UNIT 1

2.1 [5] <§2.2> For the following C statement, write the corresponding RISC-V assembly code. Assume that the C variables f, g, and h, have already been placed in registers x5, x6, and x7 respectively. Use a minimal number of RISC-V assembly instructions.

```
f = g + (h − 5);
```

2.1
```
addi x5, x7,-5
add  x5, x5, x6
[addi f,h,-5 (note, no subi) add f,f,g]
```

2.2 [5] <§2.2> Write a single C statement that corresponds to the two RISC-V assembly instructions below.

```
add f, g, h
add f, i, f
```

2.2 f = g+h+i

2.3 [5] <§§2.2, 2.3> For the following C statement, write the corresponding RISC-V assembly code. Assume that the variables f, g, h, i, and j are assigned to registers x5, x6, x7, x28, and x29, respectively. Assume that the base address of the arrays A and B are in registers x10 and x11, respectively.

```
B[8] = A[i−j];
```

2.3
```
sub     x30, x28, x29    // compute i-j
slli   x30, x30, 3      //multiply by 8 to convert the
                                  word offset to a byte offset
add     x3, x30, x10
ld      x30, 0(x3)       // load A[i-j]
sd      x30, 64(x11)     // store in B[8]
```

2.4 [10] <§§2.2, 2.3> For the RISC-V assembly instructions below, what is the corresponding C statement? Assume that the variables f, g, h, i, and j are assigned to registers x5, x6, x7, x28, and x29, respectively. Assume that the base address of the arrays A and B are in registers x10 and x11, respectively.

```
slli x30, x5, 3     // x30 = f*8
add  x30, x10, x30  // x30 = &A[f]
slli x31, x6, 3     // x31 = g*8
add  x31, x11, x31  // x31 = &B[g]
lw   x5, 0(x30)     // f = A[f]

addi x12, x30, 8
lw   x30, 0(x12)
add  x30, x30, x5
lw   x30, 0(x31)
```

2.4 B[g]= A[f] + A[f+1]

```
slli   x30, x5, 3       // x30 = f*8
add    x30, x10, x30    // x30 = &A[f]
slli   x31, x6, 3       // x31 = g*8
add    x31, x11, x31    // x31 = &B[g]
ld     x5, 0(x30)       // f = A[f]
addi   x12, x30, 8      // x12 = &A[f]+8  (i.e. &A[f+1])
ld     x30, 0(x12)      // x30 = A[f+1]
add    x30, x30, x5     // x30 = A[f+1] + A[f]
sd     x30, 0(x31)      // B[g] = x30 (i.e. A[f+1] + A[f])
```

2.5 [5] <§2.3> Show how the value `0xabcdef12` would be arranged in memory of a little-endian and a big-endian machine. Assume the data are stored starting at address 0 and that the word size is 4 bytes.

2.5

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

2.6 [5] <§2.4> Translate `0xabcdef12` into decimal.

2.6 2882400018

2.7 [5] <§§2.2, 2.3> Translate the following C code to RISC-V. Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `x5`, `x6`, `x7`, `x28`, and `x29`, respectively. Assume that the base address of the arrays `A` and `B` are in registers `x10` and `x11`, respectively. Assume that the elements of the arrays `A` and `B` are 8-byte words:

```
B[8] = A[i] + A[j];
```

2.7
```
slli    x28, x28, 3       // x28 = i*8
add     x10,x10,x28       // x10 = &A[i]
ld      x28, 0(x10)       // x28 = A[i]
slli    x29, x29, 3       // x29 = j*8
add     x12,x11,x29       // x12 = &B[j]
ld      x29,0(x12)        // x29 = B[j]
add     x29, x28, x29     // x29 = B[i]+B[j]
sd      x29, 64(x11)      // B[8] = B[i]+B[j]
```

2.8 [10] <§§2.2, 2.3> Translate the following RISC-V code to C. Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `x5`, `x6`, `x7`, `x28`, and `x29`, respectively. Assume that the base address of the arrays `A` and `B` are in registers `x10` and `x11`, respectively.

```
addi  x30, x10, 8
addi  x31, x10, 0
sw    x31, 0(x30)
lw    x30, 0(x30)
add   x5,  x30, x31
```

2.8 f = 2*(&A)
```
addi    x30, x10, 8       // x30 = &A[1]
addi    x31, x10, 0       // x31 = &A
sd      x31, 0(x30)       // A[1] = &A
ld      x30, 0(x30)       // x30 = A[1] = &A
add     x5, x30, x31      // f = &A + &A = 2*(&A)
```

2.9 [20] <§§2.2, 2.5> For each RISC-V instruction in Exercise 2.8, show the value of the opcode (op), source register (rs1), and destination register (rd) fields. For the I-type instructions, show the value of the immediate field, and for the R-type instructions, show the value of the second source register (rs2). For non U- and UJ-type instructions, show the funct3 field, and for R-type and S-type instructions, also show the funct7 field.

