Noel Ball
HW2

Question 1:

Ai.) The number of integer general purpose registers should be part of the architecture. Developers can know how many hardware resources are available for their code allowing them to write more efficient and portable code that compiles with any hardware using the same ISA.

Aii.) Hiding the cycle count of instructions within the implementation supports ongoing hardware adaptations without having to change the software.

Aiii.) The clock frequency of the processor should be hidden within the implementation to allow for more scalable architectural design. This improves the ability to meet a wider range of application needs.

Aiv.) The bandwidth bus between the memory and processor should also be hidden within the implementation for better hardware design flexibility and scalability, as well as simplifying software development.

Bi.) 0x0083b283 → 00000000100000111011001010000011
Opcode: last 7 bits '0000011' = load instruction
Rd: next 5 bits '01010' = x_{10}
Func3: next 3 bits '011' = ld
Rs1: next 5 bits '11011' = x_{27}
Imm[11:0]: remaining bits = 8
0x0083b283 → ld x_{10}, 8(x_{27})

Bii.) 0x0062c2b3 → 00000000011000101100001010110011
Opcode: last 7 bits '0110011' = r-type instruction
Rd: next 5 bits '00101' = x_5
Func3: next 3 bits '100' = xor
Rs1: next 5 bits '11000' = x_{24}
Rs2: next 5 bits '00010' = x_2
Func7: next 7 bits = 0 (no modifiers)
0x0062c2b3 → xor x_5, x_{24}, x_2

Ci.) addi x_{10}, x_{10}, 8
Opcode: addi (i-type) = '0010011'
Rd: x_{10} = 10
Func3: addi = '000'
Rs1: x_{10} = 10

Imm: 8 = ‘0…1000’
addi x_{10}, x_{10}, 8 → 00000000100001010000010100010011 → 0x00851513

Cii.) sd x_{11}, 16(x_{10})
Opcode: sd (s-type) = ‘0100011’
Imm: 16 = ‘0…10000’, [11:5] 7 upper bits = ‘0000001’
Rs2: x_{11} = 11
Rs1: x_{10} = 10
Func3: sd = ‘011’
Imm: 16 = ‘0…10000’, [4:0] 5 lower bits ‘00000’
sd x_{11}, 16(x_{10}) → 00000010101101010001100000010011 → 0x02B51813

Question 2:
Ai.) y = a - b + c; assuming a = x_{10}, b = x_{11}, c = x_{12}, and y = x_{13}
Sub x_{13}, x_{10}, x_{11} # y = a - b
Add x_{13}, x_{13}, x_{12} # y = y + c

Aii.) total += a[i]; assuming total = x_{20}, i = x_{21}, base a = x_{22}, each element element in ‘a’ is 64-bit
Slli x_{21}, x_{21}, 3   # shift i left by 3 to multiply by 8 (byte offset)
Add x_{21}, x_{22}, x_{21} # add base address of a to index offset, x_{21} = a[i]
Ld x5, 0(x_{21})   # load value of a[i] into temp register x_5
Add x_{20}, x_{20}, x_5 # add value of a[i] to total, stored in total

B.) ld x_5, 0(x_{10})   # load x_{10} into x_5
addi x_5, x_5, 1 # x_5 + 1
sd x_5, 0(x_{10}) # store x_5 into x_{10}

In C: *(x_{10})++; dereferences x_{10} loading its value, adds 1, and stores the value where x_{10} points

