CS33

Reinman

Spring 2019

Contents

CS33

Overview

- *data representation*: ints, floats, bit manipulation, casting, structs/unions
- *machine level programming*: x86, bomb-lab, attack-lab, MIPS, RISC vs. CISC
- *memory hierarchy*: caching, locality (temporal and spatial), VM, stack vs. heap
- *program optimization*: blockers, optimizations, false-sharing
- *parallelism*: OpenMP, problems (races, deadlocks), solutions (synchronization, critical sections)
- *system*: linking/compile process, traps/exceptions
- final format:
    1. fill in the blank
    2. MIPS <-> x86
    3. OpenMP
    4. structs/unions
    5. code optimization
    6. attack lab
    7. boss question

Bits and Bytes

- looking under the *abstraction* layer of the machine:
    - ISA - instruction set architecture (eg. x86-64)
        * communicates higher level languages into the underlying machine / architecture
- compilation steps:
    1. pre-processor directives
    2. compiler creates machine language code, *code generator* phase (still text readable)
    3. assembler creates object file (machine readable)
    4. linker creates an executable or *binary*
- looking at how data is handled and represented:
- based on representing information as *bits*
    - every bit is either a **0** or a **1**

  - encoding/interpreting bits in a certain way:
    * *instructions* for the computer
    * *represent* and manipulate numbers, sets, string, etc.
    * why bits? Due to their **electronic implementation**...
      · easy to store bistable elements (*voltage*), and reliably transmitted on wires
- can store/organize 8 bits in a **byte**
  - memory is usually byte-addressable
  - ranges from 8 0's (*0*) to 8 1's (*255*) (256 unique values)
  - also convenient look at bits using base 16, **hexadecimal**
    * uses 0 through 9 and A through F
    * written in C as *0x...*, eg. *0xFA1D37B*
    * one hex digit represents *four* binary bits, one byte is encoded in two hex characters (more convenient to visualize)
    * memory dumps show the address and the contents there, usually with hex-pairs (for one byte)
      · (formed in hex, but really actually *stored* in binary)
- ASCII:
  - interpreting numeric values as characters
  - what matters is how data (just a pattern of bits) is *interpreted*

Table 1: *Example Data Representations*

C Data Type	Typical 32-Bit	Typical 64-Bit	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	8	8
float	4	4	4
double	8	8	8
long double	-	-	10/16
pointer	4	8	8

Bit-Level Manipulations

- *boolean algebra* - algebraic representation of logic, applied at a bit level
  - encode 1 as **true**, 0 as **false**
  - `A & B == 1` when both A == 1 and B == 1
  - `A | B == 1` when either A == 1 or B == 1
  - `~ A == 1` when A == 0 (complementary operator)

  - `A ^ B == 1` when either A == 1 or B == 1, but *not* both (exclusive-or (*Xor*))
- when operated on bit vectors, operations are applied *bitwise*
  - eg. in an 8-bit operation, individual bits are **paired** off
- can represent and manipulate sets using bits:
  - a bit would be set to 1 if it is contained in some range
  - eg. *01101001* (corresponding to the range *76543210*) would indicate the set *{ 0, 3, 5, 6 }*
  - could use bit manipulation on multiple sets to find their intersection, union, symmetric difference, and complement...
  - operations could be cheaper and more efficient, but:
    - may have to process the data to interpret it
    - mapping is implicitly only tied to position of the bits
- all of these operations are available in C and can be applied to any built-in data type
  - arguments are viewed as a bit vector and are applied bit-wise
- examples:
  - `~0x41 == 0xBE` (`~01000001 == 10111110`)
  - `0x69 & 0x55 == 0x41` (`01101001 & 01010101 == 01000001`)
  - `0x69 | 0x55 == 0x7D` (`01101001 | 01010101 == 01111101`)
  - can use the complementary operator to quickly find the negative of a signed:
    - `x + ~x == -1` (all 1's)
    - so, `~x + 1 == -x`
- these are in **contrast** to *logical* operators!
  - view 0 as false, all *nonzero* as 1
  - *always* return 0 or 1
  - early termination / *short-circuiting*!
  - examples:
    - `!0x41 == 0x00`
    - `!0x00 == 0x01`
    - `0x69 && 0x55 == 0x01`
    - `p && !p` (avoids null pointer access)
- **shift operations** move bits so that they are of a higher or lower significance
  - *left shift* (<<) - throw away extra bits on the left, fill with 0's on the right
    - eg. `01100010 << 3 == 00010000`
  - *right shift* (>>) - throw away extra bits on the right
    - fill with either 0's (logical shift) or replicate most significant bit (arithmetic shift)
    - `logical 01100010 >> 2 == 00011000`
    - `arithmetic 01100010 >> 2 == 00011000`

* `logical 10100010 >> 2 == 00101000`
* `arithmetic 10100010 >> == 11101000`

Integers

Integers can be **unsigned** or **signed**.

Unsigned integers represent *positive* values 0 or greater. Mathematical representation (where w is the width of the integer and i is the index from the right of the integer):

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i \tag{1}$$

Signed integers represent *all* integers, positive and negative. This form is called **Two's Complement**.

$$B2T(X) = -x_{w-1} \cdot 2^w - 1 + \sum_{i=0}^{w-2} x_i \cdot 2^i \tag{2}$$

Table 2: Examples of C's *short* (two bytes long)

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

In Two's Complement form, the most **significant bit** indicates the sign: 0 for nonnegative, 1 for negative.

Unsigned values have a *discrete*, numeric range from 0 (UMin, all 0's) to $2^w - 1$ (UMax, all 1's).

Two's Complement values also have a *discrete*, numeric range from -2^{w-1} (TMin, all 0's except for 1 as the sign bit) to $2^{w-1} - 1$ (TMax, all 1's except for 0 as the sign bit).

Note that this range is **asymmetrical** ($|Tmin| = Tmax + 1$) and *less* than the range for unsigned integers ($UMax = 2 * TMax + 1$).

Interesting value: -1 in Two's Complement form is represented as all 1's.

To get the negative of a value in signed form, flip all the bits and add 1.

Listing 1: Integer Ranges in C

```
#include <limits.h>
int a = ULONG_MAX;
int b = LONG_MAX;
int c = LONG_MIN;
// note that these are platform specific!
```

More **properties** of unsigned and signed values:

- *equivalence*: same encodings for nonnegative values
- *uniqueness*: every bit pattern represents a unique integer value, and every representable integer has a unique bit pattern
    - some inverting of mappings are possible

Conversion and Casting

Mapping in either direction between unsigned and Two's Complement **maintains** the same bit pattern. The bit representation is kept **the same** and is reinterpreted, depending on the lens of either unsigned or signed (due to the difference in the highest order weighting, essentially adding or subtracting 2^w). For example, `1000` in Two's Complement is -8, `1001` is -7…and `1111` is -1. This is unlike floating point casting, in which case the bit pattern *does* change.

Positive integers are equivalent between the two representations (0 in the most significant bit position). However, for *negative* integers, the large negative weight *becomes* a large *positive* weight. This leads to an **ordering inversion**, where negative values become large positive values.

Table 3: Casting between Signed and Unsigned

Bits	Signed	Unsigned	Notes
0000	0	0	equivalent for positive values…
0001	1	1	
0010	2	2	
0011	3	3	
0100	4	4	
0101	5	5	
0110	6	6	
0111	7	7	TMax $\Rightarrow$ TMax
1000	**-8**	8	TMin $\Rightarrow$ TMax+1, $\pm$ 16
1001	**-7**	9	

Bits	Signed	Unsigned	Notes
1010	-6	10	
1011	-5	11	
1100	-4	12	
1101	-3	13	
1110	-2	14	-2 ⇒ UMax-1
1111	-1	15	-1 ⇒ UMax

Listing 2: Casting in C

```c
// constants by default considered to be signed integers.
// need a U suffix for unsigned, eg. 1234567U

// Explicit Casting:
int tx, ty; // two's complement
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;

// Implicit Casting: (occurs via assignment and procedure calls)
// if mix of types, signed (int) implicitly casts to unsigned
tx = ux;
uy = ty;
```

Table 4: Some Casting Surprises

Constant_1	Constant_2	Relation	Evaluation
`0`	`0U`	`==`	unsigned
`-1`	`0`	`<`	signed
`-1` (treated as UMAX, in bit, all 1's)	`0U`	`>`	unsigned
`2147483647` (TMAX)	`-2147483647-1` (TMIN)	`>`	signed
`2147483647U` (0 followed by all 1's)	`-2147483647-1` (1 followed by all 0's)	`<`	unsigned
`-1`	`-2`	`>`	signed
`(unsigned)-1` (all 1's)	`-2` (all 1's followed by one 0)	`>`	unsigned

Constant$_1$	Constant$_2$	Relation	Evaluation
`2147483647`	`2147483648U`	`<`	unsigned
`2147483647`	`(int)2147483648U` (TMIN when casted, 1 followed by all 0's)	`>`	signed

Sign Extension

Given a w-bit signed integer, how do we convert it to a w + k-bit integer with the same value? (eg. converting from a short to an int)
Make k copies of the sign bit to fill the k new bits that are being added.

