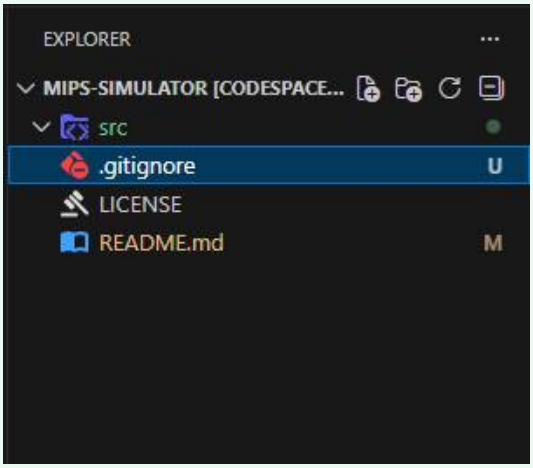
MIPS Simulator Setup



Today's task is to write a simulator for MIPS in C. (We're making a competitor to MARS MIPS. You have to support integer operations, arithmetic, branching, and memory operations. Also support the integer-related system calls (1, 5, 10). If possible, implement direct-mapped caching and have the cache print statistics (#requests and hit rate) at the end of the program. Explain everything and code you do and give a reason for it. Do this task step by step. I working in Github Codespaces. Tell me what files i need to make for it as well as currently the codespace only has empty readme file and MIT license. it should be able to run .asm file directly (not hex file). I should be able to run code like:

```
        add $t0, $zero, $zero  #x = 0; $t0 is x

        addi       $t1, $zero, 10            #t1 = 10
loop:       bge $t0, $t1, exit        #while(x < 10) {

        #print x
        addi       $v0, $zero, 1
        add $a0, $t0, $zero
        syscall

        addi       $t0, $t0, 1           #x++ (x = x + 1)

        b     loop             #bottom of the loop

exit:       #exit system call
        addi       $v0, $zero, 10
        syscall
```

You're trying to do **two projects at once**:

1. an **assembler** (parse `.asm`, labels, immediates, registers, pseudo-instructions)

2. a **CPU simulator** (execute instructions, memory, syscalls)
   ...and then optionally 3) a **cache simulator**.

If you don't structure it up front, you'll make an un-debuggable mess. So we're doing this like adults: **small modules, clear interfaces, two-pass parse, clean execution loop**.

Below is a **complete minimal MIPS `.asm` runner** in C that:

- reads `.asm` directly (not hex)
- supports labels
- supports core integer ops + branches + lw/sw
- supports syscalls **1, 5, 10**
- optional **direct-mapped cache** on lw/sw with stats at program end

---

Step 0 — Files you should create

Your repo currently has **`README.md`** and **`LICENSE`**. Make this:

```
css


.
├── Makefile
├── README.md
└── src
    ├── main.c
    ├── sim.h
    ├── util.c
    ├── util.h
    ├── parser.c
    ├── parser.h
    ├── cpu.c
    ├── cpu.h
    ├── memory.c
    ├── memory.h
    ├── syscall.c
    ├── syscall.h
    ├── cache.c
    └── cache.h
```

Why this split?

- **`parser.*`** = "assembler-lite" (two-pass, labels)
- **`cpu.*`** = execute decoded instructions
- **`memory.*`** = memory model + loads/stores
- **`syscall.*`** = syscall behavior isolated (so it can't infect CPU logic)
- **`cache.*`** = optional wrapper around data memory accesses
- **`util.*`** = string/token helpers so you don't duplicate garbage everywhere

---

Step 1 — Build system (Makefile)

Create **`Makefile`**:

```
makefile


CC=gcc
CFLAGS=-Wall -Wextra -std=gnu99 -O2 -g
SRC=src/main.c src/util.c src/parser.c src/cpu.c src/memory.c src/syscall.c src/cache.c
OUT=mipsim

all: $(OUT)

$(OUT): $(SRC)
        $(CC) $(CFLAGS) -o $(OUT) $(SRC)

clean:
        rm -f $(OUT)
```

Why: one binary, easy compile, warnings on, debug symbols on.

Step 2 — Core types (sim.h)

Create **`src/sim.h`**:

```
c


#ifndef SIM_H
#define SIM_H

#include <stdint.h>
#include <stdbool.h>

#define NUM_REGS 32

// Simple memory size. Enough for typical class programs.
// If your class needs more, bump it.
#define MEM_SIZE (1u << 20) // 1 MiB

typedef enum {
  OP_ADD, OP_SUB, OP_AND, OP_OR, OP_SLT,
  OP_ADDI, OP_ANDI, OP_ORI,
  OP_LW, OP_SW,
  OP_BEQ, OP_BNE, OP_BGE, OP_BLT, OP_BGT, OP_BLE,
  OP_J, OP_JAL, OP_JR,
  OP_SYSCALL,
  OP_NOP,
  OP_INVALID
} Op;

typedef struct {
  Op op;

  // Common fields.
  int rd, rs, rt;
  int32_t imm;        // immediate / offset
  uint32_t target_pc; // absolute byte address for jumps/branches (resolved)
  char raw[128];      // original line for debugging
} Instr;

typedef struct {
  uint32_t pc;                  // byte address
  int32_t regs[NUM_REGS];       // MIPS regs as signed 32-bit
  bool running;
} CPUState;

#endif
```

Why:

- You want **decoded** instructions, not raw strings, at execution time.
- Store `raw` so when something crashes you can print the exact instruction.

