Project 2 Documentation
CPU Simulation

Brysen Pfingsten, Nathaniel Savoury, David Anthony Fields

October 27, 2025

CSAS 3111, Seton Hall University

Contents

1 Outline

1.1 Problem Statement

This project builds upon Project 1: Basic Computer System Simulator by adding advanced features that emulate key components of a modern computer system. These enhancements include process control, memory management, multithreading, and interrupt handling. The simulation will model various system elements such as the CPU, memory, instruction cycle, Direct Memory Access (DMA), interrupt management, and the Process Control Block (PCB), among others.

Your objective is to develop a comprehensive Computer System Simulator that integrates fundamental hardware components—such as the CPU, memory, and instruction set architecture—with advanced operating system functionalities. These include process lifecycle management, context switching, and the coordination of memory and input/output operations. The simulator should consist of multiple interdependent modules capable of managing resources, handling interrupts, and supporting concurrent thread execution.

- Module 1: CPU and Instruction Cycle
- Module 2: Memory Management
- Module 3: Process Scheduling and Multitasking
- Module 4: Interrupt Handling and Dispatcher
- Module 5: Multithreading or Forking

1.2 Introduction

To complete the tasked of simulating a computer system, we elected to us the programming language, `C`, to implement this project. This documentation is structured as follows, the implementation for modules 1 through 5 can be found in Section 2, Section 3, Section 4, Section 5 and, Section 6 respectively. The testing and debugging is found in Section 7. Finally, the simulation of the described computer system is be found in Section 8.

2 Module 1: CPU and Instruction Cycle

2.1 Problem Statement

The objective of this module is to extend the basic CPU simulator by developing a detailed fetch-decode-execute cycle that accurately models CPU operations based on a chosen architecture. The system must implement essential CPU components—including the Program Counter, Accumulator, Instruction Register, and Status Register—and simulate memory capable of storing both instructions and data for multiple processes. The CPU should support at least ten arithmetic, logical, and control instructions, manage status flags for zero, carry, and overflow conditions, and include error handling for invalid instructions. Additionally, the simulator must demonstrate process support by saving and restoring CPU states during context switches, enabling the execution of multiple processes.

2.2 Implementation

CPU Design

To implement the core components of the CPU, we first chose to represent the CPU as a structure containing an array of registers and flags.

```c
typedef struct {
  dword registers[COUNT];
} Cpu;
```

The following enumeration defines the indices into the `registers` array.

```c
enum { AX , BX, CX, DX, PC, IR, ACC, FLAG, COUNT };
```

Here, `COUNT` does not represent a component of the CPU but rather serves to make the CPU easily extensible. `FLAG` is a single integer with certain bits mapping to a flag. The following enum describes these bit positions.

```c
enum {
  F_ZERO = 1 << 0,  /* zero flag */
  F_OVFLW = 1 << 1, /* overflow flag */
  F_CARRY = 1 << 2, /* carry flag */
};
```

Setting these flags are achieved through the C macro system.

```c
#define SET_FLAG(flag)   (THE_CPU.registers[FLAG] |= (flag))
#define CLEAR_FLAG(flag) (THE_CPU.registers[FLAG] &= ~(flag))
#define CLEAR_ALL_FLAGS  (THE_CPU.registers[FLAG] = 0)
```

An example of how to access these registers can be seen in the `init_cpu` function.

```c
void init_cpu(Cpu* cpu)
{
  cpu->registers[PC]   = MEM_START;
  cpu->registers[IR]   = EMPTY_REG;
  cpu->registers[FLAG] = F_ZERO;
  cpu->registers[ACC] = 0;
  printf("Initialized the cpu!\n");
  cpu_print_state();
}
```

Fetch-Decode-Execute Cycle

The main function for running the CPU proceeds through the fetch decode and execute cycle while the `program_size` has not been reached and while a `HALT` signal has not be sent. We use the assembly-like `goto` and labels for efficiency in the main loop. During each iteration, we begin by displaying the current cycle and checking for a halt signal. If there was a halt signal, we display that and terminate execution. If there was not halt, we proceed by fetching the next instruction, executing that instruction, checking and handling interrupts, then finally printing the new state of the CPU.

```c
void cpu_run(const int program_size) {
  int i = 0;

  start:
    if ((i >= program_size) || (THE_CPU.registers[PC] != CPU_HALT))
      goto end;

    printf("=== Cycle %d ===\n", i + 1);

    if (THE_CPU.registers[PC] == CPU_HALT) {
      printf("CPU Halted!\n");
      goto end;
    }

    fetch();
    execute();
    check_for_interrupt();

    cpu_print_state();

    i++;
    goto start;

  end:
    ;
}
```

The helpers `fetch`, `decode`, and `execute` are implemented as follows.

```
void fetch()
{
  THE_CPU.registers[IR] = read_mem(THE_CPU.registers[PC]);
  THE_CPU.registers[PC]++;
}

static dword decode(dword instruction)
{
  dword op = (instruction >> OPCODE_SHIFT);
  return op;
}

void execute()
{
  dword instruction = THE_CPU.registers[IR];
  dword op = decode(instruction);

  execute_instruction(op, instruction);
}
```

The `fetch` functions reads the address in the program counter and loads the data associated with it (from main memory) into the instruction register. Finally, it increments the program counter. The decode function extracts the opcode by obtaining the upper bits of the instruction. Lastly, the `execute` function dispatches the obtained opcode and instruction to be executed by the ISA module.

Instruction Set Architecture (ISA)

We extend our ISA from Project 1 with logical bit operations, conditional and unconditional branching, and more loading/storing operations. Again, we use an enumeration to define our opcodes:

```
enum {
  ADD, // ACC = ACC + operand
  SUB, // ACC = ACC - operand
  MUL, // ACC = ACC * operand
  DIV, // ACC = ACC / operand

  AND, // ACC = ACC & operand
  OR,  // ACC = ACC | operand
  XOR, // ACC = ACC ^ operand
  NOT, // ACC = ~ACC

  BRANCH, // Conditional branch based on flag(s)
  JUMP,   // PC = address
  JUMPR,  // PC = PC + offset
  JUMPZ,  // if (ZF == 1) PC = address

  STORE, // MEM[address] = ACC
  STRR,  // MEM[reg] = ACC
  STRI,  // MEM[IMM] = ACC

  LOAD, // ACC = MEM[address]
  LEA,  // ACC = &operand
  LDR,  // ACC = MEM[reg]
  LDI,  // ACC = IMM

  INTR // Trigger interrupt or handle ISR
};
```

Do to having more than $2^4 = 16$ opcodes, our operating system is now 32 bit based.

