

Assignment

CSE 112 Computer Organization

Introduction and Instructions

- This will be a group assignment in groups of 3 students. Each student in the group will be marked separately. Therefore try to make sure that work is roughly divided equally among all the members of the group.
- In this assignment, you will have to design and implement a custom assembler and a custom simulator for a given ISA.
- You are not restricted to any programming language. However, your program must read from stdin and write to stdout.
- You must use GitHub to collaborate. You must track your progress via **git**.
- The automated testing infrastructure assumes that you have a working Linux-based shell. For those who are using Windows, you can either use a VM or WSL.
- **TAs will conduct a separate session to explain the whole assignment.** They will also show you the sample solution code and will explain to you how to run the automated testing scripts.
- Start the assignment early and ask the queries well in advance. Do not expect any reply on weekends and 10 PM - 6 AM on working days. Do not escalate your query to instructors directly. **Write any queries you have in the comments section.** Wait at least 24 hours before any reply to your comment. If there is no reply then you can mail it to the content TAs (Nikhil Kushwah , Jatin Agarwal , Sunishka Sharma , Rishabh Oberoi), if still there is no response, then mail the TFs, and if still there is no response, then mail the instructors for clarification.
- **No last-minute deadline extensions will be considered whatsoever.** This includes but is not limited to connectivity issues, one group member is not working or not cooperating, group member is/are affected by covid etc. The duration of the deadline is sufficient enough to complete the assignment.
- Commit your code to the repository periodically to prevent any loss of your code due to system failures or any other issues. In case of system failures of all the members of the group, your last committed code on github before the deadline will be considered for evaluation.

QUESTION DESCRIPTION:

There are a total of **five** questions in this assignment:

1. Designing and Implementing the assembler.
2. Designing and Implementing the simulator.
3. Extending the functionality of the assembler-simulator set-up to handle simple floating-point computations.
4. A **bonus** question based on the simulator. **The bonus will be worth 3%.**
5. A **bonus** question on memory and ISA. **The bonus will be worth 7%.**

TEST CASES:

We will release some test cases with the assignment so that you can test your implementations. During the evaluations, a superset of these test cases would be provided to you, on which you will get graded.

DEADLINES:

You will have two deadlines for this assignment:

1. **The mid evaluation:**
 - a. By this deadline you must have the assembler ready.
 - b. You will be tested mostly on the test cases already provided to you with the assignment.
 - c. However, we might add some other test cases as well.
 - d. **You will only be evaluated on the assembler.** (20%)
 2. **The final evaluation:**
 - a. By this deadline you must have both the assembler and the simulator ready(70%).
 - b. You should also have completed Q3(10%).
 - c. You will be evaluated on a much larger set of test cases this time.
 - d. You will also be evaluated on the bonus question at this stage.
- ❖ **The mid evaluation will be worth 20% of your final assignment grade. The final evaluation will be worth the rest 80% of your final assignment grade. The bonus will be worth 10% making the total 110%.**

GRADING:

1. **Q1** and **Q2** are **mandatory** questions.
2. In Q1 you will have to make an assembler.
3. In Q2 you have to make a simulator for which detailed information is mentioned in the respective questions.
4. Q3 is an extension of the functionality of the assembler- simulator set-up that you build.
5. Q4 and Q5 are **bonus questions**.

- **For Q1 and Q2:** Grading will be based on the number of test cases that your program passes.
 - a. Assembler: The test cases are divided into 3 sets:
 - i. ErrorGen: These tests are supposed to generate errors
 - ii. simpleBin: These are simple test cases which are supposed to generate a binary.
 - iii. hardGen: These are hard test cases which are supposed to generate a binary.
 - b. Simulator: The test cases are divided into 2 sets:
 - i. simpleBin: These are simple test cases which are supposed to generate a trace.
 - ii. hardGen: These are hard test cases which are supposed to generate a trace.

→ ***The TA will grade the errorGen cases manually.***

- **For Q3:**
 - a. Assembler: The test cases are divided into 3 sets:
 - ErrorGen: These tests are supposed to generate errors
 - simpleBin: These are simple test cases which are supposed to generate a binary.
 - b. Simulator: The test cases are divided into 2 sets:
 - simpleBin: These are simple test cases which are supposed to generate a trace.

→ ***The TA will grade the errorGen cases manually.***

For Q4:

- For the bonus question, you need to generate some graphs which you must show to the TA during the final evaluation.
- The test cases for the bonus question will be provided to you separately well before the date of the deadline.
- Till then you can make some new test cases to try this on your own.

