# Indian Institute of Technology (IIT-Bombay)

## AUTUMN Semester, 2025
## COMPUTER SCIENCE AND ENGINEERING

### CS230: Digital Logic Design and Computer Architecture

### Tutorial - III

### Full Marks: 0

### Time allowed: $\infty$ hours

**1.** Consider a processor with 128 registers and an instruction set of size 20. Each instruction has five distinct fields, namely, opcode, two source register identifiers, one destination register identifier, and a sixteen-bit immediate value. Each instruction must be stored in memory in a byte-aligned fashion. If a program has 250 instructions, the amount of memory (in bytes) consumed by the program text is _____.

Solution:
$log_2128 = 7$, so 7 bits to address registers
$log_220 \approx 5$, so 5 bits to address opcode
So in total, $5 + 7 + 7 + 7 + 16 = 42bits$, considering 16 bit immediate field
Since it is byte-aligned, the size of an instruction must be a multiple of 8, so we pad 6 bits to make it 48 bits or 6 bytes.
So 250 instructions require, $250 * 6$ =1500 bytes

**2.** Assuming the system has 32-bit integers, answer the following questions.
Put the byte with the lowest address on the left (and the byte with the highest address on the right)

- Write the decimal number 13 in Binary (Base-2) as a 32-bit Big Endian Int
- Write the decimal number 13 in Hexadecimal (Base-16) as a 32-bit Big Endian Int
- Write the decimal number 13 in Hexadecimal (Base-16) as a 32-bit Little Endian Int
- Write the decimal number 13 in Hexadecimal (Base-16) as a 32-bit Little Endian Int

Solution :

- 00000000 00000000 00000000 00001101
- 0x 00 00 00 0D
- 00001101 00000000 00000000 00000000
- 0x 0D 00 00 00

**3.** For the following, assume that values A, B, and C reside in memory. Also assume that instruction operation codes are represented in 8 bits, memory addresses are 64 bits, register addresses are 6 bits and and data values are 32-bit integers.

Write down how many addresses, or names, appear in each instruction for the code to compute $C = A + B$, and what is the total code size for each of the following Instruction Set Architectures?

- Stack
- Accumulator
- Register-Memory
- Register (load-store)

Solution :

| Stack | Accumulator | Register-Memory | Register Load-Store |
|---|---|---|---|
| Push A | Load A | Load R1, A | Load R1, A |
| Push B | Add B | Add R3, R1, B | Load R2, B |
| Add | Store C | Store R3, C | Add R3, R1, R2 |
| Pop C | | | Store R3, C |

| **Arch. type** | **Stack** | **Accumulator** | **Register (register-memory)** | **Register (load-store)** |
|---|---|---|---|---|
| # Addresses | 3 | 3 | 7 (3 mem. + 4 reg.) | 9 (3 mem. + 6 reg.) |
| Code size | 224 bits | 216 bits | 240 bits | 260 bits |

**4.** We have 32-bit ISAs, 64-bit ISAs, well, there are also 16-bit ISAs. It's been two decades since the use of 64-bit ISAs.

a) What does 16-bit/32-bit/64-bit mean here?

b) Why not a leap towards 128-bit ISA? Yes/No? Give reasons.

Solution:

a) Size of register

b) Recall: How is memory addressed?

- 32-bit system can address $2^{32}$ bytes of memory = 4 Gigabytes (GB)
- In mid-2000s(yeah those white computers!), this became a challenge
- 64-bit system can address $2^{64}$ bytes of memory = 16 Exabytes(EB)
- 16 Exabytes = 18.4 million Terabytes - you can access astronomical sizes of RAMs
- So, moving to 128-bit ISA doesn't make any sense, when 64-bit is a sweet spot

**5.** The value represented by the hexadecimal number 4B45 5942 4F41 5244 is to be stored in an aligned 64-bit double word. The memory is byte-addressed.

a) Write the value to be stored using Big Endian byte order.

b) Write the value to be stored using Little Endian byte order.

c) What are the hexadecimal values of all misaligned 2-byte words that can be read from the given 64-bit double word when stored in Big Endian byte order?

d) What are the hexadecimal values of all misaligned 4-byte words that can be read from the given 64-bit double word when stored in Little Endian byte order?