Flag Setting For each of our 4 arithmetic operations, we set the CARRY, OVERFLOW, and ZERO flags. The implementation is below.

```
void set_zero_flag(dword value)
{
  CLEAR_FLAG(F_ZERO);
```

```c
  if(value == 0){SET_FLAG(F_ZERO); return;}
}

void set_add_flags(dword a, dword b, dword r) {
  CLEAR_ALL_FLAGS;

  // Unsigned carry out of bit 15
  if ((uint64_t)a + (uint64_t)b > 0xFFFFFFFF){
    SET_FLAG(F_CARRY);
  }
  // Signed overflow: inputs same sign, result different sign
  sdword sa = (sdword)a;
  sdword sb = (sdword)b;
  sdword sr = (sdword)r;
  if((sa > 0 && sb > 0 && sr < 0) ||
      (sa < 0 && sb < 0 && sr > 0)){
    SET_FLAG(F_OVFLW);
  }

  set_zero_flag(r);
}

void set_sub_flags(dword a, dword b, dword r) {
  CLEAR_ALL_FLAGS;

  // Borrow in unsigned
  if((dword)a < (dword)b)
  {
    SET_FLAG(F_CARRY);
  }

  // Signed overflow: inputs different sign, result sign differs from a
  sdword sa = (sdword)a;
  sdword sb = (sdword)b;
  sdword sr = (sdword)r;
  if((sa >= 0 && sb <  0 && sr <  0) ||
       (sa <  0 && sb >= 0 && sr >= 0))
  {
    SET_FLAG(F_OVFLW);
  }
  set_zero_flag(r);
}

void set_mul_flags(dword a, dword b, dword r)
{
  CLEAR_ALL_FLAGS;

  uint64_t p = (uint64_t)a * (uint64_t)b;
  if(p >> 32 != 0)
  {
    SET_FLAG(F_CARRY);
  }

  int64_t sp = (sdword)a * (sdword)b;
  if(sp > INT32_MAX || sp <INT32_MIN)
  {
    SET_FLAG(F_OVFLW);
  }

  set_zero_flag(r);
}

void set_div_flags(dword a, dword b, dword r)
{
  CLEAR_ALL_FLAGS;

  CLEAR_FLAG(F_CARRY);

  if((sdword)a == INT32_MIN && (sdword)b == -1){SET_FLAG(F_OVFLW);}

  set_zero_flag(r);
}
```

Arithmetic Operations Our four arithmetic operations follows a similar pattern.

1. Obtain the destination register, source register, and immediate mode flag.
2. If the immediate flag is set, perform sign extension and do the operation.

3. If the immediate flag is not set, obtain the value from the source register and do the operation.

4. Finally, set the flags for that operation.

As they are all so similar, we will only show addition here.

```
void add(const dword instruction)
{
  // Destination register
  dword dr = (instruction >> DR_SHIFT) & 0xF;
  // Firsr operand (source register one)
  dword sr1 = (instruction >> SR1_SHIFT) & 0xF;
  // Immediate mode flag
  dword imm_flag = (instruction >> MODE_SHIFT) & 0x1;

  if(imm_flag)
  {
    sdword imm = sign_extend(instruction & OPERAND_MASK, 12);
    THE_CPU.registers[dr] = THE_CPU.registers[sr1] + imm;
    set_add_flags(THE_CPU.registers[sr1], imm, THE_CPU.registers[dr]);
  }
  else
  { // Not in immediate mode so we must get our value from a second source register
    dword sr2 = instruction & 0xF;
    THE_CPU.registers[dr] = THE_CPU.registers[sr1] + THE_CPU.registers[sr2];
    set_add_flags(THE_CPU.registers[sr1],
        THE_CPU.registers[sr2],
        THE_CPU.registers[dr]);
  }
}
```

Bitwise Operations Our bitwise operations are similar to our arithmetic operations. We provide bitwise and here as an example.

```
void bit_and(const dword instruction)
{
  // Destination register
  dword dr = (instruction >> DR_SHIFT) & 0xF;

  // Firsr operand (source register one)
  dword sr1 = (instruction >> SR1_SHIFT) & 0xF;
  // Immediate mode flag
  dword imm_flag = (instruction >> MODE_SHIFT) & 0x1;

  if(imm_flag)
  {
    sdword imm = sign_extend(instruction & OPERAND_MASK, 12);
    THE_CPU.registers[dr] = THE_CPU.registers[sr1] & imm;
  }
  else
  { // Not in immediate mode so we must get our value from a second source register
    dword sr2 = instruction & 0xF;
    THE_CPU.registers[dr] = THE_CPU.registers[sr1] & THE_CPU.registers[sr2];
  }
  set_zero_flag(THE_CPU.registers[dr]);
}
```

Branching We provide new operations for conditional and unconditional branching.

- `BRANCH` — Branch by some offset is there are flags set.
- `JUMP` — Begin execution at the given memory address.
- `JUMPR` — Jump to the address in the given register.
- `JUMPZ` — Jump to the given address is the zero flag is set.

They are implemented as follows.

```
void branch(const dword instruction)
{
  sdword offset = sign_extend(instruction & 0x000FFFFF, 20);
  dword cond = (instruction >> DR_SHIFT) & 0x7;

  if (cond & THE_CPU.registers[FLAG])
    THE_CPU.registers[PC] += offset;
}

void jump(const dword instruction)
{
  dword dest = (instruction >> SR1_SHIFT) & 0xF;
  THE_CPU.registers[PC] = THE_CPU.registers[dest];
}
```

```
 void jump_register(const dword instruction)
{
  THE_CPU.registers[AX] = THE_CPU.registers[PC];

  dword flag = (instruction >> DR_SHIFT) & 0x1;

  if(flag)
  {
    sdword offset = sign_extend(instruction & REG_MASK, 16);
    THE_CPU.registers[PC] += offset;
  }
  else
  {
    dword br = (instruction >> SR1_SHIFT) & 0xF;
    THE_CPU.registers[PC] = THE_CPU.registers[br];
  }
}

void jump_zero(const dword instruction)
{
  if((THE_CPU.registers[FLAG] >> 0) & 1)
  {
    dword flag = (instruction >> DR_SHIFT) & 0x1;
    if(flag)
    {
      sdword offset = sign_extend(instruction & REG_MASK, 16);
      THE_CPU.registers[PC] += offset;
    }
    else
    {
      dword br = (instruction >> SR1_SHIFT) & 0xF;
      THE_CPU.registers[PC] = THE_CPU.registers[br];
    }
  }
}
```

Storing Our four storing operations are

- `STORE` — Stores the data in the ACC register at the given memory address.