Table 5: Sign-extending short to int

	Decimal	Hex	Binary
`short int x;`	15213	3B 6D	00111011 01101101
`int ix = (int) x;`	15213	00 00 3B 6D	00000000 0000000 00111011 01101101
`short int y;`	-15213	C4 93	11000100 10010011
`int iy = (int) y;`	-15213	FF FF C4 93	11111111 1111111 11000100 10010011

Expanding:

- unsigned: zeros added
- signed: sign extension
- both yield expected results

Truncating:

- unsigned/signed: bits are truncated
- result reinterpreted
- unsigned: equivalent to a mod operation (throwing away higher bits)
- signed: similar to mod operation
- for small numbers yields expected behavior

Addition

Unsigned Addition: If the operands are integers of w bits, the *true sum* to account for all possible additions would be w+1 bits (to account for adding some of the largest possible values). If we *discard* the carry, the sum would be w bits. In standard addition, the carry is ignored, and this operation is the same as the operation $UAdd_w(u, v) = (u + v) mod 2^w$ (a *modular sum*). That is, the value is too large and *overflows*/wraps around (will only wrap around at most once).
Two's Complement Addition: Signed addition works in the same way as unsigned at

the *bit level* (casting to unsigned yields the same result). However, the difference is that the overflow can be both *positive and negative.* A value that becomes too *negative* becomes a positive value at most once, while a value too *large* becomes a negative value at most once. Only values in the middle area follow the true sum.

Multiplication

Exact results (*true product*) of product of two *w*-bit numbers can be bigger than *w* bits: up to *2w* bits for unsigned, *2w-1* bits for signed negative (min), and *2w* bits for signed positive (max).

- to maintain exact results, we would have to keep expanding the word size with every product.
- in unsigned multiplication in C, *w* bits are discarded, so high order *w* bits are ignored. Similar to operation $UMult_w(u, v) = (u * v) mod 2^w$.

Multiplication is an expensive operation made up of numerous additions and shifts. When applicable, the compiler replaces some multiplication operations with simpler ones, eg. `u << k` gives $u * 2^k$, as long as the values aren't too large. Therefore, `u << 3 == u * 8`. However, we can still use this technique to calculate multiplication involving non-powers of two: `(u << 5) - (u << 3) == u * 24`. This shift/add operation is generally faster than multiply so, compilers will automatically generate this new code in a process called *strength-reduction* (but can only happen at runtime, not compile-time).

Division

Has a similar *strength-reduction* where quotient of unsigned by power of 2 can be simplified (floor-function, however). For example, `u >> k` gives the floor of $u/2^k$.

Listing 3: Mistakes with Unsigned

```
unsigned i;
for (i = cnt-2; i >= 0; i--)
  // if cnt is small, cnt-2 is a negative value, and will be casted to a very large
    unsigned value...
  // leads to an array access out of bounds!
  a[i] += a[i+1];

// instead:
unsigned i;
for (i = cnt-2; i < cnt; i--)
```

```
  // comparison is done relative to an unsigned value, so array indices are now bound
     properly
  ...

// even better:
size_t i; // size_t is defined as an unsigned value where length = word size
for (i = cnt-2; i < cnt; i--)
  ...

// another example:
#define DELTA sizeof(int)
int i;
for (i = cnt; i-DELTA >= 0; i -= DELTA)
  // since sizeof() returns an unsigned, we are essentially doing an unsigned
    operation, may decrement to a large unsigned value
  // will never not be greater than 0 in this loop
  ...
```

When to use Unsigned?

- do use when performing modular arithmetic (no negatives)
- do use when using bits to represent sets (logical right shift, no sign extension)

Representation in Memory

- programs refer to data by address
  - can envision memory as a very large array of bytes
  - address in index into an array
  - pointers store those addresses
- private address spaces for a single process, *virtual memory* is where memory can be shared between processes
  - illusion of more memory
- any computer has a *word size* - nominal size of integer-valued data / addresses
  - eg. 32 or 64-bit word size
  - storage is different for data types in different word sizes
- addresses specify byte locations, usually first byte in a word
- for the actual ordering of multi-word bytes in memory, there are two conventions:
  - *big endian*: least significant byte has highest address (left to right)
    * eg. 0x100: 01 23 45 67

    - *little endian*: least significant byte has lowest address (right to left)
        - eg. 0x100: 67 45 23 01 (ordering of bits remains in order **within** a byte)
- **note**: endian applies to integral types, not arrays
    - eg. strings in C are simply array of characters:
        - allocating a C-string is **not** a multi-byte type, still an *integral* type of char
        - so, the C-string is still stored in order, *regardless* of endian, ie. there is no byte ordering
    - for all arrays: arr[0] always at the lowest address, arr[len-1] always at the highest address
        - **but** each element *in* the array is still stored according to its endian

Integer C Puzzles

Given:

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

Table 6: Integer Puzzles

Statements	Solution	Explanation
`x < 0` $\xrightarrow{?}$ `(x*2) < 0`	false	Negative value could overflow and become positive.
`ux >= 0`	true	Unsigned comparison is always greater than 0.
`x & 7 == 7` $\xrightarrow{?}$ `(x<<30) < 0`	true	7 in binary is ...111, so x ends in 111.
`ux > -1`	false	-1 in unsigned comparison is UMax.
`(3-5U) > 0`	true	Unsigned comparison is always greater than 0.
`(3-5) > 0`	false	
`x > y` $\xrightarrow{?}$ `-x < -y`	false	If we attempt to invert TMin, we would get itself.
`x * x >= 0`	false	Could overflow and become negative.

Statements	Solution	Explanation
`x > 0 && y > 0` $\overset{?}{\rightarrow}$ `x + y > 0`	false	Could overflow and become negative.
`x >= 0` $\overset{?}{\rightarrow}$ `-x <= 0`	true	
`x <= 0` $\overset{?}{\rightarrow}$ `-x >= 0`	false	If we attempt to invert TMin, we would get itself.
`(x | -x) >> 31 == -1`	false	Not when x is 0.
`ux >> 3 == ux/8`	true	Property of strength reduction.
`x >> 3 == x/8`	false	Negative division doesn't work properly.
`x & x-1 != 0`	false	Not when x is 1.

Floating Point

- overal form: $(-1)^S M2^E$
- numerical form includes:
  - a sign bit
  - exponent that weights the value by power of 2 (encoded in exp field)
    - increasing bits for E increases the range
  - a significand or mantissa (encoded in frac field, usually fractional value between 1 and 2)
    - increasing bits for M increases the precision
- note that in floating point, there is no overflow, instead values *saturate* at +- infinity
  - no *asymmetry* in floating point notation either
- trade off: larger exp field means a longer range, while larger frac field means more precision
- *normalized values*: when exp is not all 0's or all 1's
  - exp is coded as a **biased** value, or $E = Exp - Bias$ -depending on sign bit allows for positive and negative exponents
    - bias is calculated as $2^{k-1} - 1$, where k is the number of exponent bits
    - this bias calculation allows for easy comparisons, because the exp field can be interpreted as unsigned
  - significand is coded with an implied leading 1, eg. 1.xxxx
    - each field correspondes to 2^{-k}, where k is the field from the left
      - ie. fractional powers of 2

        * minimum when frac is all 0's (1)
        * maximum when frac is all 1's (~2)
        * similar to an extra bit
- *denormalized values*: only when exp is all 0's
    - exp is interpreted as $1 - Bias$ instead of $0 - Bias$
    - significand is coded with implied leading 0, eg 0.xxxx
    - creates a more continuous spectrum
    - all 0's can represent both +-0 (depending on sign bit)
    - otherwise, represents the numbers closest to 0, *equispaced*
        * distribution gets denser towards 0
- *special values*: only when exp is all 1's
    - if frac is all 0's, represents infinity
        * can be positive and negative
        * operation that overflows
        * eg. 1.0/0.0, -1.0/-0.0, 1.0/-0.0
    - if frac is nonzero, represents NaN
        * no numeric value can be determined
        * eg. sqrt(-1), inf - inf, inf * 0
- for a float, M takes 23 bits, E takes 8 bits
- for a double, M takes 52 bits, E takes 11 bits

s	exp	frac	E	Value
0	0000	000	-6	0, closest to zero
0	0000	001	-6	1/8 * 2^{-6}
0	0000	010	-6	1/4 * 2^{-6}
0	0000	111	-6	7/8 * 2^{-6}, largest denorm
0	0001	000	-6	8/8 * 2^{-6}, smallest norm
0	0001	001	-6	9/8 * 2^{-6}
0	0110	111	-1	15/8 * 2^{-1}, closest to 1 below
0	0111	000	0	8/8 * 1
0	0111	001	0	9/8 * 1, closest to 1 above
0	1110	111	7	15/8 * 128, largest norm
0	1111	000	na	infinity

Operations

- first compute exact result, then make it fit into desired precision
- possibly overflow if exponent is too large or round to fit into frac
    - overflowing the **precision**

