ECE-6913 Computer System Architecture
Homework 2

New York University, Fall 2024

Due on 09/28, 11:59 PM

Name: Raman Kumar Jha
NYU ID: N13866145

Problem 1: RISC-V Instructions

Given:

- Base addresses: A[0] in x27, B[0] in x30, C[0] in x31
- Variables: f = x5, g = x6, h = x7, i = x28, j = x29
- Assume 32-bit words (4 bytes each)

(a) Load Register x5 with content of A[10]

```
lw x5, 40(x27)     # Load word from A[10] = A[0] + 10*4 = base + 40 bytes
```

(b) Store contents of Register x5 into A[17]

```
sw x5, 68(x27)     # Store word to A[17] = A[0] + 17*4 = base + 68 bytes
```

(c) Add operands in x5 and x6, store result in x7

```
add x7, x5, x6     # x7 = x5 + x6
```

(d) Copy memory location: C[g] = A[i+j+31]

```
add x1, x28, x29   # x1 = i + j (temporary register)
addi x1, x1, 31    # x1 = i + j + 31
slli x1, x1, 2     # x1 = (i + j + 31) * 4 (word offset to byte offset)
add x1, x27, x1    # x1 = address of A[i+j+31]
lw x2, 0(x1)       # x2 = A[i+j+31] (load value)

slli x3, x6, 2     # x3 = g * 4 (word offset to byte offset)
add x3, x31, x3    # x3 = address of C[g]
sw x2, 0(x3)       # C[g] = x2 (store value)
```

(e) Implement C code in RISC-V

(i) f = g - A[B[9]]

```
lw x1, 36(x30)     # x1 = B[9] = B[0] + 9*4 = base + 36 bytes
slli x1, x1, 2     # x1 = B[9] * 4 (convert to byte offset)
add x1, x27, x1    # x1 = address of A[B[9]]
lw x1, 0(x1)       # x1 = A[B[9]]
sub x5, x6, x1     # f = g - A[B[9]]
```

(ii) f = g - A[C[8] + B[4]]

```
lw x1, 32(x31)     # x1 = C[8] = C[0] + 8*4 = base + 32 bytes
lw x2, 16(x30)     # x2 = B[4] = B[0] + 4*4 = base + 16 bytes
add x1, x1, x2     # x1 = C[8] + B[4]
slli x1, x1, 2     # x1 = (C[8] + B[4]) * 4 (convert to byte offset)
add x1, x27, x1    # x1 = address of A[C[8] + B[4]]
lw x1, 0(x1)       # x1 = A[C[8] + B[4]]
sub x5, x6, x1     # f = g - A[C[8] + B[4]]
```

(iii) A[i] = B[2i+1], C[i] = B[2i]

```
# A[i] = B[2i+1]
slli x1, x28, 1    # x1 = 2*i
addi x1, x1, 1     # x1 = 2*i + 1
slli x1, x1, 2     # x1 = (2*i + 1) * 4 (byte offset)
add x1, x30, x1    # x1 = address of B[2i+1]
lw x2, 0(x1)       # x2 = B[2i+1]

slli x3, x28, 2    # x3 = i * 4 (byte offset)
add x3, x27, x3    # x3 = address of A[i]
sw x2, 0(x3)       # A[i] = B[2i+1]

# C[i] = B[2i]
slli x1, x28, 1    # x1 = 2*i
slli x1, x1, 2     # x1 = 2*i * 4 (byte offset)
add x1, x30, x1    # x1 = address of B[2i]
lw x2, 0(x1)       # x2 = B[2i]

slli x3, x28, 2    # x3 = i * 4 (byte offset)
add x3, x31, x3    # x3 = address of C[i]
sw x2, 0(x3)       # C[i] = B[2i]
```

(iv) A[i] = 4B[i-1] + 4C[i+1]

```
# Load B[i-1]
addi x1, x28, -1   # x1 = i - 1
slli x1, x1, 2     # x1 = (i-1) * 4 (byte offset)
add x1, x30, x1    # x1 = address of B[i-1]
lw x1, 0(x1)       # x1 = B[i-1]
slli x1, x1, 2     # x1 = 4 * B[i-1]

# Load C[i+1]
addi x2, x28, 1    # x2 = i + 1
slli x2, x2, 2     # x2 = (i+1) * 4 (byte offset)
add x2, x31, x2    # x2 = address of C[i+1]
lw x2, 0(x2)       # x2 = C[i+1]
slli x2, x2, 2     # x2 = 4 * C[i+1]

# Compute and store result
add x1, x1, x2     # x1 = 4*B[i-1] + 4*C[i+1]
slli x3, x28, 2    # x3 = i * 4 (byte offset)
add x3, x27, x3    # x3 = address of A[i]
sw x1, 0(x3)       # A[i] = 4*B[i-1] + 4*C[i+1]
```