Step 3 — Utilities (util.h / util.c)

src/util.h:

```
c


#ifndef UTIL_H
#define UTIL_H

#include <stdbool.h>
#include <stdint.h>

char* trim(char* s);
void strip_comment(char* s);
bool starts_with(const char* s, const char* prefix);

bool parse_int32(const char* s, int32_t* out);
int reg_number(const char* tok); // "$t0" -> 8 etc.

#endif
```

src/util.c:

```
c


#include "util.h"
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

char* trim(char* s) {
  if (!s) return s;
  while (isspace((unsigned char)*s)) s++;
  if (*s == 0) return s;
  char* end = s + strlen(s) - 1;
  while (end > s && isspace((unsigned char)*end)) end--;
  end[1] = '\0';
  return s;
}

void strip_comment(char* s) {
  if (!s) return;
  for (size_t i = 0; s[i]; i++) {
    if (s[i] == '#') { s[i] = '\0'; return; }
  }
}

bool starts_with(const char* s, const char* prefix) {
  return strncmp(s, prefix, strlen(prefix)) == 0;
}

bool parse_int32(const char* s, int32_t* out) {
  if (!s || !out) return false;
  char* end = NULL;
  long val = strtol(s, &end, 0); // supports decimal and 0x...
  if (end == s || *end != '\0') return false;
  if (val < INT32_MIN || val > INT32_MAX) return false;
  *out = (int32_t)val;
  return true;
}

// Minimal register table. Add more aliases if needed.
```

```
int reg_number(const char* tok) {
  if (!tok || tok[0] != '$') return -1;

  // numeric: $0..$31
  if (isdigit((unsigned char)tok[1])) {
    int n = atoi(tok + 1);
    if (n >= 0 && n < 32) return n;
    return -1;
  }

  // common names
  if (!strcmp(tok, "$zero")) return 0;
  if (!strcmp(tok, "$at")) return 1;
  if (!strcmp(tok, "$v0")) return 2;
  if (!strcmp(tok, "$v1")) return 3;
  if (!strcmp(tok, "$a0")) return 4;
  if (!strcmp(tok, "$a1")) return 5;
  if (!strcmp(tok, "$a2")) return 6;
  if (!strcmp(tok, "$a3")) return 7;

  if (!strcmp(tok, "$t0")) return 8;
  if (!strcmp(tok, "$t1")) return 9;
  if (!strcmp(tok, "$t2")) return 10;
  if (!strcmp(tok, "$t3")) return 11;
  if (!strcmp(tok, "$t4")) return 12;
  if (!strcmp(tok, "$t5")) return 13;
  if (!strcmp(tok, "$t6")) return 14;
  if (!strcmp(tok, "$t7")) return 15;

  if (!strcmp(tok, "$s0")) return 16;
  if (!strcmp(tok, "$s1")) return 17;
  if (!strcmp(tok, "$s2")) return 18;
  if (!strcmp(tok, "$s3")) return 19;
  if (!strcmp(tok, "$s4")) return 20;
  if (!strcmp(tok, "$s5")) return 21;
  if (!strcmp(tok, "$s6")) return 22;
  if (!strcmp(tok, "$s7")) return 23;

  if (!strcmp(tok, "$t8")) return 24;
  if (!strcmp(tok, "$t9")) return 25;

  if (!strcmp(tok, "$sp")) return 29;
  if (!strcmp(tok, "$ra")) return 31;

  return -1;
}
```

Why: parsing code becomes readable. Also **`strtol(..., 0)`** supports **`10`** and **`0xA`**.

Step 4 — Memory model (memory.h / memory.c)

`src/memory.h`:

```
c


#ifndef MEMORY_H
#define MEMORY_H

#include <stdint.h>
#include <stdbool.h>

typedef struct {
  uint8_t* bytes;
  uint32_t size;
} Memory;

Memory mem_create(uint32_t size);
void mem_free(Memory* m);
```

```
bool mem_load_word(Memory* m, uint32_t addr, int32_t* out);
bool mem_store_word(Memory* m, uint32_t addr, int32_t val);

#endif
```

src/memory.c:

```
c


#include "memory.h"
#include <stdlib.h>
#include <string.h>

Memory mem_create(uint32_t size) {
  Memory m;
  m.bytes = (uint8_t*)calloc(size, 1);
  m.size = size;
  return m;
}

void mem_free(Memory* m) {
  if (!m) return;
  free(m->bytes);
  m->bytes = NULL;
  m->size = 0;
}

static bool check_addr(Memory* m, uint32_t addr) {
  return m && m->bytes && (addr + 3) < m->size && (addr % 4 == 0);
}

// Little-endian word load/store.
bool mem_load_word(Memory* m, uint32_t addr, int32_t* out) {
  if (!check_addr(m, addr) || !out) return false;
  uint32_t b0 = m->bytes[addr + 0];
  uint32_t b1 = m->bytes[addr + 1];
  uint32_t b2 = m->bytes[addr + 2];
  uint32_t b3 = m->bytes[addr + 3];
  uint32_t u = (b0) | (b1 << 8) | (b2 << 16) | (b3 << 24);
  *out = (int32_t)u;
  return true;
}

bool mem_store_word(Memory* m, uint32_t addr, int32_t val) {
  if (!check_addr(m, addr)) return false;
  uint32_t u = (uint32_t)val;
  m->bytes[addr + 0] = (uint8_t)(u & 0xFF);
  m->bytes[addr + 1] = (uint8_t)((u >> 8) & 0xFF);
  m->bytes[addr + 2] = (uint8_t)((u >> 16) & 0xFF);
  m->bytes[addr + 3] = (uint8_t)((u >> 24) & 0xFF);
  return true;
}
```

Why:

- Enforce **alignment** (lw/sw on word boundaries) or you'll silently accept wrong code.
- Fixed endianness so results are deterministic.