- when casting between int / doubles, bit patterns may change due to rounding
- uses *round even* convention to round to the nearest even number if at the halfway point
  - in binary, even when least significant bit is 0
  - half-way when bits to the right of rounding position is 1 followed by all zeros
  - eg. 10.11100 in binary will round to 11.00 (or 3)
  - eg. 10.10100 in binary will round to 10.10 (or 2.5)
- *multiplication*:
  - for sign: check S1 ^ S2
  - significand: M1 x M2
  - exponent: E1 + E2
  - don't have to check for overflow, values saturate at infinity
- still have to fix and round result
  - if M >= 2, shift M right, increment E
  - if E out of range, overflow
  - round M to fit frac precision
- *addition*:
  - align up the binary points at E1 - E2, then add
  - similar fixing steps for multiplication
- *casting* between int, float, and double does change the bit representation
  - double/float -> int truncates fractional part
  - like rounding towards zero
  - undefined when out of range
- int -> double exact conversion as long as int has `<=` 53 word size
- int -> float will round according to rounding mode

Floating Point Puzzles

- `x == (int)(float) x`, false, x may have more bits of precision and become rounded
- `x == (int)(double) x`, true, double has more precision than an int
- `f == (float)(double) f`, true, double is larger than the float in all cases
- `d == (double)(float) d`, false, double holds less space
- `f == -(-f)`, true, symmetric range
- `2/3 == 2/3.0`, false, rounding restrictions
- `d < 0.0` -> `((d*2) < 0.0)`, true, values saturate, no overflowing
- `d > f` -> `-f > -d`, true, symmetric range
- `d * d >= 0.0`, true, no overflowing
- `(d+f)-d == f`, false, there may be a precision issue
  - with very small f, the impact may be rounded away because of the smaller

precision
  - finite amount of representable precision

Machine Programming: Basics

Brief History

- Intel x86 dominates the market (except for mobile computing)
- maintained backward compatibility
- *CISC* or complex instruction set computer (used by Intel)
  - older, more compact code, less memory
  - many different instructions with different formats
- *RISC* or reduced instruction set computer
  - generally faster, focuses on parallelism, but Intel has matched performance with *CISC*
    * parallelism uses less power

Assembly / Machine Code Overview

- *ISA* or Instruction Set Architecture
  - eg. Intel x86, x86-64, ARM for mobile phones
  - parts of processor design that links the hardware to software interface
    * what's needed to program machine code / compiler
  - eg. registers in memory, or the instruction set specifications
  - does not include the more fundamental implementations of the hardare (*microarchitecture*)
- **machine code**: programs written in 0's and 1's
- **assembly code**: text representation of machine code
- computer has a **cpu** and **memory**
- in the **cpu**:
  - **PC**: program counter, address of next instruction (AKA **RIP**)
  - **register file** is for heavily used program data, faster access than normal memory
  - **condition codes** store *status* information, used for *conditionals*
- **memory** can be modeled as a byte addressable array
  - holds code, user data, heap of dynamically allocated variables
  - uses a stack for procedures
- C to object code:

  - gcc *compiler* takes C program, transforms into another text program called assembly program
    * -Og disables optimization, -S creates Asm program (.s)
    * (fundamental ISA instructions, human readable)
  - *assembler* makes Asm program into a binary object program
  - *linker* pulls in library to make a binary executable program (resolves references)
    * also deals with *dynamic libraries*
- assembly data types:
  - has an integer type (for data values and addresses) and a floating data type
  - code that holds instructions is stored in various byte sequences of different lengths
  - no aggregate types like arrays / structures
    * *does* have contiguously allocated bytes in memory
- assembly operations:
  - perform arithmetic function on register / memory data
  - transfer data between memory and register (loading or storing)
  - transfer control (by changing the program counter)
    * eg. conditional branches or jumps to/from procedures

Listing 4: Machine Instruction Example

```
// copy value t into memory address in dest
*dest = t;

// move quad word (8-byte) to memory (really a copy)
movq %rax, (%rbx)
// assume register %rax stores value of variable t
// register %rbx holds 64-bit address stored by dest
// *dest goes to a memory location at M[%rbx]
// parens indicate a dereferencing of a pointer

// instruction generated by assembly is stored in memory here
0x40059e: 48 89 03
// 3-byte instruction
```

- can break down / disassemble object code with a dissassembler
  - `objdump -d sum` where sum is a binary, examines object code
  - `gdb sum` and `disassemble sumstore` is another dissassembler

        * `x/14xb sumstore` examines memory content starting at sumstore for 14 bytes

Assembly Basics

- register files are the 16 registers / memory built into the cpu
    - registers are much faster than memory
- historical (IA32) 32-bit registers were incorporated into larger 64-bit registers over time
    - eg. `%rax` and `%r8` refer to 64-bit registers, `%eax` and `%r8d` refer to 32-bit registers
        * also has lower 8-bit (`%ah` and `%al`)and 16-bit (`%ax`) registers **within** 32-bit register (allows for backwards compatibility)
    - can still access these as *lower-order* bytes **within** the full registers (overlap)
        * allows for legacy code as well as more grainularity with data types
        * when using a 32-bit register, the rest of the 64 bits are implicitly set to 0's
    - some unique registers, eg. `%rsp` is a stack pointer
- register breakdown:
    - %rax, %eax, %ax, %ah, %al
- move data with `movq`, takes source and dest operand (quad-word or 64-bit)
    - other suffixes are B for 8 bit, L for 16 bit, W for 32 bit, Q for 64-bit
    - move is more of a copy, doeesn't remove the original source
    - operands can be memory, registers (built-in registers), immediates (constants or literals, with `$` prefix)
        * immediates are specified *directly* inside the instructions (maybe 8 bits)
        * one of the 16 integer registers
        * memory accesses 8 consecutive bytes (q) at address given by register ((`%rax`))
            · check register, and then memory (so from 8 bits to 64 bits, one level of indirection)
        * other memory addressing modes:
            · *normal*, (R) –> Mem[Reg[R]], analagous to pointer dereferencing
            · *displacement*, D(R) –> Mem[Reg[R]+D], D added to register value
            · *complete*, D(Rb,Ri,S) –> Mem[Reg[Rb]+S*Reg[Ri]+D] (checking the *contents* of Rb and Ri)
            · eg. (%rdx, %rcx, 4) - R[rdx] + 4 * R[rcx]
            · eg. 0x80(, %rdx, 2) - can skip elements with comma, 2 * R[rdx] + 0x80
            · base register, index register, scale (multiple of two), displacement

        · or some combination of these components of the complete form
    - *cannot* do a memory-memory transfer in a single instruction
    - other examples, complex because of the CISC ISA:
        * eg. `movzbq %al, %rbx` moves byte to quad and pad with zeroes
        * eg. `movsbq %al, %rbx` moves byte to quad and sign-extend
        * eg. `cltq` is analagous to `movslq %eax, %rax`, except implicitly dealing with the %rax, less operands
- % indicates a register, $ indicates a literal

Table 8: Some movq Operand Patterns

movq Command	C Analog
`movq $0x4, %rax`	temp = 0x4;
`movq $-147, (%rax)`	*p = -147;
`movq %rax, %rdx`	temp2 = temp1;
`movq %rax, (%rdx)`	*p = temp;
`movq (%rax), %rdx`	temp = *p;

Arithmetic and Logical Operations

- `leaq src, dest`: load effective address
    - where src is some address mode expression, and dest is some location to set to address
    - unlike move, **memory** is *not* dereferenced, only an effective address is being computed
        * not using the value of the register as a *pointer*
    - useful to reduce arithmetic computations, also allows for two sources, where one is not modified
    - `movq 8(&rax), %rdx` - M[8+R[rax]] => R[rdx]
    - `leaq 8(&rax), %rdx` - 8+R[rax] => R[rdx], don't use this to go to memory, just compute an effective address
        * eg. calculating a pointer address, or use for simple arithmetic computations (doesn't overwrite original dest)
        * essentially provides 3 address code
        * in this case, parens does not indicate dereferencing
- two operand instructions are where one of the operands is both a source and a destination (two address code)
    - eg. `addq src, dest` –> dest = dest + src
        * more space efficient, but overrides the instruction value
    - also subq, imulq (multiplication), salq, sarg (arithmetic right), shrq (logical

right), xorgq, andq, orq

- some single operand instructions:
  - act as both destination and source
  - incq, decq, negq, notq
- no way to preserve the destination value
- example invalid `mov` instructions:
  - `movl %eax, %rdx` - destination operand doesn't have the right size
  - `movb %di, 8(%rdx)` - source operand doesn't have the right size (moving 1 byte from 16 bits)
  - `movq (%rsi), 8(%rbp)` - can't move from memory to memory
  - `movw $0xFF, (%eax)` - all memory addresses have to be 64 bits, not 32