(v) f = g - A[C[4] + B[12]]

```
lw x1, 16(x31)     # x1 = C[4] = C[0] + 4*4 = base + 16 bytes
lw x2, 48(x30)     # x2 = B[12] = B[0] + 12*4 = base + 48 bytes
add x1, x1, x2     # x1 = C[4] + B[12]
```

```
slli x1, x1, 2     # x1 = (C[4] + B[12]) * 4 (convert to byte offset)
add x1, x27, x1    # x1 = address of A[C[4] + B[12]]
lw x1, 0(x1)       # x1 = A[C[4] + B[12]]
sub x5, x6, x1     # f = g - A[C[4] + B[12]]
```

Key RISC-V Instructions Used:

- `lw rd, imm(rs1)` - Load word from memory
- `sw rs2, imm(rs1)` - Store word to memory
- `add rd, rs1, rs2` - Add registers
- `addi rd, rs1, imm` - Add immediate
- `sub rd, rs1, rs2` - Subtract registers
- `slli rd, rs1, imm` - Shift left logical immediate

Problem 2

Assume the following register contents:
`x5 = 0x00000000AAAAAAAA`
`x6 = 0x1234567812345678`

---

a. For the register values shown above, what is the value of x7 for the following sequence of instructions?

```
srli x7, x5, 16
addi x7, x7, -128
srai x7, x7, 2
and x7, x7, x6
```

Solution

We'll execute the instructions step-by-step to find the final value of register `x7`.

1. `srli x7, x5, 16` (Shift Right Logical Immediate)
   This instruction shifts the value in `x5` right by **16 bits**. Zeros are shifted in from the left.
   - Initial `x5`: `0x00000000AAAAAAAA`
   - After shifting: `x7 = 0x0000000000000AAAA`
2. `addi x7, x7, -128` (Add Immediate)
   This instruction adds the immediate value **-128** (or `-0x80`) to `x7`.
   - Current `x7`: `0x0000000000000AAAA`
   - Calculation: `0xAAAA - 0x80 = 0xAA2A`
   - After addition: `x7 = 0x000000000000AA2A`
3. `srai x7, x7, 2` (Shift Right Arithmetic Immediate)
   This instruction shifts the value in `x7` right by **2 bits**. Since the number is positive, zeros are shifted in.
   - Current `x7`: `0x000000000000AA2A`
   - In binary, `0xAA2A` is `1010 1010 0010 1010`. Shifting right by 2 gives `0010 1010 1000 1010`, which is `0x2A8A`.
   - After shifting: `x7 = 0x0000000000002A8A`
4. `and x7, x7, x6` (Bitwise AND)
   This instruction performs a bitwise AND between `x7` and `x6`.

- Current `x7`: `0x0000000000002A8A`
- Value of `x6`: `0x1234567812345678`
- Calculation (`0x2A8A & 0x5678`):

```
                                    0010 1010 1000 1010
                  &0101 0110 0111 1000
-------------------------------------------------------
                                    0000 0010 0000 1000   →   0x0208
```

- After ANDing: `x7 = 0x0000000000000208`

The final value of `x7` is `0x0000000000000208`.

---

b. For the register values shown above, what is the value of x7 for the following sequence of instructions?

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

Solution

1. `slli x7, x6, 4` (Shift Left Logical Immediate)
   This instruction shifts the value in `x6` left by **4 bits**. Zeros are shifted in from the right. Shifting a hex number left by 4 bits moves each digit one position left and adds a `0` at the end.
   - Initial `x6`: `0x1234567812345678`
   - After shifting: `x7 = 0x2345678123456780`

The final value of `x7` is `0x2345678123456780`.