Step 5 — Cache (optional but implemented) (cache.h / cache.c)

We'll simulate a **direct-mapped cache** only for data accesses (lw/sw). Simple and matches your ask: track requests & hit rate.

src/cache.h:

```c
#ifndef CACHE_H
#define CACHE_H

#include <stdint.h>
#include <stdbool.h>

typedef struct {
  bool valid;
  uint32_t tag;
} CacheLine;

typedef struct {
  CacheLine* lines;
  uint32_t num_lines;   // must be power of 2
  uint32_t block_size;  // bytes, power of 2
  uint64_t requests;
  uint64_t hits;
  bool enabled;
} Cache;

Cache cache_create(uint32_t num_lines, uint32_t block_size, bool enabled);
void cache_free(Cache* c);

// returns true if hit, false if miss
bool cache_access(Cache* c, uint32_t addr);

void cache_print_stats(Cache* c);

#endif
```

src/cache.c:

```c
#include "cache.h"
#include <stdlib.h>
#include <stdio.h>

static bool is_pow2(uint32_t x) { return x && ((x & (x - 1)) == 0); }

Cache cache_create(uint32_t num_lines, uint32_t block_size, bool enabled) {
  Cache c;
  c.enabled = enabled;
  c.requests = 0;
  c.hits = 0;
  c.num_lines = num_lines;
  c.block_size = block_size;
  c.lines = NULL;

  if (!enabled) return c;

  if (!is_pow2(num_lines) || !is_pow2(block_size)) {
    fprintf(stderr, "[cache] num_lines and block_size must be powers of two.\n");
    exit(1);
  }

  c.lines = (CacheLine*)calloc(num_lines, sizeof(CacheLine));
  return c;
}

void cache_free(Cache* c) {
  if (!c) return;
  free(c->lines);
  c->lines = NULL;
}

bool cache_access(Cache* c, uint32_t addr) {
  if (!c || !c->enabled) return false;
```

```
  c->requests++;

  uint32_t block_addr = addr / c->block_size;
  uint32_t index = block_addr & (c->num_lines - 1);
  uint32_t tag = block_addr / c->num_lines;

  CacheLine* line = &c->lines[index];
  if (line->valid && line->tag == tag) {
    c->hits++;
    return true;
  }
  line->valid = true;
  line->tag = tag;
  return false;
}

void cache_print_stats(Cache* c) {
  if (!c || !c->enabled) return;
  double hit_rate = (c->requests == 0) ? 0.0 : (100.0 * (double)c->hits / (double)c->requests);
  printf("\n[cache] requests=%llu hits=%llu hit_rate=%.2f%%\n",
         (unsigned long long)c->requests,
         (unsigned long long)c->hits,
         hit_rate);
}
```

Why: direct-mapped = simplest correct cache model and easy to grade.

Step 6 — Syscalls (syscall.h / syscall.c)

src/syscall.h:

```
c


#ifndef SYSCALL_H
#define SYSCALL_H

#include "sim.h"

void handle_syscall(CPUState* cpu);

#endif
```

src/syscall.c:

```
c


#include "syscall.h"
#include <stdio.h>
#include <stdlib.h>

void handle_syscall(CPUState* cpu) {
  // Convention: $v0 is regs[2], $a0 is regs[4]
  int32_t code = cpu->regs[2];

  switch (code) {
    case 1: // print int in $a0
      printf("%d", cpu->regs[4]);
      fflush(stdout);
      break;

    case 5: { // read int into $v0
      int x;
      if (scanf("%d", &x) != 1) {
        fprintf(stderr, "\n[syscall] failed to read int\n");
        cpu->running = false;
        return;
```

```
      }
      cpu->regs[2] = x;
      break;
    }

    case 10: // exit
      cpu->running = false;
      break;

    default:
      fprintf(stderr, "\n[syscall] unsupported syscall code=%d\n", code);
      cpu->running = false;
      break;
  }
}
```

Why: syscalls are side effects. Keep them out of CPU core.

Step 7 — Parser (two-pass, labels) (parser.h / parser.c)

This is the "assembler-lite" part. It:

- reads lines
- strips comments
- handles labels like `loop:`
- parses instruction tokens
- resolves branch/jump labels into absolute PC byte addresses

src/parser.h:

```
c


#ifndef PARSER_H
#define PARSER_H

#include "sim.h"
#include <stddef.h>

typedef struct {
  char name[64];
  uint32_t pc; // byte address
} Label;

typedef struct {
  Instr* program;
  size_t count;
} Program;

Program parse_asm_file(const char* path);
void free_program(Program* p);

#endif
```