Listing 5: Understanding Arithmetic Expression

```
long arithm (long x, long y, long z)
{
  long t1 = x + y;
  long t2 = z + t1;
  long t3 = x + 4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}

// in assembly:
leaq (%rdi, %rsi), %rax    // effectively adding %rdi and %rsi
addq %rdx, %rax            // could have also written as a leaq, would take more
   memory
leaq (%rsi, %rsi, 2), %rdx // multiplying %rsi by 3
salq $4, %rdx              // arithmetic left shift by 4, multiplying by 16 (48 = 16*
   3), strength reduction
leaq 4(%rdi, %rdx), %rcx   // x + 4 + t3
imulq %rcs, %rax           // t2 * t5
ret
// compiler keeps track of what variables are "live" or needed
// can improve performance by limiting live variables
```

```
// knows when it can overwrite values

// registers and uses
// %rdi holds argument x
// %rsi holds argument y
// %rdx holds argument z
// %rax holds t1, t2, rval at different times
// (registers get reused often, register coloring, limiting live variables)
// %rdx holds t4
// %rcs holds t5
```

Machine Programming: Control

- the register file is a set of registers within the cpu, vs. separate data in memory
- special registers include the *%rsp*, or stack top pointer (one of the 16 registers for x86-64)
  - and the *%rip* instruction pointer: tracks current instruction to execute (is a program counter or pc)
    - can be changed explicitly as a part of control flow
    - any instruction ends up implicitly changing `%rip` to the next instruction
- the instructions are held in main memory (place in memory called *text*)
  - instructions can be fixed length or variable length
  - all instruction have an op code that specifies the operation
  - and then there are operands
    - eg. RET takes 0 operands, ++ takes 1, add takes 2
- **condition codes** are CF - carry flag, ZF - zero flag, SF - sign flag, OF - overflow flag
  - allow for more complex operations / comparison
  - are single-bit registers
  - are *implicitly set* after arithmetic operations (not set by `leaq`, but set by `movq`)
  - *characterizes* the result of an operation
- CF set when unsigned overflow occurs / carry out of msb
- ZF set if result is 0
- SF set if result is < 0 (if most significant bit indicates < 0)
- OF set when signed overflow occurs `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`
  - where t is result of a op b

- condition codes are also set *explicitly* when during comparisons or tests
  - *only* write to condition codes, without setting a destination
  - `cmpq b, a`, like computing a-b without setting destination
  - `testq b, a` like computing a&b without setting destination
    - useful for masking
- reading condition codes:
  - can save into a general register value
  - or, can just jump, like a goto
- `setX` sets the 8 lower-order bits of a destination based on combinations of condition codes
  - doesn't overwrite the other bits
  - `sete` checks ZF, `setne` check ~ZF, `sets` checks SF, `setns` checks ~SF
    - `setg` check greater for signed, `setl` check less for signed, `seta` check above for unsigned
    - setg conditions: $_{\text{(SF\^{}OF)\&}}$ZF, setl conditions: SF^OF, seta conditions: $_{\text{CF\&}}$ZF
  - usually has to be combined with another command to zero out the bytes
    - eg. can use `movzbl` to move from lower byte register to the same larger byte register
    - here, byte to a long, zero specified padding with zero
    - also zeroes the upper 32 bits by convention
- `jX` jumps to different part of code depending on condition codes
  - branches to different paths, sets the `%rip` if some condition is true
    - similar to a if or other control statements
    - have to be used in conjunction with another operation that sets the condition codes
  - `je` checks equal, `js` check negative, `jg` and `jge` check `>=`, `jl` and `jle` check `<=`
    - `ja` check above unsigned, `jb` check below unsigned
    - eg. `jle` is true when `(SF^OF)|ZF`, and `jg` is true when `~(SF^OF)&~ZF`
  - eg. `jle .L4` checks if comparison just done is less than or equal
    - if it is, jumps to label L4 (labels are converted to actual addresses during linking)
  - `jmp` is an unconditional jump
    - eg. `jmp *.L4(,%rdi,8)` dereferences the contents of label + 8*%rdi as an instruction address to pull into the instruction pointer
- other ways to translate/interpret jump operation
  - could express with `goto` in C to simulate the jump operation
  - could also express with a ternary to translate the general condition expression:
    - `val = x>y ? x-y : y-x;`

Listing 6: Assembly Jump

```
// if statement in c:
long absdiff(long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}

// expressed with goto:
long absdiff_j(long x, long y)
{
  long result;
  int ntest = x <= y;
  if (ntest) goto Else;
  result = x-y;
  goto Done;
Else:
  result = y-x;
Done:
  return result;
}

// in assembly:
absdiff:
  cmpq  %rsi, %rdi # x:y
  jle   .L4        # jump if <=
  movq  %rdi, %rax
  subq  %rsi, %rax ret
.L4:    # x <= y
  movq  %rsi, %rax
```

```
  subq  %rdi, %rax
  ret
```

- **conditional moves** avoids the need to branch an instruction, just checks to do if a move is done
  - instead of passing instruction flow through pipeline, conditional moves do not require a control transfer
  - eg. `if (nt) result = eval`; no change in instruction flow, just a conditional move
    * only works if there is a single result to be output
  - eg. `cmovle %rdx, %rax` is a conditional move if result was less than or equal
  - bad cases for conditional move:
    * expensive computations (both values would get computed)
      · eg. `val = Test(x) ? Hard1(x) : Hard2(x);`
    * risky computation, may have undesired effects, eg. checking for nullptr
      · eg. `val = p ? *p : 0;`
    * computation with side effects, both values get computed, may override each other
      · eg. `val = x > 0 ? x*=7 : x+=3;`

Listing 7: Conditional Move in Assembly

```
// general conditional move in C:
val = test ? Then_Expr : Else_Expr;
// goto translation:
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;

// in assembly:
absdiff:
  movq    %rdi, %rax # x
  subq    %rsi, %rax # result = x-y
  movq    %rsi, %rdx
  subq    %rdi, %rdx # eval = y-x
```

```
  cmpq    %rsi, %rdi # x:y
  cmovle  %rdx, %rax # if <=, result = eval
  ret
```

- **loops**:
  - in do-while loop, loop is executed at least once
    - can use a go-to version to simulate the assembly version
    - eg. `loop` label and a `goto loop`;
    - eg. `.L2L`: label and a `jne .L2` to jump/goto to the loop if not zero (after a logical right shift)

Listing 8: Do-While in Assembly

```
// do-while in C, counting number of 1's in bit:
long pcount_do(unsigned long x)
{
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}

// goto version:
long pcount_goto(unsigned long x)
{
  long result = 0;
loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}

// in assembly:
  movl $0, %eax   #result = 0
```

```
.L2:                # loop:
  movq %rdi, %rdx
  andl $1, %edx     # t = x & 0x1
  addq %rdx, %rax # result += t
  shrq %rdi         # x >>= 1
  jne .L2           # if (x) goto loop
  rep; ret
```

- in a while loop, loop may not be executed at all
  - can use a jump to middle translation to simulate the assembly version
  - basically skips over the body once, and goto the test immediately
  - can also translate to a do-while loop
  - has an initial test to check if skipping the body of the do-while loop

Listing 9: While in Assembly

```
// while in C:
long pcount_while(unsigned long x)
{
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}

// jump-to-middle translation:
long pcount_goto_jtm(unsigned long x)
{
  long result = 0;
  goto test;
loop:
  result += x & 0x1;
  x >>= 1;
test:
  if(x) goto loop;
```

```
  return result;
}

// do-while translation:
long pcount_goto_dw(unsigned long x)
{
  long result = 0;
  if (!x) goto done;
loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
done:
  return result;
}
```

- in a for loop, has an initialization, a test, and an update
  - can be translated into a while loop with init outside, test, and update at the end of the body
  - can also translate into a do-while

Listing 10: For Loop in Assembly

```
// for loop in C:
#define WSIZE 8*sizeof(int)
long pcount_for(unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned bit = (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
// while translation:
long pcount_for_while(unsigned long x)
```

```
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE) {
    unsigned bit = (x >> i) & 0x1;
    result += bit;
    i++;
  } return result;
}
// do-while translation:
long pcount_for_goto_dw(unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  if (!(i < WSIZE))
    goto done; // initial test not necessary, can be optimized away
loop:
  {
  unsigned bit = (x >> i) & 0x1;
  result += bit;
  }
  i++;
  if (i < WSIZE)
    goto loop;
done:
  return result;
}
```

- **switch statements**:
  - checks multiple case values with possible fall through cases and maybe a default case
  - more complex than the other loops
    * `switch_eg`:
    * the table has a base address at some label
      - direct jump: `jmp .L8`

        - indirect jump: `jmp *.L4(,%rdi,8)`, have to scale by 8 to skip different locations in the table
        - no conditional jump, only lots of possibilities of different jumps
        - has a jump ahead in order to deal with the default case label
      - code blocks are then tied to different labels
      - in a fall through block, must goto a merge label depending on which operations are being executed in different cases

Listing 11: Switch in Assembly

```
// jumping to parts of the jump table
 switch_eg:
  movq %rdx, %rcx
  cmpq $6, %rdi      # x:6
  ja .L8             # use default
  jmp *.L4(,%rdi,8) # goto *JTab[x]
// parts in the jump table:
section .rodata
  .align 8
.L4:
  .quad .L8 # x = 0 etc...
```