2.9

	type	opcode, funct3,7	rs1	rs2	rd	imm
addi x30,x10,8	I-type	0x13, 0x0, --	10	--	30	8
addi x31,x10,0	R-type	0x13, 0x0, --	10	--	31	0
sd x31,0(x30)	S-type	0x23, 0x3, --	31	30	--	0
ld x30,0(x30)	I-type	0x3, 0x3, --	30	--	30	0
add x5, x30, x31	R-type	0x33, 0x0, 0x0	30	31	5	--

2.10 Assume that registers `x5` and `x6` hold the values `0x8000000000000000` and `0xD000000000000000`, respectively.

2.10.1 [5] <§2.4> What is the value of `x30` for the following assembly code?

```
add x30, x5, x6
```

2.10.2 [5] <§2.4> Is the result in `x30` the desired result, or has there been overflow?

2.10.3 [5] <§2.4> For the contents of registers `x5` and `x6` as specified above, what is the value of `x30` for the following assembly code?

```
sub x30, x5, x6
```

2.10.4 [5] <§2.4> Is the result in `x30` the desired result, or has there been overflow?

2.10.5 [5] <§2.4> For the contents of registers `x5` and `x6` as specified above, what is the value of `x30` for the following assembly code?

```
add x30, x5, x6
add x30, x30, x5
```

2.10.6 [5] <§2.4> Is the result in `x30` the desired result, or has there been overflow?

2.10

2.10.1 0x5000000000000000

2.10.2 overflow

2.10.3 0xB000000000000000

2.10.4 no overflow

2.10.5 0xD000000000000000

2.10.6 overflow

2.11 Assume that x5 holds the value 128_{ten}.

2.11.1 [5] <§2.4> For the instruction `add x30, x5, x6`, what is the range(s) of values for x6 that would result in overflow?

2.11.2 [5] <§2.4> For the instruction `sub x30, x5, x6`, what is the range(s) of values for x6 that would result in overflow?

2.11.3 [5] <§2.4> For the instruction `sub x30, x6, x5`, what is the range(s) of values for x6 that would result in overflow?

2.11

2.11.1 There is an overflow if $128 + x6 > 2^{63} - 1$.
In other words, if $x6 > 2^{63} - 129$.
There is also an overflow if $128 + x6 < -2^{63}$.
In other words, if $x6 < -2^{63} - 128$ (which is impossible given the range of x6).

2.11.2 There is an overflow if $128 - x6 > 2^{63} - 1$.
In other words, if $x6 < -2^{63} + 129$.
There is also an overflow if $128 - x6 < -2^{63}$.
In other words, if $x6 > 2^{63} + 128$ (which is impossible given the range of x6).

2.11.3 There is an overflow if $x6 - 128 > 2^{63} - 1$.
In other words, if $x6 < 2^{63} + 127$ (which is impossible given the range of x6).
There is also an overflow if $x6 - 128 < -2^{63}$.
In other words, if $x6 < -2^{63} + 128$.

2.12 [5] <§§2.2, 2.5> Provide the instruction type and assembly language instruction for the following binary value:

$0000\ 0000\ 0001\ 0000\ 1000\ 0000\ 1011\ 0011_{two}$

Hint: Figure 2.20 may be helpful.

2.12 R-type: add x1, x1, x1

2.13 [5] <§§2.2, 2.5> Provide the instruction type and hexadecimal representation of the following instruction:

```
sw x5, 32(x30)
```

2.14 [5] <§2.5> Provide the instruction type, assembly language instruction, and binary representation of instruction described by the following RISC-V fields:

```
opcode=0x33, funct3=0x0, funct7=0x20, rs2=5, rs1=7, rd=6
```

2.15 [5] <§2.5> Provide the instruction type, assembly language instruction, and binary representation of instruction described by the following RISC-V fields:

```
opcode=0x3, funct3=0x3, rs1=27, rd=3, imm=0x4
```

2.13 S-type: 0x25F3023 (0000 0010 0101 1111 0011 0000 0010 0011)

2.14 R-type: sub x6, x7, x5 (0x40538333: 0100 0000 0101 0011 1000 0011 0011 0011)

2.15 I-type: ld x3, 4(x27) (0x4DB183: 0000 0000 0100 1101 1011 0001 1000 0011)

2.16 Assume that we would like to expand the RISC-V register file to 128 registers and expand the instruction set to contain four times as many instructions.

2.16.1 [5] <§2.5> How would this affect the size of each of the bit fields in the R-type instructions?

2.16.2 [5] <§2.5> How would this affect the size of each of the bit fields in the I-type instructions?