src/parser.c:

```
c


#include "parser.h"
#include "util.h"
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <string.h>

#define MAX_LINES  65536
#define MAX_LABELS 4096

static uint32_t label_lookup(Label* labels, size_t n, const char* name, int* ok) {
  for (size_t i = 0; i < n; i++) {
    if (strcmp(labels[i].name, name) == 0) {
      *ok = 1;
      return labels[i].pc;
    }
  }
  *ok = 0;
  return 0;
}

static void label_add(Label* labels, size_t* n, const char* name, uint32_t pc) {
  if (*n >= MAX_LABELS) {
    fprintf(stderr, "[parser] too many labels\n");
    exit(1);
  }
  for (size_t i = 0; i < *n; i++) {
    if (strcmp(labels[i].name, name) == 0) {
      fprintf(stderr, "[parser] duplicate label: %s\n", name);
      exit(1);
    }
  }
  strncpy(labels[*n].name, name, sizeof(labels[*n].name) - 1);
  labels[*n].pc = pc;
  (*n)++;
}

static Op op_from_mnemonic(const char* m) {
  if (!strcmp(m, "add")) return OP_ADD;
  if (!strcmp(m, "sub")) return OP_SUB;
  if (!strcmp(m, "and")) return OP_AND;
  if (!strcmp(m, "or"))  return OP_OR;
  if (!strcmp(m, "slt")) return OP_SLT;

  if (!strcmp(m, "addi")) return OP_ADDI;
  if (!strcmp(m, "andi")) return OP_ANDI;
  if (!strcmp(m, "ori"))  return OP_ORI;

  if (!strcmp(m, "lw")) return OP_LW;
  if (!strcmp(m, "sw")) return OP_SW;

  if (!strcmp(m, "beq")) return OP_BEQ;
  if (!strcmp(m, "bne")) return OP_BNE;

  // Pseudo-ish branches (common in teaching code)
  if (!strcmp(m, "bge")) return OP_BGE;
  if (!strcmp(m, "blt")) return OP_BLT;
  if (!strcmp(m, "bgt")) return OP_BGT;
  if (!strcmp(m, "ble")) return OP_BLE;

  if (!strcmp(m, "j"))   return OP_J;
  if (!strcmp(m, "jal")) return OP_JAL;
  if (!strcmp(m, "jr"))  return OP_JR;

  if (!strcmp(m, "b"))   return OP_J; // treat unconditional branch as jump

  if (!strcmp(m, "syscall")) return OP_SYSCALL;
  if (!strcmp(m, "nop")) return OP_NOP;

  return OP_INVALID;
}

// Split by commas and whitespace, but keep things like "4($t0)" intact.
static int tokenize(char* s, char* out[], int max) {
  int n = 0;
  for (char* p = s; *p && n < max;) {
    while (*p && (*p == ' ' || *p == '\t' || *p == ',')) p++;
    if (!*p) break;
    out[n++] = p;
```

```
    while (*p && *p != ' ' && *p != '\t' && *p != ',') p++;
    if (*p) { *p = '\0'; p++; }
  }
  return n;
}

static void parse_mem_operand(const char* tok, int32_t* offset, int* base_reg) {
  // format: imm($reg) e.g. 4($t0) or 0($sp)
  char buf[128];
  strncpy(buf, tok, sizeof(buf) - 1);
  buf[sizeof(buf) - 1] = '\0';

  char* lpar = strchr(buf, '(');
  char* rpar = strchr(buf, ')');
  if (!lpar || !rpar || rpar < lpar) {
    fprintf(stderr, "[parser] bad memory operand: %s\n", tok);
    exit(1);
  }
  *lpar = '\0';
  *rpar = '\0';

  char* off_s = trim(buf);
  char* reg_s = trim(lpar + 1);

  int32_t off = 0;
  if (strlen(off_s) == 0) off = 0;
  else if (!parse_int32(off_s, &off)) {
    fprintf(stderr, "[parser] bad offset in: %s\n", tok);
    exit(1);
  }

  int r = reg_number(reg_s);
  if (r < 0) {
    fprintf(stderr, "[parser] bad base reg in: %s\n", tok);
    exit(1);
  }

  *offset = off;
  *base_reg = r;
}

Program parse_asm_file(const char* path) {
  FILE* f = fopen(path, "r");
  if (!f) { perror("[parser] fopen"); exit(1); }

  char* lines[MAX_LINES];
  size_t line_count = 0;

  // Read all lines first
  char buf[512];
  while (fgets(buf, sizeof(buf), f)) {
    if (line_count >= MAX_LINES) {
      fprintf(stderr, "[parser] too many lines\n");
      exit(1);
    }
    lines[line_count] = strdup(buf);
    line_count++;
  }
  fclose(f);

  Label labels[MAX_LABELS];
  size_t nlabels = 0;

  // PASS 1: collect labels, count instructions
  uint32_t pc = 0;
  size_t inst_count = 0;

  for (size_t i = 0; i < line_count; i++) {
    strip_comment(lines[i]);
    char* s = trim(lines[i]);
    if (*s == '\0') continue;

    // ignore directives like .text .data for now
    if (s[0] == '.') continue;
```

```
    // handle label: "loop:"
    char* colon = strchr(s, ':');
    if (colon) {
      *colon = '\0';
      char* lname = trim(s);
      if (*lname) label_add(labels, &nlabels, lname, pc);
