CS33 Lecture 1: Bits and Bytes

- Pre-processor → compiler → assembler → linker
  - ↳ text to binary thru assembler
  - ↳ assembler is machine specific
    - ↳ on a binary level, program is run for a particular machine architecture
- Hex - denoted by 0x preceding hex number
- Send help
- Left shift → shift bit vector left, fill w/ zeroes
- Right Shift
  - ↳ Arithmetic → shift bit vector right, replicate most significant digit
  - ↳ Logical → shift bit vector right, fill w/ zeroes

CS33 Lecture 2: Integers

- Encoding integers
  - ↳ Unsigned: $B2U(x) = \sum_{i=0}^{w-1} x_i \cdot 2^i$
  - ↳ Two's Complement: $B2T(x) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-1} x_i \cdot 2^i$
    - ↳ most significant bit indicates sign

↳ 0000	0001	0010	0011	. .	0111	1000	. . .	1111
U: 0	1	2	3	. . .	7	8	. . .	15
T: 0	1	2	3	. . .	7	-8	. . .	-1

  - ↳ Maxes/Mins
    - ↳ $|TMin| = TMax + 1$ → asymmetric range
    - ↳ $UMax = 2 * TMax + 1$
    - ↳ #include <limits.h> for C Programming
      - ↳ constants: ULONG_MAX, LONG_MAX, LONG_MIN
  - ↳ Word Size - size used for pointers (ie 64-bit machine encodes pointers as 64-bit)
  - ↳ Casting
    - ↳ signed values implicitly cast to unsigned
      - ↳ -1 > 0U → -1 unsigned is UMax
      - ↳ 2147483647U < -2147483647-1 → 011...11 < 11...110
      - ↳ (unsigned)-1 > -2
      - ↳ 2147483647 < 2147483648U → $TMax < TMax + 1$
      - ↳ 2147483647 > (int)2147483648U

CS33 Lecture 3: Machine-Level Programming: Basics

- CPU puts addresses into memory
  - ↳ Memory returns data/instructions
- Register is part of CPU
  - ↳ faster than memory (SRAM vs. DRAM)
  - ↳ compiler-visible like memory
- PC - program counter - tells us what instruction is being executed
- Registers and memory together hold the state of the program
- Memory is byte-addressable
- x86-64 has 16 Integer Registers
  - ↳ %rax has 64 bits
    - ↳ contains %eax → lower 32-bits, can reference individually
    - ↳ can reference lower, but not upper
  - ↳ %rsp points to the current location in memory where the stack is
  - ↳ Memory is contiguous in each register file
- Moving Data
  - ↳ movq → copies from source → destination
    - ↳ operand can be 1 of 3 types:
      - ↳ immediate - constant integer data
        - ↳ Ex) $0x400, $-533
      - ↳ register - one of 16 integer registers
        - ↳ Ex) %rax, %r13 → location, not value
      - ↳ memory - 8 bytes of memory at address given by register
        - ↳ Ex) (%rax) → pointer for movq instruction (dereference)
    - ↳ imm → reg: movq $0x4, %rax → temp = 0x4
    - ↳ imm → mem: movq $-147, (%rcx) → *p = -147
    - ↳ reg → reg: movq %rax, %rdx → temp2 = temp1
    - ↳ reg → mem: movq %rax, (%rdx) → *p = temp
    - ↳ mem → reg: movq (%rax), %rdx → temp = *p
      - ↳ mem denoted with parens
- Complete Memory Addressing Modes
  - ↳ D(Rb, Ri, S) → Mem[Reg(Rb) + S*Reg(Ri) + D]
    - ↳ Ex) 8(%rax, %rdx, 2) → accesses rax + 2*rdx + 8 location

- Address Computation Instruction
  - ↳ leaq → doesn't dereference, doesn't go into memory
    - ↳ setting up an address (finding a pointer)
    - ↳ leaq (rax), rdx vs. movq (rax), rdx
      - ↳ leaq (rax), rdx → movq rax, rdx
    - ↳ leaq (rax, rbx), rdx → adds rax and rbx and writes it to rdx
  - ↳ useful for 2 address code (an address is an input and output)