---

c. 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
```

Solution

1. `srli x7, x5, 3` (Shift Right Logical Immediate)
   This instruction shifts the value in `x5` right by **3 bits**.
   - Initial `x5`: `0x00000000AAAAAAAA`
   - Calculation: `0xAAAAAAAA >> 3 = 0x15555555`
   - After shifting: `x7 = 0x0000000015555555`
2. `andi x7, x7, 0xFEF` (Bitwise AND Immediate)
   This instruction performs a bitwise AND between `x7` and the immediate value `0xFEF`.
   - Current `x7`: `0x0000000015555555`
   - Calculation (`0x5555 & 0x0FEF`):

```
                                    0101 0101 0101 0101
                  &0000 1111 1110 1111
-------------------------------------------------------
                                    0000 0101 0100 0101   →   0x0545
```

   - After ANDing: `x7 = 0x0000000000000545`

The final value of `x7` is `0x0000000000000545`.

Problem 3: RISC-V Instruction Encoding

For each instruction, we identify its format and the values of its fields. All values are in decimal unless specified otherwise.

`add x5, x6, x7`

R-type

- `opcode`: `0x33` (OP)
- `rd`: 5 (`x5`)
- `func3`: 0 (add)
- `rs1`: 6 (`x6`)
- `rs2`: 7 (`x7`)
- `func7`: 0 (add)

`0000000 00111 00110 000 00101 0110011`

`0x007302B3`

`addi x8, x5, 512`

I-type

- `opcode`: `0x13` (OP-IMM)
- `rd`: 8 (`x8`)
- `func3`: 0 (addi)
- `rs1`: 5 (`x5`)
- `imm`: 512

`001000000000 00101 000 01000 0010011`

`0x20028413`

`ld x3, 128(x27)`

I-type

- `opcode`: `0x03` (LOAD)
- `rd`: 3 (`x3`)
- `func3`: 3 (ld)
- `rs1`: 27 (`x27`)
- `imm`: 128

`000010000000 11011 011 00011 0000011`

`0x080DB183`

`sd x3, 256(x28)`

S-type

- `opcode`: `0x23` (STORE)
- `imm[11:5]`: 8, `imm[4:0]`: 0 (Total imm = 256)
- `func3`: 3 (sd)
- `rs1`: 28 (x28)
- `rs2`: 3 (x3)

```
0001000 00011 11100 011 00000 0100011
0x103E3023
```

`beq x5, x6, ELSE` **(ELSE is 16 bytes away)**

B-type

- `opcode`: `0x63` (BRANCH)
- `func3`: 0 (beq)
- `rs1`: 5 (x5)
- `rs2`: 6 (x6)
- `imm`: 16

```
0000001 00110 00101 000 00000 1100011
0x02628063
```

`add x3, x0, x0`

R-type

- `opcode`: `0x33` (OP)
- `rd`: 3 (x3)
- `func3`: 0 (add)
- `rs1`: 0 (x0)
- `rs2`: 0 (x0)
- `func7`: 0 (add)

```
0000000 00000 00000 000 00011 0110011
0x000001B3
```

`auipc x3, 0xFFEFA`

U-type

- `opcode`: `0x17` (AUIPC)
- `rd`: 3 (x3)
- `imm`: `0xFFEFA`

```
11111111111011111010 00011 0010111
0xFFEFA197
```

`jal x3, ELSE` (ELSE is 16 bytes away)

J-type

- `opcode`: `0x6F` (JAL)
- `rd`: 3 (`x3`)
- `imm`: 16

`00000010000000000000 00011 1101111`

`0x010001EF`

Problem 4: RISC-V Assembly Sequences

(a) `A = C[0] << 16;`

Assuming `x5` holds `A` and `x11` holds the base address of array `C`. We also assume `C` contains 64-bit (doubleword) values. A minimal sequence is:

```
# x5 = A, x11 = base address of C
ld   x5, 0(x11)     # Load C[0] into x5 (A)
slli x5, x5, 16      # A = A << 16
```

(b) Bitfield Extraction and Insertion

This task is to extract bits 12 down to 7 from register `x3` and use them to replace bits 28 down to 23 in register `x4`. The following sequence uses the "AND/OR" method, which is clear and works for all initial values. We use `t0` and `t1` as temporary registers.