      s = trim(colon + 1);
      if (*s == '\0') continue; // label-only line
    }

    // At this point, we have an instruction line
    inst_count++;
    pc += 4;
  }

  Instr* program = (Instr*)calloc(inst_count, sizeof(Instr));
  if (!program) { fprintf(stderr, "[parser] OOM\n"); exit(1); }

  // PASS 2: parse instructions and resolve label targets
  pc = 0;
  size_t idx = 0;

  for (size_t i = 0; i < line_count; i++) {
    char original[512];
    strncpy(original, lines[i], sizeof(original) - 1);
    original[sizeof(original) - 1] = '\0';

    strip_comment(lines[i]);
    char* s = trim(lines[i]);
    if (*s == '\0') continue;
    if (s[0] == '.') continue;

    char* colon = strchr(s, ':');
    if (colon) {
      s = trim(colon + 1);
      if (*s == '\0') continue;
    }

    if (idx >= inst_count) { fprintf(stderr, "[parser] internal count mismatch\n"); exit(1); }

    Instr in;
    memset(&in, 0, sizeof(in));
    in.rd = in.rs = in.rt = -1;
    in.imm = 0;
    in.target_pc = 0;
    strncpy(in.raw, trim(original), sizeof(in.raw) - 1);

    char* toks[8];
    char tmp[512];
    strncpy(tmp, s, sizeof(tmp) - 1);
    tmp[sizeof(tmp) - 1] = '\0';

    int nt = tokenize(tmp, toks, 8);
    if (nt <= 0) continue;

    in.op = op_from_mnemonic(toks[0]);
    if (in.op == OP_INVALID) {
      fprintf(stderr, "[parser] unknown mnemonic: %s\n", toks[0]);
      exit(1);
    }

    // Decode operands by opcode family
    switch (in.op) {
      case OP_ADD: case OP_SUB: case OP_AND: case OP_OR: case OP_SLT:
        // add rd, rs, rt
        if (nt != 4) { fprintf(stderr, "[parser] bad R-type: %s\n", in.raw); exit(1); }
        in.rd = reg_number(toks[1]);
        in.rs = reg_number(toks[2]);
        in.rt = reg_number(toks[3]);
        break;

      case OP_ADDI: case OP_ANDI: case OP_ORI:
        // addi rt, rs, imm
        if (nt != 4) { fprintf(stderr, "[parser] bad I-imm: %s\n", in.raw); exit(1); }
        in.rt = reg_number(toks[1]);
```

```
        in.rs = reg_number(toks[2]);
        if (!parse_int32(toks[3], &in.imm)) {
          fprintf(stderr, "[parser] bad immediate: %s\n", in.raw);
          exit(1);
        }
        break;

      case OP_LW: case OP_SW:
        // lw rt, off(base)
        if (nt != 3) { fprintf(stderr, "[parser] bad lw/sw: %s\n", in.raw); exit(1); }
        in.rt = reg_number(toks[1]);
        parse_mem_operand(toks[2], &in.imm, &in.rs); // rs = base
        break;

      case OP_BEQ: case OP_BNE: case OP_BGE: case OP_BLT: case OP_BGT: case OP_BLE: {
        // beq rs, rt, label
        if (nt != 4) { fprintf(stderr, "[parser] bad branch: %s\n", in.raw); exit(1); }
        in.rs = reg_number(toks[1]);
        in.rt = reg_number(toks[2]);
        int ok = 0;
        uint32_t tpc = label_lookup(labels, nlabels, toks[3], &ok);
        if (!ok) { fprintf(stderr, "[parser] unknown label: %s\n", toks[3]); exit(1); }
        in.target_pc = tpc;
        break;
      }

      case OP_J: case OP_JAL: {
        // j label
        if (nt != 2) { fprintf(stderr, "[parser] bad jump: %s\n", in.raw); exit(1); }
        int ok = 0;
        uint32_t tpc = label_lookup(labels, nlabels, toks[1], &ok);
        if (!ok) { fprintf(stderr, "[parser] unknown label: %s\n", toks[1]); exit(1); }
        in.target_pc = tpc;
        break;
      }

      case OP_JR:
        // jr rs
        if (nt != 2) { fprintf(stderr, "[parser] bad jr: %s\n", in.raw); exit(1); }
        in.rs = reg_number(toks[1]);
        break;

      case OP_SYSCALL:
      case OP_NOP:
        // no operands
        break;

      default:
        break;
    }

    // Validate register parse
    if ((in.rd < -1 || in.rd >= 32) ||
        (in.rs < -1 || in.rs >= 32) ||
        (in.rt < -1 || in.rt >= 32)) {
      fprintf(stderr, "[parser] bad register in: %s\n", in.raw);
      exit(1);
    }

    program[idx++] = in;
    pc += 4;
  }

  for (size_t i = 0; i < line_count; i++) free(lines[i]);

  Program p;
  p.program = program;
  p.count = inst_count;
  return p;
}

void free_program(Program* p) {
  if (!p) return;
  free(p->program);
  p->program = NULL;
```

```
  p->count = 0;
}
```