2.16.3 [5] <§§2.5, 2.8, 2.10> How could each of the two proposed changes decrease the size of a RISC-V assembly program? On the other hand, how could the proposed change increase the size of an RISC-V assembly program?

2.16

2.16.1 The opcode would expand from 7 bits to 9.

The rs1, rs2, and rd fields would increase from 5 bits to 7 bits.

2.16.2 The opcode would expand from 7 bits to 12.

The rs1 and rd fields would increase from 5 bits to 7 bits. This change does not affect the imm field per se, but it might force the ISA designer to consider shortening the immediate field to avoid an increase in overall instruction size.

2.16.3 * Increasing the size of each bit field potentially makes each instruction longer, potentially increasing the code size overall.

* However, increasing the number of registers could lead to less register spillage, which would reduce the total number of instructions, possibly reducing the code size overall.

2.17 Assume the following register contents:

```
x5 = 0x00000000AAAAAAAA, x6 = 0x1234567812345678
```

2.17.1 [5] <§2.6> For the register values shown above, what is the value of x7 for the following sequence of instructions?

```
slli x7, x5, 4
or   x7, x7, x6
```

2.17.2 [5] <§2.6> For the register values shown above, what is the value of x7 for the following sequence of instructions?

```
slli x7, x6, 4
```

2.17.3 [5] <§2.6> For the register values shown above, what is the value of x7 for the following sequence of instructions?

```
srli  x7, x5, 3
andi  x7, x7, 0xFEF
```

2.17

2.17.1 0x1234567ababefef8

2.17.2 0x2345678123456780

2.17.3 0x545

2.18 [10] <§2.6> Find the shortest sequence of RISC-V instructions that extracts bits 16 down to 11 from register `x5` and uses the value of this field to replace bits 31 down to 26 in register `x6` without changing the other bits of registers `x5` or `x6`. (Be sure to test your code using `x5 = 0` and `x6 = 0xffffffffffffffff`. Doing so may reveal a common oversight.)

2.18 It can be done in eight RISC-V instructions:

```
addi x7, x0, 0x3f// Create bit mask for bits 16 to 11
slli x7, x7, 11   // Shift the masked bits
and  x28,x5, x7   // Apply the mask to x5
slli x7, x6, 15  //Shift the mask to cover bits 31
                          to□26
xori x7, x7, -1   // This is a NOT operation
and  x6, x6, x7  //"Zero out" positions 31 to
                          26 of x6
slli x28,x28,15  //Move selection from x5 into
                          positions 31 to 26
or   x6, x6, x28 // Load bits 31 to 26 from x28
```

2.19 [5] <§2.6> Provide a minimal set of RISC-V instructions that may be used to implement the following pseudoinstruction:

```
not x5, x6      // bit-wise invert
```

2.19 xori x5, x6, -1

2.20 [5] <§2.6> For the following C statement, write a minimal sequence of RISC-V assembly instructions that performs the identical operation. Assume `x6 = A`, and `x17` is the base address of C.

```
A = C[0] << 4;
```

```
2.20 ld    x6, 0(x17)
     slli x6, x6, 4
```

2.21 [5] <§2.7> Assume `x5` holds the value `0x0000000001010000`. What is the value of `x6` after the following instructions?

```
        bge x5, x0, ELSE
        jal x0, DONE
ELSE:   ori x6, x0, 2
DONE:
```

2.21 x6 = 2

2.22 Suppose the *program counter* (PC) is set to `0x20000000`.

2.22.1 [5] <§2.10> What range of addresses can be reached using the RISC-V *jump-and-link* (`jal`) instruction? (In other words, what is the set of possible values for the PC after the jump instruction executes?)

2.22.2 [5] <§2.10> What range of addresses can be reached using the RISC-V *branch if equal* (`beq`) instruction? (In other words, what is the set of possible values for the PC after the branch instruction executes?)

2.22

2.22.1 [0x1ff00000, 0x200FFFFE]

2.22.2 [0x1FFFF000, 0x20000ffe]

2.23 Consider a proposed new instruction named `rpt`. This instruction combines a loop's condition check and counter decrement into a single instruction. For example `rpt x29, loop` would do the following:

```
if (x29 > 0) {
        x29 = x29 -1;
        goto loop
    }
```

2.23.1 [5] <§2.7, 2.10> If this instruction were to be added to the RISC-V instruction set, what is the most appropriate instruction format?

2.23.2 [5] <§2.7> What is the shortest sequence of RISC-V instructions that performs the same operation?

2.23

2.23.1 The UJ instruction format would be most appropriate because it would allow the maximum number of bits possible for the "loop" parameter, thereby maximizing the utility of the instruction.