- STRR — Store the contents of the given register in the given memory address.
- STRI — Writes data from a register to a memory location specified by an address held in another register

They are implemented as follows:

```c
void store(const dword instruction)
{
  dword sr = (instruction >> DR_SHIFT) & 0xF;
  sdword offset = sign_extend(instruction & IMM_MASK, 20);
  write_mem(THE_CPU.registers[PC] + offset, THE_CPU.registers[sr]);
}

void store_register(const dword instruction)
{
  dword sr = (instruction >> DR_SHIFT) & 0xF;
  dword br = (instruction >> SR1_SHIFT) & 0xF;
  sdword offset = sign_extend(instruction & REG_MASK, 16);

  write_mem(THE_CPU.registers[br] + offset, THE_CPU.registers[sr]);
}

void store_indirect(const dword instruction)
{
  dword sr = (instruction >> DR_SHIFT) & 0xF;
  sdword offset = sign_extend(instruction & IMM_MASK, 20);
  write_mem(read_mem(THE_CPU.registers[PC] + offset), THE_CPU.registers[sr]);
}
```

Loading We implement four mechanism of loading data into the registers.

- LOAD — Loads the data at the given memory address into ACC.
- LEA — Load effective address.
- LDR — Load register.
- LDI — Load indirect.

Their implementations follow:

```
// Loads the data at the given memory address into cpu's ACC register
void load(const dword instruction)
{
  dword dr = (instruction >> DR_SHIFT) & 0xF;
  sdword offset = sign_extend(instruction & IMM_MASK, 20);

  THE_CPU.registers[dr] = read_mem(THE_CPU.registers[PC] + offset);
  set_zero_flag(THE_CPU.registers[dr]);
}

// An address is computed by sign-extending bits [8:0] to 16 bits and adding this
// value to the incremented PC. This address is loaded into DR. The condition
// codes are set, based on whether the value loaded is negative, zero, or positive.
void load_effective_address(const dword instruction)
{
  dword dr = (instruction >> DR_SHIFT) & 0xF;
  sdword offset = sign_extend(instruction & IMM_MASK, 20);

  THE_CPU.registers[dr] = THE_CPU.registers[PC] + offset;
  set_zero_flag(THE_CPU.registers[dr]);
}

// An address is computed by sign-extending bits [5:0] to 16 bits and adding this
// value to the contents of the register specified by bits [8:6]. The contents of memory
// at this address are loaded into DR. The condition codes are set, based on whether
// the value loaded is negative, zero, or positive
void load_register(const dword instruction)
{
  dword dr = (instruction >> DR_SHIFT) & 0xF;
  dword br = (instruction >> SR1_SHIFT) & 0xF;
  sdword offset = sign_extend(instruction & REG_MASK, 16);

  THE_CPU.registers[dr] = read_mem(THE_CPU.registers[br] + offset);
  set_zero_flag(THE_CPU.registers[dr]);
}

// An address is computed by sign-extending bits [8:0] to 16 bits and adding this
// value to the incremented PC. What is stored in memory at this address is the
// address of the data to be loaded into DR. The condition codes are set, based on
// whether the value loaded is negative, zero, or positive
void load_indirect(const dword instruction)
```

```c
{
  dword dr = (instruction >> DR_SHIFT) & 0xF;
  sdword offset = sign_extend(instruction & IMM_MASK, 20);

  THE_CPU.registers[dr] = read_mem(read_mem(THE_CPU.registers[PC] + offset));

  set_zero_flag(THE_CPU.registers[dr]);
}
```

Execution The `execute_instruction` function dispatches a given instruction to one of the previously mentioned functions for execution.

```c
void execute_instruction(const dword op, const dword instruction) {
  switch (op) {
    case ADD: add(instruction); break;
    case SUB: sub(instruction); break;
    case MUL: mul(instruction); break;
    case DIV: divide(instruction); break;
    case AND: bit_and(instruction); break;
    case OR: bit_or(instruction); break;
    case XOR: bit_xor(instruction); break;
    case NOT: bit_not(instruction); break;
    case BRANCH: branch(instruction); break;
    case JUMP: jump(instruction); break;
    case JUMPR: jump_register(instruction); break;
    case JUMPZ: jump_zero(instruction); break;
    case STORE: store(instruction); break;
    case STRR: store_register(instruction); break;
    case STRI: store_indirect(instruction); break;
    case LOAD: load(instruction); break;
    case LEA: load_effective_address(instruction); break;
    case LDR: load_register(instruction); break;
    case LDI: load_indirect(instruction); break;
    case INTR: interrupt(instruction); break;
    default: report_invalid_opcode(op, instruction);
  }
}
```

Invalid opcodes are reported as an error and execution is terminated.

```c
static inline void report_invalid_opcode(const dword opcode, const dword instruction) {
  printf("ERROR: Invalid opcode %u (IR=0x%04X)\n", (unsigned)opcode,
            (unsigned)instruction);
    THE_CPU.registers[PC] = CPU_HALT;
}
```

3 Module 2: Memory Management System

3.1 Problem Statement

The objective of this module is to enhance the existing memory management system by implementing a hierarchical memory architecture consisting of main memory (RAM) and two levels of cache (L1 and L2). This design aims to simulate realistic memory behavior, where data is accessed through progressively slower layers of storage, improving efficiency through caching frequently used data. Additionally, a Memory Table will be introduced to track memory allocation and availability for each process, ensuring proper management of system resources. The module will support dynamic memory allocation and deallocation, allowing processes to request and release memory as needed using strategies such as First-Fit or Best-Fit. Through these implementations, the system will demonstrate efficient memory utilization, process-level memory tracking, and realistic cache behavior for multi-process and multithreaded environments.

3.2 Implementation

3.2.1 hierarchical Memory System

To simulate a hierarchical memory system, we implemented the RAM as an array of `word` and the L1 and L2 caches are structures:

```
typedef struct
{
  Entry* items;
  int front;
  int count;
  int size;
} Cache;
```

where `Entry` is a structure:

```
typedef struct
{
  word val;
  mem_addr addr;
} Entry;
```

The two main functions for this module is `read_mem` and `write_mem`. The former takes in as input a memory and returns the value at that address.