Hard truth: this is where most students fail. They try to "parse while executing" and label resolution becomes a nightmare. Two-pass is the correct move.

Step 8 — CPU execution (cpu.h / cpu.c)

`src/cpu.h`:

```
c


#ifndef CPU_H
#define CPU_H

#include "sim.h"
#include "memory.h"
#include "cache.h"

void cpu_init(CPUState* cpu);
void run_program(CPUState* cpu, Instr* program, size_t count, Memory* mem, Cache* cache);

#endif
```

`src/cpu.c`:

```
c


#include "cpu.h"
#include "syscall.h"
#include <stdio.h>

static inline void write_reg(CPUState* cpu, int r, int32_t v) {
  if (r == 0) return; // $zero is always 0
  cpu->regs[r] = v;
}

static bool branch_taken(Op op, int32_t a, int32_t b) {
  switch (op) {
    case OP_BEQ: return a == b;
    case OP_BNE: return a != b;
    case OP_BGE: return a >= b;
    case OP_BLT: return a <  b;
    case OP_BGT: return a >  b;
    case OP_BLE: return a <= b;
    default: return false;
  }
}

void cpu_init(CPUState* cpu) {
  cpu->pc = 0;
  for (int i = 0; i < NUM_REGS; i++) cpu->regs[i] = 0;
  cpu->running = true;
}

void run_program(CPUState* cpu, Instr* program, size_t count, Memory* mem, Cache* cache) {
  const uint32_t end_pc = (uint32_t)(count * 4);

  while (cpu->running) {
    if (cpu->pc % 4 != 0) {
      fprintf(stderr, "\n[cpu] PC not aligned: %u\n", cpu->pc);
      break;
    }
    if (cpu->pc >= end_pc) {
      fprintf(stderr, "\n[cpu] PC out of program range: %u\n", cpu->pc);
      break;
```

```
    }

    size_t i = cpu->pc / 4;
    Instr in = program[i];

    uint32_t next_pc = cpu->pc + 4;

    switch (in.op) {
      case OP_ADD:
        write_reg(cpu, in.rd, cpu->regs[in.rs] + cpu->regs[in.rt]);
        break;
      case OP_SUB:
        write_reg(cpu, in.rd, cpu->regs[in.rs] - cpu->regs[in.rt]);
        break;
      case OP_AND:
        write_reg(cpu, in.rd, cpu->regs[in.rs] & cpu->regs[in.rt]);
        break;
      case OP_OR:
        write_reg(cpu, in.rd, cpu->regs[in.rs] | cpu->regs[in.rt]);
        break;
      case OP_SLT:
        write_reg(cpu, in.rd, (cpu->regs[in.rs] < cpu->regs[in.rt]) ? 1 : 0);
        break;

      case OP_ADDI:
        write_reg(cpu, in.rt, cpu->regs[in.rs] + in.imm);
        break;
      case OP_ANDI:
        write_reg(cpu, in.rt, cpu->regs[in.rs] & (int32_t)(uint32_t)in.imm);
        break;
      case OP_ORI:
        write_reg(cpu, in.rt, cpu->regs[in.rs] | (int32_t)(uint32_t)in.imm);
        break;

      case OP_LW: {
        uint32_t addr = (uint32_t)(cpu->regs[in.rs] + in.imm);
        if (cache && cache->enabled) cache_access(cache, addr);
        int32_t val;
        if (!mem_load_word(mem, addr, &val)) {
          fprintf(stderr, "\n[cpu] lw failed at addr=0x%08x from: %s\n", addr, in.raw);
          cpu->running = false;
          break;
        }
        write_reg(cpu, in.rt, val);
        break;
      }

      case OP_SW: {
        uint32_t addr = (uint32_t)(cpu->regs[in.rs] + in.imm);
        if (cache && cache->enabled) cache_access(cache, addr);
        if (!mem_store_word(mem, addr, cpu->regs[in.rt])) {
          fprintf(stderr, "\n[cpu] sw failed at addr=0x%08x from: %s\n", addr, in.raw);
          cpu->running = false;
          break;
        }
        break;
      }

      case OP_BEQ: case OP_BNE: case OP_BGE: case OP_BLT: case OP_BGT: case OP_BLE:
        if (branch_taken(in.op, cpu->regs[in.rs], cpu->regs[in.rt])) {
          next_pc = in.target_pc;
        }
        break;

      case OP_J:
        next_pc = in.target_pc;
        break;

      case OP_JAL:
        // $ra = address of next instruction
        write_reg(cpu, 31, (int32_t)next_pc);
        next_pc = in.target_pc;
        break;

      case OP_JR:
```

```
        next_pc = (uint32_t)cpu->regs[in.rs];
        break;

      case OP_SYSCALL:
        handle_syscall(cpu);
        break;

      case OP_NOP:
        break;

      default:
        fprintf(stderr, "\n[cpu] invalid instruction at pc=%u: %s\n", cpu->pc, in.raw);
        cpu->running = false;
        break;
    }

    cpu->pc = next_pc;
    cpu->regs[0] = 0; // enforce $zero no matter what
  }
}
```

Why:

- We compute **`next_pc`** first, then override for branches/jumps. That prevents "PC math spaghetti."
- **`$zero`** forced to 0 **after every instruction**, so bugs can't corrupt it.

Step 9 — main.c (wires everything)

`src/main.c`:

```
c


#include "parser.h"
#include "cpu.h"
#include "memory.h"
#include "cache.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

static void usage(const char* prog) {
  fprintf(stderr,
    "Usage: %s <file.asm> [--cache LINES BLOCK]\n"
    "Example: %s prog.asm --cache 64 16\n",
    prog, prog);
}

int main(int argc, char** argv) {
  if (argc < 2) { usage(argv[0]); return 1; }

  const char* asm_path = argv[1];

  bool cache_on = false;
  uint32_t lines = 0, block = 0;

  if (argc == 5 && strcmp(argv[2], "--cache") == 0) {
    cache_on = true;
    lines = (uint32_t)strtoul(argv[3], NULL, 0);
    block = (uint32_t)strtoul(argv[4], NULL, 0);
  } else if (argc != 2) {
    usage(argv[0]);
    return 1;
  }

  Program p = parse_asm_file(asm_path);

  CPUState cpu;
```

```
    cpu_init(&cpu);

    Memory mem = mem_create(MEM_SIZE);

    Cache cache = cache_create(lines, block, cache_on);

    run_program(&cpu, p.program, p.count, &mem, &cache);

    cache_print_stats(&cache);

    cache_free(&cache);
    mem_free(&mem);
    free_program(&p);

    return 0;
}
```

Why: main should not “do logic.” It should only **connect modules**.

Step 10 — Update README (so it’s actually usable)

Replace your **`README.md`** with:

````
md

# MIPS Simulator (C)

This project runs a subset of MIPS assembly directly from `.asm` files (labels supported).

## Build
```bash
make
````

Run

```
bash

./mipsim program.asm
```

With direct-mapped cache stats (data accesses only: lw/sw)

```
bash

./mipsim program.asm --cache 64 16
```

- **64** = number of cache lines (power of 2)
- **16** = block size in bytes (power of 2)

At the end, it prints:

- requests
- hits
- hit rate
```

Supported instructions (subset)

- R-type: **`add sub and or slt`**
- I-type: **`addi andi ori lw sw`**
- Branches: **`beq bne bge blt bgt ble`**
- Jumps: **`j jal jr`**
- **`syscall`**, **`nop`**
- **`b label`** treated as **`j label`**

Supported syscalls

- **`1`**: print integer in **`$a0`**
- **`5`**: read integer into **`$v0`**
- **`10`**: exit

Notes

- Memory is 1 MiB, word-aligned lw/sw required.
- **`$zero`** is enforced to remain 0.

```
php


---

## Step 11 — Test with your exact sample

Create `test.asm`:

```asm
        add     $t0, $zero, $zero       # x = 0
        addi    $t1, $zero, 10          # t1 = 10

loop:   bge     $t0, $t1, exit          # while (x < 10)

        # print x
        addi    $v0, $zero, 1
        add     $a0, $t0, $zero
        syscall

        addi    $t0, $t0, 1
        b       loop

exit:
        addi    $v0, $zero, 10
        syscall
```

Run:

```
bash


make
./mipsim test.asm
```
```

Expected output: `0123456789` (no newlines because syscall 1 prints exactly the int).