- switches are implemented with a *jump table*
  - would appear as an array of pointer with different possible jump targets
  - targets would point to code blocks actually in memory

Machine Programming: Procedures

- *procedures* are a form of control that change the `%rip`
  - some differences:
    - procedures allow for a *return* to the calling site
    - allow for *passing* data
      - eg. parameters/arguments or a return value
    - there is an *allocation* of state during procedure
  - mechanisms are all implemented with machine instructions

The Stack

- one portion of memory, *grows* and *shrinks*
  - **accounts** for different locations in memory at different times
- stack bottom stays in place (starts at a higher address)
  - and grows *downward* towards lower addresses
  - `%rsp` contains the lowest stack address (top of stack, last value that was stored)
  - can visualize as virtually allocating space (growing or shrinking)
    * but **no** memory is being created/destroyed
    * stack simply **occupies** more or less of the memory space
- some stack operations:
  - `pushq src`
    * fetch operand at `src`
    * decrement `%rsp` by 8 (bytes)
    * write operand to address at `%rsp`
    * implicitly made up of two commands:
      - eg. `sub $0x8, %rsp` and `mov [val], (%rsp)`
  - compiler may also simply do a bulk allocation for the stack, instead of calling push multiple times
    * or modify/access data beyond the stack pointer without modifying it
  - `popq dest`
    * read value at address given by `%rsp`
    * increment `%rsp` by 8 (bytes)
    * store value at `dest` (must be a register)
    * implicitly made up of two commands:
      - eg. `mov (%rsp), [dst]` and `add $0x8, %rsp`

Calling Conventions

Passing Control

- stack supports procedure call and return
  - *procedure call*: `call label` (ie. push and a jump)
    * pushes return address on stack
      - address of next instruction right after call
    * jump to `label` (or move instruction address into `%rip`)
      - `label` will be linked to actual memory later
  - *procedure return*: `ret` (ie. pop and a jump)
    * pop address from stack into the `%rip`
    * thus, jumps to address

Passing Data

- two places to store parameters to procedures:
  - in the registers themselves, or the stack
  - by convention for x86-64, first 6 arguments are stored in registers
    - rdi, rsi, rdx, rcx, r8, r9 (in order)
  - stack parameters are stored in reverse order
    - only when over 6
  - return value stored in `%rax`

Managing Local Data

- has to do with scoping rules / local variables
- for languages that support recursion, code must be *reentrant*
  - multiple simultaneous instantiations of a procedure
  - need to store *state* of each instantiation
    - eg. arguments, local variables, return pointer
- requires **stack discipline**:
  - set of constraints in assembly, order to store state, and how to clean up
  - stack is allocated in *frames* for a single procedure instantiation
- frames hold:
  - the return information (where to go when done)
  - local storage (if required)
  - temporary space (if required)
    - eg. saved registers
  - another optional argument build (calling another function)
  - ie. the current *context* of some procedure call
- `%rbp` *used* to point to the beginning of the frame (called fram pointer)
- "set-up" code is the space allocation upon entry to a procedure
  - includes push by `call`
- "finish" code is the space deallocation
  - includes pop by `ret`
- register saving convetion:
  - needs to be agreement between which registers can be changed/restored
  - *caller saved*: the caller saves temporary values in its frames before the call
    - caller must restore temporary values
      - eg. `%rax`, return value, is caller-saved because callee is free to modify it
      - 6 argument registers (`%rdi` through `%r9`) are also caller-saved, callee also uses these
      - `%r10` and `%r11` also caller saved
  - *callee saved*: callee saves temporary values in its frames before using

        - callee restores them before returning to caller
            - eg. `%rbx`, `%r12` - `%r14`
            - also `%rbp` and `%rsp`
            - `%rsp` special form of callee save: must be restored to original value upon exit (must line up return address)
- observations about recursion:
- each function call has private storage
    - can save registers / return pointer
    - follows stack discipline

Machine Programming: Data

Arrays

- arrays have some data type and a length, `T A[L]`
    - stored in a contiguously allocated region of `length * sizeof(type)` bytes in memory
    - Endian type does not affect the overall array layout, just the storage of each individual type
- can offset to different elements using array subscripts or pointer arithmetic
    - `&val[2]` gives back address of third position
    - `*(val+1)` gives back second element
    - C automatically scales pointers by the size of the array type
- successive arrays *not* neccessarily allocated in successive blocks
    - but each array is stored *contiguously*
- in assembly:
    - could use `movl (%rdi, %rsi, 4), %eax` to generate desired location in memory
- *multidimensional* or nested arrays: `T A[R][C]`
    - overall size would be `R * C * sizeof(T)`
    - in memory, has **row-major ordering**, where array rows are laid out consecutively/contiguously
    - `A[i]` would be a single array/row with C elements (an address)
    - to get to a particular row: `A + (i*C*4)`
    - to get to a particular item: `A + (i*C*4) + (j*4)`
        - `Mem[nestedarr + C*4*index + 4*digit]`
- *multi-level* arrays:
    - every element in the array is a **pointer** to an array of some data type

  - no necessary relationship between each type arrays in memory
  - same formula in C to find an element: `A[R][C]`
  - but in assembly:
    * have to dereference another array with some memory offset
    * two memory reads, first pointer to row array, and then access element within array
    * `Mem[Mem[multiarr + 8*index] + 4*digit]`
- static vs. dynamic dimensionality:
- if the array is *static* (dimensions are known at compile time):
  - compiler can do strength-reductions and optimize operations
- if the dimensions are *variable*:
  - in explicit indexing, uses pointer to the array of a whole
  - in implicit indexing, uses listed dimensions in the parameter list
  - usually have to perform actual, more expensive multiply operation

Structs

- a block of contiguous memory that holds different discrete types / variables
- at least big enough to hold all fields, may be larger for *alignment*
- the fields are ordered according to the declaration
- machine program has no understanding of the structures in the source code
- **alignment**: useful when moving memory around so that data will not be spanned across page or block boundaries
  - restrictions on the *starting* address of different types
  - the primitive types that make up a struct, when aligned, their address must be a multiple of the size of that type (eg. end in 3 0's, 2 0's etc.)
    * the *overall* structure has to be aligned at the *largest* k (size for *integral* type)
    * thus some memory is skipped by the compiler in order to align the elements of the struct
    * since char is 1 byte, has no address restrictions
  - for arrays of structures, has to satisfy alignment for every element
    * have to consider alignment when computing indices
    * to save space, place large data types first

Machine Programming: More Advanced Topics

Memory Layout

- the memory layout includes:
  - *stack* grows downward
    - used for local variables and procedure frames
  - *heap* grows upward
    - dynamically allocated as needed (malloc), used for data
  - *global data* holds statically allocated data, such as globals / constants
  - *text* holds instructions, read-only, and shared libraries

Buffer Overflow

- trying to access memory out of bounds in an array / struct
  - called a **buffer overflow**
    - could overwrite a return address and lead to:
      - seg fault
      - or branch to some unintended instruction
  - usually caused by unchecked lengths on string input
- Unix functions gets, strcpy, scanf all have issues with buffer overflow

```c
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 123456789; // could be out of bounds
  return s.d;
}

fun(0) // 3.14
fun(1) // 3.14
fun(2) // 2.13999999
fun(3) // 2.00000000
fun(4) // 3.14
fun(6) // seg fault, overwriting a critical address
```

- **code injection**:
    - could write executable code in the input string to exploit something
    - overwrite and force a return to the exploit code instead of the callee
    - stack smashing
    - eg. internet worm (1988), IM war
- solutions:
    1. avoid overflow vulnerabilities with safer input functions that limit lengths:
    - fgets instead of gets, strncpy instead of strcpy
    2. system level protection:
    - randomize allocation on stack to shift addresses for entire program
    - mark regions of memory as read-only or non-executable
    3. stack canary:
    - special value called a canary just beyond buffer, check for corruption
- **return-oriented** programming attacks:
    - no injecting, instead make use of existing code
        * string together fragments of library code to achieve desired outcome
    - still doesn't overcome stack canaries
    - construct a program from *gadgets*, code would be executable
        * need a return, so utilize the tail end of existing functions
            · don't have to start at beginning of a function either
    - write each address to stack, and chain together the gadgets with one return

Unions

- similar to structs
- allocated according to the largest integral element (or strictest alignment rule)
- can only use **one** field at a time
    - less memory
- can be set only once
    - thus, reading the values may result in undesired results

Optimization

Limitation

- fundamental constraint that cannot change program behavior
- behavior that may be obvious to the programmer can be obfuscated to the machine

- most analysis is performed only within procedures
  - and based only on *static* information
  - when in doubt, compiler must be **conservative**
- compiler should be designed to optimize, instead of the underlying hardware
  - but dynamic architecture can also be optimized
- *latency* is the amount of time to process an instruction through a pipe
- *throughput* is the amount of time to finish a **parallelized** instruction