It first checks the L1 cache, then the L2 cache, then finally the RAM. It also updates the hit/miss stats for the different caches. The latter takes a memory address and a value and writes that value at the given memory address. These are the core operations used to fetch and store information from the CPU to the main memory and vice versa.

```
//return the value at the given memory address
word read_mem(const mem_addr addr)
{
  int index;
  index = cache_search(&L1, addr);
  if(index != EMPTY_ADDR)
  {
    //Cache hit at L1
    L1cache_hit++;
    return L1.items[index].val;
  }

  //cache miss at L1
  L1cache_miss++;

  index = cache_search(&L2, addr);
  if(index != EMPTY_ADDR)
  {
    //Cache hit at l2
    L2cache_hit++;
    word val = L2.items[index].val;
    //Update L1 cache to prevent future cache misses
    update_cache(&L1, addr, val);
    return val;
  }

  //Cache miss at L2
  L2cache_miss++;

  //Complete cache miss, so read RAM and update cache
  word val = RAM[addr];
  update_cache(&L1, addr, val);
  update_cache(&L2, addr, val);
  return val;
}
```

```
//write the given value to the given memory address.
void write_mem(const mem_addr addr, const word val)
{
  RAM[addr] = val;

  int index;

  //Update L1 Cache
  index = cache_search(&L1, addr);
  if(index != EMPTY_ADDR)
  {
    L1.items[index].val = val;
  }

  //Update L2 Cache
  index = cache_search(&L2, addr);
  if(index != EMPTY_ADDR)
  {
    L2.items[index].val = val;
  }
}
```

The helper functions `cache_search` and `update_cache` are used to manage interfacing with the caches. The former checks if the data can be found in the cache and if so returns the index. The latter inserts data into the cache following the interface of a double ended queue.

```
//find the address of the value if it exists in cache
int cache_search(Cache* cache, const mem_addr addr)
{
  for(int i = 0; i < cache->size; i++)
  {
    //the address we want was found in cache
    //so return the index of that address
    if(cache->items[i].addr == addr)
    {
      return i;
    }
  }
  //address not found so return signifier
  return EMPTY_ADDR;
}
```

```c
//Update the given cache in case of misses
void update_cache(Cache* cache, const mem_addr addr, const word val)
{
  //calculate where in the cache to store the value
  int index = (cache->front + cache->count) \% cache->size;
  cache->items[index].addr = addr;
  cache->items[index].val = val;

  //update the size and count of the cache
  if(cache->count < cache->size)
  {
    //cache isn't full so we can just put the new
    //value in the next index in the cache
    cache->count++;
  }
  else
  {
    //cache is full, so loop around to put the new
    //value at the front
    cache->front = (cache->front + 1) % cache->size;
  }
}
```

3.2.2 Memory Table

We implement our Memory Table as a structure containing an array of memory blocks and a count. Each memory block contains the ID of the process it belongs to, the starting address in memory, the ending address in memory, and a flag for if that block is free.

```c
typedef struct {
  int pid;
  dword start_addr;
  dword end_addr;
  bool is_free;
} MemoryBlock;

typedef struct
{
  MemoryBlock *blocks;
  int block_count;
} MemoryTable;

void init_memtable(const int size)
{
  //make the first entry into the table one large free block of memory
  MEMORY_TABLE.blocks = (MemoryBlock*)malloc(sizeof(MemoryBlock) * size);
  MEMORY_TABLE.blocks[0].pid = NO_PID;
  MEMORY_TABLE.blocks[0].is_free = true;
  MEMORY_TABLE.blocks[0].start_addr = 0;
  MEMORY_TABLE.blocks[0].end_addr = (size - 1);

  MEMORY_TABLE.block_count = 1;
  printf("initialized memory table with size %d\n" , size);
}
```

3.2.3 Dynamic Memory Allocation and Deallocation

We use the bes fit method to dynamically allocate memory.

```c
// allocate memory for a specific process
// using the best fit method
dword mallocate(int pid, int size)
{
  int best_size = WRS + 1;
  int index = -1;

  for(int i = 0; i < MEMORY_TABLE.block_count; i++)
  {
    if (MEMORY_TABLE.blocks[i].is_free)
    {
      int mem_block_size = (MEMORY_TABLE.blocks[i].end_addr - MEMORY_TABLE.blocks[i].sta
      if(mem_block_size >= size && mem_block_size < best_size)
      {
        best_size = mem_block_size;
        index = i;
      }
    }
  }

  //No memory free so ...idk
  if(index == -1)
  {
    printf("Could not fullfill process(PID %d)'s request for a (%d byte) chunk of memory
    return -1;
  }

  //Modify the free space to house our process
  //@david Boo! spooky mutation ~~ooooh~~
  MemoryBlock* best_fit = &MEMORY_TABLE.blocks[index];

  //save the old start and end addresses
  dword old_start_addr = best_fit->start_addr;
  dword old_end_addr = best_fit->end_addr;

  dword new_end_addr = (old_start_addr + size) - 1;

  //give the process the space
  best_fit->pid = pid;
  best_fit->is_free = false;
  best_fit->end_addr = new_end_addr;

  //cut down the size of the block
  //to free up unused space
  if(new_end_addr < old_end_addr)
  {
    //shift all the blocks to the right to make room
    for(int i = MEMORY_TABLE.block_count; i > index + 1; i--)
    {
      MEMORY_TABLE.blocks[i] = MEMORY_TABLE.blocks[i - 1];
    }

    MEMORY_TABLE.blocks[index +1].pid = NO_PID;
    MEMORY_TABLE.blocks[index +1].is_free = true;
    MEMORY_TABLE.blocks[index +1].start_addr = new_end_addr + 1;
    MEMORY_TABLE.blocks[index +1].end_addr = old_end_addr;

    MEMORY_TABLE.block_count++;
  }

  printf("Process (PID %d) given (%d bytes) of memory from [%d --> %d]\n", pid, size, be
  return best_fit->start_addr;
}
```

```c
// free up the memory block associated with the process
void liberate(int pid)
{
  int index = 0;
  for(index = 0; index < MEMORY_TABLE.block_count; index++)
  {
    if(MEMORY_TABLE.blocks[index].pid == pid && !MEMORY_TABLE.blocks[index].is_free)
    {
      MEMORY_TABLE.blocks[index].pid = NO_PID;
      MEMORY_TABLE.blocks[index].is_free = true;
      printf("Freed (PID %d) at memory [%d --> %d]\n", pid, MEMORY_TABLE.blocks[index].s
      break;
    }
  }
  //if the pid was not found
  if(index == MEMORY_TABLE.block_count){return;}

  //merge newly freed block with the previous block if it's also free
  if(index > 0 && MEMORY_TABLE.blocks[index - 1].is_free)
  {
    MEMORY_TABLE.blocks[index -1].end_addr = MEMORY_TABLE.blocks[index].end_addr;
    //shift everything left to clean the gap
    for(int i = index; i < MEMORY_TABLE.block_count - 1; i++)
    {
      MEMORY_TABLE.blocks[i] = MEMORY_TABLE.blocks[i + 1];
    }
    MEMORY_TABLE.block_count--;
    index--;
  }

  //merge with the next memory block if it's also free
  if(index < MEMORY_TABLE.block_count - 1 && MEMORY_TABLE.blocks[index + 1].is_free)
  {
    MEMORY_TABLE.blocks[index].end_addr = MEMORY_TABLE.blocks[index + 1].end_addr;
    //shift right to clean the gap
    for(int i = index + 1; i < MEMORY_TABLE.block_count - 1; i++)
    {
      MEMORY_TABLE.blocks[i] = MEMORY_TABLE.blocks[i + 1];
    }
    MEMORY_TABLE.block_count--;
  } }
```