```
# Goal: x4[28:23] = x3[12:7]

# 1. Isolate the 6 source bits from x3
srli t0, x3, 7          # Shift bits 12:7 down to bits 5:0
andi t0, t0, 0x3F         # Clear upper bits, leaving only the 6 bits

# 2. Create a mask to clear the destination bits in x4
li   t1, 0x3F           # Load the 6-bit mask (111111)
slli t1, t1, 23         # Position the mask over bits 28:23
xori t1, t1, -1         # Invert mask to create a "clearing" mask
and  x4, x4, t1          # Clear bits 28:23 in x4

# 3. Shift the source bits to the destination and combine
slli t0, t0, 23         # Shift the isolated bits up to position 28:23
or   x4, x4, t0          # OR them into the cleared space in x4
```

(c) Implementing `not x5, x6`

The bit-wise `NOT` operation flips every bit in a register. This is equivalent to performing a bit-wise `XOR` with a register containing all 1s. In 64-bit two's complement, the number **`-1`** is represented as all 1s (`0xFFFFFFFFFFFFFFFF`). The `xori` instruction can be used to achieve this in a single instruction.

```
# Implements: not x5, x6
xori x5, x6, -1    # x5 = x6 XOR 0xFF...FF
```

Problem 5: RISC-V Branch and Jump Ranges

Given the Program Counter (PC) is at `0x60000000`.

(a) Range of the `jal` instruction

The `jal` (jump-and-link) instruction is a **J-type** instruction. It uses a 20-bit sign-extended immediate, which is shifted left by one bit to produce a 21-bit signed byte offset. This allows for jumps to word-aligned addresses.

The range of a 21-bit signed offset is from -2^{20} to $+2^{20} - 2$ bytes.

- 2^{20} bytes = 1 MiB = `0x100000` in hexadecimal.

The target address is calculated as `PC + offset`.

Minimum Address:

$$\texttt{0x60000000} - \texttt{0x100000} = \texttt{0x5FF00000}$$

Maximum Address:

$$\texttt{0x60000000} + (\texttt{0x100000} - 2) = \texttt{0x600FFFFE}$$

The address range reachable by `jal` is from `0x5FF00000` to `0x600FFFFE`.

(b) Range of the `beq` instruction

The `beq` (branch if equal) instruction is a **B-type** instruction. It uses a 12-bit sign-extended immediate, which is shifted left by one bit to produce a 13-bit signed byte offset.

The range of a 13-bit signed offset is from -2^{12} to $+2^{12} - 2$ bytes.

- 2^{12} bytes = 4096 bytes = 4 KiB = `0x1000` in hexadecimal.

The target address is calculated as `PC + offset`.

Minimum Address:

$$\texttt{0x60000000} - \texttt{0x1000} = \texttt{0x5FFFF000}$$

Maximum Address:

$$\texttt{0x60000000} + (\texttt{0x1000} - 2) = \texttt{0x60000FFE}$$

The address range reachable by `beq` is from `0x5FFFF000` to `0x60000FFE`.

Problem 6: Loop Analysis

The final value in register `x5` is **20**. The loop runs 10 times (for `x6` values 10 down to 1). In each iteration, `x5` is incremented by 2. Thus, the final value is $10 \times 2 = 20$.

(a) Equivalent C code

Assuming `x5` is an integer `acc` and `x6` is an integer `i`.

```
// acc is initialized to 0
// i is initialized to 10
while (i != 0) {
    i = i - 1;
    acc = acc + 2;
}
```

(b) Number of instructions executed

Let N be the initial value of register `x6`. The loop body consists of 4 instructions.

- The loop iterates N times. For each iteration, the `beq` fails and the three subsequent instructions execute. This accounts for $4 \times N$ instructions.
- After N iterations, `x6` is 0. The code jumps back to `LOOP`, and the `beq` is executed one final time, succeeding. This is 1 more instruction.

The total number of instructions executed is $\mathbf{4N + 1}$.

(c) Replacing `beq` with `blt`

The new code with `blt x6, x0, DONE` changes the loop condition to terminate if `x6 < 0`. The loop will now execute as long as `x6 >= 0`. If `x6` starts at 10, it will execute for values 10, 9, ..., 1, 0. This is a total of 11 iterations. The equivalent C code is:

```
// acc is initialized to 0
// i is initialized to 10
while (i >= 0) {
    i = i - 1;
    acc = acc + 2;
}
```

Problem 7: C to RISC-V Translation

(a) C to RISC-V Assembly Code

The C code to translate is:

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

Register mapping: `x5=a, x6=b, x7=i, x29=j, x10=base of D`. We assume `D` is an array of 64-bit values (8 bytes per element). The address of `D[4*j]` is `base + (4*j) * 8 = base + 32*j`.

