Chapter 2 Exercise Solutions Guide
Principles of Computer Systems

Abolfazl Naderi
avn@disroot.org

November 22, 2025

1 Question 1: Endianness

1.1 Problem Data

- Register x7 contains: `0xabcdef01`
- Stored at memory address: `0x10000000`

1.2 a) Big-Endian at Address 0x10000003

In big-endian format, the most significant byte (MSB) is stored at the lowest address.

Memory Layout:

Address	**Byte Value**	**Description**
`0x10000000`	`0xab`	MSB (bits [31:24])
`0x10000001`	`0xcd`	bits [23:16]
`0x10000002`	`0xef`	bits [15:8]
`0x10000003`	`0x01`	LSB (bits [7:0])

Visual Representation:

```
Value: 0xabcdef01

Big-Endian Storage:
    +-------------+-------------+-------------+-------------+
    |     0xab    |     0xcd    |     0xef    |     0x01    |
    +-------------+-------------+-------------+-------------+
    0x10000000    0x10000001    0x10000002    0x10000003
       (MSB)                                       (LSB)
```

Answer: `0x01`

1.3 b) Little-Endian at Address 0x10000003

In little-endian format, the least significant byte (LSB) is stored at the lowest address.

Memory Layout:

Address	**Byte Value**	**Description**
`0x10000000`	`0x01`	LSB (bits [7:0])
`0x10000001`	`0xef`	bits [15:8]
`0x10000002`	`0xcd`	bits [23:16]
`0x10000003`	`0xab`	MSB (bits [31:24])

Visual Representation:

```
Value: 0xabcdef01

Little-Endian Storage:
    +-------------+-------------+-------------+-------------+
    |    0x01     |    0xef     |    0xcd     |    0xab     |
    +-------------+-------------+-------------+-------------+
    0x10000000   0x10000001   0x10000002   0x10000003
       (LSB)                                     (MSB)
```

Answer: `0xab`

2 Question 2: Instruction Encoding

2.1 Problem Data

Instruction: `sw x15, 40(x19)`

This is an S-type instruction in RISC-V.

2.2 S-Type Format

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

2.3 Field Values

- **opcode**: `0100011` (store instruction)
- **funct3**: `010` (word, 32-bit)
- **rs1**: x19 = `10011`
- **rs2**: x15 = `01111`
- **immediate**: 40 = `000000101000`
  - imm[11:5] = `0000001`
  - imm[4:0] = `01000`

2.4 Binary Representation

```
0000001 | 01111 | 10011 | 010 | 01000 | 0100011
```

2.5 Hexadecimal Representation

Grouping into 4-bit chunks:

```
0000 0010 1111 1001 1010 0100 0010 0011
```

Machine Code: `0x027CA423`

3 Question 3: Bit Manipulation Operations

3.1 Problem Data

- x5 = 0x0000ABCD
- x6 = 0x12345678

3.2 a) `slli x7, x6, 1`

Shift left logical immediate: multiply by 2.

Calculation:

$$\texttt{x6} = \texttt{0x12345678}$$

$$\texttt{x7} = \texttt{x6} << 1 = \texttt{0x2468ACF0}$$

Answer: `x7 = 0x2468ACF0`

3.3 b) Two Instruction Sequence

Instructions:

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

Step 1: `slli x7, x5, 4`

$$\texttt{x7} = \texttt{0x0000ABCD} << 4 = \texttt{0x000ABCD0}$$

Step 2: `or x7, x7, x6`

$$\texttt{x7} = \texttt{0x000ABCD0} | \texttt{0x12345678}$$

Binary OR:

$$\begin{array}{r} \texttt{0000 0000 0000 1010 1011 1100 1101 0000} \\ | \; \texttt{0001 0010 0011 0100 0101 0110 0111 1000} \\ \hline \texttt{0001 0010 0011 1110 1111 1110 1111 1000} \end{array}$$

Answer: `x7 = 0x123EFEF8`

3.4 c) Two Instruction Sequence

Instructions:

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

Step 1: `srli x7, x5, 3`

$$\texttt{x7} = \texttt{0x0000ABCD} >> 3 = \texttt{0x0000157A}$$

Binary: `0xABCD` = `1010 1011 1100 1101`
After right shift by 3: `0001 0101 0111 1010` = `0x157A`
Step 2: `andi x7, x7, 0xFEF`

$$\texttt{x7} = \texttt{0x0000157A} \& \texttt{0x00000FEF}$$

Binary AND:

$$\begin{array}{r} \texttt{0001 0101 0111 1010} \\ \& \ \texttt{0000 1111 1110 1111} \\ \hline \texttt{0000 0101 0110 1010} \end{array}$$

Answer: `x7 = 0x0000056A`

4 Question 4: Bit Field Replacement

4.1 Problem

Write a RISC-V program with minimum instructions to replace bits [14:10] of register `x6` with bits [28:24] of register `x5`, keeping all other bits of `x6` unchanged.