4 Module 3: Process Scheduling and Multitasking

4.1 Problem Statement

The objective of this module is to implement process management through efficient CPU scheduling and multi-process handling. The system must allow multiple processes to share the CPU by integrating the previously developed fetch-decode-execute cycle with a scheduling mechanism. Each process will be represented by a Process Control Block (PCB) containing its unique identifier, register values, program counter, accumulator, and current process state. A scheduling algorithm—either Round-Robin or Priority-Based Scheduling—will be implemented to determine the order and duration of process execution. The module must also support context switching, ensuring that a process's state is properly saved and restored when switching between processes. Finally, the simulator should demonstrate realistic multi-process behavior by handling CPU-bound, I/O-bound, and mixed processes, ensuring fair and efficient CPU utilization according to the selected scheduling strategy.

4.2 Implementation

Representation We represent our processes as a structure containing a process ID, state, and a CPU that contains the registers and flags of that process. A process state is either `READY`, `RUNNING`, `WAITING`, or `FINISHED`. We store our processes in a `PROCESS_TABLE`.

```
#define MAX_PROCESSES 5
#define PROCESS_TIME 5

typedef enum
{
  READY,
  RUNNING,
  WAITING,
  FINISHED
} ProcessState;

typedef struct
{
  int pid;
  ProcessState state;
  Cpu cpu_state;
} Process;

//Array to keep track of all the processes
extern Process PROCESS_TABLE[MAX_PROCESSES];
```

Implementation Initially, our process table is empty. As processes are added, we use a round-robin scheduler to schedule execution and a context switcher to change which process is being executed.

```
//the index into the process table of the
//current process being run
int current_process = 0;

//the number of active processes, either:
int num_processes = 0;

//Get the next process
//(round robin style so its just the next index of current)
#define next_process ((current_process + 1) % num_processes)

Process PROCESS_TABLE[MAX_PROCESSES];

void init_processes()
{
  for(int i = 0; i < MAX_PROCESSES; i++)
  {
    PROCESS_TABLE[i].pid = i;
    PROCESS_TABLE[i].cpu_state = THE_CPU;
    PROCESS_TABLE[i].state = READY;
  }
  num_processes = MAX_PROCESSES;
  current_process = 0;
  PROCESS_TABLE[0].state = RUNNING;
}

void context_switch(int current, int next)
{
  Process* curr = &PROCESS_TABLE[current];
  Process* nxt = &PROCESS_TABLE[next];
  //save current's state
  curr->cpu_state = THE_CPU;
  curr->state = READY;

  //start the next process
  THE_CPU = nxt->cpu_state;
  nxt->state = RUNNING;

  printf("Switched from process (PID: %d) to process (PID: %d)", curr->pid, nxt->pid);
}

void scheduler()
{
  while(true)
  {
    Process* proc = &PROCESS_TABLE[current_process];
    if(proc->state == FINISHED)
    {
      current_process = next_process;
      continue;
    }

    proc->state = RUNNING;
    THE_CPU = proc->cpu_state;

    for(int i = 0; i < PROCESS_TIME; i++)
    {
      fetch();
      execute();
      //check_for_interrupt();

      if(proc->cpu_state.registers[PC] == CPU_HALT)
        break;

    }

    proc->cpu_state = THE_CPU;

    context_switch(current_process, next_process);
    current_process = next_process;
    bool all_done = true;
    for (int j = 0; j < num_processes; j++)
    {
      if (PROCESS_TABLE[j].state != FINISHED)
      {
        all_done = false;
        break;
      }
    }
    if (all_done) break;
  }
}
```

5 Module 4: Interrupts

5.1 Problem Statement

The objective of this module is to develop an advanced interrupt handling and dispatching system that enables the CPU to manage asynchronous events and process transitions efficiently. The system must allow the CPU to pause its current execution in response to various interrupts—such as timer, I/O, system call, trap, and priority interrupts—and appropriately service each event before resuming or switching processes. An Interrupt Vector Table (IVT) will be implemented to map interrupt types to their corresponding handlers, enabling fast and accurate interrupt resolution. Each handler will process its specific event, update process states, and invoke the dispatcher when necessary. The dispatcher will perform context switching by saving the current process state and restoring the next process's context based on the selected scheduling algorithm (Round-Robin or Priority-Based). This module ensures smooth and controlled transitions between processes, accurate interrupt servicing, and efficient CPU utilization in a multitasking environment.

5.2 Implementation

Interrupt Types

```c
typedef enum irq{
    SAY_HI = 0x1,
    SAY_GOODBYE,
    TIMER_INTR,
    IO_INTR,
    SYS_CALL_INTR,
    TRAP_INTR,
    PRIORITY_INTR,
    EOI, //end of interrupt, make sure this is the last in the list
} IRQ;
```

Interrupt Vector Table

Interrupt Handler

Dispatcher

Context Switching

We provide three types of interrupt opcodes `SAY_HI`, `SAY_GOODBYE`, and `EOI`.

```c
typedef enum irq{
    SAY_HI = 0x1,
    SAY_GOODBYE,
    EOI, //end of interrupt
} IRQ;
```

Each interrupt instance contains one of these opcodes and a priority with 0 being the most important.

```c
typedef struct {
    IRQ irq;
    int priority;
} Interrupt;
```

The interrupt controller is structured as an `InterruptHeap` which is a min-heap with constant time access to the highest priority interrupt and logarithmic insertion for new interrupts.

```c
typedef struct {
    Interrupt data[MAX_INTERRUPTS];
    int size;
} InterruptHeap;
```

Finally, we store CPU states in a stack which allows for the pausing and resumption of processes by storing the information necessary to restart them.

```c
typedef struct {
    Cpu* items;
    int SP;
} stack;
```

Interrupts are checked for every CPU cycle. If there is no interrupt then nothing happens. If there is an interrupt, its priority is checked against the potential current interrupt; if its priority is higher then it is immediately executed, if not it is added to the heap.

```
void check_for_interrupt() {
  if (!CPU.flags.INTERRUPT) return;
  if (INTERRUPTCONTROLLER.size == 0){
    //no more interrupts in que, so clear flag
    set_interrupt_flag(false);
    return;
  }

  Interrupt intrpt = next_interrupt();
  if(curr_intrrpt.irq == -1 || intrpt.priority < curr_intrrpt.priority){
    curr_intrrpt = intrpt;
    interrupt_handler(curr_intrrpt);
  }else{
    interrupt_handler(curr_intrrpt);
    add_interrupt(intrpt.irq, intrpt.priority);
  }

  //clear flag if heap is empty after handle
  if(INTERRUPTCONTROLLER.size == 0) set_interrupt_flag(false);
}
```