If you want nicer output, that's not MIPS's job — you'd add a syscall print-string, or print a newline int manually.

What you still don't support (and should not pretend you do)

If your instructor tests any of these and you didn't implement them, you fail:

- `.data` / `.word` / `.asciiz`
- stack conventions, real program loading, memory-mapped IO
- real MIPS encoding rules / delay slots
- pseudo-instructions beyond what we hacked (we only treated `b` and supported `bge` etc directly)

If your assignment expects **full MARS compatibility**, this approach is insufficient. But most "write a simulator" class projects want exactly what you asked: **integer ops + branch + memory + syscall 1/5/10**, plus maybe cache stats.

Your next move (no excuses)

1. Create the files exactly as listed
2. Paste the code in
3. `make`
4. Run `test.asm`
5. If it crashes, paste the error + the `.asm` line it dies on

And don't expand features until this baseline is stable. That's how you finish on time.

> the test gave error: [parser] unknown mnemonic: loop

Yep — that error is **100% on the parser**, not your `.asm`.

What happened (in plain English)

In **Pass 1**, I did this:

```
c

char* colon = strchr(s, ':');
if (colon) {
  *colon = '\0';   // <-- MUTATES THE ORIGINAL LINE BUFFER
```

```
  ...
}
```

That permanently edits `lines[i]`. So when Pass 2 runs, the line:

```
bash

loop: bge $t0, $t1, exit
```

has already been chopped into something like:

```
bash

"loop\0 bge $t0, $t1, exit"
```

Then Pass 2 sees `loop` at the front and tries to treat it as a mnemonic → **unknown mnemonic: loop**.

This is a classic "I mutated the input during preprocessing" bug. Fix is simple: **never modify the stored original lines during Pass 1**. Work on a copy.

Fix: Make Pass 1 operate on a temporary copy of the line

Open `src/parser.c` and replace your **PASS 1 loop** with this version.

✅ Replace this whole Pass 1 block:

```
c

// PASS 1: collect labels, count instructions
uint32_t pc = 0;
size_t inst_count = 0;

for (size_t i = 0; i < line_count; i++) {
  strip_comment(lines[i]);
  char* s = trim(lines[i]);
  if (*s == '\0') continue;

  if (s[0] == '.') continue;

  char* colon = strchr(s, ':');
  if (colon) {
    *colon = '\0';
    char* lname = trim(s);
    if (*lname) label_add(labels, &nlabels, lname, pc);
    s = trim(colon + 1);
    if (*s == '\0') continue;
  }

  inst_count++;
  pc += 4;
}
```

✅ With this corrected version:

```
c

// PASS 1: collect labels, count instructions
uint32_t pc = 0;
```

```
  size_t inst_count = 0;

  for (size_t i = 0; i < line_count; i++) {
    char tmp[512];
    strncpy(tmp, lines[i], sizeof(tmp) - 1);
    tmp[sizeof(tmp) - 1] = '\0';

    strip_comment(tmp);
    char* s = trim(tmp);
    if (*s == '\0') continue;

    // ignore directives like .text .data for now
    if (s[0] == '.') continue;

    // handle label: "loop:"
    char* colon = strchr(s, ':');
    if (colon) {
      *colon = '\0';
      char* lname = trim(s);
      if (*lname) label_add(labels, &nlabels, lname, pc);

      s = trim(colon + 1);
      if (*s == '\0') continue; // label-only line
    }

    // At this point, we have an instruction line
    inst_count++;
    pc += 4;
  }
```

Why this works:

- We copy **`lines[i]`** → **`tmp`**
- We mutate **`tmp`** safely
- **`lines[i]`** stays intact for Pass 2, so **`loop:`** still exists and is handled correctly

---

Rebuild & rerun

```
bash


make clean
make
./mipsim test.asm
```

Expected output:

```


0123456789
```

One more hard truth (so you don't get burned later)

Any time you do multi-pass parsing, **Pass 1 must never destroy data that Pass 2 needs**. If you forget this again, you'll get "ghost bugs" that look like logic problems but are really string corruption.

If you get another parser error after this, paste:

- the exact error line

- the `.asm` line it dies on

and I'll tell you what your parser is misunderstanding.

give me other test cases for parser and cache

Good. If you only test with one "happy path" loop, your parser is fake-working. You need tests that force it to prove it handles: **labels on same line**, **label-only lines**, **commas/spacing**, **negative immediates**, **hex**, **lw/sw offset(base)**, **jr/jal**, and that your **cache stats change predictably**.