2.23.2 It can be done in three instructions:

```
loop:
        addi  x29, x29, -1    // Subtract 1 from x29
        bgt   x29, x0, loop  //Continue if x29 not
                                          negative
        addi  x29, x29, 1     //Add back 1 that shouldn't
                                          have been subtracted.
```

2.24 Consider the following RISC-V loop:

```
LOOP: beq  x6, x0, DONE
      addi x6, x6, -1
      addi x5, x5, 2
      jal  x0, LOOP
DONE:
```

2.24.1 [5] <§2.7> Assume that the register x6 is initialized to the value 10. What is the final value in register x5 assuming the x5 is initially zero?

2.24.2 [5] <§2.7> For the loop above, write the equivalent C code. Assume that the registers x5 and x6 are integers acc and i, respectively.

2.24.3 [5] <§2.7> For the loop written in RISC-V assembly above, assume that the register x6 is initialized to the value N. How many RISC-V instructions are executed?

2.24.4 [5] <§2.7> For the loop written in RISC-V assembly above, replace the instruction "beq x6, x0, DONE" with the instruction "blt x6, x0, DONE" and write the equivalent C code.

2.24

2.24.1 The final value of xs is 20.

2.24.2
```
acc = 0;
i = 10;
while (i ! = 0) {
    acc += 2;
    i--;
}
```

2.24.3 4*N + 1 instructions.

2.24.4 (Note: change condition ! = to > = in the while loop)

```
acc = 0;
i = 10;
while (i >= 0) {
    acc += 2;
    i--;
}
```

2.25 [10] <§2.7> Translate the following C code to RISC-V assembly code. Use a minimum number of instructions. Assume that the values of a, b, i, and j are in registers x5, x6, x7, and x29, respectively. Also, assume that register x10 holds the base address of the array D.

```
for(i=0; i<a; i++)
   for(j=0; j<b; j++)
      D[4*j] = i + j;
```

2.25 The C code can be implemented in RISC-V assembly as follows:

```
LOOPI:
        addi   x7,    x0, 0         // Init i = 0
        bge    x7,    x5, ENDI      // While i < a
        addi   x30,   x10, 0        // x30 = &D
        addi   x29,   x0, 0         // Init j = 0
LOOPJ:
        bge    x29,   x6, ENDJ      // While j < b
        add    x31,   x7, x29       // x31 = i+j
        sd     x31,   0(x30)        // D[4*j] = x31
        addi   x30,   x30, 32       // x30 = &D[4*(j+1)]
        addi   x29,   x29, 1        // j++
        jal    x0,    LOOPJ

ENDJ:
        addi   x7,    x7, 1         // i++;
        jal    x0,    LOOPI
ENDI:
```

2.29 [30] <§2.8> Implement the following C code in RISC-V assembly. Hint: Remember that the stack pointer must remain aligned on a multiple of 16.

```
int fib(int n){
    if (n==0)
       return 0;
    else if (n == 1)
       return 1;
    else
       return fib(n-1) + fib(n-2);
}
```

2.29

```
        // IMPORTANT! Stack pointer must reamin a multiple
         of 16!!!!

fib:
        beq    x10,   x0, done      // If n==0, return 0
        addi   x5,    x0, 1
        beq    x10,   x5, done      // If n==1, return 1
        addi   x2,    x2, -16       //Allocate 2 words of stack
                                                space
        sd     x1,    0(x2)         // Save the return address
        sd     x10,   8(x2)         // Save the current n
        addi   x10,   x10, -1       // x10 = n-1
        jal    x1,    fib           // fib(n-1)
        ld     x5,    8(x2)         // Load old n from the stack
        sd     x10,   8(x2)         // Push fib(n-1) onto the stack
        addi   x10,   x5, -2        // x10 = n-2
        jal    x1,    fib           // Call fib(n-2)
        ld     x5,    8(x2)         // x5 = fib(n-1)
        add    x10,   x10, x5       // x10 = fib(n-1)+fib(n-2)
// Clean up:
        ld     x1,    0(x2)         // Load saved return address
        addi   x2,    x2, 16        // Pop two words from the stack
done:
        jalr   x0,    x1
```

2.31 [20] <§2.8> Translate function `f` into RISC-V assembly language. Assume the function declaration for `g` is `int g(int a, int b)`. The code for function `f` is as follows:

```
int f(int a, int b, int c, int d){
  return g(g(a,b), c+d);
}
```

2.31

```
// IMPORTANT! Stack pointer must remain a multiple of 16!!!
f:
    addi  x2, x2, -16     //Allocate stack space for 2 words
    sd      x1, 0(x2)       // Save return address
    add    x5, x12, x13  // x5 = c+d
    sd      x5, 8(x2)       // Save c+d on the stack
    jal    x1, g             // Call x10 = g(a,b)
    ld      x11, 8(x2)     // Reload x11= c+d from the stack
    jal    x1, g             // Call x10 = g(g(a,b), c+d)
    ld      x1, 0(x2)       // Restore return address
    addi  x2, x2, 16      // Restore stack pointer
    jalr  x0, x1
```

2.33 [5] <§2.8> Right before your function `f` from Exercise 2.31 returns, what do we know about contents of registers `x10`-`x14`, `x8`, `x1`, and `sp`? Keep in mind that we know what the entire function `f` looks like, but for function `g` we only know its declaration.