Interrupts are dispatched to their appropriate functions via the `interrupt_handler`:

```
void interrupt_handler(Interrupt intrpt) {
    //push the current CPU state to stack
    Cpu init_cpu_state = CPU;
    callstack.items[callstack.SP] = init_cpu_state;
    callstack.SP++;

    //decode the given interrupt and handle it
    switch(intrpt.irq) {
    case SAY_HI :
      printf("INTERRUPT: hello\n");
      set_interrupt_flag(false);
      reset_curr_interrupt();
      break;
    ...
    case EOI :
      set_interrupt_flag(false);
      reset_curr_interrupt();
      break;
    default:
      printf("ERROR: Invalid irq -> %u <-\n", (unsigned)intrpt.irq);
      CPU.PC = CPU_HALT;
      break;
    }

    //decrement the CPU stack
    callstack.SP--;
    //reset the CPU to it's original state
    CPU = callstack.items[callstack.SP];
    //increment the PC to start normal execution
    //CPU.PC++;
}
```

Handling Interrupts We use a min-heap to store our interrupts based on priority giving us logarithmic insertions and constant time removals.

```
static void swap(Interrupt* a, Interrupt* b)
{
  Interrupt tmp = *a;
  *a = *b;
  *b = tmp;
}

static int parent(int i) { return (i - 1) / 2; }
static int left(int i)   { return 2 * i + 1; }
static int right(int i)  { return 2 * i + 2; }

/* --------------------- Core Functions --------------------- */

void init_interrupt_controller(void)
{
  INTERRUPTCONTROLLER.size = 0;

  for (int i = 0; i < MAX_INTERRUPTS; i++) {
    INTERRUPTCONTROLLER.data[i].irq = EOI;
    INTERRUPTCONTROLLER.data[i].priority = 100000;
  }

  callstack.SP = 0;
  callstack.items = malloc(sizeof(Cpu) * CALLSTACK_SIZE);
  if (!callstack.items)
  {
```

```c
    fprintf(stderr, "Failed to allocate callstack\n");
    exit(EXIT_FAILURE);
  }

  curr_intrrpt.irq = EOI;
  curr_intrrpt.priority = 100000;

  printf("Initialized interrupt controller.\n");
}

// No more memory leaks yay :)
void free_interrupt_controller(void) {
    free(callstack.items);
    callstack.items = NULL;
}

// Add an interrupt to the heap (lower number = higher priority)
void add_interrupt(IRQ irq, int priority)
{
  if (INTERRUPTCONTROLLER.size >= MAX_INTERRUPTS)
  {
    fprintf(stderr, "Interrupt queue full!\n");
    return;
  }

  int i = INTERRUPTCONTROLLER.size++;
  INTERRUPTCONTROLLER.data[i].irq = irq;
  INTERRUPTCONTROLLER.data[i].priority = priority;


  while (i != 0 && INTERRUPTCONTROLLER.data[parent(i)].priority > INTERRUPTCONTROLLER.da
  {
    swap(&INTERRUPTCONTROLLER.data[i], &INTERRUPTCONTROLLER.data[parent(i)]);
    i = parent(i);
  }

  printf("[INTERRUPT] Queued IRQ %d (priority %d)\n", irq, priority);
}

/* Pop the highest-priority interrupt */
static Interrupt next_interrupt(void)
```

```
{
  if (INTERRUPTCONTROLLER.size ⩽ 0)
  {
  Interrupt none = { EOI, 100000 };
  return none;
  }

  Interrupt root = INTERRUPTCONTROLLER.data[0];
  INTERRUPTCONTROLLER.data[0] = INTERRUPTCONTROLLER.data[--INTERRUPTCONTROLLER.size];

  int i = 0;
  while (1)
  {
    int l = left(i), r = right(i), smallest = i;

    if (l < INTERRUPTCONTROLLER.size &&
            INTERRUPTCONTROLLER.data[l].priority < INTERRUPTCONTROLLER.data[smallest].pr
            smallest = l;

    if (r < INTERRUPTCONTROLLER.size &&
            INTERRUPTCONTROLLER.data[r].priority < INTERRUPTCONTROLLER.data[smallest].pr
            smallest = r;

    if (smallest ≠ i)
    {
      swap(&INTERRUPTCONTROLLER.data[i], &INTERRUPTCONTROLLER.data[smallest]);
      i = smallest;
    }
    else break;
  }

  return root;
}
```

6 Module 5: Multithreading or Forking

6.1 Problem Statement

The objective of this module is to extend the system to support concurrent execution through the use of multiple threads or forked processes. This enhancement will enable simultaneous handling of key system functions, including CPU operations, memory management, process scheduling, and interrupt handling. Each thread or process will be responsible for executing a specific subsystem, simulating real-world multitasking behavior. Threads will share a common memory space, while forked processes will operate independently with separate memory. The system must coordinate interactions between these concurrent tasks—such as context switching and shared resource management—to ensure efficient, synchronized execution across all modules.

6.2 Implementation

We use three volatile variables to keep track of thread progress.

```
// Thread synchronization flags
volatile int process1_done = 0;
volatile int process2_done = 0;
volatile int process3_done = 0;
```

We then create threads for timers and I/O along with three proceses.

```
pthread_t proc1_thread, proc2_thread, proc3_thread;
pthread_t timer_thread, io_thread;

// Launch interrupt generators
pthread_create(&timer_thread, NULL, timer_interrupt_thread, NULL);
pthread_create(&io_thread, NULL, io_interrupt_thread, NULL);

// Launch process threads
pthread_create(&proc1_thread, NULL, process1_hello_professor, NULL);
pthread_create(&proc2_thread, NULL, process2_arithmetic, NULL);
pthread_create(&proc3_thread, NULL, process3_dma_transfer, NULL);

// Wait for all processes to complete
pthread_join(proc1_thread, NULL);
pthread_join(proc2_thread, NULL);
pthread_join(proc3_thread, NULL);
```