For Q5:

- For the bonus question, you need a basic command line interface which you must show to the TA during the final evaluation.
- The prompts or the format of the output (how it is printed) does not matter as long as the outputs are correct.
- Test cases for the bonus question will not be provided to you, the testing will be done by your TAs during your demo as you run the command line interface.
- In the meantime, you can make some new test cases to try this on your own.

EVALUATION PROCEDURE:

1. The date for the demo of the mid and final evaluation will be announced in due time.
2. On the day of your demo, a compressed archive of all tests will be shared with you on the google classroom. This archive will include other test cases as well which will not be provided to you beforehand.
3. On the day of evaluation, you must
 - a. Prove that you are not running code written after the deadline by running “git log HEAD” which prints the date and time of the commit pointed to by the HEAD. You must also run “git status” to show that you don’t have any uncommitted changes.
 - b. Prove the integrity of the tests archive by computing the sha256sum of the archive. To compute the checksum, you can run “**sha256sum**
i. <path/to/the/archive>”. The TA will then match the checksum to verify the integrity.
4. Then you can extract the archive and replace the “automatedTesting/tests” directory.
5. Then you need to execute the automated testing infrastructure, which will run all the tests and finally print your score.
6. The TA will verify the correctness for the test cases which are supposed to generate errors. You do not need to run these tests by yourself. The testing infrastructure will do this automatically for you.

PLAGIARISM

1. Any copying of code from your peers or from the internet will invoke institute Plagiarism policy.
2. Provide proper references if you're taking your code from some other resource. Needless to say, if the said code is the main part of the assignment, You will be awarded 0 marks.
3. If you are found indulging in any bad practice to circumvent the above mentioned evaluation procedure, you will be awarded 0 marks and institute plagiarism policy will be applied.

Assignment Description

ISA description:

Consider a 16 bit ISA with the following instructions and opcodes, along with the syntax of an assembly language which supports this ISA.

The ISA has 6 encoding types of instructions. The description of the types is given later.

Opcode	Instruction	Semantics	Syntax	Type
10000	Addition	Performs $\text{reg3} = \text{reg1} + \text{reg2}$. If the computation overflows, then the overflow flag is set	add reg1 reg2 reg3 what will be written to the reg3 in case of overflow?	A
10001	Subtraction	Performs $\text{reg3} = \text{reg1} - \text{reg2}$. In case $\text{reg2} > \text{reg1}$, 0 is written to reg3 and overflow flag is set.	sub reg1 reg2 reg3	A
10010	Move Immediate	Performs $\text{reg1} = \$\text{Imm}$ where Imm is a 8 bit value.	mov reg1 \$Imm	B
10011	Move Register	Performs $\text{reg2} = \text{reg1}$.	mov reg1 reg2	C
10100	Load	Loads data from mem_addr into reg1.	ld reg1 mem_addr	D
10101	Store	Stores data from reg1 to mem_addr.	st reg1 mem_addr	D
10110	Multiply	Performs $\text{reg3} = \text{reg1} \times \text{reg2}$. If the computation overflows, then the overflow flag is set.	mul reg1 reg2 reg3	A

10111	Divide	Performs reg3/reg4. Stores the quotient in R0 and the remainder in R1.	div reg3 reg4	C
11000	Right Shift	Right shifts reg1 by \$Imm, where \$Imm is an 8 bit value.	rs reg1 \$Imm	
11001	Left Shift	Left shifts reg1 by \$Imm, where \$Imm is an 8 bit value.	ls reg1 \$Imm	B
11010	Exclusive OR	Performs bitwise XOR of reg1 and reg2. Stores the result in reg3.	xor reg1 reg2 reg3	A

11011	Or	Performs bitwise OR of reg1 and reg2. Stores the result in reg3.	or reg1 reg2 reg3	A
11100	And	Performs bitwise AND of reg1 and reg2. Stores the result in reg3.	and reg1 reg2 reg3	A
11101	Invert	Performs bitwise NOT of reg1. Stores the result in reg2.	not reg1 reg2	C
11110	Compare	Compares reg1 and reg2 and sets up the FLAGS register.	cmp reg1 reg2	C