```
# Register mapping: x5=a, x6=b, x7=i, x29=j, x10=base(D)
# Temporary registers: t0, t1

    mv x7, x0               # i = 0

OUTER_LOOP:
    bge x7, x5, END_OUTER   # if (i >= a) goto END_OUTER

    mv x29, x0              # j = 0

INNER_LOOP:
    bge x29, x6, END_INNER  # if (j >= b) goto END_INNER

    # --- Loop Body ---
    add  t0, x7, x29        # t0 = i + j
    slli t1, x29, 5        # t1 = j * 32 (byte offset)
    add  t1, x10, t1        # t1 = base address of D + offset
    sd   t0, 0(t1)          # D[4*j] = i + j

    # --- End of Loop Body ---
    addi x29, x29, 1        # j++
    jal  x0, INNER_LOOP

END_INNER:
    addi x7, x7, 1          # i++
    jal  x0, OUTER_LOOP

END_OUTER:
    # Program continues
```

(b) Instruction Count

Static Instruction Count: The assembly code above uses **12** unique instructions.

Dynamic Instruction Count: We can derive a formula for the number of executed instructions based on `a` and `b`.

- **Setup:** 1 instruction (`mv i, 0`)
- **Outer Loop:** Runs `a` times, with one final check. Contains 3 instructions outside the inner loop. Total = $(a+1)+3a$
- **Inner Loop:** Runs `b` times for each of the `a` outer loops. Contains 6 instructions. Total = $a \times (b+1) + a \times b \times 6$

Total executed instructions = $1 + (a+1) + 3a + a(b+1) + 6ab = \mathbf{2 + 5a + 7ab}$.

For `a = 10` and `b = 1`:

$$\text{Total} = 2 + 5(10) + 7(10)(1) = 2 + 50 + 70 = \mathbf{122}$$

The total number of RISC-V instructions executed is **122**.

Problem 8: Endianness

Given `x7 = 0x10000000` and the 64-bit value at that address is `0x1122334455667788`. The instructions `lb x6, 0(x7)` and `sd x6, 8(x7)` are executed. The question asks for the value at memory address `0x10000007`. The instructions do not modify the memory in the range `0x10000000` to `0x10000007`, so we only need to determine what byte was originally stored at that address based on the machine's endianness.

(a) What value is stored in `0x10000007` on a big-endian machine?

On a big-endian machine, the most significant byte (`0x11`) is stored at the lowest address (`0x10000000`). The bytes are stored in order from most to least significant.

- Address `0x10000000`: `0x11`
- Address `0x10000001`: `0x22`
- ...
- Address `0x10000007`: `0x88`

The value stored at address `0x10000007` is `0x88`.

(b) What value is stored in `0x10000007` on a little-endian machine?

On a little-endian machine, the least significant byte (`0x88`) is stored at the lowest address (`0x10000000`). The bytes are stored in reverse order.

- Address `0x10000000`: `0x88`
- Address `0x10000001`: `0x77`
- ...
- Address `0x10000007`: `0x11`

The value stored at address `0x10000007` is `0x11`.

Problem 9: Creating a 64-bit Constant

To create the 64-bit constant `0x1234567812345678` and store it in register `x10`, we must build it in pieces, as immediates are not large enough. A standard sequence uses `lui` to build the upper 32 bits, shifts them into place, and then builds and adds the lower 32 bits.

```
# Goal: Load 0x1234567812345678 into x10

# 1. Construct the upper 32 bits (0x12345678) in x10
lui  x10, 0x12345          # x10 = 0x000...012345000
addi x10, x10, 0x678        # x10 = 0x000...012345678

# 2. Shift the upper 32 bits into the high-order bits of x10
slli x10, x10, 32         # x10 = 0x1234567800000000

# 3. Construct the lower 32 bits (0x12345678) in a temp register
lui  t0, 0x12345           # t0 = 0x000...012345000
addi t0, t0, 0x678         # t0 = 0x000...012345678

