the address 0x00FF0004. s1 will hold the word at 36 = 0x24, so 0xDEADB33F. Finally, s2 will hold 0xFFFFFFC5, taking the most significant byte and sign-extending it.
2. Memory updated:

5.5.5 Exercise 5

1. Sets arr[1] to arr[0] + arr[2].
2. Increments all values in the linked list by 1.
3. Negates all elements in arr.

5.5.6 Exercise 6

1. s0, s3, s9, ra, s7, and s8 We must save all s-registers we modify (note that since s7 and s8 were used, it is assumed that they were modified in omitted code), and it is conventional to store ra in the prologue (rather than just before calling a function) when the function contains a function call.
2. t0, a7
   Under calling conventions, all the t-registers and a-registers may be changed by generate random, so we must store all of these which we need to know the value of after the call. t0 is used on line 18 and a7 is used on line 29. Note that while t1 and t5 are used later, we don't care about its value before calling generate random (they are set after the call, on lines 14-15), so we don't need to store them.
3. t1, t5, a7
   As before, we must save t-registers and a-registers we need to read later.
4. s0, s3, s9, ra, s7, and s8
   This mirrors what we saved in the prologue.

5.5.7 Exercise 7

1. **False**. This only holds for data types that are four bytes wide, like int or float. For data-types like char that are only one byte wide, 4(a0) is too large of an offset to return the element at index 1, and will instead return a char further down the array (or some other data beyond the array, depending on the array length).
2. **True**. If your compiler/OS allows it (some do not, for security reasons), it is possible for your code to jump to and execute instructions passed into the program via an array. Conversely, it's also possible for your code to treat itself as normal data (search up self-modifying code if you want to see more details).
3. **False**. jalr is used to return to the memory address specified in the second argument. Keep in mind that jal jumps to a label (which is translated into an immediate by the assembler), whereas jalr jumps to an address stored in a register, which is set at runtime. Related, j label is a pseudo-instruction for jal x0, label (they do the same thing).
4. **False**. a0 and a1 registers are often used to store the return value from a function, so the function can set their values to the its return values before returning.
5. **True**. The saved registers are callee-saved, so we must save and restore them at the beginning and end of functions. This is frequently done in organized blocks of code called the "function prologue" and "function epilogue".
6. **False**. While it is a good idea to create a separate 'prologue' and 'epilogue' to save callee registers onto the stack, the stack is mutable anywhere in the function. A good example is if you want to preserve the current value of a temporary register, you can decrement the sp to save the register onto the stack right before a function call.
7. - blt s0, s1, LABEL
   - bne s0, s1, LABEL
   - bge s1, s0, LABEL
   - blt s1, s0, LABEL

   Note that RISC-V does not provide a bgt instruction because you can manipulate the blt instruction to get an equivalent result. Also note that the above solutions assume that s0 and s1 contained signed integers. If they are unsigned, then we would use the unsigned variants of the above commands (namely, bltu, bgeu).

5.5.8 Exercise 8

1. addi s0, x0, 5
   addi s1, x0, 10

```
add t0, s0, s0
bne t0, s1, else
xor s0, x0, x0
jal x0, exit
else:
addi s1, s0, -1
exit:
```

2. // computes s1 = 2^{30}
```
// assume int s1, s0; was declared above
s1 = 1;
for(s0 = 0; s0 != 30; s0++) {
s1 *= 2;
}
```

3.
```
addi s1, x0, 0
loop:
beq s0, x0, exit
add s1, s1, s0
addi s0, s0, -1
jal x0, loop
exit:
```

5.5.9 Exercise 9

1. Assembly language instruction and binary representation.
   - R-type: sub x6, x7, x5 (0x40538333: 0100 0000 0101 0011 1000 0011 0011 0011)
   - I-type: lw x3, 4(x27) (0x4DB183: 0000 0000 0100 1101 1011 0001 1000 0011)
2. Machine code in Hex:
   - 0x003100b3: 00000000001100010000000010110011
   - 0x06410093; 00000110010000010000000010010011
   - 0x00410083: 00000000010000010000000010000011
   - 0x40830063: 01000000100000110000000001100011
   - 0x011FAF03: 00000001000111111010111100000011
3. The instructions are:
   - slt x5, x2, x3
   - lb x15, 8(x4)
   - sh x2, 4(x3)
   - auipc x18, 0xEE151
4. Question 4
   - 0x00048C63
   - Line: 7
   - Line: 8
5. **<lower bound>**:
   $-8192 = [-2^{13}, 2^{13} - 1]$ Since we now only need 4 bits for the register fields, for Branch instructions, the immediate field will have 14 bits.
   <upper bound>:
   $8191 = [-2^{13}, 2^{13} - 1]$ Since we now only need 4 bits for the register fields, for Branch instructions, the immediate field will have 14 bits.

5.5.10 Exercise 10

1. &year: static
2. name: stack
3. game: static
4. ver: heap
5. &ver: stack