11111	Unconditional Jump	Jumps to mem_addr, where mem_addr is a memory address.	jmp mem_addr	E
0110 0	Jump If Less Than	Jump to mem_addr if the less than flag is set (less than flag = 1), where mem_addr is a memory address.	jlt mem_addr	E
0110 1	Jump If Greater Than	Jump to mem_addr if the greater than flag is set (greater than flag = 1), where mem_addr is a memory address.	jgt mem_addr	E
01111	Jump If Equal	Jump to mem_addr if the equal flag is set (equal flag = 1), where mem_addr is a memory address.	je mem_addr	E
01010	Halt	Stops the machine from executing until reset	hlt	F

where reg(x) denotes register, mem_addr is a memory address (must be an 8-bit binary number), and Imm denotes a constant value (must be an 8-bit binary number). The ISA has 7 general purpose registers and 1 flag register. The ISA supports an address size of 8 bits, which is double byte addressable. Therefore, each address fetches two bytes of data. This results in a total address space of 512 bytes. **This ISA only supports whole number arithmetic.** If the subtraction results in a negative number; for example “3 - 4”, the reg value will be set to 0 and overflow bit will be set. All the representations of the number are hence unsigned.

The registers in assembly are named as R0, R1, R2, ... , R6 and FLAGS. Each register is 16 bits.

Note: “mov reg \$Imm”: This instruction copies the Imm(8bit) value in the register’s lower 8 bits. The upper 8 bits are zeroed out.

Example:

Suppose R0 has 1110_1010_1000_1110 stored, and **mov R0 \$13** is executed. The final value of R0 will be 0000_0000_0000_1101.

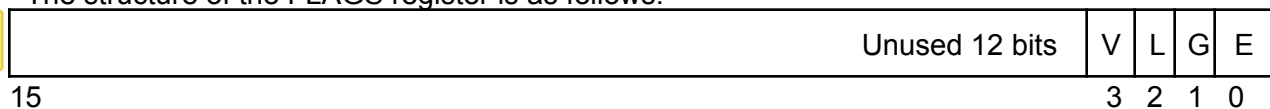
FLAGS semantics

The semantics of the flags register are:

- Overflow (V): This flag is set by `add`, `sub` and `mul`, when the result of the operation overflows. This shows the overflow status for the last executed instruction.
- Less than (L): This flag is set by the "`cmp reg1 reg2`" instruction if `reg1 < reg2`
- Greater than (G): This flag is set by the "`cmp reg1 reg2`" instruction if the value of `reg1 > reg2`
- Equal (E): This flag is set by the "`cmp reg1 reg2`" instruction if `reg1 = reg2`

The default state of the FLAGS register is all zeros. If an instruction does not affect the FLAGS register, then the state of the FLAGS register is reset to 0 upon the execution.

The structure of the FLAGS register is as follows:



The only operation allowed in the FLAGS register is "`mov reg1 FLAGS`", where `reg1` can be any of the registers from `R0` to `R6`. This instruction reads FLAGS register and writes the data into `reg1`. All other operations on the FLAGS register are prohibited.

The `cmp` instruction can implicitly write to the FLAGS register. Similarly, conditional jump instructions can implicitly read the FLAGS register.

Example:

`R0` has 5, `R1` has 10

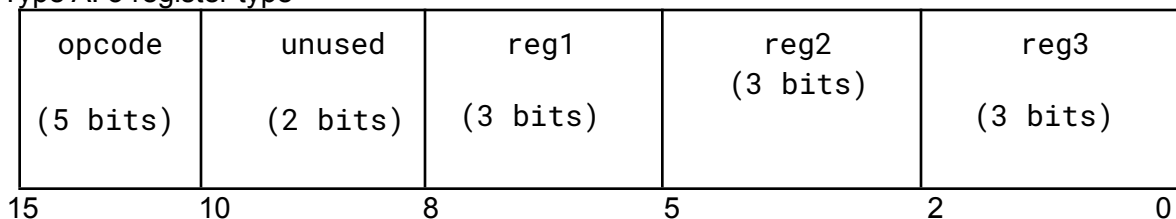
Implicit write: `cmp R0 R1` will set the L (less than) flag in the FLAGS register. Implicit read:

`jlt 10001001` will read the FLAGS register and figure out that the L flag was set, and then jump to address 10001001.