CS33 Lecture 4: Machine-Level Programming - Control

- Condition codes: 1-bit registers
  - ↳ CF - carry flag (unsigned)
  - ↳ ZF - zero flag
  - ↳ SF - sign flag (signed)
  - ↳ OF - overflow flag (signed)
    - ↳ Ex) using addq : t = a+b
      - ↳ CF set if MSB carried out
      - ↳ ZF set if t == 0
      - ↳ SF set if t < 0 (set if signed or unsigned)
      - ↳ OF set if 2s complement overflow " → set w/ CF
    - ↳ not set by leaq *
- cmpq instruction (Src2, Src1) → (Src1 - Src2)
  - ↳ performs subtraction without setting destination
    - ↳ saves register space when performing comparisons
    - ↳ CF - carry out, ZF: a==b, SF: (a-b) < 0, OF: 2s complement overflow
- Jumping - jump to diff. parts of code depending on condition codes
  - ↳ can be set explicitly (cmpq) or implicitly (arithmetic)
  - ↳ modifies RIP → points to current instruction (instruction pointer)
  - ↳ jmp → unconditional jump, UBR
  - ↳ rest of instructions are CBR → conditional
    - ↳ Ex) je/jne → jumps if equal/zero or not equal/nonzero
    - ↳ Ex) ja/jb → unsigned
      - ↳ check if < 0 or > 10 → can use ja → all neg. values interpreted as > 10
  - ↳ linker replaces labels with memory addresses for jumping
- Conditional move: using more registers, allows us to avoid control flow issues
  - ↳ Ex) if (x < y) → return x-y, else return y-x
    - ↳ writes both results into registers before evaluating condition
    - ↳ decides which to return based on condition
  - ↳ compiler stores alternatives that will be performed to be returned/selected later
  - ↳ cmov → conditional move instruction
  - ↳ Bad when expensive (hard evaluations), risky (dereferencing), side effects present (value modification)

- All loops can generally be modified to the same structure
  - Do-while
    - .L2:

```
movq    %rdi, %rdx  -> rdi is x
andl    $1, %edx    -> edx is either 0 or 1
addq    %rdx, %rax
shrq    %rdi        -> sets ZF when x=0, x>>=1
jne     .L2
rep; ret
```

  - Switch
    - case 100:

```
i++; j++;
break;
case 102:
i+=2; j+=2;
break;
case 103: <- fall through
i++;
case 104:
j+=2;
break;
case 106;
i+=2;
j++;
break;
default:
i=0; j=0;
```

    - jump table (values close together)
      - one jump instruction sends the RIP to find the correct memory address
      - cases merge back to the break
- Direct jump -> jbe -> to a location
- Indirect jump -> jmpq *(ADDR) -> used in the jump table

CS33 Lecture 5: Machine-Level Programming: Procedures

- The stack grows down in addresses (FFF -> 000)
  - ↳ rsp points to the top of the stack (lowest address)
  - ↳ all programs have their own stack/heap -> virtual memory
- Ex)

```
400540 <multstore>:
    :
400544: callq 400550 <mult2>
400549: mov %rax,(%rbx)
    :
400550 <mult2>:
400550: mov %rdi,%rax
    :
400557: retq
```

  - ↳ rip increments with the instruction addresses
    - ↳ naturally goes to the next instruction
      - ↳ changes w/ jumps, callq, etc
  - ↳ callq leaves the original caller's memory address on the stack and changes rip to the new instruction address
  - ↳ retq -> pop + jump -> consumes data in the stack, changes rip back to caller's address
    - ↳ uses the rsp to locate the caller's address
- Procedure Data Flow
  - ↳ 1st 6 arguments stored in registers
    - ↳ rdi, rsi, rdx, rcx, r8, r9
  - ↳ 7+ arguments placed on stack
- x86-64/Linux Stack Frame
  - ↳ Frame - data of some function call
    - ↳ Arguments (if 7+), return address -> callee frame
      - ↳ calleeframe: Registers + vars, argument build (if caller of another function)
- Register Saving Conventions
  - ↳ Caller saved -> temp. values saved in frame before call
  - ↳ Callee saved -> temp. values saved in frame after call, before using
    - ↳ restored by callee before returning to caller

- x86-64 Linux Register Usage
  - %rax → return value
    - caller saved, can be modified by procedure
  - rdi, rsi, rdx, rcx, r8, r9 → arguments
    - caller saved, can be modified by procedure
  - r10, r11
    - caller saved
  - rbx, r12, r13, r14 → callee saved
  - rbp → callee saved
    - may be used as a frame-pointer
  - rsp → special form of callee save, stack pointer

