Stanley Wei

# COM SCI 33

Glenn Reinman

Spring 2023

**Table of Contents:**

**Bits & Bit Operators**

- Byte - sequence of 8 bits (encodings: 8 digits in binary, 0-255 in decimal, 00-FF in hex)
    - Encodings determine how meaningless bits are interpreted
        - Binary - prepended by 0b, hexadecimal - prepended by 0x
- *Bit pattern operators:* ~ (unary operator),
    - **~** (***unary*** operator/complement): given a bit pattern, flips all bits of a bit pattern
    - **&** (intersection/**AND**): given two bit patterns, output bit is 1 iff both input bits are 1
    - **|** (union/**OR**): given two bit patterns, output bit is 1 if at least one input bit is 1
    - **^** (exclusive-or/XOR): given two bit patterns, output bit is 1 iff only one input bit is 1
    - ~ (unary/complement), & (intersection), | (union), ^ (exclusive-or) bitwise operators implemented in C (treat arguments as bit vectors, applied bit-by-bit)
        - Note: AND, OR, XOR are commutative
        - Vs logical operators &&, e.g.
- *Shift operators*:
    - ***Left shift*** (**<<**) vs ***right shift*** (**>>**) - shift bit vector x to left/right by y positions (i.e. begin at x_0 -> begin at x_3/x_(-3))
        - Left shift - fills with 0s on right
        - Right (***logical***) - fill with 0s on left
            - Only relevant for C, unsigned
        - Right (***arithmetic***) - replicate most significant bit (bit position-wise) on left
            - Preserves negative sign, in the case of signed integers
        - Shifting left/right overwrites bits on left/right
    - Undefined - negative shift amounts, shift amounts > word length
- Instructions stored in [readable/read-only] addresses in memory [stored in bytes], referenced by program calls
    - Size of an instruction may vary based on contents
    - Data can be placed in various places, depending on the compiler - read-only memory, dynamically allocated memory, registers (e.g.)

**Computers**

- **[Instruction Set] Architecture (ISA)** - interface between hardware and software
    - ISA understands instructions, registers, condition codes of the hardware
- **Microarchitecture** - implementation of the architecture
- Programmers typically deal with 2 parts of a computer - **CPU** and **memory**
    - Parts of the **CPU**
        - **Programmer counter (PC)** - contains address of next instruction for computer to execute ("RIP" on Intel)
        - **Register file** - contains heavily-used program data
        - **Condition codes** - store status information about recent [arithmetic/logical] operations (used for conditional branching)
        - **Cache** - smaller memory container located directly on the CPU, allowing for faster, less expensive memory accessing
    - **Memory** - contains code and user data
        - Also has an *internal stack* to support procedures
        - Stored in the form of a byte-addressable array
    - CPU pulls data and instructions from memory; accesses memory using addresses
    - GPUs usually have simpler core designs
- Transferring memory: memory transferred in contiguous fixed-size chunks
    - ***Blocks*** used for transferring between cache, DRAM
    - ***Pages*** used for transferring between disk, DRAM (generally larger than blocks)
    - Computer minimizes the number of blocks, pages used for transferring information
        - Primitive types are best optimized by transferring in single blocks/pages, rather than being split across multiple
- *Virtual memory* - the "memory" that programs see is not the same as the physical memory of the system (DRAM)
    - When programs allocate [virtual] memory, the addresses they use do not necessarily correspond to addresses on physical memory
        - A *memory management unit* (MMU) on the CPU translates from logical addresses (in virtual memory) to physical addresses (on the hardware itself)

**Integers**

- Integer encodings:
    - **Unsigned**: every bit represents a power of 2
        - B2U(x) [bit to unsigned] = sum(x_i * 2^i), 0 < i < w-1 [w = num digits - 1]
        - An unsigned integer represented by a w-digit bit sequence has a range of 0 [UMin] < B2U(x) < 2^w - 1 [UMax]
        - Used for representing nonnegative values - can fail
    - **Two's complement** (signed integers in ANSI C):
        - T2U(x) = (x_(w-1) * -2^(w-1)) sum(x_i * 2^i), 0 < i < w-2
        - Most significant digit indicates sign: 0 nonnegative, 1 negative
        - A two's complement integer represented by a w-digit bit sequence has a range of -2^(w-1) [TMin] < B2T(x) < 2^(w-1) - 1 [TMax]
        - Given bit sequence x in two's complement form: -x = ~x + 1
            - Exception: x = TMin -> -x evaluates to x [overflow]
        - Sign extension [by *k* bits, i.e. a w-digit bit pattern -> w+k digits, preserving value] - copy the most significant bit k times [arithmetic shift]
- Default C integer sizings: **short = 2 bytes, unsigned/signed int = 4 bytes, long = 8 bytes**
    - Integers (ints/constants) considered *signed by default* in C, unless explicitly cast to unsigned (keyword "unsigned") or declared as unsigned (e.g. x = 5U;)
    - Comparing unsigned and signed integers: utilizes *unsigned comparison*
        - Also uses unsigned arithmetic over signed arithmetic
        - Can cast unsigned to signed for signed comparison
        - Unsigned comparisons between negatives/positives (< TMax) are preserved; otherwise, generally flipped
    - Casting the same bit pattern to unsigned vs signed [two's complement] int in C interprets the number based on the encoding [the cast]
        - Signed vs unsigned take the 1st digit as large negative/positive bias
- Integer addition
    - Problems with integer addition arise due to limited [or constrained] space
        - Exceeding MAX results in overflow; falling below MIN, underflow
        - Unsigned addition to a w-digit bit pattern - (x+y) becomes (x+y) mod 2^w (overflows if x+y > 2^w)

        - Signed addition to a w-digit bit pattern
            - Adding one to TMax evaluates to TMin; subtracting one from TMin evaluates to TMax
- Integer multiplication: multiplying an integer (signed or unsigned) by $2^k$ equivalent to a leftward shift of *k* digits
    - Leftmost bits may be discarded
    - Breaking down multiplication factors in terms of powers of 2 then adding & shifting can be faster than true multiplication (i.e. repeated addition)
        - Ex: 14 = 8 + 4 + 2 -> x * 14 = x * 8 + x * 4 + x * 2 = x << 3 + x << 2 + x << 1
        - Shifting and adding is faster than multiplying
- Integer division: can be implemented via rightward shifts, similar to reverse multiplication
    - Shifting loses precision – left shifting an unsigned integer *k* digits gives floor($u/2^k$)
        - Adding a bias term (1 less than divisor) changes behavior to always round toward 0 (as opposed to toward -infinity, default behavior of a simple shift)
            - Dividing by y: (x + y - 1) / y [bias for negative division to round toward 0, rather than flooring]
- Memory is byte-addressable: each location stores a byte; later bytes are 4 or 8 bits after
    - Multi-byte quantities: *Big-Endian* vs *Little-Endian* ordering
        - Given a quantity stored in multiple bytes (e.g. addresses x, x+1, …, x+w) [address &x given by compiler]:
            - ***Big-Endian***: More significant values stored at lower addresses (i.e. most significant value stored at x)
            - ***Little-Endian*** (x86): More significant values stored at higher addresses (i.e. most significant value stored at x+w)
            - Hex quantity 0x123456: more significant (12) -> less significant (34)
                - Big-Endian: 12 -> 34 -> 56
                - Little-Endian: 56 -> 34 -> 12
            - Actual internal contents of bytes unchanged, just outside ordering
                - Orderings within (of bits): most to least significant
- Misc.
    - C: <limits.h> declares platform-specific constants, e.g. LONG_MIN/MAX, etc.

**x86**

**Assembly**

- A single line [instruction] in Assembly is 8 bytes
- Data types (Assembly):
    - Integer values (1, 2, 4, 8 or bytes)
        - Data values, stored in memory addresses [untyped pointers]
    - Floating point - 4, 8, 10 bytes
    - Code - byte sequences encoding series of instructions
    - Note: no aggregate types (e.g. arrays, structures), just [contiguously allocated] bytes in memory
- Layers of code
    - Machine code – each instruction stored at some address in memory
        - Instructions are variable length: contain a certain number of bits (depending on command) storing needed information (e.g. integer indicating the command, SRC address, DEST address, etc.)
    - Assembly: human-readable version of machine code
- Assembly commands
    - “q” (quad) at the end of a command indicates that it handles 64 bits; also: “l” (long - 32-bit), “w” (word - 16-bit)

| Operation | Command | Operands | Operation |
|---|---|---|---|
| Move [copy] | movq | SRC, DEST | Copies value of/at SRC to DEST |
| Load effective address | leaq | SRC, DST | Copies address at SRC to DST |
| Add | addq | SRC, DST | Replaces value at DST with SRC + DST |
| Subtraction | subq | | |
| Multiplication | imulq | | Signed |
| Multiplication | mulq | | Unsigned |

| | | | |
|---|---|---|---|
| Left shift | salq/shlq | SHIFTAMT, SRC = DST | Replaces value of SRC with (SRC << 2) |
| Arithmetic right shift | sarq | SHIFTAMT, SRC = DST | Replaces value of SRC with (SRC >> 2) |
| Logical right shift | shrq | SHIFTAMT, SRC = DST | Replaces value of SRC with (SRC >> 2) |
| Exclusive or | xorq | | |
| And | andq | | |
| Or | orq | | |
| Increment | incq | *one operand | |
| Decrement | decq | *one operand | |
| Negate | negq | *one operand | |
| Not | notq | *one operand | |
| Return | ret | N/A | Terminates function |