2.33 *We have no idea what the contents of x10-x14 are, g can set them as it pleases.

*We don't know what the precise contents of x8 and sp are; but we do know that they are identical to the contents when f was called.

*Similarly, we don't know what the precise contents of x1 are; but, we do know that it is equal to the return address set by the "jalx1,f" instruction that invoked f.

Note : Register Mapping for variables a, b, c, d, g should be given as x10-14, x9

2.34 [30] <§2.9> Write a program in RISC-V assembly to convert an ASCII string containing a positive or negative integer decimal string to an integer. Your program should expect register `x10` to hold the address of a null-terminated string containing an optional "+" or "−" followed by some combination of the digits 0 through 9. Your program should compute the integer value equivalent to this string of digits, then place the number in register `x10`. If a non-digit character appears anywhere in the string, your program should stop with the value −1 in register `x10`. For example, if register `x10` points to a sequence of three bytes 50_{ten}, 52_{ten}, 0_{ten} (the null-terminated string "24"), then when the program stops, register `x10` should contain the value 24_{ten}. The RISC-V `mul` instruction takes two registers as input. There is no "`muli`" instruction. Thus, just store the constant 10 in a register.

2.34

```
a_to_i:
    addi     x28,   x0, 10        # Just stores the constant 10
    addi     x29,   x0, 0         # Stores the running total
    addi     x5,    x0, 1         #Tracks whether input is positive
                                             or negative
    # Test for initial '+' or '-'
    lbu      x6, 0(x10)           # Load the first character
    addi     x7, x0, 45           # ASCII '-'
    bne      x6, x7, noneg
    addi     x5, x0, -1           # Set that input was negative
    addi     x10, x10, 1          # str++
    jal      x0, main_atoi_loop

noneg:
    addi     x7, x0, 43           # ASCII '+'
    bne      x6, x7, main_atoi_loop
    addi     x10, x10, 1          # str++

main_atoi_loop:
    lbu      x6, 0(x10)           # Load the next digit
    beq      x6, x0, done         #Make sure next char is a digit,
                                             or fail
    addi     x7, x0, 48           # ASCII '0'
    sub      x6, x6, x7
    blt      x6, x0, fail       # *str < '0'
    bge      x6, x28, fail      # *str >= '9'

    # Next char is a digit, so accumulate it into x29

    mul      x29, x29, x28        # x29 *= 10
    add      x29, x29, x6         # x29 += *str - '0'
    addi     x10, x10, 1          # str++
    jal      x0, main_atoi_loop

done:
    addi     x10, x29, 0          # Use x29 as output value
    mul      x10, x10, x5         # Multiply by sign
    jalr     x0, x1               # Return result

fail:
    addi     x10, x0, -1
    jalr     x0, x1
```

2.35 Consider the following code:

```
lb x6, 0(x7)
sw x6, 8(x7)
```

Assume that the register x7 contains the address 0×10000000 and the data at address is 0×1122334455667788.

2.35.1 [5] <§2.3, 2.9> What value is stored in 0×10000008 on a big-endian machine?

2.35.2 [5] <§2.3, 2.9> What value is stored in 0×10000008 on a little-endian machine?

2.35

2.35.1 0x11

2.35.2 0x88

2.36 [5] <§2.10> Write the RISC-V assembly code that creates the 32-bit constant $0x12345678_{hex}$ and stores that value to register x10.

```
2.36 lui    x10,   0x11223
     addi   x10,   x10, 0x344
     slli   x10,   x10, 32
     lui    x5,    0x55667
     addi   x5,    x5, 0x788
     add    x10,   x10, x5
```

UNIT 2

4.1 Consider the following instruction:

Instruction: `and rd, rs1, rs2`

Interpretation: `Reg[rd] = Reg[rs1] AND Reg[rs2]`

4.1.1 [5] <§4.3> What are the values of control signals generated by the control in Figure 4.10 for this instruction?

4.1.2 [5] <§4.3> Which resources (blocks) perform a useful function for this instruction?

4.1.3 [10] <§4.3> Which resources (blocks) produce no output for this instruction? Which resources produce output that is not used?

4.1

4.1.1 The value of the signals is as follows:

RegWrite	ALUSrc	ALUoperation	MemWrite	MemRead	MemToReg
true	0	"and"	false	false	0

Mathematically, the MemRead control wire is a "don't care": the instruction will run correctly regardless of the chosen value. Practically, however, MemRead should be set to false to prevent causing a segment fault or cache miss.

