Binary

- Given n bits, we can store 2^n different numbers from 0 to 2^n - 1 (unsigned)
-- sizeof(char) = 1 byte (8 bits); sizeof(int) = 4 bytes (32 bits) = 1 word (for MIPS)
- Signed int: Give up first bit to be the sign bit, i.e. 0 = positive and 1 = negative. Allows us to quickly tell the sign by just looking at a number's first bit.
-- Two's Complement: Split range into halves for negative and nonnegative, so we have one more negative than positive number
-- Finding the complement (both negative and positive): Invert all bits and +1, i.e. ~x + 1
- Overflow: Exceeding the max positive int loops back to the most negative int, and vice versa
- To convert to hex, every 4 bits = one hex value

Boolean algebra:
- OR represented by +, AND by *
- Identities:
-- Law of 0's: X * 0 = 0, X + 0 = X
-- Law of 1's: X * 1 = X, X + 1 = 1
-- X X = X, (X+Y) X = X, X + X = X, X Y + X = X
-- X_{not} Y + X = X + Y, (X_{not} + Y) X = X Y
-- $(X\ Y)_{not} = X_{not} + Y_{not}$, $(X + Y)_{not} = X_{not}\ Y_{not}$
- Bit masking: X * 0 = 0 and X * 1 = X, so AND with string of 0's and 1's to get desired bits

MIPS

- 32 registers, but we only use $s0 - $s7 for C/Java vars and $t0 - $t7 for temporary vars
- Registers have no type; operations determine how contents are treated
- Arithmetic: add, addu (unsigned), sub, subu, sll, srl (shift left/right logical)
- Immediates allow you to use constants: e.g. addi $s0, $s1, 10
- Loads and stores allow us to derefence pointers stored in a register
-- We usually use lw or sw (load/store word) since MIPS stores data in terms of words (4 bytes), in which case the offset must always be a multiple of 4
-- Example: To access arr[3], we need to add an offset of 4x3=12: lw $t0, 12($s3)
-- Other types also available, e.g. lbu (load byte unsigned) for single chars, or li (load immediate)
- Conditionals: beq (branch if equals), bne (if not equal), j (jump), slt (set on less than)
- jal (jump and link) will store (PC+4) to $ra, allowing us to jump back using jr $ra (jump to register)
- slt is often followed by bne or beq against $0
-- Example: slt $t0, $left, $right; bne $t0, $0, Less; will goto Less if $left < $right
- Pseudo-instructions:
-- move command does not actually exist, but is instead assembled into addi $a, $b, $zero
-- mul: m bits * n bits = (m+n) bits result, so multiplying 32-bit integers will give a 64-bit integer. mult $s0, $s1 will automatically store the result into the hi and lo registers, which is then copied out using mfhi and mflo (move from hi/lo)

-- Note: We usually only care about the lower half
-- Similarly, div $s0, $s1 puts the quotient ($s0 / $s1) in lo and the remainder ($s0 % $s1) in hi

Subroutines (Functions):
- By convention, each subroutine can take 4 arguments $a0 to $a3, and can have 2 return values $v0, $v1
- Procedure for a subroutine:
[Prologue]
1. Move stack pointer ($sp) down to make space: addi $sp, $sp, -8 (number of items to store * 4 bytes per word)
2. Store $ra and any other variables to stack: e.g. sw $ra, 0($sp)
-- For optimization, convention is for subroutines to preserve $ra, $sp, $gp, $fp and saved registers ($s0 - $s7), but not return value, argument and temp registers
-- If you intend to use any saved pointers, be sure to save their old value to the stack first!
-- We then usually cache arguments to a saved pointer, since there is no guarantee they will be preserved following an inner subroutine call
[If calling another subroutine]
3. Set up $a0 ... $a3
4. Call the subroutine: jal <func> (jump and link)
5. Return values are now in $v0, $v1. Use them if desired.
[Epilogue]
6. Restore $ra and any saved registers used: lw $ra, 4($sp)
7. Restore stack pointer: addi $sp, $sp, 8
8. Exit: jr $ra

Instructions: Also stored as 32-bit words
- R-format (for most instructions):
-- opcode (6 bits) | rs (5) | rt (5) | rd (5) | shamt (5) | funct (6)
-- opcode = 0, so funct selects variant of operation
-- rs (source register) = first operand, rt (target reg) = second, rd = destination reg; all 5 bits since there are only 32 registers
-- shamt contains the shift amount for shift instructions, and is 0 for all other instructions. Also 5 bits since shifting a 32-bit word by more than 31 is useless.
- I-format (immediates): Same as R-format, except we need more than a 5-bit field since immediates may be large, so we combine rd, shamt, funct into a 16-bit field
-- Since rd is removed, rt now denotes the destination register
-- Branching is also suitable for immediates since we are likely only applying an offset of a small number of lines, which can be saved in an immediate. We then use relative address: PC (program counter) = (PC + 4) + (offset * 4). NOTE: Beginning of offset is counted from one instruction AFTER the branch!
-- Can reach +- 2^{15} instructions (words) = +- 2^{17} bytes total around PC
-- If immediate is >16 bits, compiler splits instruction up into lui (load upper immediate) on the upper 16 bits, then ori (or immediate) on the lower 16

- J-format (jumps): Needs more space than branches since we can jump anywhere in memory
-- Ideally, we would specify a 32-bit memory address to jump to, but since we need 6 bits for opcode, we use the remaining 26 bits for the target address. To extend it to 32 bits, we add 0b00 as the last two bits since addresses are all multiples of 4, then take the highest 4 bits from the PC. Hence, new PC = { (PC+4)[31:28] (4 bits), target addr (26 bits), 00 (2 bits) }
-- Adequate since most programs will fit within 256 MB, but if necessary, we can always use 2 jumps instead
- Self-modifying instructions: Since instructions are also stored as words, we can modify them by using la (load address) on a label, then lw on that pointer to access a instruction, before replacing it with sw.

Compilation Process:
I. (C program) -- Compiler -->
II. (Assembly program, i.e. MIPS with pseudo-instructions) -- Assembler -->
- Assembler takes in assembly language code and outputs object code and information tables:
1. Convert pseudo-instructions, assign addresses to each line
2. Create relocation and symbol table
-- Relocation info: Identifies lines of code containing dependencies that need to be fixed, including static data (like strings), lib functions, jump addresses but NOT branches
-- Symbol table: List of publicly accessible labels (e.g. that of main()) and static data
3. Resolve PC-relative labels, i.e. branches. The first pass has already stored the labels in the symbol table, so we just need a second one to fill in their relative offsets.
4. Generate object file, which contains:
-- Text segment: The machine code
-- Data segment: Binary representation of static data
-- Our symbol and relocation tables
III. (Object file) -- Linker, links libs -->
- Linker combines object files and info tables, outputting executable code:
1. Merge text and data segments of each object file
2. Calculate absolute address of each referenced label and piece of data
3. Go through relocation table and fill in absolute addresses for each reference
-- To resolve each reference, search in symbol tables and then library files
- Note: This is the approach for statically-linked libraries. Dynamically-linked libraries (DLL) would reduce program sizes and allow for upgrades, but require overhead to do link at runtime
IV. (Executable) -- Loader --> Load into memory
- Loader creates new address space larger enough for text, data and a stack
- Copies in instructions and data from executable, and arguments passed to the program onto the stack