Ci.) if (x == 4) { x = 0; }
    Li x_{11}, 4   # Load 4 into x_{11}
    Beq x_{10}, x_{11} set_x # Compare x_{10} (x) with x_{11} (4), if equal branch to set_x
    J end   # Jump to end if condition false
Set_x:
    Li x_{10}, 0   # Load 0 into x_{10}, x = 0
end:

```
Cii.) if (y >= 2) { y = 5; }
        Li $x_{12}$, 2            # Load 2 into $x_{12}$
        Bge $x_{11}$, $x_{12}$, set_y # Compare $x_{11}$ (y) with $x_{12}$ (2), if greater or equal branch to set_y
        J end                     # Jump to end if condition false
Set_y:
        Li $x_{11}$, 5            # Load 5 into $x_{11}$, y = 5
end:
```

```
D.)     slt $x_{10}$, $x_5$ , $x_6$      # If $x_5$ < $x_6$ sets $x_{10}$ = 1, $x_{10}$ = 0 otherwise
        beq $x_{10}$, $x_0$, L1    # Branch to L1 if $x_{10}$ equals $x_0$ (typically 0), checking if $x_5$ >= $x_6$
        xor $x_7$, $x_0$, $x_5$     # Xor $x_0$ and $x_5$, since xor with 0 doesn't change other value, $x_7$ = $x_5$
        j L2                       # Jumps to L2 if condition above isn't met, skipping L1
L1: or $x_7$, $x_6$, $x_0$         # Or $x_6$ and $x_0$, since or with 0 doesn't change other value, $x_7$ = $x_6$
L2:
```

```
In C: if ($x_5$ >= $x_6$) { $x_7$ = $x_5$; }
      else               { $x_7$ = $x_6$; }
```

Question 3:

```
A.)     bge r0, a0, L2       # If $r_0$ >= $a_0$ branch to L2 and skip loop
L1:
        andi t0, a0, 1       # And $a_0$ and 1, isolating the LSB of $a_0$
        add a1, a1, t0       # $a_1$ = $t_0$ + $a_1$
        srai a0, a0, 1       # Shifts $a_0$ to the right 1 bit
        blt r0, a0, L1       # If $r_0$ < $a_0$ branch to L1 for another loop
L2:
```

```
In C: while ($r_0$ < $a_0$) {
        Int $t_0$ = $a_0$ & 1;
        $a_1$ += $t_0$;
        $a_0$ >>= 1; }
```

B.) assuming a_0 holds pointer to array 'a', a_1 holds value of 'n', a_0 will return result

```
Array_total:
        Addi sp, sp, -16       # Adjust stack pointer
        Sd ra, 8(sp)           # Save return address
        Sd $s_0$, 0(sp)        # Save frame pointer
        Addi $s_0$, sp, 16     # Set new frame pointer

        Addi $t_0$, zero, 0    # Initialize total to 0, using $t_0$ as accumulator
        Addi t1, zero, 0        # Initialize loop counter i to 0, using t1
Loop_start:
        Bge t1, a1, loop_end # If i >= n, break

        Slli t2, t1, 3          # t2 = i * 8 (byte offset for a[i])
        Add t2, a0, t2          # t2 = address of a[i]
        Ld t3, 0(t2)            # Load a[i] into t3
        Add t0, t0, t3          # Add a[i] to total

        Addi t1, t1, 1          # Increment loop counter i
        J loop_start

Loop_end:
        Mv a0, t0               # Move total from t0 to a0 for return
        Ld ra, 8(sp)            # Restore return address
        Ld s0, 0(sp)            # Restore frame pointer
        Addi sp, sp, 16         # Adjust stack pointer back
        Ret
```

Question 4:

A.) Main:

```
        Li a0, 0x4d808 # Load first argument into a0
        Li a1, 0xbab   # Load second argument into a1
        Call foo       # Call the function foo
```

B.) Foo:

```
        Addi sp, sp, -16        # Adjust stack pointer
        Sd ra, 8(sp)            # Save return address
        Sd s0, 0(sp)            # Save frame pointer
        Addi s0, sp, 16         # Set new frame pointer

        Bne a0, a1, not_equal   # If x != y, branch to not_equal
        J exit                  # If x == y, prepare to return x in a0

Not_equal:
        Blt a0, a1, x_less_y    # If x < y, go to x_less_y

        Sub a0, a0, a1          # x > y, store x - y in a0
        Call foo                # recursive call with new x and same y
        J exit
```

```
X_less_y:
        Sub a1, a1, a0          # store y - x in a1
        Call foo                # recursive call foo with same x and new y

Exit:
        Ld ra, 8(sp)            # Restore the return address
        Ld s0, 0(sp)            # Restore the frame pointer
        Addi sp, sp, 16         # Adjust stack pointer back
        Ret                     # Return from function
```