CS33 Lecture 6: Machine-Level Programming: Data

- Multidimensional Arrays
  - ↳ Declaration: T A[R][C]
    - ↳ 2D array of type T
    - ↳ R rows, C columns
      - ↳ Type T element requires K bytes
  - ↳ Array Size: R * C * K bytes
  - ↳ Arrangement: Row-major ordering
- Multi-Level Array
  - ↳ Array with a first array that contains pointers to secondary arrays
  - ↳ More flexible, more complex
  - ↳ Can have varying lengths at each unit
- Machine-Level Translation
  - ↳ Fixed-Size → C=16, K=4 → A + i*(16*K) + j*K
    - ↳ a in rdi, i in rsi, j in rdx ↳ i*(16*k) positions which nested array
    - ↳ salq $6, %rsi # 64*i ↳ j*K positions inside the nested array

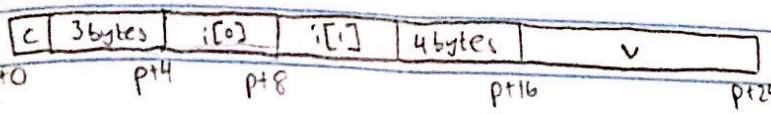
```
addq %rsi, %rdi  # a + 64*i
movl (%rdi,%rdx,4), %eax  # M[a+64*i+4*j]
ret
```

  - ↳ Dynamic → C=n, K=4 → A + i*(C*K) + j*K
    - ↳ must use multiplication rather than shifts
    - ↳ n in rdi, a in rsi, i in rdx, j in rcx
    - ↳ imulq %rdx, %rdi # n*i

```
leaq (%rsi,%rdi,4), %rax  # a + 4*n*i
movl (rax,%rcx,4), %rax  # a + 4*n*i + 4*j
ret
```

- Structs
  - ↳ must satisfy alignment requirement K = largest alignment of any element
    - ↳ prevents overlapping of blocks/pages/etc
  - ↳ struct S1 {

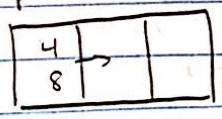
```
char c;
int i[2];
double v;
}
```

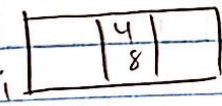


CS33 Lecture 8: Floating-Points

- Tiny FP example (8 bits)
  - ↳ the sign is in the MSB
  - ↳ next 4 bits are the exponent -> bias of $7 (2^{4-1}-1)$
  - ↳ the last 3 bits are the frac
  - ↳ normalized: $(-1)^S 1.\text{frac} \times 2^E$, $E = \text{exp} - \text{bias}$
    - ↳ exp != 000...0, exp != 111...1
  - ↳ denormalized -> exp = 000...0
    - ↳ E = 1-Bias, M = 0.frac
  - ↳ infinity: exp is 111...1, frac is zero
  - ↳ NaN: exp is 111...1, frac is non-zero
    - ↳ instead of overflow, floating-point arithmetic saturates
- x =/= (int)(float) x -> float doesn't have enough precision
- x == (int)(double) x ✓
- f == (float)(double) f ✓
- d =/= (double)(float) d
- f == -(-f) ✓ -> float doesn't have the asymmetry of int
- 2/3 =/= 2/3.0
- d < 0.0 -> ((d*2) < 0.0) ✓ -> no overflow concerns
- d > f -> -f > -d ✓
- d * d >= 0.0 ✓ -> no overflow concerns
- (d+f)-d =/= f -> potential for not enough precision, f rounded away

CS33 Lecture 9: Optimization

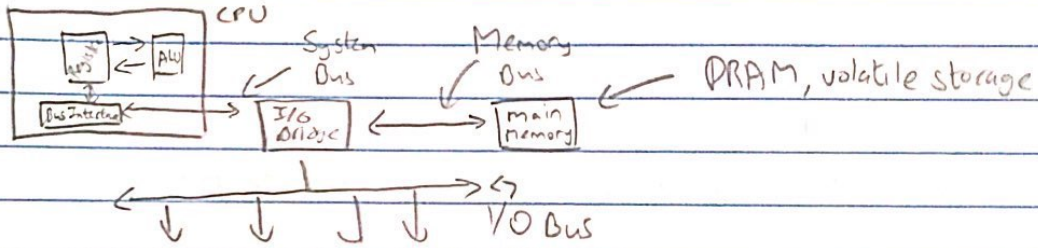
- The compiler has to be conservative
  - ↳ Branching can block optimization → not sure if code is actually executed
  - ↳ Storing to memory can block optimization → memory alteration
  - ↳ Functions can block memory
    - ↳ Inlining can help, but you lose access to interposition (updating with DLLs without recompiling)
    - ↳ Interposition leaves a gap in the code assembly
    - ↳ The compiler cannot assume the function doesn't modify memory
- Simple optimizations: code motion, strength reduction, subexpressions
- Memory conflicts: aliasing
  - ↳ No guarantee that two references to memory don't overlap → cannot optimize
  - ↳ Introducing local variables → tells compiler to not check for aliasing
- Architecture:
  - ↳ Processor can execute instructions in diff. order than compiler, must affect architectural units in correct order
    - ↳ Improves performance, reordered at end by retirement unit
  - ↳ Hardware does a lot of operation
  - ↳ Multiple functional units → instruction-level parallelism
    - ↳ 