Types of Optimization Blockers

- *CPE*: cycles to process an element
- procedures, branches, and memory aliasing are areas that can be optimized
  - dynamic branching, "guessing"
  - taking advantage of parallelism
- common types of optimizations:
  - code motions
  - strength reduction
  - sharing common subexpressions
  - loop unrolling
- common types of optimization blockers:
  - procedure calls
    * compiler treats procedure calls as a black box
  - memory aliasing
    * using temporary accumulator (registers), instead of dereferencing memory
- common types of hardware optimizations:
  - superscaling
- code **motion**:
  - reduce frequency with which a computation is performed
  - essentially moving code out of a loop;
  - compiler *cannot* optimize code motion in a procedure call
    * can move procedure call into block;
    * or **inline** code so that it is inserted
      · but increases instruction size in memory
      · and cannot inline dynamic libraries
- compiler will reuse portions of expensive expressions (memoization)
  - but can also code to reuse expressions
- *instruction* level parallelism:
  - improve latency of overlapping, independent instructions
  - eg. try *unrolling* a loop

        * avoid sequential dependence, use **reassociation**
        * less checks in a loop, can parallelize operations
        * there is a lower bound on unrolling loops
    - compared to *thread* level parallelism
        * numerous functional units
- *reordering* code or code **hoisting**:
    - pros: reduce latency time, better parallelization
    - cons: limitation on registers, ie. register pressure
        * hard for compiler to know how to optimize branching
        * can't tell which branch will be taken
- multiple points in memory get **aliased**:
    - hard to tell if there is a dependency in memory, one value reads another
    - thus, compiler can't reorder those load and store commands to optimize
        * (load and store are high latency instructions)
    - can *accumulate* in a temp variable
        * avoid pointer arithmetic

Memory Hierarchy

Technologies and Trends

- **RAM**: random-access-memory
    - packaged as a chip
    - basic storage unit is a cell (one bit per cell)
    - can be *static* (faster, more expensive, more power) or *dynamic* (better capacity)
        * these are *volatile* memories that lose information without power
- *nonvolatile* memories don't depend on power:
    - read-only memory (ROM)
    - programmable memory (PROM)
    - large scale drives: disks and flash drives
- *bus* interface includes wires that carry data and signals
    - system bus to I/O bridge
    - memory bus to main memory
    - eg. interface process to read from memory:
        * CPU places address A on memory bus
        * main memory reads A from memory bus
        * retrieves data word x, and places it on the bus

- I/O bridge can also connect to different components
  - eg. USB devices, a disk controller
  - disk controller links between disk and memory
    - provides an *interrupt* or message upon completion
- SSD has no moving parts compared to conventional rotational disks
  - significantly faster
  - not as high capacity
  - has potential to wear out that must be spread out evenly
- important notes for memory hierarchy:
  - sequential access is **faster** than random access
  - the gap continues to **widen** between DRAM, disk, and CPU speeds
    - latency has not scaled well
  - avoid going to slower storages often in programs
    - well-written programs tend to exhibit good locality

Locality of Reference

- **locality**: programs tends to use data and instructions with addresses near or equal to those used recently
  - recently referenced items likely to be referenced again (temporal)
  - items with nearby addresses tend to be referenced close together (spatial)
- for example, in a loop iteration:
  - references array elements in succession (nearby, spatial)
  - references a sum variable each iteration (frequent, temporal)
  - instructions are referenced in sequence (spatial), and cycled (temporal)
- can exploit by iterating loops in row-first order, instead of column-first order
  - larger *strides* if going by column
- these properties lead to memory being organized in a *hierarchy* depending on their locality:
  - L0 - registers, least space, fastest, most costly
  - L1-3 - SRAM, L1-3 caches
  - L4 - DRAM, main memory, more latency, more storage, cheaper
  - L5 - local secondary storage
  - L6 - remote secondary storage
- **cache** is a smaller faster storage device, acts as a staging area for a subset of data
  - physical, separate structures, usually handled by hardware, not software
  - organized and partitioned by *blocks*
    - blocks are kept consistent to avoid overlap
  - each block has an identification tag
    - tag tells system what addresses are currently cached

- can have hits or misses, depending on what blocks are required from memory
  - hits are where desired blocks are currently loaded into the cache
  - memory is loaded on a miss, on a block size granularity
    - ie. go to the next level of the memory hierarchy
- different types of caches, including registers (but managed by software)
  - TLB cache on-chip for virtual memory
  - L1 and L2 caches are on-chip
  - virtual memory, buffer cache in main memory

Cache Memories

- small, fast DRAM-based memory handled in hardware
- holds frequently accessed blocks of main memory
- CPU looks first in data in cache (pulls in more memory in case of miss)
  - **block size** is the grainularity at which data is moved on a miss
- cache organization:
  - similar to a hash table, with buckets
  - cache size: Sets * Elements * Bytes
- types of caches: *associative* (block can go anywhere, more locations to check), *direct map* (block goes in one location)
  - hybrid / set associative (mix of both)
  - increasing flexibility with more locations, but more time consuming
  - also changes the eviction policy
- cache organization is made out of **sets** with **lines**
  - lines composed of: valid bit, tag, and the actual cache block
    - valid bit unset when cache memory is invalidated by writing into memory
  - two-way associative: 2 lines per set, four-way associative, etc.
- *addressing* of a word in a cache block
  - address composed of: tag, set index, block offset
  - not actual address, but physical address translation of the address
- each core has a private L1 (i and d) and L2 cache
  - cores share an L3 unified cache
  - but block size is homogeneous
- some performance metrics: **miss rate**, **hit time**, **miss penalty**
  - huge difference in cycles between a hit and a miss
- some cores have unified caches
  - more resources to share, but easier to communicate
- caches have an *eviction policy* to determine how to clear / replace data in the

cache
  - eg. **LRU** (least recently used) or **LFU** (least frequently used)

Writing Cache Friendly Code

- focus on the innermost loops on core function
- to minimize misses, look for *locality*
  - repeated variable references
  - stride-1 reference patterns
- *memory mountain* measures read throughput as a function of spatial and temporal locality
  - number of elements influences temporal locality
  - stride influences spatial locality
  - has certain **ridges** in the graph
    * represent L1, L2, L3, and Memory caches (temporal locality)
    * past a certain memory size for each cache, have to go down the memory hierarchy
      · leads to a throughput drop
  - **slopes** in graph represent spatial locality
    * stride length
  - **aggressive prefetching** allows for even higher throughput
- eg. when optimizing matrix multiplication:
  - lowest miss rate occurs by reading sequentially with stride-1
  - that is, read both matrices row-wise in the innermost loop
- can also break up loops or matrices even further into *tiles* or *blocks*
- want largest possible tile size

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
} // a row-wise, b column-wise, c fixed
  // 0.25 + 1 misses per iteration

for (k=0; k<n; k++) {
```

```
  for (i=0; i<n; i++) {
      r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
} // a fixed, b row-wise, c row-wise
  // 0.25 + 0.25 misses per iteration

for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
      r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
} // a column-wise, b fixed, c column-wise
  // 1 + 1 misses per iteration

// Tiling Example:
for (i=0; i<n; i+=B)
  for (j=0; j<n; j+=B)
    for (k=0; k<n; k+=B)
      for (i1=i, i1<i+b; i1++)
        for (j1=j, j1<j+b; j1++)
          for (k1=k, k1<k+b; k1++)
            c[i1*n+j1] = a[i1*n+k1] * b[k1*n+j1];
```

Parallel Computing

- an attempt to speed a task by dividing into subtasks
  - execute them *simultaneously* on multiple processors
- *domain decomposition* approach:
  - divide **data** elements among processors
  - decide which tasks each processor should be doing
  - eg. for finding max of an array, break array into pieces
    - distribute segments among different CPUs

- *task/functional decomposition* approach:
  - divide **tasks** among processors
    - leads to different communication chains
  - decide which data elements are going to be accessed
- *pipelining* approach:
  - assembly line parallelism
  - overlapping stages in the assembly line
- *dependence graph*:
  - nodes for constants, operators or function calls
  - arros for use of variables or constants
    - ie. data and conditional flow
  - if there is no **cross-iteration** dependence (no overlapping)
    - can use domain decomposition (discrete, independent regions)
  - often will have some dependence that requires a rejoining across processors
- for parallel portions of program, master thread *forks* additional threads
- at *join*, extra threads are suspended or die
- forking allows for **incremental** parallelization
  - transform sequential code when possible into parallel code
- threads often share memory in order to communicate with each other
  - also would have private variables
  - must somehow *synchronize* shared variables
  - can lead to *race condition* where threads can write to shared memory

OpenMP

- OpenMP is an API for parallel programming
  - provides library functions, environment variable, compiler directives
    - compiler directive in C is called a **pragma**
    - appear before relevant construct
    - has form `#pragma omp <...>`
  - used with C, C++, Fortran
  - based on fork/join model
  - mainly suited for domain decomposition
- types of breakdown/*partitioning*:
  - static, dynamic, guided
  - with dynamic, using a finer grain breakdown
    - more work can be done by another thread if one thread is taking a while
    - allows for more load balancing

    - with guided, starts with large partitions that get smaller
- `#pragma omp parallel for`
    - a *forking* and *worksharing* construct
        * eg. in a loop running 100 times, **static** breakdown/decomposition of 50 iterations into two processors
    - tells the compiler loop immediately following can be executed in parallel
    - number of loop iterations must be computable at *runtime*
    - thus, no break, return, exit, goto
    - has an overhead:
        * want to maximize the amount of work done for each fork/join
        * ie. the *grain size*
    - automatically makes the loop index a *private variable*
        * use `private` clause
        * private variables are **undefined** at loop entry and exit
            · can not assign new value to "other" private variable (in shared memory)
    - use `firstprivate` clause
        * private variable should inherit value of shared one on entry
        * once per thread, not per iteration
    - use `lastprivate` clause
        * value of private variable after sequentially lsat loop should be assigned to shared one on exit
        * remember, different threads can complete at different times
- `parallel` pragma used when a block of code should be execued in parallel (*forking* construct)
    - forking and joining inherently has a nowait clause
- `for` pragma used inside parallel block of code (*worksharing* construct)
- `single` pragma used inside parallel block of code (*worksharing* construct)
    - only a single thread should execute the statement or block following
- `nowait` clause says there is no need for synchronization at end of `for` or `single`
    - ie. should we wait for all the threads to finish before continuing?
- `omp_get_num_procs()` returns number of physical processors/cores
- `omp_set_num_threads(int)` sets number of threads should be active when in parallel