# 4. Combine the upper and lower halves
or   x10, x10, t0          # x10 = 0x1234567812345678
```

Problem 10: Integer Overflow

We analyze overflow conditions for 64-bit signed arithmetic. The range of a 64-bit signed integer is from $S_{MIN} = -2^{63}$ to $S_{MAX} = 2^{63} - 1$. Register `x5` holds the value 128.

(a) `add x30, x5, x6`

Overflow occurs if the operands have the same sign and the result has a different sign. Since `x5` is positive, overflow can only happen if `x6` is also positive and their sum exceeds S_{MAX}.

$$128 + x6 > 2^{63} - 1$$

$$x6 > 2^{63} - 1 - 128$$

$$x6 > 2^{63} - 129$$

The range of values for `x6` that causes overflow is $(\mathbf{2^{63} - 129}, \mathbf{2^{63} - 1}]$.

(b) `sub x30, x5, x6`

This operation is `x5 - x6`. Overflow can occur if the operands have different signs. This is equivalent to `128 + (-x6)`. Positive overflow occurs if `-x6` is positive (i.e., `x6` is negative) and the result exceeds S_{MAX}.

$$128 - x6 > 2^{63} - 1$$

$$-x6 > 2^{63} - 129$$

$$x6 < -(2^{63} - 129) \implies x6 < -2^{63} + 129$$

This includes the most negative number, S_{MIN}. The boundary is $x6 = -2^{63} + 128$, where $128 - (-2^{63} + 128) = 2^{63}$, which is an overflow. The range of values for `x6` that causes overflow is $[\mathbf{-2^{63}}, \mathbf{-2^{63} + 128}]$.

(c) `sub x30, x6, x5`

This operation is `x6 - x5`. Negative overflow can occur if `x6` is negative and the result is less than S_{MIN}.

$$x6 - 128 < -2^{63}$$

$$x6 < -2^{63} + 128$$

The boundary is $x6 = -2^{63} + 127$, where $(-2^{63} + 127) - 128 = -2^{63} - 1$, which is an overflow. The range of values for `x6` that causes overflow is $[\mathbf{-2^{63}}, \mathbf{-2^{63} + 127}]$.

Problem 11: Computing's Energy Problem

Answers based on Professor Mark Horowitz's ISSCC 2014 Keynote.

1.1 How does Technology Scaling decrease the cost of Computing?

Technology scaling, the process of making transistors smaller, decreases the cost of computing by increasing the number of transistors that can be fabricated on a single silicon wafer. This dramatically lowers the manufacturing cost per transistor. This economic benefit makes it feasible to produce complex integrated circuits with billions of transistors for consumer products like smartphones and laptops, enabling the widespread use of powerful computing devices.

1.2 Why did scaling processor clock frequency become more difficult?

Scaling processor clock frequency became difficult around 2004 because of the end of Dennard scaling and the resulting "power wall". To increase frequency reliably, the supply voltage also had to be increased. Since power consumption scales with voltage squared and frequency ($P \propto V^2 f$), this led to a cubic increase in power. The resulting power density (power dissipated per unit area) became so high that chips could no longer be cooled effectively, making further frequency increases unsustainable. This made power dissipation the primary constraint on performance.

1.3 Why is Moore's Law slowing down? Why did Dennard Scaling end?

Moore's Law is slowing down due to fundamental physical and economic limits. As transistors approach atomic sizes, quantum effects become problematic, and the cost of building new fabrication plants (fabs) to produce smaller transistors has grown exponentially, making further scaling economically challenging.

Dennard Scaling ended because it was no longer possible to reduce transistor voltage proportionally with size. As transistors shrank, leakage current (power consumed even when the transistor is off) became a significant portion of total power consumption. To control this leakage, threshold voltages could not be lowered further, breaking the constant-power-density relationship that had fueled decades of performance growth.

1.4 What component of energy consumption by Memory is substantial?

The most substantial component of energy consumption related to memory is **data movement**. The energy required to move data over the long wires between the processor and external DRAM (i.e., driving the I/O interface) is significantly greater—by orders of magnitude—than the energy consumed by the actual computation performed on that data.

1.5 What solutions does Professor Mark Horowitz envision?

Professor Horowitz envisions a shift away from general-purpose computing towards more energy-efficient approaches:

- **Specialization:** Using domain-specific hardware accelerators (ASICs) designed for one task (e.g., machine learning, graphics) is vastly more energy-efficient than using a general-purpose CPU for the same task.
- **Near-Data Processing:** Reducing the energy cost of data movement by moving the computation closer to where the data is stored, for example, by integrating processing logic within memory chips.
- **Approximate Computing:** For applications that are inherently tolerant to small errors (like neural networks or signal processing), relaxing the requirement for perfect numerical precision can lead to large energy savings.

Problem 11: Open-Source vs. Proprietary ISAs

This response analyzes the 2015 debate between Professor David Patterson and Dave Christie and summarizes the technical arguments for an open-source Instruction Set Architecture (ISA) like RISC-V.

(1) Articulate your views on the topics debated in [2]. Justify your views.