→ computation not done, but pipeline has open spot
      - ↳ Computation divided into stages, independent computations can be passed from stage to stage
        - ↳ Latency bound vs. throughput bound → time vs. # of tasks
        - ↳ A straight-line chain of computation is not optimal
- SIMD – Single Instruction Multiple Data
  - ↳ vector elements → vector registers → hold multiple values
- Branch Prediction → attempt to predict on branching, penalty when wrong prediction
  - ↳ must flush pipeline if wrong, high cycle cost
  - ↳ predictable branches have lower overhead

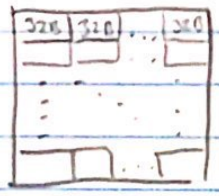
CS33 Lecture 10: Memory Hierarchy

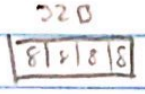
- CPU sends an address out, receives data/instructions from memory
  - ↳ Transistors scale well, memory wall means memory access doesn't
    - ↳ CPU can only be as fast as its ability to access memory
    - ↳ Bound by memory
- SRAM - Static Random Access Memory → registers, caches, etc
  - ↳ fast to access, hard to have high capacity
- DRAM - Dynamic Random Access Memory → main memory
  - ↳ slower to access higher capacity
- I/O Bus
  - ↳



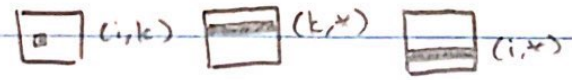
  - ↳ non-volatile storage: disks, USB drive, etc.
    - ↳ CPU request to disk, disk stored in main memory (DMA), CPU notified by interrupt
- Costs: area, material, power, monetary, etc.
- Locality: Programs tend to use data and instructions with addresses near those they have used recently
  - ↳ Temporal: recently referenced items are likely to be referenced again in the near future (loop w/ iterator)
  - ↳ Spatial: items with nearby addresses tend to be referenced close together (arrays and structs)
- General Cache Concepts
  - ↳ Cache → smaller, faster, more expensive memory caches a subset of blocks transferred from main memory
    - ↳ brings reused data in to lower latency of common accesses
    - ↳ blocks may contain larger granularity than necessary
      - ↳ why padding is necessary, takes advantage of spatial locality
    - ↳ decisions based on cache-policy combined w/ memory stream (eviction)
- Prefetching - attempt to access data ahead of time
  - ↳ further ahead of time = lower latency, balance w/ pollution of storage

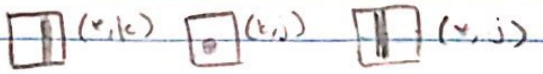
CS33 Lecture 11: Memory and Parallelism

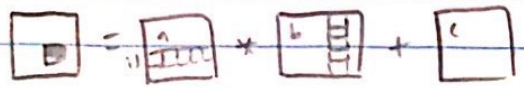
- Matrix Multiplication
  - 

    ← machine manipulated, not compiler
    → about reuse for locality
  - 1st method → k in inner-most loop
    

    - row order with i → 0.25 misses/iteration
    - column order with j → 1 miss/iteration
  - 2nd method → j in inner most loop
    

    - row-order w/ B and C → 0.25 misses/iteration
    - more operations, better locality
  - 3rd method → i in inner-most loop
    

    - worst, A and C miss every iteration
- Blocked Matrix Multiplication
  - Constrains matrix to work in smaller matrices
    