- **movq** analogous to dereferencing and assigning a value at a pointer's address in C (i.e. *dest = val -> movq %addr1, %addr2)
    - If SRC is an immediate, assigns value of that immediate to DEST
    - If SRC is a memory address [i.e. referred to by operator()], assigns value at that memory address to DEST
    - movzbl - move from byte (hence the b) to 32-bit (hence the l), filling the remaining bytes with zeros (hence the z)
    - movslq - move from long (32-bit) to quad (64-bit), filling remaining bytes by sign-extending (hence the s)
    - When moving 32-bit to 64-bit register, automatically zeroes the extra bits
- **leaq** (***load effective address***):
    - Is a means of computing addresses without a memory reference/performing pointer arithmetic without actually dereferencing the pointer [i.e. with only a register address] (analogous to p = &x[i])
        - Can use general memory addressing with D(Rb, Ri, S) to perform pointer arithmetic (i.e. SRC = (%rax, %rcx, 2) = %rax + 2 * %rcx

            - Using the same command with movq copies the value at the returned memory address to DEST; leaq simply copies the returned memory address to DEST without dereferencing
    - Computing arithmetic expressions x + k*y (k = 1, 2, 4, or 8)
        - Is an alternative to addq that does not overwrite the destination
        - E.g. leaq (%rax, %rcx, 3), %rdx places a value of %rax + 3 * %rcx in %rdx (whether %rax, %rcx held integers or memory addresses)
    - Omitting SHIFT_AMT for one of the shift commands will shift by 1

**Registers**

- Prefix 'r' denotes 64-bit registers; prefix 'e' denotes 32-bit virtual register
    - %rsp - reserved for specific purposes (stack pointer), %rbp - base pointer
    - %rbp indicates 64 bits at some place in the register file; %ebp denotes the latter 32 bits of *those* 64 bits in the register
        - %rax - 64 bits, %eax - 32 bits, %ax - 16 bits, %al - 8 bits,etc.
        - Assigning to lower-bit portions of a register will zero out upper bits
- Every *register address* (e.g. %rax) is a label for an n-bit sequence in register
- X86 contains only 16 distinct integer registers
    - Running out of registers (register coloring (?)) - must then also start to use memory in conjunction with register (register spilling)
        - Note: x86 allows direct access of memory (not all architectures)
            - Only allows 1 access of memory at a time due to physical design restrictions in x86 (e.g. cannot move from memory to another address in memory - must go through registers as an intermediary)
    - Having additional registers prevents spilling, but may result in hardware issues
        - Also requires more bits to specify a specific register in an instruction - increases size of instructions in memory
    - Other architectures may have more or less registers (e.g. ia32 - 8 registers)
- **%rdi, %rsi, %rdx** used for first three arguments (in that order); **%rax** used for return
    - Registers: rax, rbc-rdx, rsi, rdi, rbp, r8-r15 [also rsp (reserved)]

**Operands**

- *Assembly operand types:*
    - **Immediate/literal** - constant integer data
        - Akin to a C constant, but prefixed with $ (e.g. $0x400, $-533)
            - Encoded with 1/2/4 bytes (instruction always allocates 4)
        - Instruction contains an immediate as a constant (i.e. contains the actual value of the immediate, not an address to it)
    - **Register** - [value contained at] one of 16 integer registers (e.g. "%rax")
        - Akin to a variable in C (points at a certain integer register)
    - **Memory** - 8 consecutive bytes of memory at address given by register
        - E.g.: (%rax) refers to the 8 consecutive bytes in memory, at the memory address stored in %rax (%rax contains a memory address - (%rax) follows that memory address and returns the value at that address in memory)
        - Doesn't need to only be referenced as a pointer - can also be treated as a variable [register], a la pointer arithmetic
- *Addressing modes:*
    - Normal ((R), e.g. %rax) returns value at memory address contained in %rax
        - (Rb) = Mem[Reg[Rb]]
    - ***General form:* D(Rb, Ri, S) = Mem[Reg[Rb] + S*Reg[Ri] +D]**
        - D = displacement, Rb = base register, Ri = index register, S = scale (1/2/4/8)
            - Rb automatically added
            - Ri (index register) added, but multiplied by scale
                - Similar to indexing into an array via pointers
            - Scale limited to 1, 2, 4, 8 [bytes] since "multiplication" is just left shift
            - Note: D defaults to 0 (no displ), S to 1 (adds just one of Reg[Ri])
        - Displacement specified by a constant: offset by certain number of bytes
            - Can be used to index into arrays, e.g.
            - Since displacement is a constant, limitations in data type mean displacement can only be a 1/2/4-byte integer (no 8-byte)
        - May or may not address into forbidden/nonexistent memory addresses
- Compiler typically handles allocation of/dealing with registers

**Control**

- **Condition codes** are special single-bit registers [on the CPU] describing attributes of the most recent arithmetic/logical operation [excluding leaq]
    - Can be used to perform conditional branches
    - Notable codes:
        - CF (Carry flag) - most recent operation carried out of most significant bit
            - Used for detecting unsigned integer overflow
        - ZF (Zero flag) - most recent operation yielded a zero
        - SF (Sign flag) - most recent operation yielded a negative value
        - OF (Overflow flag) - most recent operation resulted in a two's complement overflow (negative or positive)
    - Can also be modified by certain commands [without altering other registers]
        - COMPARE S1, S2 -> S2-S1:
            - cmpb/w/l/q (compare byte/word/double word/quad word)
            - setg [greater] executes if S2 > S1 (e.g.)
        - TEST S1, S2 -> S1 & S2: [testb/w/l/q]
- **SET operations** set a single byte (e.g. of a register, such as a single-byte %e* register) to 0 or 1, depending on condition codes
    - Take one parameter (D = destination)
    - **Operations:**

| | | |
|---|---|---|
| sete/setz | D = ZF | [equal] |
| setne/setnz | D = ~ZF | [not equal] |
| sets | D = SF | [signed negative] |
| setns | D = ~SF | [signed nonnegative] |
| setg/setnle | D = ~(SF ^ OF) & ~ZF | [greater: signed >] |
| setge/setnl | D = ~(SF ^ OF) | [greater or equal: signed >=] |
| setl/setnge | D = SF ^ OF | [less: signed <] |
| setle/setng | D = (SF ^ OF) \| ZF | [less or equal: signed <=] |
| seta/setnbe | D = ~CF & ~ZF | [above: unsigned >] |
| setae/setnb | D = ~CF | [above or equal: unsigned >=] |
| setb/setnae | D = ZF | [below: unsigned <] |
| setbe/setna | D = CF \| ZF | [below or equal: unsigned <=] |

- **Jump instructions** cause program execution to "jump" to a specific instruction (changing the program counter/RIP)
    - Break normal sequential execution of instructions
    - An ***unconditional jump*** (**jmp**) takes a single address [to be jumped to] as input; taken as either a *label* [directly encoded in the instruction] or *operand [read from register/memory]
        - Direct jumps are jumps using labels; indirect jumps use *operands
        - Syntax: jmp [Label/*Operand]
            - A label is written as its own line in the program (i.e. the line ".L2:" labels a destination; it can then be jumped to with "jmp .L2")
                - Converted to/replaced by an actual memory address during the linking process
    - ***Conditional jumps*** jump if and only if a certain condition is fulfilled (e.g. jge only jumps if ~(SF ^ OF) evaluates as true)
        - Conditional jumps follow notation of SET operations for conditions
        - Can only be direct jumps (i.e. take labels as input)
- *Conditional branching in Assembly:*
    - Conditional jumps can be used to mimic conditional branching (e.g. C if statements)
        - Entail a conditional transfer of *control* (i.e. the program takes a different execution path depending on the evaluation of the condition)
        - Simple, general, but possibly inefficient on modern processors
        - *Pipelining*, process by which modern processors achieve higher performance by "looking ahead" in instructions, may break down for conditional jumps if the processor cannot predict jump/predicts incorrectly
            - "Falling through" (i.e. not following condjump) is typically faster than following the conditional jump for optimization purposes
    - ***Conditional moves*** can be used to mimic conditional assignment (e.g. C ternary conditional operator)
        - Conditional moves also follow SET operation condition syntax (e.g. cmovge = move if greater/equal)
        - Entail a conditional transfer of *data* (i.e. the execution path of the program is static, but the data itself might change based on the condition)
        - More restrictive than conditional jumps, but also more efficient