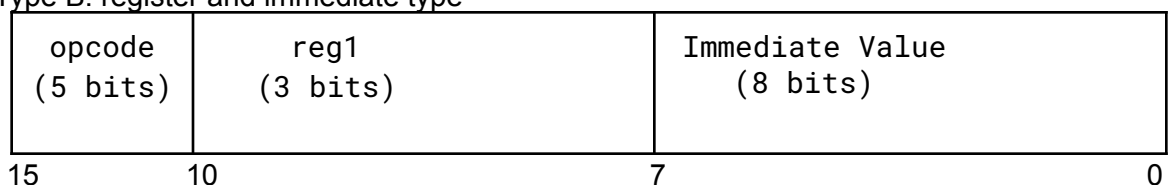
Binary Encoding

The ISA has 6 types of instructions with distinct encoding styles. However, each instruction is of 16 bits, regardless of the type.

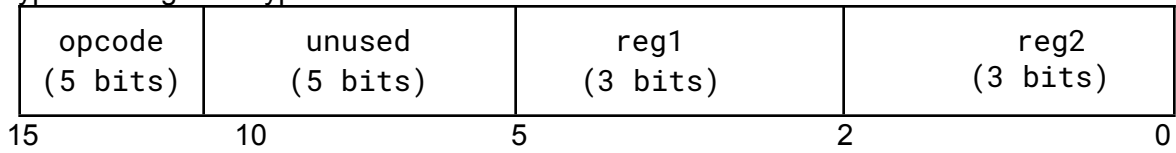
- Type A: 3 register type



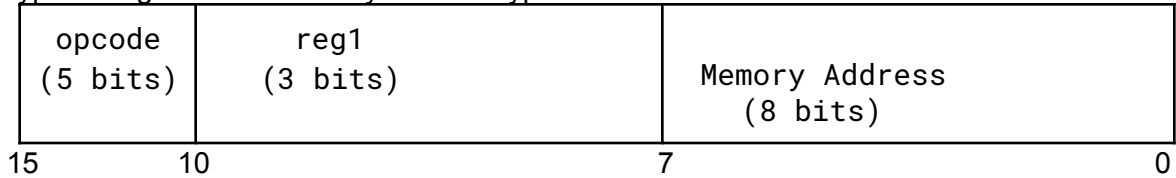
- Type B: register and immediate type



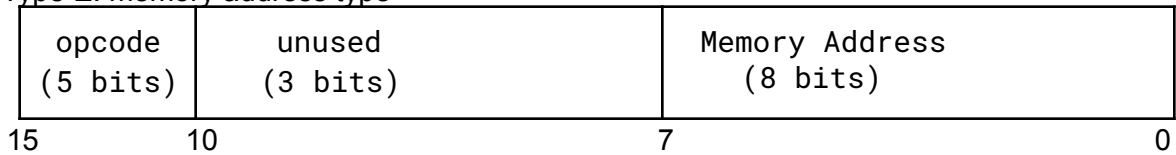
- Type C: 2 registers type



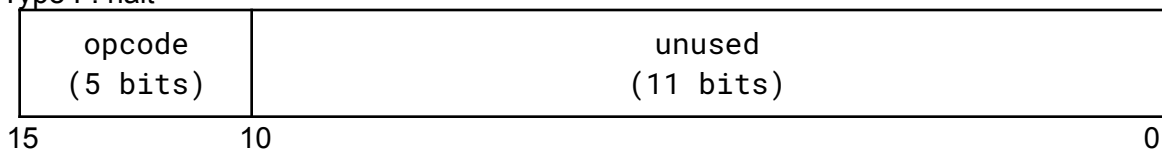
- Type D: register and memory address type



- Type E: memory address type



- Type F: halt



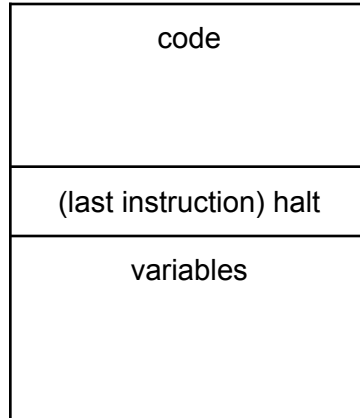
Binary representation for the register are given as follows:-

Register	Address
R0	000
R1	001
R2	010
R3	011
R4	100
R5	101
R6	110
FLAGS	111

Executable binary syntax

The machine exposed by the ISA starts executing the code provided to it in the following format, until it reaches `hlt` instruction. There can only be one `hlt` instruction in the whole program, and it must be the last instruction. The execution starts from the 0th address. The ISA follows von-neumann architecture with a unified code and data memory.