    - multiple rows/columns per block
    - constraining amt of data accessed → can control how much data the cache deals w/
    - can be worse due to time complexity / branching
- Cache Miss Analysis
  - Assume → block = 8 doubles, $C \ll n$, 3 blocks into cache ($3B^2 < C$)
    - $B^2/8$ misses for each block
    - $\frac{2n}{B} \times \frac{B^2}{8} = n\frac{B}{4}$
- MIMD
  - Multi-core → simultaneous execution when data sharing is unnecessary
  - Cooperative multithreading → separate execution, but sharing of some data

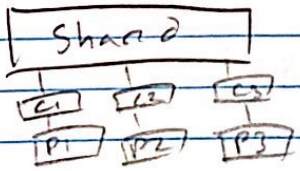
CS33 Lecture 11: Parallelism Part 1

- T0 T1 T2 T3 → ideally Parallel Latency = Latency/4
  - ↳ Sometimes less performance
    - ↳ overhead → communication between cores / OS interaction
    - ↳ ILP → may not be much parallelism in task
    - ↳ load balancing → work can't be distributed evenly
  - ↳ Sometimes more performance:
    - ↳ Memory bound applications can get much better memory locality by distributing across more caches
  - ↳ Balance computation v. communication
- Work sharing
  - ↳ for:
    - ↳ #pragma omp parallel
      :
      #pragma omp for
      for(i=0, i<10000; i++) → Static

      T0 (5000) ↙ ↘ T1 (5000)
      - ↳ static → tasks split, no further communication / load balancing
      - ↳ dynamic → requests more work when done with work
        - ↳ much more overhead due to communication
      - ↳ guided → start off with large chunks of work distributed and size decreases logarithmically
      - ↳ cannot have breaks → openmp must know at runtime how many iterations the loop will take
  - ↳ single:
    - ↳ communicates that a piece of code is only to be done by 1 thread
  - ↳ critical sections → only one thread can access this at a time → sequential
    - ↳ a lot of overhead in creating this region
  - ↳ reduction:
    - ↳ specifies associative operator and the variable
    - ↳ private variable created in each thread, after running, the associative operator is used on the variables to combine
- Deadlock → mutually exclusive resource access, threads hold resources until they have what they want, resources cannot be taken away, cyclical

CS 33 Lecture 12: Parallelism Part 2

- 2 types of parallelism
  - ↳ Data → implicit parallelism inside data itself
  - ↳ Task → specialization of task across cores
    - ↳ executes different things on each task
- OpenMP Tasks → independent units of work
  - ↳ composed of code, data, and internal control variables
  - ↳ Threads perform the work of each task
  - ↳ Runtime system decides execution → may be deferred, or executed immediately
    - ↳ guaranteed completion at barriers
      - ↳ barrier → thread barrier
      - ↳ taskwait → task barrier
- Scope - variables are default private in a task
  - ↳ shared → makes variables shared between threads
  - ↳ firstprivate → makes sure variable is private/has the correct value
- OpenMP memory model
  - ↳ supports a shared memory model → threads share an address space
    - ↳



  - ↳ defined by
    - ↳ coherence → behavior when single address is accessed by multiple threads
    - ↳ consistency → orderings of reads, writes, syncs w/ various addresses and multiple threads
      - ↳ weak consistency → can't reorder sync ops w/ read or write ops

CS33 Lecture 14 : Exceptional Control Flow and Linking

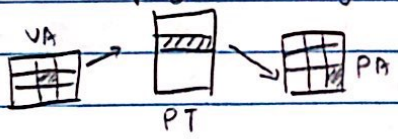
- Synchronous Exceptions
  - Traps
    - intentional → part of the actual code
    - system calls, breakpoint traps, special instructions
    - returns control to "next" instruction
  - Fault
    - unintentional but possibly recoverable
    - page faults, protection faults, floating point exceptions
    - either re-executes faulting instruction or aborts
  - Aborts
    - unintentional and unrecoverable
    - illegal instruction, parity error, machine check
    - aborts current program
- Page faults
  - user writes to memory location, which is currently on disk
    - OS copies page from disk to memory
- Invalid Memory Reference - fault
  - starts w/ page fault, OS detects invalid address, signals the process
    - sends SIGSEGV, program exits w/ segmentation fault
- System Calls - trap
  - each x86-64 system call has a unique ID #
  - OS needs to intervene (read, write, open, etc.)
  - sends exception w/ ID # in eax, performs the specified instruction, returns with status of operation
- Compilation:
  - cpp → preprocessor → #defines, #pragmas → injects source code
    - compiler → parses → forms an intermediate representation of source code across diff. languages → optimizes code
    - assembler → change into binary
      - multiple object files, may have unresolved symbols, incomplete view of memory layout
    - linker → resolves above issues
- Creating a library
  - Put all functions in a single source file → space/time inefficient
  - Put each function in a separate source file → burdensome on programmer
    - related object files into a single archive file, linker searches thru .a for symbols