        - Both sides of the ternary are always computed when using a conditional move, no matter the value of the condition
            - May cause problems, e.g. if the two expressions are expensive to compute, would cause errors [are risky], or could cause issues/side effects (e.g. if both modify an external variable)
            - Ex: “ptr ? *ptr : 0;” could attempt to dereference a null pointer, even if “ptr” evaluates to false
- Looping in Assembly:
    - C-style loops can be mimicked/implemented in Assembly using a combination of conditional tests and jumps
        - C do-while(condition): *body statements [loop body]* -> *conditional jump to top [loop condition]*
        - C while has two main implementations in GCC:
            - *Jump-to-middle*: *unconditional jump to bottom* -> *body statements [loop body]* -> *conditional jump to body statements [loop condition]*
                - Less optimized (-Og [no optimizations]), due to extra jumping compared to guarded-do
            - *Guarded-do*: *conditional jump skipping loop [loop condition, initial check]* -> *body statements [loop body]* -> *conditional jump to body statements [loop condition]*
                - Effectively transforms while loop into a do-while loop with an extra conditional jump before the loop itself
- C switch statements can be implemented using either a straightforward if-else conditional branching approach, or using a special data structure called a *jump table*
    - A ***jump table*** is an array where each entry i contains the address of the instruction to be jumped to if the switch index == i
        - Allow for constant-time switch indexing via referencing
            - Is more efficient than traditional if-else chains in situations with a significant amount of cases (e.g. >4) that only span a small range of values (such that the array does not take up too much space)

        - Composed of multiple quads in sequence (i.e. .align 8 (indicating each value is sized 8 bytes) \n  .myLabel: quad .LabelForCase1 \n quad .LabelForCase2 \n …)
            - Jmp *.L4(, *index as register value*, 8)
                - Performs an indirect jump, to a memory address relative to .L4,  + 8 * index
        - Does offset the switch cases (e.g. 5, 6, 7 -> 0, 1, 2)
            - Performing unsigned comparisons ja [jump above] filters out negatives - notably unsigned “x > 6” returns true for all positive > 6, but also all negative > 6 [since interpreting a negative two’s complement as unsigned = large [positive] integer]
    - Using conditional jumps converts the switch to an if-else chain
        - Notably, instructions for each case are likely stored sequentially and on top of each other; as such, if a case does not explicitly jump to the done state upon concluding, the code will otherwise simply “fall through” to the next case (similar to forgetting break; in a C switch)
    - Can also be implemented as a form of tree

**Procedures**

- A **procedure** is a named, independent section of code, callable by a larger body of code, that takes a designated set of arguments and performs a specific task
    - Are analogous to *functions* in C/C++
        - Similar to C/C++ functions, a procedure may or may not return a value
    - Can be referenced/called by other bits of code as a form of abstraction
        - Calling a procedure in Assembly: *call* *label* [*callq* *label*]
            - Are called by the memory address of the procedure [the first instruction of the procedure]
- Mechanisms for transferring control between procedures (via machine instructions):
    - *Passing control* - moving the program counter from one procedure to another
    - *Passing data* - passing values between procedures (i.e. as either input to the subprocedure, or output from the subprocedure)
    - *Memory allocation* - the subprocedure will need to allocate/deallocate memory to handle its own local variables
- *Picture of memory:*
    - **Stack** - at the top of memory (highest addresses), **grows downward** (i.e. toward lower memory addresses)
    - Towards the bottom of memory: text - contains instructions, e.g. (pointed to by program counter), data (global variables), heap (malloc'd/dynamically allocated memory - grows upward)
- The **x86-64 stack** is a stack data structure used for memory management and allocation/deallocation between procedures
    - Motivation: The most recent procedure called will always be the only procedure allocating memory, and all of *its* memory will be deallocated when it exits [before the memory of earlier procedures is touched]. As such, a stack is a good way to represent this pattern or allocation/deallocation
        - Is dissimilar from a stack in the sense that any address can be accessed
    - Is generally used by x86-64 when more space is needed than can be accommodate by registers; is also referred to as the *stack frame*
        - Registers are generally preferred for space and time efficiency
    - Terminology: the stack top is the active end, where insertion/deletion occurs

- Notes on the x86-64 stack:
    - The stack always grows towards lower memory addresses
    - The %rsp register points toward the top of the stack
        - Can allocate/deallocate space on the stack by decrementing/incrementing %rsp, respectively
    - ***pushq*** SRC writes SRC to %rsp, decrements %rsp by 8 [bytes] (allocates space)
    - ***popq*** DEST writes value at %rsp to DEST, increments %rsp by 8 (deallocates)
        - DEST must be a full 64-bit register
- Mechanisms for transferring control between procedures:
    - *Passing control* - moving the program counter from one procedure to another
        - Whenever a procedure is called, a return address is always added to the stack, indicating the instruction to resume at once the procedure exits
            - Return address is first item added to the stack upon a procedure call, even before the jump to the procedure is executed [decrement rsp; add return address; jump]
            - Assembly command ret pops the return address from stack, then jumps to it (resuming program)
                - To reach the return address - compiler determines how much space is allocated in a certain stack frame; at the end of procedure, simply moves stack pointer back that much space, which will be back at return address
            - Can decrease overhead by inlining functions
    - *Passing data* - passing values between procedures (i.e. as either input to the subprocedure, or output from the subprocedure)
        - First six arguments for procedure calls are held by registers **%rdi, %rsi, %rdx, %rcx, %r8, %r9**; return value is held by **%rax**
            - Stack space for additional arguments is only allocated when needed (i.e. when num_args > 6)
    - *Memory allocation* - the subprocedure will need to allocate/deallocate memory to handle its own local variables
- A programming language can support recursion only if its code is reentrant (allows multiple simultaneous instantiation of a single procedure)

    - Needs some place to store the state of each instantiation (e.g. local variables, arguments, return pointer, etc.)
- Stack allocated in ***frames*** - memory regions storing the data of a single procedure call/instantiation
    - Is typically allocated on function call (push by "call"/callq instruction), deallocated on return ("ret" instruction)
    - Contains return information, local storage, and any temporary storage (as needed)
    - Allows for handling of recursion without any special considerations
    - *Contents of the frame (in order):*
        - Caller frame: ends with arguments of the call to new procedure (if num_args > 6), return address
        - Pointer to old %rbp (optional)
        - Procedure frame [callee]:
            - Saved registers (to be restored on procedure exit)
            - Local variables [if registers are insufficient]
            - Argument build [any arguments for a subsequent function call]
            - *%rsp [stack pointer] points to top*
- Using registers for temporary storage
    - A procedure calling another procedure may cause issues relating to memory if both are modifying/accessing the same registers for temporary storage
    - Conventions:
        - *Caller saved*: caller saves any temporary values it needs (e.g. in registers that may be overwritten) in its own stack frame before procedure calls
            - Assumes registers will be overwritten
            - Registers: %rax, argument registers (%rdi, …, %r9), %r10, %r11
        - *Callee saved*: callee saves temporary values of caller (e.g. register values) in its own frame, restores them before exiting
            - Registers may still be changed in the procedure, but will be restored before returning
            - Registers: %rbx, %r12-%r14, %rbp, %rsp
    - Stack overflow - writing past the bounds of allocation on the stack
        - E.g. allocating a buffer, and then writing past the the end of it

**Data**

- **Arrays** are implemented as single contiguous regions of memory, holding variables of a certain type in sequence
    - An array of length L containing variables of type T (T A[L]) will take a contiguously allocated region of L * sizeof(T) bytes in memory
        - An array variable can thus be interpreted as a pointer to the corresponding region in memory
            - Can be indexed into via memory addressing: (arr, index, sizeof(var))
    - A **multidimensional array** can simply be implemented as an array of arrays
        - Ex: A 2D array with R rows and C columns, holding variables of type T (T A[R] [C]) will take a region of R * C * sizeof(T) bytes in memory
        - Individual subarrays can be accessed via memory addressing
            - &A[i][j] = &A[0][0] + i * C * sizeof(T) + j * sizeof(T)
        - Are ordered row-major: structured as a vector of rows
        - Implementations:
            - Simple array of array objects (*static array*)
            - Array of pointers to arrays
                - Requires one additional memory address dereference, but frees subarrays from needing to be contiguous in memory
    - NxN matrix handling in C:
        - *Fixed dimensions, known at compile time*: handled as a traditional array
        - *Variable dimensions, explicit indexing*: handled as a pointer, then offset
            - Traditional method for implementing dynamic arrays
        - *Variable dimensions, implicit indexing*: indexing into arrays (e.g. a[i][j], for array a[n][n]) more recently supported by gcc
- **Structures** (e.g. classes) are represented as singular blocks of memory, containing all of the structure's fields (i.e. member variables)
    - Overall size + positions of fields determined by compiler at compile time
        - Machine code does not understand the structures in the source code
        - C: fields ordered by declaration (explicitly not reordered for efficiency)
            - Strategies for saving space: putting large data types first
        - Ex: array fields have the requisite memory allocated at compile time

- ***Alignment/Aligned data***: If a primitive [integral type] requires ***K bytes*** of memory to store, it must be stored at a memory address that is ***a multiple of K***
    - Ex: char (1 byte), short (2 bytes), int (4 bytes), long (8 bytes)
    - Is required by some OSes; recommended for x86-64
        - Some OSes will have an explicitly-defined K (block size for memory/cache transfer) [e.g. 8 bytes on some systems]
            - If an OS has a fixed 8-byte block size, it will also require input addresses be multiples of 8 [due to the mechanism of dividing cache blocks/lines]
    - Makes accessing memory + loading/storing values more efficient, by minimizing the amount of blocks/pages needed to transfer values
        - Ensures a single primitive is only on a single block/page
    - Compiler will insert gaps in memory as needed to ensure alignment
- *Unaligned data* has no such requirement
- ***Conditions for alignment:***
    - **Within structs**: each element must satisfy its own alignment requirement
    - **Overall placement**: each structure has an alignment requirement equal to that of the largest alignment requirement of its elements
        - Initial address + structure size must be multiples of K
    - Arrays of structures must satisfy alignment requirements for all elements
        - Padding may be added between structures
    - Other: alignment requirement for an array is the alignment requirement for the elements of the array