4.1.2 Registers, ALUsrc mux, ALU, and the MemToReg mux.

4.1.3 All blocks produce some output. The outputs of DataMemory and Imm Gen are not used.

4.2 [10] <§4.4> Explain each of the "don't cares" in Figure 4.22.

4.2 MemToReg for sd and beq: Neither sd nor beq write a value to the register file. It doesn't matter which value the MemToReg mux passes to the register file because the register file ignores that value.

4.3 Consider the following instruction mix:

R-type	I-type (non-lw)	Load	Store	Branch	Jump
24%	28%	25%	10%	11%	2%

4.3.1 [5] <§4.4> What fraction of all instructions use data memory?

4.3.2 [5] <§4.4> What fraction of all instructions use instruction memory?

4.3.3 [5] <§4.4> What fraction of all instructions use the sign extend?

4.3.4 [5] <§4.4> What is the sign extend doing during cycles in which its output is not needed?

4.3

4.3.1 25 + 10 = 35%. Only Load and Store use Data memory.

4.3.2 100% Every instruction must be fetched from instruction memory before it can be executed.

4.3.3 28 + 25 + 10 + 11 + 2 = 76%. Only R-type instructions do not use the Sign extender.

4.3.4 The sign extend produces an output during every cycle. If its output is not needed, it is simply ignored.

4.5 In this exercise, we examine in detail how an instruction is executed in a single-cycle datapath. Problems in this exercise refer to a clock cycle in which the processor fetches the following instruction word: `0x00c6ba23`.

4.5.1 [10] <§4.4> What are the values of the ALU control unit's inputs for this instruction?

4.5.2 [5] <§4.4> What is the new PC address after this instruction is executed? Highlight the path through which this value is determined.

4.5.3 [10] <§4.4> For each mux, show the values of its inputs and outputs during the execution of this instruction. List values that are register outputs at `Reg [xn]`.

4.5.4 [10] <§4.4> What are the input values for the ALU and the two add units?

4.5.5 [10] <§4.4> What are the values of all inputs for the registers unit?

4.5 For context: The encoded instruction is sd x12, 20(x13)

4.5.1

ALUOp	ALU Control Lines
00	0010

4.5.2 The new PC is the old PC + 4. This signal goes from the PC, through the "PC + 4" adder, through the "branch" mux, and back to the PC.

4.5.3 ALUsrc: Inputs: Reg[x12] and 0x0000000000000014; Output: 0x0000000000000014

MemToReg: Inputs: Reg[x13] + 0x14 and <undefined>; output: <undefined>

Branch: Inputs: PC+4 and PC + 0x28

4.5.4 ALU inputs: Reg[x13] and 0x0000000000000014

PC + 4 adder inputs: PC and 4

Branch adder inputs: PC and 0x0000000000000028

4.5.5 Read register 1 = 0x13 (base address)

Read register 2 = 0x12 (data to be stored)

Write register = 0x0 or don't-care (should not write back)

Write data = don't-care (should not write back)

RegWrite = false (should not write back)

4.6 Section 4.4 does not discuss I-type instructions like `addi` or `andi`.

4.6.1 [5] <§4.4> What additional logic blocks, if any, are needed to add I-type instructions to the CPU shown in Figure 4.21? Add any necessary logic blocks to Figure 4.21 and explain their purpose.

4.6.2 [10] <§4.4> List the values of the signals generated by the control unit for `addi`. Explain the reasoning for any "don't care" control signals.

4.6

4.6.1 No additional logic blocks are needed.

4.6.2 Branch: false
MemRead: false (See footnote from solution to problem 4.1.1.)
MemToReg: 0
ALUop: 10 (or simply saying "add" is sufficient for this problem)
MemWrite: false
ALUsrc: 1
RegWrite: 1

4.7 Problems in this exercise assume that the logic blocks used to implement a processor's datapath have the following latencies:

I-Mem / D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

"Register read" is the time needed after the rising clock edge for the new register value to appear on the output. This value applies to the PC only. "Register setup" is the amount of time a register's data input must be stable before the rising edge of the clock. This value applies to both the PC and Register File.

4.7.1 [5] <§4.4> What is the latency of an R-type instruction (i.e., how long must the clock period be to ensure that this instruction works correctly)?

4.7.2 [10] <§4.4> What is the latency of `lw`? (Check your answer carefully. Many students place extra muxes on the critical path.)

4.7.3 [10] <§4.4> What is the latency of `sw`? (Check your answer carefully. Many students place extra muxes on the critical path.)

4.7.4 [5] <§4.4> What is the latency of `beq`?

4.7.5 [5] <§4.4> What is the latency of an arithmetic, logical, or shift I-type (non-load) instruction?

4.7.6 [5] <§4.4> What is the minimum clock period for this CPU?