Solution:

a) In Big Endian, the most significant byte (4B) is stored at the lowest memory address. The bytes are stored in the order they are read.

| Memory Address Offset | Hex Value |
|---|---|
| +0 | 4B |
| +1 | 45 |
| +2 | 59 |
| +3 | 42 |
| +4 | 4F |
| +5 | 41 |
| +6 | 52 |
| +7 | 44 |

b) In Little Endian, the least significant byte (44) is stored at the lowest memory address. The byte order is reversed.

| Memory Address Offset | Hex Value |
|---|---|
| +0 | 44 |
| +1 | 52 |
| +2 | 41 |
| +3 | 4F |
| +4 | 42 |
| +5 | 59 |
| +6 | 45 |
| +7 | 4B |

c) A 2-byte word is misaligned if its starting address is odd, ie, not divisible by 2(offsets 1, 3, 5). We read the bytes in the order they appear in the Big Endian storage.
Start at offset +1: Reads bytes at (+1, +2) - 45 59
Start at offset +3: Reads bytes at (+3, +4) - 42 4F
Start at offset +5: Reads bytes at (+5, +6) - 41 52

The hexadecimal values of the misaligned 2-byte words are 4559, 424F, and 4152

d) A 4-byte word is misaligned if its starting address is not a multiple of 4 (offsets 1, 2, 3). For Little Endian, we read the bytes from memory and then reverse them to interpret the value correctly.

Start at offset +1:
Bytes read from memory: 52 41 4F 42
Interpreted value (reversed): 424F4152

Start at offset +2:
Bytes read from memory: 41 4F 42 59
Interpreted value (reversed): 59424F41
Start at offset +3:
Bytes read from memory: 4F 42 59 45
Interpreted value (reversed): 4559424F
The hexadecimal values of the misaligned 4-byte words are 424F4152, 59424F41, and 4559424F

e) Consider the case of a processor with an instruction length of 12 bits and with 32 general-purpose registers, so the size of the address fields is 5 bits. Is it possible to have instruction encodings for the following?

a) 3 two-address instructions
b) 30 one-address instructions
c) 45 zero-address instructions

Solution:

(i) Encoding the 3 Two-Address Instructions

Format:

$$[Opcode][Reg1(5bits)][Reg2(5bits)]$$

Operand Bits: The two register fields use 5 + 5 = 10 bits
Opcode Bits: This leaves 12 - 10 = 2 bits for the opcode

With 2 bits, we can create $2^2 = 4$ unique opcodes for two-address instructions. The request is for 3 instructions, which fit comfortably.
Let's assign the first 2 bits of the instruction as the primary opcode identifier:

00: Two-address instruction 1
01: Two-address instruction 2
10: Two-address instruction 3

11: This will be our escape code. When the decoder sees 11 in the first two bits, it knows it's not a two-address instruction and must look at more bits.

(ii) Encoding the 30 One-Address Instructions
All of these instructions must start with our escape code 11.
Format:

$$[11][OpcodeExtension][Reg1(5bits)]$$

Operand Bits: The single register field uses 5 bits.
Opcode Bits: This leaves 12 - 5 = 7 bits for the total opcode. Since the first 2 bits are fixed as 11, we have 7 - 2 = 5 bits for our opcode extension.

With these 5 extension bits, we can create 2 = 32 unique opcodes for one-address instructions. The request is for 30 instructions, which also fits.

We can use another escape code here. Let's say we use 30 of the 32 possible patterns for the one-address instructions, leaving 2 patterns free to signal that a zero-address instruction is coming next.

(iii) Encoding the 45 Zero-Address Instructions

These instructions must start with the escape codes from the previous steps. Let's say we use one of the two remaining one-address opcode patterns as the escape.

Format:

$$[11][1 - AddrEscapeCode(5bits)][OpcodeExtension]$$

Operand Bits: There are no register fields, so 0 bits.
Opcode Bits: The total length is 12 bits. The first escape 11 uses 2 bits. The second escape (the

one-address escape code) uses 5 bits. This leaves 12 - 2 - 5 = 5 bits for the final opcode extension.

With these remaining 5 bits, we can create $2 = 32$ unique opcodes for zero-address instructions. So, not possible for 45 zero-address instructions.

f) Consider a **32-bit hypothetical CPU** which supports **1-word long instructions** stored in a **32KB memory**. Each instruction contains:

```
[ opcode | reg1 | reg2 | reg3 | immediate ]
```

- The opcode field is **6 bits**.
- There are **3 register operands**.
- Each register field must encode **64 registers**.
- The remaining bits go to the **immediate constant field**.

a) What is the size of the immediate field?

b) What is the **largest unsigned constant** that can be represented in this instruction format?

**Step 1: Word and instruction size.** Word size = 32 bits. Instruction size = 1 word = 32 bits. **Step 2: Field sizes.**

- Opcode = 6 bits (given).
- Each register field must encode 64 registers: $\lceil \log_2 64 \rceil = 6$ bits.
- Three register fields $\Rightarrow 3 \times 6 = 18$ bits.