**x86 Memory**

- X86-64 memory layout:
    - *Dynamic memory regions:*
        - **Runtime stack** (local variables) begins at the top, grows downward toward lower addresses [8 MB limit]
        - **Heap** (dynamically allocated variables) begins near the bottom, grows upward toward higher addresses [allocated by malloc(), calloc(), new()]
    - *Static memory regions:*
        - Statically allocated data (global vars, static vars, string constants) stored under the heap
        - Text/shared libraries (machine instructions) stored under the data
            - Begins at hex address 0x400000000000
    - Below text - reserve data (e.g. memory map registers)
- **Buffer overflow** - attempting to access objects past the memory size allocated for an array
    - e.g. attempting to access index 5 of a 4-element array
    - Results in possible security vulnerabilities (namely code injection)
    - Possible causes:
        - Unchecked lengths on string inputs
            - Bounded character arrays on the stack [stack smashing]
        - Unix string functions (e.g. gets [read from stdin into C string]) do not provide a means of specifying read limit
            - Other functions: strcpy, strcat, scanf/fscanf/sscanf
- ***Code injection*** - inputting a string containing executable code and overwriting the return address of a function to point to said executable code, such that it is run when the function returns
    - Allows for running arbitrary code from a remote machine
    - Location of ret address needs to be known (i.e. attacker has to have run program and found ret address, and ret address must be the same across calls)
- *Avoiding buffer overflow*
    - ***Avoid overflow vulnerabilities:***
        - Use library string functions that allow for specifying read limit lengths
    - ***System-level protections:***

        - ***Random stack offsets*** allocate a random amount of space on the stack whenever a program is run, shifting addresses for the entire program
            - Makes prediction of exact memory address difficult
            - E.g. giant block of padding before main frame
        - ***Non-executable code segments***: x86 has three types of permissions: read (always true), write (opt-in), and execute (opt-in) permissions
            - Translation look-aside buffer in MMU translates virtual addresses to physical addresses (allows for marking memory non-executable)
            - Can mark stack as non-executable, since stack is generally supposed to contain only data [not instructions]
    - ***Stack canaries*** can be placed at the end of a stack beyond a buffer, check for corruption before exiting function (*GCC: -fstack-protector*)
        - Any buffer overflow injections will then modify the canary as well - incorrect canary -> corruption
        - %fs - one of set of system registers describing memory (can be the canary)
- ***Return-oriented attacks:***
    - String together fragments from existing executable code (e.g. stdlib library code)
        - Will be executable and thus runnable
    - Make program from *gadgets*: any sequence of instructions that ends in ret (0xc3)
        - Code positions fixed from run to run, always executable
        - Uses tail ends of existing functions to string together a program
            - Overwrite return address to point to gadget
            - Can index into an instruction or even specific bytes in an instruction
        - Each gadget ends in a return; can layer gadgets consecutively on the stack to call gadgets in sequence
- **Union** - a data structure where all fields are stored in the exact same region of memory
    - Storage is allocated according to largest element
    - Can only use one field at a time
    - Can be used to handle bit patterns
        - Ex: declaring a union with an unsigned int and signed int will allocated space for a single int; referencing the unsigned int will return the bit pattern in that space interpreted as a uint, vice versa for signed int

**Floating Point**

- *Fractional binary*: Bits represent powers of 2, similar to integers; bits to the left of a binary point represent powers >= 0, bits to the right represent powers < 0 (fractional)
    - Limitations: can only approximate numbers not expressible in terms of powers of 2, only have one binary point (limiting the possible range of values)
- **Floating-point (FP) numbers** represented as specially-interpreted bit patterns
    - Created to replace fractional binary; provide larger possible range of values
- **IEEE Standard 754 (1985)** - uniform standard for floating-point arithmetic
    - Driven by numerical concerns, making standards for rounding, overflow, underflow
        - Not necessarily hardware-efficient
    - Numerical form - sign bit as the MSB [similar to ints], exponent E (exp), significand/mantissa M (frac)
        - Significand normally a fractional value in the range [1.0, 2.0)
        - Value: $float = (-1)^S * M * 2^E$
    - **Encodings:**
        - ***Single-precision*** [**32 bits**]: sign bit, 8 exp bits, 23 frac bits
        - ***Double-precision*** [**64 bits**]: sign bit, 11 exp bits, 52 frac bits
        - *Extended-precision (Intel only): 1 sign bit, 15 exp bits, 63/64 frac bits*
- Normalized [normal] vs denormalized [subnormal]] values
    - ***Normalized values*** take the form $(-1)^s * 1.[frac_2] * 2^E$
        - *Frac* field contains an implied leading 1 (saves one bit of space)
            - Significand may be padded with zeros to fit space
        - Exponent will not be all 1s, all 0s [indicates normalized value]
            - E takes the value of *exp* [exponent bit field] - bias
                - *exp* interpreted as unsigned int
                - Makes sorting floating-point numbers easier
            - ***Bias*** = $2^{k-1} - 1$, where k is number of exponent bits
                - Fixed by type size: single-precision: 127, double: 1023
    - ***Denormalized values*** take the form $(-1)^s * 0.[frac_2] * 2^E$
        - Significand contains an implied leading 0
        - Exponent will always be all 0s; E = 1 - bias always
            - E is fixed; -126 for floats

            - In conjunction with leading 0, means denormalized values effectively have exponent of [min_normalizedE - 1]
        - Used when normalized exponent exceeds values
    - **Exceptional cases:**
        - Exp field = all 1s, frac field = all 0s indicates ***positive/negative infinity***
            - No wrap-around behavior a la integer overflow
        - Exp field = all 1s, frac field not all 0s indicates ***not-a-number/NaN***
        - **0** - all zeros
    - Denormalized used for values closer to 0; normalized for values farther from 0
- Floats can utilize unsigned integer comparison [once the sign bit is compared]
    - Infinity vs normalized vs denormalized comparisons work, with the exception of NaNs [may be evaluated as larger than infinity/smaller than negative infinity

**Floating-Point Arithmetic**

- Involves first computing an exact value, then fitting into a desired precision
    - Value may be *truncated* to fit into frac, or *overflow* if the exponent is too large
- Rounding modes: towards zero, towards +infinity, towards -infinity, round-to-even
    - *Round-to-even* - default rounding mode in most computers
        - Rounds up or down to nearest whole value if not halfway between whole values; rounds to nearest even value if halfway
        - Other rounding modes arithmetically biased in one direction (will add up bias over time); round-to-even is more roughly random in direction of rounding (“even” value: least significant frac bit = 0)
    - Floating-point values:
        - Even when MSB is 0; “halfway” occurs when the digits to the right of the rounding position are 1000….000 [1 followed by all 0s]
- *Float multiplication:*
    - Sign: Exclusive or’d; significands multiplied directly; exponents added
    - Other operations: shifting if new significand >= 2, overflowing to infinity if E out of range [float saturation], rounding M to fit precision
- *Float addition:*
    - Signed align & add sign bit, significand

            - May result in difficulties when signed align + add results in a number with more digits than the frac portion can accommodate
                - Will result in truncating the frac portion when rounding
                - **Consequence: *Floating point addition is not associative***
        - Exponent of 1st taken
        - Fixing potential errors: shift [left or right] significand as needed, overflows to infinity, rounding M to fit frac precision
- C provides for *float* (single-precision - 4 bytes), *double* (double-precision - 8 bytes)
    - Int vs float: float has a larger dynamic range than int, but ints have more granularity than floats (32 vs 23 numerical bits)
    - Int/float/double conversions change bit representations
        - double/float to int truncates fractional portion (rounds toward zero)
            - Not defined when out of range/NaN; generally sets to TMin
        - Int to double is an exact conversion; int to float may need to round
- **Floating-point (FP) numbers** are stored in special *XMM registers*
    - 16 XMM registers - each XMM register is 16 bytes
        - An XMM register can hold:
            - 4 32-bit/8 16-bit/16 8-bit integers, 1 or 4 single-precision floats (32 bits each), 1 or 2 double-precision floats (64 bits each)
    - Have many special operations related to dealing with FP numbers
        - Require special operations for transferring to/from memory (vmov), converting to/from integers
            - Any FP operations referencing memory are scalar operations (operate on individual, not packed, values)
                - SIMD operations operate on packed values
            - Converting FP values to integers performs truncation, rounding values toward zero
    - XMM register conventions:
        - Arguments passed in %xmm0, %xmm1, etc.
        - Result stored in %xmm0
        - All XMM registers are caller-saved