The variables must be allocated in the binary in the program order.



Questions:

Q1: Assembler:

Program an assembler for the aforementioned ISA and assembly. The input to the assembler is a text file containing the assembly instructions. Each line of the text file may be of one of 3 types:

- Empty line: Ignore these lines
- A label
- An instruction
- A variable definition

Each of these entities have the following grammar:

- The syntax of all the supported instructions is given above. The fields of an instruction are whitespace separated. The instruction itself might also have whitespace before it. An instruction can be one of the following:
 - The opcode must be one of the supported mnemonic.
 - A register can be one of R0, R1, ... R6, and FLAGS.
 - A `mem_addr` in jump instructions must be a label.
 - A `Imm` must be a whole number ≤ 255 and ≥ 0 .
 - A `mem_addr` in load and store must be a variable.
- A label marks a location in the code and must be followed by a colon (:). No spaces are allowed between label name and colon(:)
- A variable definition is of the following format:
`var xyz`
which declares a 16 bit variable called xyz. This variable name can be used in place of `mem_addr` fields in load and store instructions. **All variables must be defined at the very beginning of the assembly program.**

The assembler should be capable of:

1. Handling all supported instructions
2. Handling labels
3. Handling variables
4. Making sure that any illegal instruction (any instruction (or instruction usage) which is not supported) results in a syntax error. In particular you must handle:
 - a. Typos in instruction name or register name
 - b. Use of undefined variables
 - c. Use of undefined labels
 - d. Illegal use of FLAGS register
 - e. Illegal Immediate values (more than 8 bits)
 - f. Misuse of labels as variables or vice-versa
 - g. Variables not declared at the beginning
 - h. Missing `hlt` instruction
 - i. `hlt` not being used as the last instruction

You need to generate distinct readable errors for all these conditions. If you find any other illegal usage, you are required to generate a "General Syntax Error". **The assembler must print out all these errors.**

5. If the code is error free, then the corresponding binary is generated. The binary file is a text file in which each line is a 16bit binary number written using 0s and 1s in ASCII. The assembler can write less than or equal to 256 lines.

Input/Output format:

- The assembler must read the assembly program as an input text file (stdin).
- The assembler must generate the binary (if there are no errors) as an output text file (stdout).
- The assembler must generate the error notifications along with line number on which the error was encountered (if there are errors) as an output text file (stdout). **In case of multiple errors, the assembler may print any one of the errors.**

Example of an assembly program

```
var X
mov R1 $10
mov R2 $100
mul R3 R1 R2
st R3 X
hlt
```

The above program will be converted into the following machine code

```
1001000100001010
1001001001100100
1011000011001010
1010101100000101
0101000000000000
```

Q2: Simulator:

You need to write a simulator for the given ISA. The input to the simulator is a binary file (the format is the same as the format of the binary file generated by the assembler in **Q1**). The simulator should load the binary in the system memory at the beginning, and then start executing the code at address 0. The code is executed until `hlt` is reached. After execution of each instruction, the simulator should output one line containing an 8 bit number denoting the program counter. This should be followed by 8 space separated 16 bit binary numbers denoting the values of the registers (R0, R1, ... R6 and FLAGS).

<PC (8 bits)><space><R0 (16 bits)><space>...<R6 (16 bits)><space><FLAGS (16 bits)>.

The output must be written to *stdout*. Similarly, the input must be read from *stdin*. After the program is halted, print the memory dump of the whole memory. This should be 256 lines, each having a 16 bit value

<16 bit data>

<16 bit data>

.....

<16 bit data>

Your simulator must have the following distinct components:

1. Memory (MEM): MEM takes in an 8 bit address and returns a 16 bit value as the data.
The MEM stores 512bytes, initialized to 0s.
2. Program Counter (PC): The PC is an 8 bit register which points to the current instruction.
3. Register File (RF): The RF takes in the register name (R0, R1, ... R6 or FLAGS) and returns the value stored at that register.
4. Execution Engine (EE): The EE takes the address of the instruction from the PC, uses it to get the stored instruction from MEM, and executes the instruction by updating the RF and PC.