**Step 3: Immediate field size.**

$$\text{Immediate size} = 32 - (6 + 18) = 32 - 24 = \boxed{8 \text{ bits}}$$

**Step 4: Largest unsigned constant.** An $n$-bit immediate can represent unsigned values from 0 to $2^n - 1$:

$$0 \text{ to } 2^8 - 1 = 255$$

$$\Rightarrow \boxed{255}$$

g) A hypothetical CPU supports **64 opcodes**, **256 registers**, and **32K memory cells**. Every instruction is fixed-format:

```
[ opcode | reg1 | reg2 | mem | immediate14 ]
```

A program contains **200 instructions**. Instructions must be stored *aligned to the memory cell size*. **Note:** The immediate field is explicitly specified to be **14 bits**.

a) Compute the *field widths* using $\lceil \log_2 N \rceil$ and hence the ***instruction size*** *in bits*.

b) For each memory organization, give the ***program size*** and ***internal fragmentation*** (padding) **per instruction** and **in total**:

(i) **Byte-addressable** memory (cell = 8 bits),

(ii) **Word-addressable** memory (cell = 16 bits),

(iii) **Cell size equals the instruction size**.

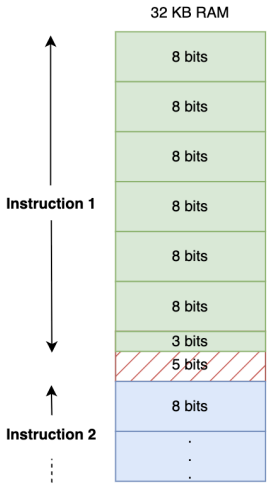
a) **Field widths and instruction size.**

| – | **opcode** | **reg1** | **reg2** | **mem** | **immediate14** |
|---|---|---|---|---|---|
| Choices (N) | 64 | 256 | 256 | $32K$ | 14 |
| $\lceil \log_2 N \rceil$Bits used | 6 | 8 | 8 | 15 | 14 |

$$\Rightarrow \text{Instruction size } = 6 + 8 + 8 + 15 + 14 = \boxed{51 \text{ bits}}$$

(i) **Byte-addressable memory (cell = 8 bits).**

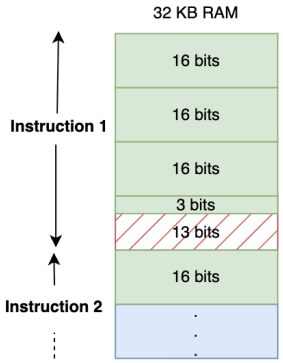
- Instruction length = 51 bits.
- One memory cell stores = 8 bits.
- Number of cells per instruction = $\lceil 51/8 \rceil = 7$ bytes.
- Therefore, program size = $200 \times 7$ bytes = $\boxed{1400 \text{ bytes}}$.
- Actual allocated per instruction = $7 \times 8 = 56$ bits.
- Waste per instruction = $56 - 51 = \boxed{5 \text{ bits}}$.
- Total waste for 200 instructions = $200 \times 5 = 1000$ bits = $\boxed{125 \text{ bytes}}$.




Byte-addressable case: each instruction uses 7 bytes, last 5 bits wasted.

(ii) **Word-addressable memory (cell = 16 bits).**

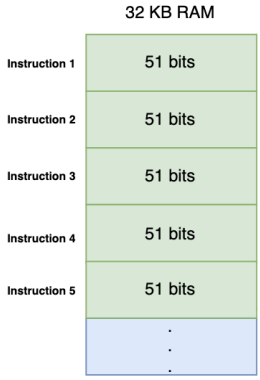
- Instruction length = 51 bits.
- One memory cell stores = 16 bits.
- Number of cells per instruction = $\lceil 51/16 \rceil = 4$ words.
- Each word = 16 bits, so allocated per instruction = $4 \times 16 = 64$ bits.
- Waste per instruction = $64 - 51 =$ $\boxed{13 \text{ bits}}$.
- Program size = $200 \times 4$ words = 800 words = $\boxed{1600 \text{ bytes}}$.
- Total waste = $200 \times 13$ bits = 2600 bits = $\boxed{325 \text{ bytes}}$.




Word-addressable case: each instruction uses 4 words (8 bytes), last 13 bits wasted.

(iii) **Cell size = instruction size (cell = 51 bits).**

- Instruction length = 51 bits.
- One cell = 51 bits (exact match).
- Cells per instruction = 1.
- Program size = $200 \times 51 = 10{,}200$ bits = $\boxed{1275 \text{ bytes}}$.
- Waste per instruction = 0, total waste = 0.




Exact-fit case: one cell holds the whole instruction, no waste.

**Summary Table.**

| Case | Cells/Instr | Program Size | Waste/Instr (Total) |
|---|---|---|---|
| Byte (8 b) | 7 bytes | 1400 B | 5 b (125 B) |
| Word (16 b) | 4 words = 8 B | 1600 B | 13 b (325 B) |
| Cell = 51 b | 1 cell | 1275 B | 0 |