7 Testing and Debugging

This test program evaluates the complete functionality and integration of the simulated computer system, including CPU execution, memory management, interrupt handling, DMA transfers, and multithreading. It launches three concurrent processes: one that prints a message using CPU interrupts, another that performs intensive arithmetic operations and memory storage, and a third that simulates DMA data transfers from SSD and HDD into RAM. Meanwhile, separate threads generate timer and I/O interrupts to test asynchronous event handling and process coordination. The system also runs a demo CPU program to verify correct instruction execution, memory access, and cache behavior. Overall, this test assesses whether all system components—CPU, memory hierarchy, interrupt controller, DMA, and multithreaded process handling—work together smoothly to emulate a functioning multitasking operating system environment.

```c
// Initialize test environment
void init_test_environment() {
    init_cache(&L1, L1CACHE_SIZE);
    init_cache(&L2, L2CACHE_SIZE);
    init_ram(RAM_SIZE);
    init_HDD(HDD_SIZE);
    init_SSD(SSD_SIZE);
    init_interrupt_controller();
    init_cpu(&THE_CPU);
    init_processes();
}

// Cleanup test environment
void cleanup_test_environment() {
    free_interrupt_controller();
    free(L1.items);
    free(L2.items);
    free(RAM);
    free(HDD);
    free(SSD);
}

// ============ CPU TESTS ============
void test_cpu_initialization() {
    TEST_START("CPU Initialization");
    Cpu test_cpu;
    init_cpu(&test_cpu);

    TEST_ASSERT(test_cpu.registers[PC] == MEM_START, "PC initialized to MEM_START");
    TEST_ASSERT(test_cpu.registers[IR] == EMPTY_REG, "IR initialized to EMPTY_REG");
    TEST_ASSERT(test_cpu.registers[FLAG] == F_ZERO, "FLAG initialized to F_ZERO");
    TEST_ASSERT(test_cpu.registers[ACC] == 0, "ACC initialized to 0");
}

void test_add_instruction() {
    TEST_START("ADD Instruction");

    // Test immediate mode ADD: ADD AX, BX, #5
    // Opcode: ADD (0x0), DR: AX (0), SR1: BX (1), IMM: 1, Operand: 5
    THE_CPU.registers[BX] = 10;
    dword instruction = (ADD << 24) | (AX << 20) | (BX << 16) | (1 << 12) | 5;

    execute_instruction(ADD, instruction);

    TEST_ASSERT(THE_CPU.registers[AX] == 15, "AX = BX + 5 = 15");
    TEST_ASSERT(!(THE_CPU.registers[FLAG] & F_ZERO), "Zero flag not set for non-zero res
}

void test_sub_instruction() {
    TEST_START("SUB Instruction");

    THE_CPU.registers[BX] = 20;
    dword instruction = (SUB << 24) | (AX << 20) | (BX << 16) | (1 << 12) | 5;

    execute_instruction(SUB, instruction);

    TEST_ASSERT(THE_CPU.registers[AX] == 15, "AX = BX - 5 = 15");
}

void test_mul_instruction() {
    TEST_START("MUL Instruction");

    THE_CPU.registers[BX] = 10;
    dword instruction = (MUL << 24) | (AX << 20) | (BX << 16) | (1 << 12) | 5;

    execute_instruction(MUL, instruction);
    TEST_ASSERT(THE_CPU.registers[AX] == 50, "AX = BX * 5 = 50");
}

void test_div_instruction() {
    TEST_START("DIV Instruction");

    THE_CPU.registers[BX] = 100;
    dword instruction = (DIV << 24) | (AX << 20) | (BX << 16) | (1 << 12) | 5;

    execute_instruction(DIV, instruction);

    TEST_ASSERT(THE_CPU.registers[AX] == 20, "AX = BX / 5 = 20");
}

void test_bitwise_operations() {
    TEST_START("Bitwise Operations");

    // AND test
    THE_CPU.registers[BX] = 0xFF;
    dword instruction = (AND << 24) | (AX << 20) | (BX << 16) | (1 << 12) | 0x0F;
    execute_instruction(AND, instruction);
    TEST_ASSERT(THE_CPU.registers[AX] == 0x0F, "AND operation works");

    // OR test
    THE_CPU.registers[BX] = 0xF0;
    instruction = (OR << 24) | (AX << 20) | (BX << 16) | (1 << 12) | 0x0F;
    execute_instruction(OR, instruction);
    TEST_ASSERT(THE_CPU.registers[AX] == 0xFF, "OR operation works");

    // XOR test
    THE_CPU.registers[BX] = 0xFF;
    instruction = (XOR << 24) | (AX << 20) | (BX << 16) | (1 << 12) | 0xFF;
    execute_instruction(XOR, instruction);
    TEST_ASSERT(THE_CPU.registers[AX] == 0, "XOR operation works");
}

// ============ MEMORY TESTS ============
void test_memory_read_write() {
    TEST_START("Memory Read/Write");
```

```c
    dword test_addr = 100;
    dword test_val = 0xDEADBEEF;

    write_mem(test_addr, test_val);
    dword read_val = read_mem(test_addr);

    TEST_ASSERT(read_val == test_val, "Memory read/write consistency");
}

void test_cache_hierarchy() {
    TEST_START("Cache Hierarchy");

    // Reset cache stats
    extern int L1cache_hit, L1cache_miss, L2cache_hit, L2cache_miss;
    L1cache_hit = L1cache_miss = L2cache_hit = L2cache_miss = 0;

    dword addr = 50;
    dword val = 0x12345678;

    write_mem(addr, val);

    // First read should miss all caches
    read_mem(addr);
    TEST_ASSERT(L1cache_miss == 1, "L1 cache miss on first read");
    TEST_ASSERT(L2cache_miss == 1, "L2 cache miss on first read");

    // Second read should hit L1
    read_mem(addr);
    TEST_ASSERT(L1cache_hit == 1, "L1 cache hit on second read");
}

void test_memory_allocation() {
    TEST_START("Memory Allocation");

    int pid1 = 1;
    int size1 = 100;

    dword addr1 = mallocate(pid1, size1);
    TEST_ASSERT(addr1 != (dword)-1, "Memory allocation successful");

    int pid2 = 2;
    int size2 = 50;
    dword addr2 = mallocate(pid2, size2);
    TEST_ASSERT(addr2 != (dword)-1, "Second allocation successful");
    TEST_ASSERT(addr2 != addr1, "Allocations don't overlap");

    liberate(pid1);
    liberate(pid2);
}

void test_memory_deallocation() {
    TEST_START("Memory Deallocation");

    int pid = 3;
    dword addr = mallocate(pid, 100);

    liberate(pid);

    // Try to allocate again in freed space
    dword addr2 = mallocate(4, 50);
    TEST_ASSERT(addr2 == addr, "Freed memory is reusable");

    liberate(4);
}

// ============ INTERRUPT TESTS ============
void test_interrupt_queue() {
    TEST_START("Interrupt Queueing");

    add_interrupt(SAY_HI, 5);
    add_interrupt(SAY_GOODBYE, 3);
    add_interrupt(SAY_HI, 7);

    TEST_ASSERT(1, "Interrupts queued without error");

    // Process interrupts
    check_for_interrupt();
    check_for_interrupt();
    check_for_interrupt();
}

void test_interrupt_priority() {
    TEST_START("Interrupt Priority");

    add_interrupt(SAY_HI, 10);
    add_interrupt(SAY_GOODBYE, 1);  // Higher priority (lower number)

    // Should handle SAY_GOODBYE first due to priority
    check_for_interrupt();
    TEST_ASSERT(1, "Higher priority interrupt handled first");
}

// ============ PROCESS TESTS ============
void test_process_creation() {
    TEST_START("Process Creation");

    TEST_ASSERT(PROCESS_TABLE[0].state == RUNNING, "First process is RUNNING");
    TEST_ASSERT(PROCESS_TABLE[1].state == READY, "Other processes are READY");
}

void test_context_switching() {
    TEST_START("Context Switching");

    // Set up different register values for processes
    PROCESS_TABLE[0].cpu_state.registers[AX] = 100;
    PROCESS_TABLE[1].cpu_state.registers[AX] = 200;

    THE_CPU.registers[AX] = 100;

    context_switch(0, 1);

    TEST_ASSERT(THE_CPU.registers[AX] == 200, "Context switch loads new process state");
    TEST_ASSERT(PROCESS_TABLE[0].state == READY, "Previous process marked READY");
    TEST_ASSERT(PROCESS_TABLE[1].state == RUNNING, "New process marked RUNNING");
}

// ============ DMA TESTS ============
void test_dma_transfer() {
    TEST_START("DMA Transfer");

    dword source[5] = {1, 2, 3, 4, 5};
    dword destination[5] = {0, 0, 0, 0, 0};
```

```c
    initiateDMA(source, destination, 5);

    int match = 1;
    for (int i = 0; i < 5; i++) {
        if (source[i] != destination[i]) {
            match = 0;
            break;
        }
    }

    TEST_ASSERT(match, "DMA transfer copies data correctly");
}

// ============ INSTRUCTION SET TESTS ============
void test_load_store() {
    TEST_START("LOAD/STORE Instructions");

    dword addr = 10;
    dword value = 0xABCDEF12;

    // Write value to memory
    write_mem(addr, value);

    // LOAD instruction: LOAD AX, PC+offset
    THE_CPU.registers[PC] = 0;
    dword load_instr = (LOAD << 24) | (AX << 20) | addr;
    execute_instruction(LOAD, load_instr);

    TEST_ASSERT(THE_CPU.registers[AX] == value, "LOAD retrieves value from memory");

    // STORE instruction
    THE_CPU.registers[BX] = 0xCAFEBABE;
    THE_CPU.registers[PC] = 0;
    dword store_instr = (STORE << 24) | (BX << 20) | 20;
    execute_instruction(STORE, store_instr);

    TEST_ASSERT(read_mem(20) == 0xCAFEBABE, "STORE writes value to memory");
}

void test_jump_instructions() {
    TEST_START("JUMP Instructions");
    // JUMP instruction
    THE_CPU.registers[PC] = 10;
    THE_CPU.registers[BX] = 50;
    dword jump_instr = (JUMP << 24) | (BX << 16);
    execute_instruction(JUMP, jump_instr);

    TEST_ASSERT(THE_CPU.registers[PC] == 50, "JUMP sets PC to register value");

    // JUMPZ instruction with zero flag set
    THE_CPU.registers[PC] = 10;
    THE_CPU.registers[FLAG] = F_ZERO;
    THE_CPU.registers[CX] = 100;
    dword jumpz_instr = (JUMPZ << 24) | (CX << 16);
    execute_instruction(JUMPZ, jumpz_instr);

    TEST_ASSERT(THE_CPU.registers[PC] == 100, "JUMPZ jumps when zero flag is set");
}

// ============ FLAG TESTS ============
void test_flag_operations() {
    TEST_START("Flag Operations");

    // Test zero flag
    THE_CPU.registers[BX] = 5;
    dword sub_instr = (SUB << 24) | (AX << 20) | (BX << 16) | (1 << 12) | 5;
    execute_instruction(SUB, sub_instr);
    TEST_ASSERT(THE_CPU.registers[FLAG] & F_ZERO, "Zero flag set when result is zero");

    // Test overflow flag
    THE_CPU.registers[BX] = 0x7FFFFFFF;  // Max positive int
    dword add_instr = (ADD << 24) | (AX << 20) | (BX << 16) | (1 << 12) | 1;
    execute_instruction(ADD, add_instr);
    TEST_ASSERT(THE_CPU.registers[FLAG] & F_OVFLW, "Overflow flag set on overflow");
}
```