**Optimization**

- Compiler has two functions: parsing the language
    - Compilers may try to parse languages into an intermediate level, optimize, and then finally generate language
        - Intermediate level aimed to be consistent across different language + language parser combos
- Optimized compilers provide efficient program-machine instruction mappings, reducing the constant factors of programs
    - In terms of register allocation, code selection/ordering [scheduling], dead code elimination, removal of minor inefficiencies
    - Compiler is designed to be *conservative*; limitations:
        - Cannot alter program behavior, even if only to safeguard against errors
            - The program itself may limit/hide opportunities for optimization
        - Most optimize only within individual procedures, based on static info
            - Does not see runtime inputs, e.g.
        - Have difficulty with potential optimization blockers: memory aliasing, procedural side-effects
- General optimizations (programmer or compiler)
    - **Code motion**: reduce instances of redundant/repeated calculations of one value
        - Can share common subexpressions
            - Ex: split {5 * i + 1, 5 * i, 5 * i - 1} -> x=5 * i; {x + 1, x, x - 1}
    - **Reduction in strength**: replace costly operations with cheaper ones
        - Ex: replace multiplications with shifts
    - Dead code elimination, code hoisting
- Compiler optimization blockers:
    - *Procedure calls*: compiler treats procedure calls as black boxes, and generally avoids optimizing around them
        - Procedures may have side effects, alter global variables, etc.; cannot be freely moved around the code
        - *Inlining* - replacing function calls with simply splicing in the function body can allow the compiler to make optimizations to/around it
            - Removes the benefits of using a function

        - *Interposition* - functions imported dynamically (e.g. from the OS, e.g. stdlib) may not be imported until linkage time
            - Removes need to recompile [becomes modular], but means compiler cannot optimize around it
    - Branching
    - ***Memory aliasing*** (disambiguation): a single memory location may be referred to by multiple different memory references; compilers generally hesitant to optimize around memory references as a result
        - May occur in the case of memory arithmetic, esp. with variables
        - Using local variables avoids this, so long as the compiler can know no pointers to the variables will ever be generated
    - Arrays & loops with arrays may be difficult for a compiler to optimize around, if the length is not known at compile time
    - ***Loop unrolling*** - trying to reduce loop iterations, increase work per iteration
        - Techniques to avoid ***dependency chains***: Reassociation (reordering operations), separating accumulators
            - Can use local variables instead of pointer references (i.e. array stores) to eliminate store/load dependencies
        - Loops in general are difficult to optimize around - a computer may not know what to set instruction counter to, if the instruction that computes said value is not yet known
- ***Parallelism***: modern processors can execute multiple instructions in parallel; can be exploited to boost performance on an instruction level
    - ***Instruction-level parallelism*** (**ILP**) - finding independent instructions that do not overlap in terms of the data accessed, such that they can be executed in parallel
        - Alt: thread-level parallelism (TLP) - used by GPUs, e.g.
    - Still limited by data dependencies, + lack of associativity/distributivity in floating-point arithmetic
        - Compilers generally do not perform such optimizations
    - ***Out-of-order execution*** (**OoOE**) - hardware-level optimization, “finds” opportunities for parallelism by freeing program to perform operations out-of-order
        - Involves performing operations out-of-order, but delaying the write-back to mimic sequential execution

- *Performance metrics:*
    - ***Latency*** – the time to execute a single instruction
        - Latency bound - the minimum time it takes for an operation to complete
    - ***Throughput*** – the number instructions executed per cycle (analogous to capacity)
    - ***Cycles per element*** (**CPE**) - notation for expressing performance of programs dealing with vectors and lists
        - Expressed in terms of the length of the vector/list (n)
- *Superscalar processor* - processor capable of issuing and executing multiple instructions in a single cycle (most modern CPUs are superscalar)
    - Are typically retrieved from sequential instruction stream, scheduled dynamically
        - Can be used to take advantage of ILP without extra programming effort
    - Instructions: fetch instruction, decode instruction, perform operation, write back
        - Registers are actually just logical registers, mapped to physical registers (uses register stations)
- Modern CPUs composed of two halves - instruction control and execution
    - Instruction control contains instruction cache + decoder
        - Fetch control used in the case of branching, to determine which instructions to fetch
        - Retirement unit contains register file; used to store results of operations before writing to registers as part of retirement (finishing an instruction)
    - Modern CPUs divide execution components into distinct ***functional units***, handling different operations (e.g. load, store, arithmetic, branch, etc.)
        - May have more than one functional unit for a single task (e.g. arithmetic)
- *Pipelined* functional units divide computations into stages, pass partial computations from stage to stage
    - If a computation must pass through 3 stages M1, M2, M3: once it completes M1 and is in M2, M1 is now free to start another computation before the first computation has “finished”, e.g.
    - Latches/blocks (?): registers holding intermediate results of computations
    - Can be used to begin new computations before the original finishes, decreasing total number of cycles required (increases throughput, but not latency)
    - Haswell CPUs contain 8 functional units
- Branching:

    - Challenge: Instruction control unit must work ahead of execution unit to generate enough operations for execution unit to stay busy, but cannot easily determine where to fetch when a conditional branch is hit
        - Cannot resolve which branch to take until branch/integer unit determined
    - Branch prediction: guessing the branch that will be taken, and begin executing at predicted position (but without modifying registers, memory)
        - Forces reloading of pipeline if misprediction
- Performance optimization: general tips
    - Good compiler, compiler flags
    - Write good code
        - Watch out for hidden algorithmic inefficiencies, especially inside loops
        - Make code compiler-friendly
            - Reduce optimization blockers: procedure calls, memory aliases
    - Tune code for machine
        - Take advantage of instruction-level parallelism
        - Avoid unpredictable branches
        - Make code cache-friendly
- Computer architectures:
    - ***Single-instruction, single-data*** (**SISD**) - each instruction operates on only a single stream of data
    - ***Single instruction, multiple data*** (**SIMD**) – applying a single instruction simultaneously to multiple streams of data (e.g. through parallelism/pipelining)
        - GPU divergence - code may act differently on different threads
        - Lacks flexibility, but reduces complexity of processor
        - Implemented with *YMM registers* - registers capable of holding multiple values simultaneously (e.g. in vector format)
            - 16 YMM registers, 32 bytes each
            - Can be used with vector operations (AVX) - special instructions acting across the values in a YMM register
                - Requires functional units capable of accommodating this
    - Multiple instructions, multiple data (MIMD)

**Memory**

**Memory in Computing**

- *CPU-memory gap*: the gap between CPU cycle times [actual & effective cycle times] and memory seek/access times has been widening over time
    - Applies to all forms of memory (disk drives, SSDs, DRAM/SRAM)
- **Principle of Locality** - programs tend to use data and instructions with addresses near or equal to those they have used recently
    - ***Temporal locality*** - recently referenced items are more likely to be referenced again in the near future
        - Ex: referencing the same variables repeatedly, cycling through the same instructions in a loop
    - ***Spatial locality*** - items with addresses near each other tend to be referenced close together in time
        - Ex: adjacent items in an error, reading of instructions in sequence
    - Well-written programs tend to exhibit good locality
- **Memory hierarchy** - an approach for organization of memory and storage systems, keeping in mind the benefits/drawbacks of various memory types + program properties
    - Ex: a hierarchy might have registers at the top, 3 levels/lines of SRAM cache below, main memory (DRAM), and finally secondary storage (local disks, web servers)
        - Higher-up memory types use storage formats with less capacity, but which are faster in terms of accessing, reading/writing
            - Each storage device in the hierarchy serves as a cache (of sorts) for the device below it

**Caches**

- **Cache** - a smaller, faster storage device used as a staging area for a subset of the data in a larger, slower storage device
    - Due to locality, computers can move recently-referenced data from memory to cache as a means of increasing read/write speed
    - **Cache memories** - small, fast SRAM-based memories on the CPU, managed automatically in hardware

        - Hold frequently-accessed blocks of memory, and are generally the first location searched by the CPU
        - Intel Core i7 has three levels of caches: the first two are divided between cores, and the third level is a unified cache for all cores
- *Cache terminology:*
    - A larger memory device is usually referred to as being partitioned into ***blocks***; a cache holds copies of a subset of these blocks
        - **Blocks** - information unit for transfer between cache, memory
            - Holds data bytes + metadata: valid bit (indicating if data is valid cache data), tag (indicating source location in DRAM)
        - Caches divided into ***sets***, divided into ***lines/blocks***
            - Cache size: ***C*** = ***S*** (# sets) * ***E*** (# lines) * ***B*** (block offset/size)
            - Number of physical address bits: m = t + s + b (t = # tag bits; s = # set index bits = $\log_2(S)$; b = # block offset bist = $\log_2(B)$)
        - Cache association:
            - **Direct-mapped**: exactly one cache line per set
                - A single given memory address only has one possible line
                - Fast (no need to search across lines), but increases miss rate: references to any addresses on same line result in a miss
            - **Fully associative**: one set containing all cache lines
                - A single given memory address can occupy any line
                - Most complex: decreases miss rate, since all blocks are contained in one set, but requires searching entire set
            - **Set associative**: anything in between direct, full
    - A program requesting memory from a block that has been copied to the cache can draw directly from the cache (a ***hit***)
        - *Hit time* - time needed to deliver a cache line to the processor (~4 clock cycles for L1, 10 for L2)
    - A program requesting memory from a block not in the cache must first copy the block from memory to cache, then draw from cache (***miss***)
        - *Placement policies* determine where the new block is placed in the cache; *replacement policies* determine which block is replaced