The simulator should follow roughly the following pseudocode:

```
initialize(MEM); // Load memory from stdin
PC = 0; // Start from the first instruction
halted = false;

while(not halted)
{
    Instruction = MEM.getData(PC); // Get current instruction
    halted, new_PC = EE.execute(Instruction); // Update RF compute new_PC
    PC.dump(); // Print PC
    RF.dump(); // Print RF state
    PC.update(new_PC); // Update PC
}
MEM.dump() // Print memory state
```

Q3: Floating-Point Arithmetic:

*CSE112_Floating_point_representation: **NO** sign bit, 3 exponent bits, and 5 mantissa bits.*

- In the registers, only the last 8 bits will be used in computations and initialization for the floating-point numbers.*

Modify the assembler and simulator to include arithmetic operations for floating-point numbers of the form(precision) given above.

Specifically, include the following functions:

00000	F_Addition	Performs $\text{reg3} = \text{reg1} + \text{reg2}$. If the computation overflows, then the overflow flag is set	<code>addf reg1 reg2 reg3</code>	A
00001	F_Subtraction	Performs $\text{reg3} = \text{reg1} - \text{reg2}$. In case $\text{reg2} > \text{reg1}$, 0 is written to reg3 and overflow flag is set.	<code>subf reg1 reg2 reg3</code>	A
00010	Move F_Immediate	Performs $\text{reg1} = \$\text{Imm}$ where Imm is an 8-bit floating-point value.	<code>movf reg1 \$Imm</code>	B

Note:

- For moving 1.5 into reg1 . The instruction(in assembly language) should be:
`movf reg1 $1.5`
- The students must only apply the operations for the floating-point numbers that can be represented in the given system(8 bits), else they should report it as an error.

Q4: (Bonus) Memory Access Trace:

In Q2, generate a scatter plot with the cycle number on the x-axis and the memory address on the y-axis. You need to plot which memory address is accessed at what time.

Q5: (Bonus) Memory Mumbo Jumbo:

This question is aimed at helping students visualize how memory is organized inside of a computer.

4 Types of Memory

1. Bit Addressable Memory - Cell Size = 1 bit
2. Nibble Addressable Memory - Cell Size = 4 bit
3. Byte Addressable Memory - Cell Size = 8 bits(standard)
4. Word Addressable Memory - Cell Size = Word Size (depends on CPU)

Note : Byte Addressable Memory is industry standard (if not mentioned use this as default).

Imp Note : CPU is always word addressable(ie the number of bits of cpu indicate its word size) so we need an interface to connect CPU with memory.

Your aim is to create a command line program to solve questions related to memory organization:

Firstly input:

1. The space in memory (eg 16 MB) Make sure you recognize inputs in this very format Mb should be read as mega bits and MB as mega byte.
2. Then input how the memory is addressed as mentioned above (either of the four options)

QUERIES: Now you have to do processing on these types of queries:

- The first type of question is ISA and Instructions related:
 1. Type A: <Q bit opcode> <P-bit address> <7 bit register>
 2. Type B: <Q bit opcode> <R bits filler> <7 bit register> <7 bit register>

Here, input the following:

- length of one instruction in bits
- length of register in bits

and output the following:

- How many minimum bits are needed to represent an address in this architecture
- Number of bits needed by opcode
- Number of filler bits in Instruction type 2
- Maximum numbers of instructions this ISA can support
- Maximum number of registers this ISA can support

- The second type of question is System enhancement related

TYPE 1:

Input the following

1. Firstly input how many bits the cpu is
2. Then input how you would want to change the current addressable memory to any of the rest 3 options

And Output the following:

How many address pins are saved or required (*indicate saved by [- how many ever we don't need] and + [by how many more we need]*) (also do note that the memory size and current addressability will be carried over from the initial inputs taken before any of the queries were taken)

Example -

Initial input (before query processing) :

256 kB of memory space

Byte addressable memory

Current input:

32 bit CPU supporting System

enhanced with word addressable memory

Output:

-2 (ie 2 pins saved)

TYPE 2:

Input the following

1. Firstly input how many bits the cpu is
2. Then input how many address pins it has
3. Then input what type of addressable memory it has (either of the four options)

And Output the following:

How big the main memory can be in Bytes (here we neglect the very first inputs of the addressability and the space of the memory that we take before processing any sort of queries are taken or processed)

Example -

Current input:

64 bit CPU supporting System

34 address pins

Byte addressable memory

Output:

16 GB (*please output in B or Bytes*)

Example 2 -

Current input:

64 bit CPU supporting System

34 address pins

Word addressable memory

Output:

128 GB (*please output in B or Bytes*)