After reviewing the debate, my view aligns strongly with Professor Patterson's position in favor of open-source ISAs. While Dave Christie's arguments regarding the established ecosystem of proprietary ISAs like x86 are valid from a business and legacy perspective, the open-source model presented by RISC-V represents a more compelling and sustainable future for computing.

My justification is based on three key points:

1. Democratization of Innovation: The most significant advantage of an open-source ISA is that it dramatically lowers the barrier to entry for hardware design. Christie argues that the ISA is a small part of the effort, but it is the fundamental, gatekeeping part. Without a license, which can be prohibitively expensive, small companies, startups, and academic researchers cannot even begin to design and tape out a custom processor. An open, royalty-free standard like RISC-V democratizes chip design, fostering a competitive and innovative ecosystem. This allows for the creation of highly specialized processors for emerging domains like AI/ML, IoT, and automotive applications, where the general-purpose nature of x86 is often inefficient.

2. Simplicity and Extensibility over Legacy Baggage: Christie correctly notes that backward compatibility is a powerful feature for the x86 ecosystem. However, this compatibility has turned the x86 ISA into an incredibly complex and bloated architecture, burdened by decades of legacy instructions. Patterson's argument for a simple, modular, clean-slate design is far more forward-looking. A small base integer ISA makes RISC-V easy to learn, implement, and verify. Its modular design, with optional standard extensions, allows designers to create processors with only the features they need, leading to smaller, lower-power, and more efficient cores. This "less is more" philosophy is a direct and necessary response to the power and complexity constraints that now limit modern computing.

3. A Thriving, Collaborative Ecosystem: The primary risk Christie highlights is the challenge of building a software ecosystem to compete with x86. While this is a significant hurdle, the open-source model has repeatedly proven its ability to overcome it. Linux, for example, built a robust ecosystem to challenge proprietary operating systems. Similarly, RISC-V is not just a specification; it is a community. The collaboration of hundreds of companies and universities under the umbrella of RISC-V International ensures the development of a shared set of tools (compilers, simulators, debuggers) and IP. This shared investment prevents fragmentation of the core standards while still allowing for the custom extensions that drive specialization, providing the best of both worlds.

In conclusion, while the inertia of proprietary ISAs is immense, the technical and economic benefits of an open, simple, and extensible standard are poised to drive the next wave of computing innovation.

(2) Review and summarize technical reasons for Open-Source ISAs in [3]

The UC Berkeley technical report outlines several key technical advantages of an open-source ISA, using RISC-V as its prime example. The core reasons are:

- **A Small, Simple Base ISA:** RISC-V is designed around a minimal base integer instruction set (RV32I or RV64I) with fewer than 50 instructions. This simplicity reduces the complexity and cost of designing a basic processor core, making it easier to verify and more suitable for educational purposes.
- **Modularity through Optional Extensions:** Functionality is added via optional standard extensions (e.g., 'M' for integer multiplication, 'F' for single-precision floating-point). A designer can mix and match these extensions to include only the hardware they need, which is ideal for creating optimized processors ranging from tiny, low-power microcontrollers to powerful application processors.
- **Support for Specialization and Customization:** The ISA encoding space explicitly reserves opcodes for custom, non-standard extensions. This allows companies to add specialized instructions for their unique applications (e.g., cryptography, machine learning) without conflicting with the standard ISA or causing fragmentation.
- **Clean-Slate Design:** Having been designed recently, RISC-V avoids the architectural baggage of older ISAs like x86. It was designed from the ground up to support modern design principles, such as pipelining, superscalar execution, and 64-bit addressing, without being constrained by legacy compatibility requirements.
- **Variable-Length Instruction Support:** The instruction encoding is designed to accommodate variable-length instructions. This is exemplified by the standard 'C' extension for compressed instructions, which reduces 32-bit instructions to 16-bit versions for common operations, significantly improving static and dynamic code density. This is a critical feature for memory-constrained embedded systems.
- **Clear Separation of ISA and Microarchitecture:** The RISC-V specification rigidly defines the instruction set and visible architectural state but intentionally avoids dictating the microarchitectural implementation. This gives designers complete freedom to innovate in hardware, allowing for a wide variety of compliant processors, from simple in-order pipelines to complex out-of-order superscalar designs.