CS33 Lecture 15: Virtual Memory

32-bit
2^{32} B

↳ Prog $\xrightarrow{\text{Addr.}}$ Mem
Data

↳ Model cannot handle simultaneous, independently compiled programs
  ↳ wastes memory, must account for all possible combinations
↳ VM: Prog $\xrightarrow{VA}$ MMU $\xrightarrow{PA}$ Mem.
  ↳ Applications have own view of memory, translated into single set of main mem.
  ↳ Excess memory not in DRAM, in disk → exploit locality
• Page Table - takes VA, translates into PA → dynamic by necessity
↳ resides in physical memory - may not fit in memory as a whole
↳



  ↳ Communication between VM and MM done in pages
  ↳ Relatively high access cost → common swaps = thrashing
↳ A page table register per process
  ↳ can protect from duplicate addresses across processes or allow sharing
↳ Page table start denoted by page table base register (PTBR)
↳ Each entry has a valid bit that tells if the entry is in physical memory
  ↳ accessing an entry with an invalid bit results in a page fault →
    evicts old page from memory and replaces w/ requested page
    ↳ behavior is non-deterministic
↳ Dirty page → page in memory doesn't match page on disk → consistency
↳ Breakdown of address
  ↳ Page offset → index within page → unchanged between VA and PA
  ↳ Page number → specifies which page in VM is translated into MM
• TLB - Translation Lookaside Buffer
↳ Hardware structure acts as a cache for the page table
↳ Takes you to physical address directly

- Multi-Level Page Tables
  - Level 1 -> each PTE points to a page table
    - allows for only part of page table in DRAM
    - more loads -> less severe if good locality
  - 

- Balancing page size, spatial locality, and performance

CS33 Lecture 16: MIPS

- CISC v. RISC
  - ↳ CISC motivated by capacity of memory limitations
    - ↳ complex instructions → mem. addressing mode
    - ↳ memory access or register address → all in 1 instruction = compact
    - ↳ small # of registers
    - ↳ variable length instructions
    - ↳ difficult to decode/ppeline/implement
  - ↳ Compilers becoming more powerful, complexity rising → move to RISC
    - ↳ simpler instructions, fixed length
      - ↳ wastes space, simplifies decoding
  - ↳ Boundaries blurred to allow both architectures to coexist
- MIPS
  - ↳ Fixed length → 32 bits
    - ↳ R type → register specifiers (Rs, Rt, Rd) → 5 bit, ALU oriented
      - ↳ 32 registers
      - ↳ dest, src
    - ↳ I type - load word and store word → data transfer
      - ↳ load = mem. → reg., store = reg. → mem.
      - ↳ LW $t0, 4($s0) → R[$t0] ← M[R[$s0]+4] (src, dest)
        - ↳ Base (register) + displacement (immediate)
    - ↳ J type - jump (j) and jump and link (jal)
      - ↳ j changes the PC
      - ↳ jal is like a call → $RA set to PC+4, PC to new PC
        - ↳ nested calls → $RA saved on the stack

CS33 Lecture 17: MIPS

- Pseudoinstructions
  - ↳ translated into other machine instructions
  - ↳ move -> move $t, $s -> addiu $t, $s, 0
  - ↳ clear -> clear $t -> addu $t, $zero, $zero
  - ↳ load 16-bit i -> li $t, C -> addiu $t, $zero, C_lo
  - ↳ load 32-bit i -> li $t, C -> lui $t, C_hi, ori $t, $t, C_lo
  - ↳ load label address -> la $t, A -> lui $t, A_hi, ori $t, $t, A_lo
    - ↳ instructions have 6 bits for op-code, 5 for each register -> only 16 to hold immediate -> must separate C_hi and C_lo
- System calls -> traps are recoverable and intentional
  - ↳ application trying to perform I/O that requires communication w/ OS -> syscall
  - ↳ passes a code as a parameter to identify the specific syscall -> $V0
    - ↳ switch statement - cspu
- $a0 and $a1 hold parameters