4.7

4.7.1 R-type: 30 + 250 + 150 + 25 + 200 + 25 + 20 = 700ps

4.7.2 ld: 30 + 250 + 150 + 25 + 200 + 250 + 25 + 20 = 950ps

4.7.3 sd: 30 + 250 + 150 + 200 + 25 + 250 = 905

4.7.4 beq: 30 + 250 + 150 + 25 + 200 + 5 + 25 + 20 = 705

4.7.5 I-type: 30 + 250 + 150 + 25 + 200 + 25 + 20 = 700ps

4.7.6 950ps

4.8 [10] <§4.4> Suppose you could build a CPU where the clock cycle time was different for each instruction. What would the speedup of this new CPU be over the CPU presented in Figure 4.25 given the instruction mix below?

R-type/I-type (non-ld)	lw	sw	beq
52%	25%	11%	12%

4.8 Using the results from Problem 4.7, we see that the average time per instruction is

.52*700 + .25*950 + .11*905 + .12 * 705 = 785.6ps.

In contrast, a single-cycle CPU with a "normal" clock would require a clock cycle time of 950.

Thus, the speedup would be 925/787.6 = 1.174.

4.10 When processor designers consider a possible improvement to the processor datapath, the decision usually depends on the cost/performance trade-off. In the following three problems, assume that we are beginning with the datapath from Figure 4.25, the latencies from Exercise 4.7, and the following costs:

I-Mem	Register File	Mux	ALU	Adder	D-Mem	Single Register	Sign extend	Single gate	Control
1000	200	10	100	30	2000	5	100	1	500

Suppose doubling the number of general purpose registers from 32 to 64 would reduce the number of `lw` and `sw` instructions executed by 12%, but increase the latency of the register file from 150 ps to 160 ps and double the cost from 200 to 400. (Use the instruction mix from Exercise 4.8.)

4.10.1 [5] <§4.4> What is the speedup achieved by adding this improvement?

4.10.2 [10] <§4.4> Compare the change in performance to the change in cost.

4.10.3 [10] <§4.4> Given the cost/performance ratios you just calculated, describe a situation where it makes sense to add more registers and describe a situation where it doesn't make sense to add more registers.

4.10.1 The additional registers will allow us to remove 12% of the loads and stores, or (0.12)*(0.25 + 0.1) = 4.2% of all instructions. Thus, the time to run n instructions will decrease from 950*n to 960*.958*n = 919.68*n. That corresponds to a speedup of 950/919.68 = 1.03.

4.10.2 The cost of the original CPU is 4496; the cost of the improved CPU is 4696.

PC:	5
I-Mem:	1000
Register file:	200
ALU:	100
D-Mem:	2000
Sign Extend:	100
Controls:	1000
adders:	30*2
muxes:	3*10
single gates:	1*1

Thus, for a 3% increase in performance, the cost of the CPU increases by about 4.4%.

4.10.3 From a strictly mathematical standpoint, it does not make sense to add more registers because the new CPU costs more per unit of performance. However, that simple calculation does not account for the utility of the performance. For example, in a real-time system, a 3% performance may make the difference between meeting or missing deadlines. In which case, the improvement would be well worth the 4.4% additional cost.

4.12 Examine the difficulty of adding a proposed `swap rs1, rs2` instruction to RISC-V.

Interpretation: `Reg[rs2]=Reg[rs1]; Reg[rs1]=Reg[rs2]`

4.12.1 [5] <§4.4> Which new functional blocks (if any) do we need for this instruction?

4.12.2 [10] <§4.4> Which existing functional blocks (if any) require modification?

4.12.3 [5] <§4.4> What new data paths do we need (if any) to support this instruction?

4.12.4 [5] <§4.4> What new signals do we need (if any) from the control unit to support this instruction?

4.12.5 [5] <§4.4> Modify Figure 4.25 to demonstrate an implementation of this new instruction.

4.12

4.12.1 No new functional blocks are needed.

4.12.2 The register file needs to be modified so that it can write to two registers in the same cycle. The ALU would also need to be modified to allow read data 1 or 2 to be passed through to write data 1.

4.12.3 The answer depends on the answer given in 4.12.2: Whichever input was not allowed to pass through the ALU above must now have a data path to write data 2.

4.12.4 There would need to be a second RegWrite control wire.

4.12.5 Many possible solutions.

4.13 Examine the difficulty of adding a proposed `ss rs1, rs2, imm` (Store Sum) instruction to RISC-V.

Interpretation: `Mem[Reg[rs1]]=Reg[rs2]+immediate`

4.13.1 [10] <§4.4> Which new functional blocks (if any) do we need for this instruction?

4.13.2 [10] <§4.4> Which existing functional blocks (if any) require modification?

4.13.3 [5] <§4.4> What new data paths do we need (if any) to support this instruction?