        - ***Miss rate*** - fraction of memory references that are misses (3-10% for L1, possibly <1% for L2)
        - ***Miss penalty*** - additional time required when a miss occurs (50-200 cycles for main memory; has been increasing)
- Writing cache-friendly code
    - Focus on the locations where the most computations + memory accesses occur, e.g. in the inner loops of core functions
    - Can utilize locality to minimize misses in inner loops
        - Using repeated references to variables boosts temporal locality
        - Stride-1 reference patterns boost spatial locality
            - Note: spatial locality more important for loads than for stores
- *Memory throughput* (read bandwidth) - number of bytes read from memory/second (MB/s)
    - *Memory mountain* - a measure of memory throughput as a function of spatial, temporal locality (i.e. as a 3D surface)
        - Throughput generally increases with smaller strides, cache sizes
        - Can be used to characterize memory system performance
- ***Tiling*** – when performing computations involving matrices/matrix arithmetic, can perform operations on individual block matrices instead to improve performance

**Parallelism**

- Moore's law - semiconductor density doubles every 18 months
- Single-core hardware performance hits the power wall - power consumption increases exponentially relative to scalar performance [thermal throttling]
    - Mobile CPUs with shallow pipelines use less power
- Multi-core processors avoid increased power consumption of single-core
    - May share caches for communication
    - Require applications (programmers) to split across cores
    - Different cores may be specialized for different tasks
- Definitions:
    - **Concurrency** - a system state where multiple tasks are *logically* active at a time
        - Tasks logically occurring at once are not necessarily running simultaneously in real-time (e.g. computer might be switching control)
    - **Parallelism** - a system state in which multiple tasks are *actually* active at a time
        - Parallelism implies concurrency; vice versa is not necessarily true

**OpenMP**

- OpenMP - set of compiler directives + libraries (API) for writing multithreaded (parallel) applications for Fortran, C, C++ (#include <omp.h>)
    - Runs at program layer (like compiler, env variables), system layer (runtime library)
    - Constructs are mostly compiler directives (included using pragma statements; imported during compilation)
        - Apply to a structured block - block of one or more statements with point of entry at top, point of exit at bottom
        - Ex: *#pragma omp parallel num_threads* (creates multiple workers)
            - Runs a given structured block once for each thread
- Shared memory programming
    - ***Process*** - a single instance of a program execution
        - Contains execution context - resources associated with program execution
        - May use specialized cores depending on the task/program
        - Simultaneous multithreading (hyperthreading) - running multiple processes simultaneously on a single core

            - Can be used in the case that each individual process has lots of downtime that would waste processor time, e.g.
    - ***Threads*** - "lightweight processes"
        - Share process state among multiple threads - reduces cost of switching
        - Operate on an MIMD/SPMD (single-program, multiple-data) model
- OpenMP: a single application (***process***) may have multiple ***threads*** running
    - Each thread has their own private memory; interact with other threads through reads/writes to ***shared address space***
            - Synchronization used to ensure correct results
    - Threads may run in an interleaved fashion (as determined by OS scheduler)
        - Result in no guarantees in terms of program ordering
    - Uses a ***fork-join parallelism model*** - a master thread spawns a team of worker threads over the course of execution [fork], synchronizing intermittently [join]
        - Parallelism added/removed incrementally, as needed, to hit performance goals [sequential program becomes parallel]
        - Worker threads may make their own sub-threads (nested parallel regions)
        - *#pragma omp master* – only master thread executes block
- OpenMP - multithreading, shared address model
    - Different threads communicate by sharing variables
        - Unintended sharing causes ***race conditions***: when program's outcome changes depending on thread scheduling (prevented via synchronization)
    - Can change manner of data access to prevent expensive synchronization
- OpenMP programming
    - *omp_set_num_threads* requests certain number of threads; *omp_get_thread_num()* returns thread ID (can only be run outside of/before parallel sections)
- ***Synchronization*** - bringing one or more threads to well-defined, known points of execution
    - Synchronization ***barriers*** may prevent a program [main thread] from continuing until all threads have hit a certain point [the barrier]
        - *#pragma omp barrier* - creates barrier
        - Implied barriers at end of worksharing/parallel constructs; can be removed with nowait clause
    - *Mutual exclusion* - defined blocks/portions of code [***critical regions***] that only one thread can execute

        - Upon hitting a critical region, only one thread can perform the read/write at a time (temporarily stops parallelism)
            - pragma omp critical declares one-statement critical region
        - pragma omp atomic is much lower overhead than omp critical
            - Requires subsequent statement to be one of the following forms: x binop = expr, x++/++x, x−/−x
            - Requires clause specifying operation type:
                - update [default] – protects an update of value
                - capture – protects an update and assignment
                - read – protects a load; write – protects a store
- Parallelism - scaling issues
    - **False sharing** - accesses of multiple independent data elements sitting on the same cache line by different threads may force the cache to continually cycle between different points in the cache
        - Solution: pad arrays so elements are on distinct cache lines
    - ***Cache coherence*** – if multiple threads have cached the same memory address, one thread writing to that address will invalidate all said other caches
        - Snooping - a cache listens to bus transactions
            - Does not scale well
        - Directory - a directory in cache contains who (which thread) has what block, and in what state
            - L3/L2, typically
            - MOESI [modified owned exclusive shared invalid]

**Worksharing**

- **Loop worksharing** - splitting up loop iterations among threads in a team
    - Used for loops of unknown/dynamic runtime, e.g.
- ***pragma omp for*** [omp do in Fortran] splits up execution of loop iterations
    - Can be combined with parallelism construct: ***pragma omp parallel for*** for parallelized worksharing (splitting up iterations between threads)
        - Equivalent to pragma omp parallel; pragma omp for
        - Loop variable implicitly made private to each thread

    - Only applies to outermost loop in a nested for loop; can be combined with collapse(*depth*) clause in the case of perfectly nested inner loops
    - ***Loop-carrying dependencies*** - cases where each loop iteration depends on the results of previous iterations
        - May increase efficiency, but prevent out-of-order execution
            - Can be removed to take advantage of out-of-order execution
- ***Schedule*** clause - affect mapping of loop iterations onto threads
    - *pragma omp for schedule(dynamic)*, e.g.
    - Types:
        - Static - handing out blocks of iterations
            - Inefficient in the case that one thread takes longer than the others, e.g., but does not incur much overhead
        - Dynamic - each thread grabs a certain number of iterations off a queue, until all iterations have been handled
        - Guided - threads dynamically grab blocks of iterations
            - Blocks initially large; shrink as calculation proceeds
        - Runtime - Schedule, chunk size taken from OMP_SCHEDULE environment variable (or runtime library)
        - Auto - schedule left up to runtime choice
            - May not choose the other predefined types
- OpenMP reduction: *pragma omp *statements* reduction (op : list)*
    - Parallelizes a loop: a “global variable” is separated into different local variables [one per iteration], then combined at the end (more scalable than critical)
        - Operators: +, *, -; min, max;, &, |, ^; &&, ||
- OpenMP sections:
    - Defines task-level parallelism - gives different structured block to each thread
        - Max number - number of active threads
    - Syntax: *#pragma omp sections -> #pragma omp section*
- OpenMP lock routines
    - Lock a single piece of data to only allow writing by a single thread
        - May lock a single element in an array, allowing for access by other threads to all other elements

        - Is an alternative to critical sections - allows for parallelism to be used without having errors, in the case that threads may be modifying the same data but are not dependent on each other
    - Deadlock - locks may be placed in such a way that no thread can act (threads may have locked each other's data, e.g.)
    - Syntax [within a #pragma omp parallel for]:
        - *omp_init_lock(memory_addr)* allows for creating a lock a memory address
            - Repeated for each element in an array, e.g.
        - *omp_set_lock(memory_addr)*; *omp_unset_lock(memory_addr)*
            - Operates by invoking a system call (similar to atomic)
        - *omp_destroy_lock(memory_addr)*
    - Nested lock - can be accessed if unlocked, or by locking thread
        - Must be unset as many times as was set to unlock
- Data:
    - *Shared memory programming model*: most global/externally-allocated variables are shared, local [thread-specific] variables are private
    - Can attach clauses to change storage attributes for constructs: SHARED, PRIVATE
        - FIRSTPRIVATE(varname) gives each thread a private variable (with name varname), with initial value the value of the variable (with name varname) in shared memory [copy-constructed]
            - Private variable is not initialized otherwise
        - LASTPRIVATE(variable) - private variable updates shared variable with value from last sequential iteration [functionally last]
        - Set default permissions: default(private|shared|none)

**Tasks**

- Non-deterministic work (i.e. exact number of steps not known) difficult for worksharing
    - Recursion (e.g. pointer chasing)
- **Tasks** (OpenMP) - independent work units, containing: code to execute, data environment, internal [control] variables (ICV)
    - Tasks are added to a queue [buffer]; threads pull tasks from the queue and perform work dynamically