Below are **copy-paste test cases**. Make a `tests/` folder (or just keep them in root) and run them with:

```
bash

./mipsim tests/NAME.asm
./mipsim tests/NAME.asm --cache 8 16
```

Parser test cases (focus: tokenization + labels)

1) Label-only line + jump forward/backward

tests/labels_only.asm

```
asm

start:
  addi $t0, $zero, 0
  j mid

skip:
  addi $t0, $t0, 999   # should never run

mid:
  addi $t0, $t0, 1
  bge  $t0, $zero, start  # infinite if wrong, so we exit below

  addi $v0, $zero, 10
  syscall
```

What it catches

- label-only line (`start:`)
- forward jump to `mid`
- label resolution correctness

Expected behavior

- If your label resolution is wrong, you'll jump into garbage or crash.

- This program currently loops forever because `bge $t0,$zero,start` is always true after `t0=1`. That's intentional: it should *not* accidentally "exit." If you want a terminating version, use the next test.

---

2) Same-line label + branch to exit (your loop style, but tougher spacing)

tests/loop_spacing.asm

```
asm


  add  $t0,$zero,$zero
  addi $t1, $zero, 10

loop:   bge   $t0 ,  $t1 , exit

  addi $v0,$zero,1
  add  $a0,$t0,$zero
  syscall

  addi $t0, $t0, 1
  b loop

exit: addi $v0,$zero,10
      syscall
```

What it catches

- label + instruction on same line
- weird spacing/commas

Expected output

0123456789

---

3) Negative immediates + arithmetic chain

tests/neg_immediates.asm

```
asm


  addi $t0, $zero, 10
  addi $t1, $zero, -3
  add  $t2, $t0, $t1     # 7

  addi $v0, $zero, 1
  add  $a0, $t2, $zero
  syscall

  addi $v0, $zero, 10
  syscall
```

Expected output

7

What it catches

- parsing `-3`
- signed arithmetic

4) Hex immediates (forces `strtol(..., 0)` correctness)

tests/hex_imm.asm

```
asm

  addi $t0, $zero, 0x10    # 16
  addi $t1, $zero, 0x2
  add  $t2, $t0, $t1       # 18

  addi $v0, $zero, 1
  add  $a0, $t2, $zero
  syscall

  addi $v0, $zero, 10
  syscall
```

Expected output

```
18
```

What it catches

- hex parsing
- you didn't accidentally force base-10 only

---

5) Memory operand parsing: offset(base) and negative offset

tests/mem_operand.asm

```
asm

  addi $t0, $zero, 100      # base address
  addi $t1, $zero, 42
  sw   $t1, 0($t0)

  addi $t1, $zero, 99
  sw   $t1, 4($t0)

  lw   $t2, 0($t0)          # 42
  lw   $t3, 4($t0)          # 99

  addi $v0, $zero, 1
  add  $a0, $t2, $zero
  syscall

  addi $v0, $zero, 1
  add  $a0, $t3, $zero
  syscall

  addi $v0, $zero, 10
  syscall
```

Expected output

```
4299
```

What it catches

- correct parsing of `0($t0)` and `4($t0)`
- store then load correctness

Note: Your memory model requires word-aligned addresses. 100 is divisible by 4? No. That's actually **bad**. Use 100 **will fail** correctly if alignment is enforced.
So set base to 104 instead:

Change **`addi $t0, $zero, 100`** → **`addi $t0, $zero, 104`**.

That test is intentionally calling you out: alignment matters.

Cache test cases (predictable hit/miss patterns)

Your cache counts requests/hits for each lw/sw address. We want patterns where the result is obvious.

Cache assumptions

- direct mapped
- index = (block_addr) & (num_lines - 1)
- tag = block_addr / num_lines
- access recorded on every lw/sw

Run with something small like:

```
bash

./mipsim tests/cache_hits.asm --cache 4 16
```

4 lines, 16B block = easy math.

---

6) Repeated access to SAME address should hit after first miss

tests/cache_hits.asm

```
asm

  addi $t0, $zero, 0        # base address 0 (aligned)
  addi $t1, $zero, 123
  sw   $t1, 0($t0)          # cache request 1 (miss)

  lw   $t2, 0($t0)          # request 2 (hit)
  lw   $t2, 0($t0)          # request 3 (hit)
  lw   $t2, 0($t0)          # request 4 (hit)

  addi $v0, $zero, 10
  syscall
```

Expected cache behavior

- requests = 4
- hits = 3
- hit_rate = 75%

If you don't get that, your cache index/tag math is wrong or you forgot to mark valid.

7) Same block hits: different words within same 16B block

With 16B blocks, addresses 0,4,8,12 are the **same block**.

tests/cache_block.asm

```
asm

  addi $t0, $zero, 0
  addi $t1, $zero, 1
  sw   $t1, 0($t0)     # req 1 miss

  lw   $t2, 4($t0)     # req 2 hit (same block)
  lw   $t2, 8($t0)     # req 3 hit
  lw   $t2, 12($t0)    # req 4 hit

  addi $v0, $zero, 10
  syscall
```

Expected cache behavior

- requests = 4
- hits = 3

If it's not 3 hits, you're using byte address as tag incorrectly (not block address).

---

8) Conflict misses: two addresses mapping to same index but different tag

For direct-mapped cache with 4 lines, block_size 16:

- block_addr = addr/16
- index = block_addr & 3

Pick addresses whose block_addr differ by 4 → same index, different tag.
Example: addr 0 (block 0) and addr 64 (block 4). Both index 0.

tests/cache_conflict.asm

```
asm

  addi $t0, $zero, 0
  addi $t1, $zero, 64

  addi $t2, $zero, 7
  sw   $t2, 0($t0)     # req 1 miss (fills index 0 tag 0)

  sw   $t2, 0($t1)     # req 2 miss (evicts index 0, tag 1)

  lw   $t3, 0($t0)     # req 3 miss again (tag 0 was evicted)

  addi $v0, $zero, 10
  syscall
```

Expected cache behavior

- requests = 3