4.2 Solution

Strategy:

1. Extract bits [28:24] from x5
2. Shift them to position [14:10]
3. Clear bits [14:10] in x6
4. OR the results together

Important Note: The mask `0xFFFF83FF` is too large for an immediate field in I-type instructions (which only support 12-bit immediates). We must use `lui` and `addi` to load it.

4.3 RISC-V Code (5 instructions)

```
srli x7, x5, 24         # Extract bits [28:24]
andi x7, x7, 0x1F       # Mask to keep only 5 bits
slli x7, x7, 10         # Shift to position [14:10]
lui  x8, 0xFFFF8        # Load 0xFFFF8000 into x8
addi x8, x8, 0x3FF      # x8 = 0xFFFF83FF (mask)
and  x6, x6, x8         # Clear bits [14:10] in x6
or   x6, x6, x7         # Insert new bits
```

Explanation:
The mask `0xFFFF83FF` in binary is:

`1111 1111 1111 1111 1000 0011 1111 1111`

Breaking it down:

- `lui x8, 0xFFFF8` loads `0xFFFF8000`
- `addi x8, x8, 0x3FF` adds `0x3FF`, giving `0xFFFF83FF`
- This clears bits [14:10] = positions 10, 11, 12, 13, 14

5 Question 5: Array Operations

5.1 Problem Data

- Array C base address is in register `x20`
- Elements are 32-bit integers (4 bytes each)
- Result stored in A (assume register `x5`)

5.2 a) `A = C[2] << 3`

Calculation:

- Offset for C[2] $= 2 \times 4 = 8$ bytes

Memory Layout:

Element	Address
C[0]	x20 + 0
C[1]	x20 + 4
C[2]	x20 + 8
C[3]	x20 + 12

RISC-V Code:

```
lw   x5, 8(x20)          # Load C[2] into x5
slli x5, x5, 3           # Shift left by 3 (multiply by 8)
```

5.3 b) `A = C[2] << n`

Assume `n` is in register `x6` (user input).

RISC-V Code:

```
lw  x5, 8(x20)           # Load C[2] into x5
sll x5, x5, x6           # Shift left by n
```

6 Question 6: Even/Odd Number Sorting

6.1 Problem

Read two integers. If both have the same parity (both even or both odd), print in ascending order. Otherwise, print the odd number first, then the even number.

6.2 RISC-V Assembly Code

```
    # Read first number
    li a7, 5                # syscall: read integer
    ecall
    mv x5, a0               # Store in x5

    # Read second number
    li a7, 5
    ecall
    mv x6, a0               # Store in x6

    # Check parity of x5
    andi x7, x5, 1          # x7 = 1 if x5 is odd, 0 if even

    # Check parity of x6
    andi x8, x6, 1          # x8 = 1 if x6 is odd, 0 if even

    # Compare parities
    xor x9, x7, x8          # x9 = 0 if same parity

    beqz x9, same_parity    # Branch if same parity

different_parity:
    # One is odd, one is even - print odd first
    bnez x7, print_x5_x6    # If x5 is odd

    # x6 is odd, x5 is even
    mv a0, x6
    li a7, 1
    ecall
    li a0, '\n'
    li a7, 11
    ecall
    mv a0, x5
    li a7, 1
    ecall
    j end

print_x5_x6:
    # x5 is odd, x6 is even
    mv a0, x5
    li a7, 1
    ecall
```

```
    li a0, '\n'
    li a7, 11
    ecall
    mv a0, x6
    li a7, 1
    ecall
    j end

same_parity:
    # Print in ascending order
    blt x5, x6, print_x5_first

    # x6 <= x5
    mv a0, x6
    li a7, 1
    ecall
    li a0, '\n'
    li a7, 11
    ecall
    mv a0, x5
    li a7, 1
    ecall
    j end

print_x5_first:
    mv a0, x5
    li a7, 1
    ecall
    li a0, '\n'
    li a7, 11
    ecall
    mv a0, x6
    li a7, 1
    ecall

end:
    li a7, 10                  # syscall: exit
    ecall
```

7 Question 7: Digit Count Classification

7.1 Problem

Read n integers and count how many are 1-digit, 2-digit, and 3-or-more-digit numbers.

7.2 RISC-V Assembly Code

```
    # Read count
    li a7, 5
    ecall
    mv x5, a0              # x5 = n (loop counter)

    # Initialize counters
    li x10, 0              # 1-digit count
    li x11, 0              # 2-digit count
    li x12, 0              # 3+ digit count

loop:
    beqz x5, print_results  # Exit loop when counter is 0

    # Read number
    li a7, 5
    ecall
    mv x6, a0

    # Get absolute value
    bgez x6, classify
    sub x6, x0, x6         # Negate if negative

classify:
    li x7, 10
    blt x6, x7, one_digit   # If x6 < 10

    li x7, 100
    blt x6, x7, two_digits  # If x6 < 100

    # Three or more digits
    addi x12, x12, 1
    addi x5, x5, -1        # Decrement counter
    j loop

one_digit:
    addi x10, x10, 1
    addi x5, x5, -1        # Decrement counter
    j loop

two_digits:
    addi x11, x11, 1
    addi x5, x5, -1        # Decrement counter
    j loop
```

```

print_results:
    # Print 1-digit count
    mv a0, x10
    li a7, 1
    ecall
    li a0, '\n'
    li a7, 11
    ecall

    # Print 2-digit count
    mv a0, x11
    li a7, 1
    ecall
    li a0, '\n'
    li a7, 11
    ecall

    # Print 3+ digit count
    mv a0, x12
    li a7, 1
    ecall

    # Exit
    li a7, 10
    ecall
```