```
for (k = 0; k < N; k++)     /* has loop-carried dependencies */


  /* don't want too fine-grained, so parallelize middle loop */
  #pragma omp parallel for private(j) /* private clause */
  /* i is automatically a private variable */
```

```
  for (i = 0; i < N; i++)   /* can be parallelized */

    /* j is a SHARED variable, will NOT work unless made private */
    for (j = 0; j < N; j++) /* can be parallelized */

#pragma omp parallel for private(tmp)
for (i = 0; i < N; i++)
{
  tmp = a[i] / b[i];
  c[i] = tmp * tmp;
}
```

Examples with private clauses:

```
/* firstprivate example: */
for (i = 1; i < N; i++)
  a[i] = alpha(i, a[i-1]);
#pragma omp parallel for firstprivate(a)
/* copies values of the existing a into all private copies of a */
for (i = 0; i < N; i++)
{
  b[i] = beta(a[i]);
  b[i] = gamma(a[i]);
  b[i] = delta(a[i]);
}
/* lastprivate example: */
#pragma omp parallel for lastprivate(x)
for (i = 0; i < N; i++)
{
  x = foo(i);
  y[i] = bar(i, x);
}
last_x = x;
```

Example with for and single pragmas:

```
#pragma omp parallel        /* forking */
{
  #pragma omp for nowait    /* worksharing */
  for (i = 0; i < N; i++)
    a[i] = alpha(i);
  #pragma omp single nowait /* only one of the threads will print */
  if (delta < 0.0) printf (...);
  #pragma omp for           /* worksharing */
  for (i = 0; i < N; i++)
    b[i] = beta(...);
  /* inherent synchronization barrier here */
}
```

Extended example:

```
for (i = 0; i < m; i++)
{
  low = a[i];
  high = b[i];
  if (low > high)
  {
    printf(...);
    break; /* cannot workshare this loop! */
  }
  #pragma omp parallel for
  for (j = low; j < high; j++)
    c[j] += alpha(...);
}
```

Even better:

```
#pragma omp parallel private(i, j, low, high)
/* private(j) is redundant */
for (i = 0; i < m; i++)
{
  low = a[i];
```

```
  high = b[i];
  if (low > high)
  {
    #pragma omp single nowait
    printf(...);
    break;
  }
  #pragma omp for nowait
  /* removes the overhead of forking on every iteration! */
  for (j = low; j < high; j++)
    c[j] += alpha(...);
}
```

Race Conditions

- prevalant pitfall with parallel computing
- occur when there are shared memory accesses / read and writes
  - may lead to errors when one thread is reading memory before the other has written to it
  - exhibit **nondeterministic** behavior, influenced by timing, unpredictable
- threads are *racing* each other
- need to have *mutual exclusion*, a type of synchronization where only a single thread or process can access a shared resource
- could we manually solve by having a flag that is set when only one thread should be running?
  - no, what if two threads still read the flag at the same time and both execute
  - need an *atomic* test-and-set operation that is indivisible
- `#pragma omp critical` tells compiler next block of code can only be executed by **one** thread at a time
  - similar to `single`, but all threads end up executing the code
  - drawback is the code block is executed *sequentially*
  - can hoist code to help deter this

```
for (i = 0; i < n; i++)
{
x = (i + 0.5) / n;
area += 4.0 / (1.0 + x * x);
```

```
/* accumulator variables can't simply be private,
   want to eventually combine them.
   We need a check for this. */
}
pi = area / n;
```

Fix with critical pragma:

```
#pragma omp parallel for private(x)
for (i = 0; i < n; i++)
{
x = (i + 0.5) / n;
#pragma omp critical
area += 4.0 / (1.0 + x * x);
}
pi = area / n;
```

Reduce time in critical section with hoist:

```
#pragma omp parallel for private(x, tmp)
for (i = 0; i < n; i++)
{
x = (i + 0.5) / n;
tmp = 4.0 / (1.0 + x * x);
#pragma omp critical
area += tmp;
}
pi = area / n;
```

Even less time in critical section:

```
#pragma omp parallel private(tmp)
{
tmp = 0.0;
#pragma omp for private(x)
for (i = 0; i < n; i++)
```

```
{
  x = (i + 0.5) / n;
  tmp = 4.0 / (1.0 + x * x);
}
#pragma omp critical
  /* only occurs once */
  area += tmp;
}
pi = area / n;
```

- common pattern of parallelism called a *reduction*
  - using some associate binary operator to accumulate
- `reduction` clause
  - eliminates overhead for private variable and dividing computations