        - May also be deferred, or executed immediately (runtime system decides)
    - Syntax: *#pragma omp task* will add the following structured block to the task queue
        - Tasks are automatically handled by other threads, if inside omp parallel
        - Structured block: default next line [e.g. function call]/curly brace contents
        - Task barrier: wait until all tasks in buffer are completed, before continuing
            - *#pragma omp taskwait* [omp barrier also does the same]
        - #pragma omp task within #pragma omp parallel will create one of that task for each thread
        - *#pragma omp single* { *code* } [inside an omp parallel] - only one thread executes the contained code
            - Has an implicit barrier at the end [nowait clause -> no barrier]
    - #pragma omp task
- Task construct - explicit tasks
    - Create team of threads [pragma omp parallel]
    - One thread [in omp single] makes tasks [omp task]; other threads handle those tasks automatically as they are created
        - Tasks defined inside omp single are still handled by other threads
            - Original thread will also work on tasks once omp single completes

**Memory Models**

- Memory models defined in terms of: coherence and consistency
    - *Coherence* - behavior of system when single address is accessed by multiple threads
        - Multiple cores may look at the same value, e.g.
    - *Consistent* - ordering of reads/writes/syncs (RWS) with various addresses, by multiple threads
    - OpenMP shared memory model - all threads share an address space
- Consistency models define constraints on RWS ordering
    - Limit reordering - changes in the order of operations
    - ***Sequential consistency*** - all operations remain in program order [i.e. order specified in the program]
        - Program order = code [compiled] order = memory commit [machine instruction] order; all processors see the same overall order

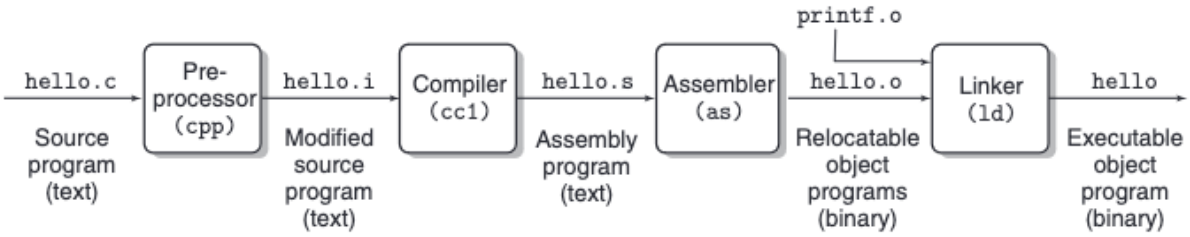
    - ***Relaxed consistency*** removes some of the constraints of sequential
    - OpenMP weak consistency - reads/writes to diff addresses can be freely reordered
        - Synchronizations create wall (***memory fence***) - cannot reorder across synchronizations
- **Flush** - defines a sequence point at which a thread is guaranteed to have the same view of memory as a flush set
    - **Flush set** – all thread-visible variables [*flush()*], or just a specified set/list of variables [*omp flush(list)]*
        - Flushes with overlapping flush sets cannot be reordered
    - Flush forces data to be updated in memory, so other threads see most recent value
        - All R/W operations overlapping the flush set must either complete before, or execute after, flush
        - OpenMP synchronizations have implied flushes
            - Entry/exit to parallel regions, barriers, entry/exit to critical regions, setting/unsetting of locks
        - pragma atomic write, pragma atomic read prevent ***race conditions*** [only one reads/writes at a time]
- *Threadprivate* - makes global data private to a thread
    - Private vs threadprivate - private masks global variables [private-ness ends once structured block completes], threadprivate preserves global scope within thread [variable remains private for lifetime of thread]
        - threadprivate(variable) [variable name]
        - copyin(variable) - analogous to firstprivate
        - copyprivate(variable) - analogous to lastprivate

**Exceptional Control Flow**

- The system: I/O connected to CPU via the system
- Main purpose of ***CPU architecture*** is to ***execute instructions***
    - May be user [application] or kernel instructions
- **Control flow** - sequence of instructions executed by the CPU
    - Mechanisms for changing control flow: jumps and branches, calls and returns
    - **Exceptional control flow** - different events/changes to the system state may warrant changes to the control flow
        - ***System state events***: data arrives from disk/network adapter [into memory], program error (e.g. division by zero, invalid memory accesses), user interrupts [C-c], system timer expires, etc.
- **Exception** - transfer of control to the OS kernel in response to some event (i.e. some change in processor state)
    - ***Kernel*** - memory-resident part of the OS
        - Has system-level view (can access disk, e.g.)
    - Temporarily redirects control from user code to ***exception handler*** before returning
        - May return to same instruction that was left, or next instruction
    - Built on ***exception tables*** – array in memory, mapping different exceptions to pointers to code for exception handlers (similar to a jump table)
- **Classes of exceptions:**
    - ***Asynchronous exceptions*** (***interrupts***) – caused by events external to the processor, and can occur at any time
        - Indicated by setting processor's interrupt pin
        - Ex: timer interrupt, I/O interrupt from external device (e.g. C-c)
    - ***Synchronous exceptions*** – caused by events arising from execution of an instruction
        - ***Traps*** – intentional (programer-set) exceptions
            - Returns control to next instruction upon resuming
            - Ex: system calls (memory reads/writes, requests to the kernel), breakpoints, special instructions
        - ***Faults*** – unintentional, but possibly recoverable exceptions
            - Either re-executes faulting instruction, or aborts

            - Ex: page faults [accesses to unmapped/protected memory, sigseg faults], protection faults, floating point exceptions
        - ***Aborts*** - unintentional, unrecoverable exceptions; immediately abort the program upon occurring
            - Typically occur as a result of system check failures
                - Certain portions of memory may be shut off; different "unwanted" instructions may be turned off [set read-only]
            - Ex: illegal instructions (OS-unsupported instructions, divides by 0, e.g.), parity errors, machine checks
- ***System calls*** are calls by a program to system call instructions
    - Are associated with unique ID numbers
    - Ex: read/write/open/close files
- ***Page faults*** involve attempts to access memory locations not currently on DRAM [in memory]; require kernel copying from disk to memory
    - Also requires OS decide what data to eliminate from memory
    - Returns to and re-executes the faulting instruction upon resolution
    - Invalid memory references result in page faults, but do not have a corresponding place in disc to draw from
        - SIGSEGV – page fault returns signal to user process, signaling abort [fault was not recoverable]

**Linking**




- Steps of compilation:
    - Preparation
    - Parsing – discerning the meaning of different program tokens (“for”, “while”, etc.)
    - Code generation – conversion of code into machine instructions
    - ***Linking*** – combination of multiple object files into a single executable file
- **Types of linking:**
    - ***Static linking*** – code included during the compilation phase
        - Code from program files are compiled into .o object files
            - Relocatable – contains text and data segments, but have not yet been placed in memory (associated with memory addresses)
        - Linker creates a single fully linked, executable object file from relocatable object files, with all code/data required for the program
            - Has actual text, data segments associated with memory addresses
    - ***Dynamic linking*** – putting “hooks” to external libraries in compiled executables, rather than actually including the literal code of the libraries
        - Removes the requirement to actually put said external libraries in every program that is compiled
            - Drawback: does not allow for cross-program optimization
- Reasons for linking:
    - *Modularity* - programs can be written as collections of smaller source files, rather than one massive file
        - Can build libraries of common functions (e.g. math, C stdlib)
    - *Efficiency* – can compile different source files separately
        - Dynamic linking saves space relative to static linking
- *Linker functions:*
    - **Symbol resolution** - define and reference ***symbols*** (global variables/functions)

        - Store symbol definitions in object file ***symbol tables***
            - **Symbol table** – an array of structs, containing the name, size, and location of each symbol
                - Are associated with one symbol definition during symbol resolution step of linking
    - **Relocation** – linker merges separate code, data sections into single sections
        - Replaces relative locations of symbols in .o files, with final absolute memory addresses in the executable
            - Updates references to symbols with new positions
- *Types of object files:*
    - **Relocatable object files** (.o) – compiled, but not linked files
    - **Executable object files** (a.out default) – statically linked, executable file
    - **Shared object file** (.so) – special form of relocatable object, can be pulled in dynamically at runtime via dynamic linking
- File contents of different relocatable object files all combined into a single file during the relocation process
    - Contains *headers*, *text* (system code, user functions), *data* (system/program data)
- **Packaging common functions:**
    - ***Approach 1*** – creating a large source file (.o) containing functions; statically linked into each importing program
        - Inefficient to link a large source file every time
    - ***Approach 2*** – placing every function in a separate source file (.o), which may be specifically linked
        - More efficient than a single .o, but annoying to program
    - ***Approach 3*** – ***static libraries*** (.a – *archive files*) combine a number of relocatable object files within a single index
        - Linker then knows how to individually link archive relocatables into executable as needed
        - Archiver creates static libraries from a set of .o files
        - Statically linked; has all data, instructions needed for execution without pulling from external libraries
        - Downsides:

            - Multiple programs using the same archive functions would each need to individually store the instructions (redundant code)
            - Minor changes to system libraries may require explicit relinking of all importing applications
    - ***Approach 4*** (***dynamic linking***): code and data are stored in ***shared libraries*** are loaded and linked into an application dynamically
        - Static linking process creates only partially linked executable; fully linked executable only created in memory by dynamic linker, later
        - May be loaded and linked at either ***load-time*** or ***run-time***
            - Load/run time: process of loading executable into memory, dynamically linking libraries, and executing
        - Also called dynamic link libraries, DLLs, .so files
- **Library interpositioning** – linking technique allowing programmers to intercept/observe calls to arbitrary functions
    - May occur at compile time, link time, or load/run time
    - Functions:
        - Monitoring/profiling – tracking number of calls to functions + call sites, function arguments
            - Malloc tracing – detecting memory leaks, generating address traces