Optimization Notes:

- Removed the `next` label for efficiency
- Each classification path decrements the counter and jumps directly to `loop`
- Reduced total jumps from 2 per iteration to 1 per iteration

8 Question 8: Date-Time Storage in 32-bit Register

8.1 Problem

Store date and time (year, month, day, hour, minute) in a single 32-bit register.

8.2 Bit Field Allocation

Field	**Bits**	**Size**	**Range**
Minute	[5:0]	6 bits	0-63 (need 0-59)
Hour	[10:6]	5 bits	0-31 (need 0-23)
Day	[15:11]	5 bits	0-31 (need 1-31)
Month	[19:16]	4 bits	0-15 (need 1-12)
Year	[31:20]	12 bits	0-4095

Visual Representation:

```
32-bit Register Layout:
+------------+-------+-----+------+--------+
|    Year    | Month | Day | Hour | Minute |
+------------+-------+-----+------+--------+
[31......20] [19.16] [15.11] [10.6]  [5..0]
  12 bits     4 bits  5 bits 5 bits  6 bits
```

8.3 a) Increment One Minute Function

Function Logic:

1. Increment minute
2. If minute = 60, set to 0 and increment hour
3. If hour = 24, set to 0 and increment day
4. If day exceeds month's days, set to 1 and increment month
5. If month = 13, set to 1 and increment year

RISC-V Code:

```
increment_minute:
    # Extract minute [5:0]
    andi t0, a0, 0x3F
    addi t0, t0, 1

    li t1, 60
    blt t0, t1, update_minute

    # Minute overflow
    li t0, 0
    srli t1, a0, 6
    andi t1, t1, 0x1F        # Extract hour
    addi t1, t1, 1

    li t2, 24
    blt t1, t2, update_time

    # Hour overflow
    li t1, 0
    srli t2, a0, 11
    andi t2, t2, 0x1F        # Extract day
    addi t2, t2, 1

    # Check day overflow (simplified: assume 31 days)
    li t3, 32
    blt t2, t3, update_time

    # Day overflow
    li t2, 1
    srli t3, a0, 16
    andi t3, t3, 0xF         # Extract month
    addi t3, t3, 1

    li t4, 13
    blt t3, t4, update_date

    # Month overflow
    li t3, 1
    srli t4, a0, 20          # Extract year
    addi t4, t4, 1
    slli t4, t4, 20
    slli t3, t3, 16
    or a0, t4, t3
    slli t2, t2, 11
    or a0, a0, t2
    slli t1, t1, 6
    or a0, a0, t1
    or a0, a0, t0
    ret

```

```
update_date:
    li t5, 0xFFF00000
    and a0, a0, t5          # Clear date/time
    slli t3, t3, 16
    or a0, a0, t3

update_time:
    li t5, 0xFFFF0000
    and a0, a0, t5
    slli t2, t2, 11
    or a0, a0, t2
    slli t1, t1, 6
    or a0, a0, t1

update_minute:
    li t5, 0xFFFFFFC0
    and a0, a0, t5
    or a0, a0, t0
    ret
```

8.4 b) Display Date-Time Function

Display format: `YYYY/MM/DD, HH:MM`

RISC-V Code:

```
display_datetime:
    mv s0, a0              # Save packed date-time

    # Extract and print year [31:20]
    srli a0, s0, 20
    li a7, 1               # Print integer
    ecall

    li a0, '/'
    li a7, 11              # Print character
    ecall

    # Extract and print month [19:16]
    srli a0, s0, 16
    andi a0, a0, 0xF
    li a7, 1
    ecall

    li a0, '/'
    li a7, 11
    ecall

    # Extract and print day [15:11]
    srli a0, s0, 11
    andi a0, a0, 0x1F
    li a7, 1
    ecall

    # Print comma and space
    li a0, ','
    li a7, 11
    ecall
    li a0, ' '
    ecall

    # Extract and print hour [10:6]
    srli a0, s0, 6
    andi a0, a0, 0x1F
    li a7, 1
    ecall

    li a0, ':'
    li a7, 11
    ecall

    # Extract and print minute [5:0]
    mv a0, s0
```

```
    andi a0, a0, 0x3F
    li a7, 1
    ecall

    ret
```