4.13.4 [5] <§4.4> What new signals do we need (if any) from the control unit to support this instruction?

4.13.5 [5] <§4.4> Modify Figure 4.25 to demonstrate an implementation of this new instruction.

4.13

4.13.1 We need some additional muxes to drive the data paths discussed in 4.13.3.

4.13.2 No functional blocks need to be modified.

4.13.3 There needs to be a path from the ALU output to data memory's write data port. There also needs to be a path from read data 2 directly to Data memory's Address input.

4.13.4 These new data paths will need to be driven by muxes. These muxes will require control wires for the selector.

4.13.5 Many possible solutions.

4.14 [5] <§4.4> For which instructions (if any) is the Imm Gen block on the critical path?

4.14 None: all instructions that use sign extend also use the register file, which is slower.

4.16 In this exercise, we examine how pipelining affects the clock cycle time of the processor. Problems in this exercise assume that individual stages of the datapath have the following latencies:

IF	ID	EX	MEM	WB
250ps	350ps	150ps	300ps	200ps

Also, assume that instructions executed by the processor are broken down as follows:

ALU/Logic	Jump/Branch	Load	Store
45%	20%	20%	15%

4.16.1 [5] <§4.6> What is the clock cycle time in a pipelined and non-pipelined processor?

4.16.2 [10] <§4.6> What is the total latency of an `lw` instruction in a pipelined and non-pipelined processor?

4.16.3 [10] <§4.6> If we can split one stage of the pipelined datapath into two new stages, each with half the latency of the original stage, which stage would you split and what is the new clock cycle time of the processor?

4.16.4 [10] <§4.6> Assuming there are no stalls or hazards, what is the utilization of the data memory?

4.16.5 [10] <§4.6> Assuming there are no stalls or hazards, what is the utilization of the write-register port of the "Registers" unit?

4.16

4.16.1 Pipelined: 350; non-pipelined: 1250

4.16.2 Pipelined: 1750; non-pipelined: 1250

4.16.3 Split the ID stage. This reduces the clock-cycle time to 300ps.

4.16.4 35%

4.16.5 65%

4.17 [10] <§4.6> What is the minimum number of cycles needed to completely execute `n` instructions on a CPU with a `k` stage pipeline? Justify your formula.

4.17 n + k − 1. Let's look at when each instruction is in the WB stage. In a k-stage pipeline, the 1st instruction doesn't enter the WB stage until cycle k. From that point on, at most one of the remaining n − 1 instructions is in the WB stage during every cycle.

This gives us a minimum of k + (n − 1) = n + k − 1 cycles.

4.18 [5] <§4.6> Assume that x11 is initialized to 11 and x12 is initialized to 22. Suppose you executed the code below on a version of the pipeline from Section

4.6 that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP instructions where necessary). What would the final values of registers x13 and x14 be?

```
addi   x11, x12, 5
add    x13, x11, x12
addi   x14, x11, 15
```

4.18 x13 = 33 and x14 = 36

4.19 [10] <§4.6> Assume that x11 is initialized to 11 and x12 is initialized to 22. Suppose you executed the code below on a version of the pipeline from Section 4.6 that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP instructions where necessary). What would the final values of register x15 be? Assume the register file is written at the beginning of the cycle and read at the end of a cycle. Therefore, an ID stage will return the results of a WB state occurring during the same cycle. See Section 4.8 and Figure 4.68 for details.

```
addi   x11, x12, 5
add    x13, x11, x12
addi   x14, x11, 15
add    x15, x11, x11
```

4.19 x15 = 54 (The code will run correctly because the result of the first instruction is written back to the register file at the beginning of the 5th cycle, whereas the final instruction reads the updated value of x1 during the second half of this cycle.)

4.20 [5] <§4.5> Add NOP instructions to the code below so that it will run correctly on a pipeline that does not handle data hazards.

```
addi   x11, x12, 5
add    x13, x11, x12
addi   x14, x11, 15
add    x15, x13, x12
```

4.20

```
addi x11, x12, 5
NOP
NOP
add x13, x11, x12
addi x14, x11, 15
NOP
add x15, x13, x12
```

4.24 [10] <§4.8> Which of the two pipeline diagrams below better describes the operation of the pipeline's hazard detection unit? Why?

Choice 1:

```
lw x11, 0(x12):      IF ID EX ME WB
add x13, x11, x14:    IF ID EX..ME WB
or x15, x16, x17:      IF ID..EX ME WB
```

Choice 2:

```
lw x11, 0(x12):    IF ID EX ME WB
add x13, x11, x14:  IF ID..EX ME WB
or x15, x16, x17:    IF..ID EX ME WB
```

4.24 The second one. A careful examination of Figure 4.59 shows that the need for a stall is detected during the ID stage. It is this stage that prevents the fetch of a new instruction, effectively causing the add to repeat its ID stage.