```
#pragma omp parallel for private(x) reduction(+:area)
for (i = 0; i < n; i++)
{
x = (i + 0.5) / n;
area += 4.0 / (1.0 + x * x);
}
pi = area / n;
```

- specify operator in clause
- automatically optimizes and creates an accumulator

Deadlock

- in general, even more efficient to lock *data*, instead of the *code*
- for example, in a hash table, race condition invalid if writing into different buckets
  - but with `critical` pragma, every write would be done synchronously
  - need a *finer-grain* solution
- can use a lock *table* for each element in the table
  - `omp_set_lock`() is an **atomic** operation to check AND set
    - at the grainularity of the indices of the table
  - `omp_unset_lock`() unsets
- however, can lead to an issue with **deadlock**
  - both threads waiting for each other to unlock locked data, forever

    - cyclic resource allocation lock
    - also nondeterministic
- deadlock only occurs under four conditions:
    - mutually exclusive access to shared resource, lock
    - threads hold onto resources they have while waiting
    - resources cannot taken away from threads
    - cycle in resource allocation graph
- can solve by ranking resources
    - always must acquire one lock before the other
        * only necessary when threads are locking multiple resources
    - every lock needs an unlock

Exceptions

- want to handle *system state* instead of simply the *program state*
    - eg. data from disk, divide by zero, ctrl-c
- mechanisms for **exceptional control flow**
    - low level: exceptions
    - higher level: process switching, signals between processes, nonlocal jumps
- *exception*: transfer of control to the OS *kernel* in response to some event
    - kernel has higher-level privileges
    - after processing with exception handler, can:
        * return to current instruction to execute
        * return to next instruction
        * abort
    - exception table, similar to a jump table
        * unique indices for different events and their handlers
- different types of exceptions, async and sync
- *asynchronous* exceptions include system interrupts
    - events external to the processor
    - indicated by processor's *interrupt* pin
    - returns to next instruction
    - eg. timer interrupt, I/O interrupt
- *synchronous* exceptions include:
    - result of *executing* an instruction
    - **traps**: intentional, recoverable
        * calling OS for assistance, higher permissions
        * returns to next instruction
        * eg. a sys call, breakpoints

  - **faults**: unintentional, possibly recoverable
    - either re-executes or aborts
    - eg. page faults, segmentation faults, floating point exceptions
      - could re-execute move command after loading from disk
  - **aborts**: completely unexpected, not recoverable
    - abort program
    - eg. parity error, machine check, illegal instruction
- *system calls*:
  - user needs higher permissions to perform file operations, etc.
  - eg. read, write, open, close, stat, fork, etc.

Linking

- compilation steps: compilation (cpp), translation (cc1), assembler(as), linker(ld)
- once program compiled into .o file, becomes a relocatable object file
  - memory addresses not solidified yet, contains memory labels
  - each .o file produces from a single sources file
- **linking** combines together these relocatable object files into executable
  - allows for *modularity* of files
  - allows *libraries* to be easily used in programs
    - optimized, specialized, common code/tools
  - allows for separate compilation and relinking
  - efficiency in time and space
  - ELF format (ELF binaries) is a standard format for object files
    - unified format for .o, a.out, .so
- **symbols** are keywords that can be defined and referenced to
  - functions, variables, etc.
  - eg. `void swap() {...}` defines a symbol
    - `swap();` is a reference to a symbol
  - can be global, external (defined in another module), or local
- linkers use a **symbol table** to keep track of symbols and their definitions
  - array of structs with name, size, and location of a symbol
  - associates reference with definition during symbol resolution step
- relocatable object files have a text segment (instructions) and a data segment (globals)
- linker merges separate code sections
  - then, relocates relative symbol locations into final absolute locations
  - finally, updates all symbol references
- *static* linking links together code at **compile-time**

    - larger executable
        * less overhead, portable
        * can embed a specific library version
    - archive/static *libraries*: .a archive file
        * concatenation of related object files
        * linker can distinguish between different object files within the archive
            · searches the library for unresolved external references
            · linker scans files in the *command line order* (libraries at the end)
            · only links specific necessary object files
- *dynamic* linking links together code at **load-time** or **runtime**
    - avoids duplication of code, and easy updating of libraries without relinking
    - when run, there is an expected path to find libraries
    - executables are only *partially-linked*, does not contain actual code for the libraries
    - can link at *load-time*:
        * handled automatically by linker
        * C library is usually dynamically linked
    - can link at *runtime*:
        * dynamically loading
        * allows for inter-positioning
    - some extra latency
        * only pull in what is needed from libraries
            · pulled into memory, not disk
    - shared libraries: .so files on Linux, .dll files on Windows
- **library interpositioning**: allows programmers to intercept calls to functions
    - could occur at compile, link, load, and run-time
    - wrap shared libraries in extra auxiliary code
    - provide an extra layer of indirection, programmer freedom
- *applications*:
    - confinement, security measures
    - behind-the-scenes encryption
    - debugging
    - monitoring and profiling
        * malloc tracing
        * detecting memory leaks or generating address traces

Virtual Memory

- when CPU's are shared between processes (eg. parallelism):

  - use a memory management unit to map virtual address to a physical one
  - don't have to compile programs with every possible memory location
  - allows programs to have the *illusion* of having *full* virtual address space
- simplifies memory management, each process gets the same uniform linear address space
- isolates address spaces
  - programs can't access each others memory
  - protection and sharing of data
- use DRAM as cache for parts of virtual address space
- **virtual memory** is an array of contiguous bytes on disk
  - contents of the array on disk are cached in **physical memory** (DRAM cache)
  - cache blocks are called **pages**
    * since disk is slow, page blocks are large
    * large granularity
- *page table* is an array of page table entries that maps virtual pages to physical pages
  - memory resident structure
  - different page tables for different processes
  - valid bit and physical page number or address
    * PTE's can also be extended with permission bits
  - page *hit*: reference that is in physical memory, cache hit
  - page *fault*: reference that is not in physical memory, cache miss
    * page fault handler must evict a block from DRAM cache and bring in new block from disk
    * instruction is then rerun
- works because of *locality*:
  - if working set size is less than main memory size, good performance
  - otherwise, leads to thrashing: pages are swapped in and out continuously
- virtual memory allows each process has its own virtual address space:
  - memory is viewed as a simple linear array
  - compile programs can be run in any machine and memory locations
  - then, can map those virtual addresses into the same, shared pool of physical memory
    * allows for sharing of data between processes
- simplifies linking and loading:
  - each program has similar virtual address space
  - code, data, and heap always start at the same addresses

Address Translation

- virtual addresses are either invalid and stored on disk, or stored in memory
- virtual address (handled by the MMU):
  - page number: index into the page table
  - page offset: offset from start of the page
- *page table base register* points to the beginning of the page table
  - page table stored in memory
  - made up of PTE's
    - valid bit
    - physical page number
- if valid bit in PTE is 1,
  - can take physical page number and append to the original page offset
  - forms overall physical address
  - can go to physical address in memory immediately
- otherwise, *page fault*
  - page fault handled by kernel
  - the instruction is restarted after page on *disk* is brought into *physical memory*
- CPU communicates with MMU
  - two requests from memory for one memory request: PTE and actual data
- **Translation Lookaside Buffer** (TLB):
  - small hardware cache in MMU
    - usually more associative
  - caches PTE's
  - TLB itself has a valid bit and tag for associating PTE's
  - VPN of the PTE is split up into a TLB tag and TLB index for looking up line in set
  - TLB hit: still need to go to cache/memory hierarchy with physical address after translation
  - TLB miss: incurs an additional memory access to find the PTE in actual page table
  - can be multiple TLB's for different cores, similar to L1/L2 caches
- page tables can be very large
  - solution is to have extra level of indirection
  - level 1 table is memory resident
  - each PTE in level 1 table is indexed and points to another page table in disk
    - other higher level pages can be paged in and out of physical memory

RISC vs. CISC

- *reduced* instruction vs. *complex* instruction sets
  - fewer registers and only one register classes
  - can only operate on registers, not memories + registers
  - utilizes 3-address instructions, instead of 2
  - only one addressing mode (base-offset)
  - fixed instruction length (32 bits)
- CISC determined at a time where memory was expensive
  - RISC promotes more granular, can more easily be parallelize or pipelined
- **MIPS** is a RISC
  - all arithmetic operations have the form $Rd < -RsopRt$
  - MIPS is a load-store architecture, ALU only operates on registers
  - basic operations:
    * arithmetic
    * logical
    * comparison
    * control
    * memory access (load and store)
- MIPS registers:
  - 32, 32-bit registers name $0 - $31 for general use
  - 32-bit program counter (PC), equivalent to $rip
  - special registers for multiply, division, and floating point
- register *conventions*:
  - zero register always 0
  - $v0-v1 used for function return / sys. calls
  - $a0-a3 used for function parameters
  - $t0-t7 and $t8-t9 not saved on call (callee saved)
  - $s0-s7 are saved on call (caller saved)
  - $gp global pointer
  - $sp stack pointer
  - $fp frame pointer
  - $ra return address
- register *notation*:
  - rd: destination
  - rs: source
  - rt: source/destination (read+modified)
  - immed: 16-bit immediate
- **load/store**:

  - `LW rt, offset(base)` loads word from memory into register
    * base register + literal offset
  - `SW rt, offset(base)` stores word into memory from register
  - LB load byte and sign-extend
  - LBU load byte and zero-extend
  - SB store byte
- **arithmetic instructions**:
  - `ADD rd, rs, rt` : rd = rs + rt
  - `ADDI rt, rs, immed` : rd = rs + immed
  - `SUB rd, rs, rt` : rd = rs - rt
- **control flow**:
  - `BEQ rs, rt, target` branches if registers are equal
    * target is a PC-relative address (4, next instruction + offset)
  - `BNE rs, rt, target` branches if registers are not equal
  - comparison between registers:
    * `SLT rd, rs, rt` sets rd to 1 if rs < rt, otherwise 0
    * `SLTU rd, rs, rt` similar, but unsigned
    * allows for branch if var1 < var2
- **jumping**: (uses 26-bit immediates that are appended onto PC to calculate PC-relative address)
  - `J target` jumps to target
  - `JR rs` jumps to address in register
  - `JAL target` jumps to target, ra = PC + 4
  - `JALR rs, rd` jumps to rs, rd = PC + 4
- **logic instructions**:
  - `AND rd, rs, rt` : rd = AND(rs, rt)
  - `ANDI rd, rs, immed` : rt = AND(rs, immed)
  - OR, ORI, XOR, XORI
  - `LUI rt, immed` loads upper immediate into upper 16 bits of register
- **pseudo-instructions**:
  - not real machine instruction
  - assembly instructions that are broken down by compiler
  - `MOVE $t, $s` -> `ADDIU $t, $s, 0` (t = s)
  - `CLEAR $t` -> `ADDU $t, $zero, $zero` (t = 0)
  - `LI $t, immed` -> `ADDIU $t, $zero, immed_lo` (t = immed)
    * load 16-bit immediate
  - `LI $t, immed`
    * load 32-bit immediate (or a label address with `LA`)
    * load upper immediate: `LUI $t, immed_hi`
    * or lower immediate: `ORI $t, $t, immed_lo`

- **system calls**:
  - each have their own code to distinguish
  - print integers (1) or strings (4)
  - read integers (5) or strings (8)
  - exit (10)
- MIPS code sample:

```
    .data
A:  .word 5
B:  .word 10

    .text
    .globl foo
foo:
    lw $t0, 0($a0)
    lw $t1, 0($a1)
    sw $t0, 0($a1)
    sw $t1, 0($a0)
    jr $ra

    .globl main
main:
    addu $s7, $0, $ra
    la $a0, A
    la $a1, B
    jal foo

    li $v0, 4
    la $a0, A
    syscall
    li $v0, 4
    la $a0, B
    syscall

    addu $ra, $0, $s7
    jr $ra
```

```
    add $0, $0, $0
```