**Virtual Memory**

- **Virtual memory** –the "memory" that programs see is not the same as the physical memory of the system (DRAM)
    - Applications do not directly see where their data is stored in memory/on disk
    - Requires translation between virtual, physical addresses, incurring some overhead
    - *Benefits:*
        - Can run the same programs on different systems, including with different memory structures/addressing
        - Can run several applications together without needing to modify them to not overwrite each other's space
    - *Drawbacks:*
        - Allocation of physical memory may not be perfectly efficient
        - No easy solution for when physical memory runs out
- Virtual memory built on **page tables** – stores **page table entries** (**PTEs**), mapping page numbers (virtual addresses, as given by the application) to physical addresses in memory
    - Recall: memory divided into ***pages*** (chunks of memory)
        - Page size operating system-specific
        - Total memory space = page size * # pages
    - Also contains protection bits (e.g. read-write permission), valid bit (whether address exists in physical memory)
        - Valid bit false results in a ***page fault***
    - Page table stored in memory
        - **Page table base register** (**PTBR**) points to physical page table address for current process
    - Page table may be broken up into levels of page tables, for space efficiency
        - **Memory management unit** contains ***translation look-aside buffer*** [special cache containing portion of the page table], ***page walker*** [specialized hardware for traversing/searching levels of a page table]
- Virtual addresses consist of a ***virtual page number*** and ***page offset***
    - Virtual page number essentially acts as an index for the page table
    - ***Page offset*** - bits for storing offset from initial memory address of page to desired byte within the page

        - Offset size = $\log_2$(page size) [in bits]
        - Virtual page offset (VPO) equal to physical page offset (PPO)
    - Virtual page number represents upper bits of a physical address; page offset represents lower bits
- **Steps of page retrieval:**
    - *(1)* Processor sends virtual address to MMU
    - *(2)* MMU fetches PTE from page table in memory
    - *(3)* MMU sends fetched physical address to cache/memory
    - *(4)* Cache/memory returns the value at that address to the processor
    - *Page faults*: MMU fetches PTE with valid bit false at (2)
        - *(3)* Handler identifies victim [if dirty, pages out to disk]
        - *(4)* Handler pages in new page, with help from OS
            - Updates PTE in memory to map to new page
        - *(5)* Returns to original process, resuming halting instruction
- **Translation look-aside buffer** (**TLB**) is a special cache/caches in the MMU, containing a portion of the overall page table
    - In the case of a TLB hit, a memory access to the page table is skipped [faster]
- Page tables may be broken up into ***multi-level page tables***, where each level points to tables of the next level (and final level contains desired virtual memory-physical memory PTEs)
    - Only first level table needs to be stored in physical memory; can page in other levels of the page table (otherwise stored in secondary storage) as needed
        - Saves space on memory, e.g. in the case of an overly large page table
    - PTEs in a page table may be left unallocated if not needed
    - Virtual address contains indexes for each level of page table

**RISC**

- Reminder: ***Instruction set architecture*** (***ISA***) – part of a computer, serving as an intermediary between hardware and software
- Types of ISA: ***CISC*** vs ***RISC***
    - **Complex instruction-set computers** (**CISC**) – instruction set supporting a large number of different instructions
        - Efficient memory-wise; initially popular due to the high cost of memory historically (used in x86-64, e.g.)
        - A single instruction may take several cycles to complete
    - **Reduced instruction-set computers** (**RISC**) – uses simpler set of instructions/operations, allowing for a simpler associated hardware
        - Con: Simple nature of instruction means performing a single task requires more instructions (using more space in memory); a single instruction only takes one cycle to complete
        - Pro: Since instructions are smaller and simpler than CISC, they can be rearranged more easily (e.g. by a compiler) for better optimization
            - Can be parallelized more easily, e.g. (better performance)
            - Essentially breaks up single CISC instructions into sequences of smaller instructions
- *CISC vs RISC:*
    - RISC typically has a larger number of registers than CISC
        - RISC only has one register type, relative to CISC's many
        - Done to minimize number of memory accesses needed, for performance
    - RISC architectures do not allow for taking memory addresses as arithmetic operands (must perform load, arithmetic operation in separate instructions)
    - RISC architectures have three-address instructions, where CISC have two
    - CISC has variable-length instructions; RISC originally only supported 32-bit
- Misc:
    - Pipelining (performing different operations simultaneously) vs parallelism (performing the same operation multiple times simultaneously)
    - Intel gradually moving from CISC to RISC

        - x86-64 is built on CISC, but has been trying to avoid using more complex instructions (i.e. use more RISC-like instructions)
    - ARM built on RISC
    - CISC vs RISC is not a clear line
        - RISC originally only had one size of instruction (32-bit); has been branching out into supporting different types of instructions (e.g. 16-bit)
            - Has introduced macro op fusion to fuse instructions into more complex instructions
    - Fixed-length vs variable-length instructions
        - Fixed-length is easier to decode (can always read the same number of bytes), but less space-efficient (cannot make smaller instructions, even if the full length results in wasted space)
            - Is notably also less susceptible to code injection with gadgets (cannot splice into the middle of an instruction as easily, e.g.)

**MIPS**

- **MIPS** – basic RISC architecture
    - Words 4 bytes in memory; aligned at multiples of 4
    - Is a *load-store architecture*: every operand of a MIPS instruction must be in a register [generally speaking]
        - Requires explicit data movement (i.e. explicit loads, stores), as opposed to x86's memory-addressing operands (e.g. 8(%rax))
    - Only a few basic operations
        - Instruction format: INST DEST, SRC1, SRC2
            - Convention: **rd** = destination register, **rs** = source register, **rt** = source or destination register (depending on instruction), immed = a 16-bit value
- MIPS registers specified with $; can be specified by register number or by name
    - MIPS has 32 32-bit registers for general operations; 32 for floating-point
        - Temporary registers: $t0 - $t9
        - Callee-saved registers: $s0 - $s3

    - Also contains 32-bit program counter (PC), registers HI and LO for multiplication/division, other special-purpose registers
        - $0 ($zero) always holds the value zero (can be specified with 5-bit register name rather than 32-bit integer value, for compactness)
- Only a few types of MIPS instructions: **R[egister]**, **I[mmediate]**, **J[ump]**
    - **R-type instructions**: 6-bit opcode; 5-bits each for RS, RT, RD; 5-bit shift amount (used for shift instructions); 6-bit func field
        - All R-types use the same 6-bit opcode (all 0s); store actual function encoding in 6-bit func field
            - Allows for $2^6$ - 1 different I/J instructions (denoted by opcode) + $2^6$ different R instructions (denoted by func)
    - **I-type**: 6-bit opcode, 5-bits for RS, RT; 16-bits for immediate
    - **J-type**: 6-bit opcode, 26-bit immediate (used for jumps)

**MIPS Instructions**

- MIPS relies on explicit load/store instructions (**LW** = ***loadword***/**SW** = ***storeword*** read from/write to memory, respectively, e.g.)
    - LW is I-format: LW RT, IMMED(RS) places value of IMMED(RS) into RT
        - Memory address to be read represented as value of RS + sign-extended immediate [displacement]
    - SW: SW RT, IMMED(RS) stores RT at IMMED(RS)
        - Memory address to be stored into represented as value of RS + sign-extended immediate
- Arithmetic operations: ADD vs ADDI [*add immediate*], sub
    - ADD: ADD rd, rs, rt vs ADDI: ADDI rt, rs, immed
        - R-type: R[RS] FUNC R[RT] -> R[RD]
    - ADDIU [add unsigned immediate]
    - No single analogous LEAQ instruction
    - addu $s7, $0, $ra – push return address to $s7
- Branching: BEQ (branch equal) and BNE (branch not equal)
    - Branching: BEQ rs, rt, target [target representing some address or label in memory], change program counter on jump

    - Comparison instructions: SLT (set less than) + SLTU (set less than [unsigned])
        - SLT: SLT rd, rs, rt (set rd to 1 if true (rs < rt), 0 if false)
- Jump instructions are unconditional
    - J target [jump to target], JR rs [jump to target stored in register]
    - JAL [jump and link] used to call procedures; JR $31 [jump to return address] for returning from procedures
        - JAL: JAL target jumps to target; stores value of PC [return address, essentially] + 4 [next instruction] in $31
        - JALR: JALR rs, rd jumps to address in rs; stores value of rd in $31
- Logic: AND, OR, XOR [+I]
    - Immed constant limited to 16 bits – LUI loads constant to upper bits of a register
- Pseudo-instructions: instructions available in assembly, but eventually broken down into series of simpler instructions in machine code
    - Move, clear, li [load 16-bit/32-bit immediate], la [load label address: la $t, A]
- Syscalls: print/read integers and strings [from memory; address as input]; exit/end program
- MIPS instructions also stored in text, global segments
    - .text – defines where program is stored
    - .global__start: calls start function in global scope
        - __start: defines start function (label)
    - Can label strings (e.g str: .asciiz “hi”), use label (str) when referring to memory addresses (with la, e.g.) in assembly