8 The Simulation

This program serves as an integrated system test for an advanced operating system simulator, demonstrating the interaction between multiple subsystems including the CPU, memory hierarchy, interrupt controller, DMA, and process management. It initializes all system components and launches three concurrent threads (processes) — one prints "Hello, Professor" using CPU interrupts, another performs repeated arithmetic calculations and memory operations, and a third simulates DMA data transfers from SSD and HDD into RAM. At the same time, separate timer and I/O interrupt threads generate asynchronous events to test interrupt handling and CPU responsiveness. After these processes finish, a demo CPU program is loaded into memory and executed to verify instruction execution, arithmetic logic, and memory storage. Finally, the program outputs memory contents, CPU register states, and cache statistics before freeing all resources. Overall, it tests the system's ability to handle multithreading, interrupt-driven execution, DMA transfers, and coordinated CPU-memory operations within a simulated multitasking environment.

```c
// Thread synchronization flags
volatile int process1_done = 0;
volatile int process2_done = 0;
volatile int process3_done = 0;

// ============ PROCESS 1: Print "Hello, Professor" using CPU Interrupts ============
void* process1_hello_professor(void* arg) {
    printf("\n[Process 1 Started] - Printing 'Hello, Professor' using CPU interrupts\n")

    // Allocate memory for this process
    dword proc1_mem = mallocate(1, 100);

    // Store the string "Hello, Professor!\n" in memory (one char per dword)
    const char* message = "Hello, Professor!\n";
    int msg_len = 0;
    for (int i = 0; message[i] != '\0'; i++) {
        write_mem(proc1_mem + i, (dword)message[i]);
        msg_len++;
    }
    write_mem(proc1_mem + msg_len, 0); // Null terminator

    printf("[Process 1] Loaded message into memory at address 0x%X\n", proc1_mem);
    // Print the message 5 times using CPU interrupts
    for (int i = 0; i < 5; i++) {
        printf("[Process 1] Iteration %d/5: ", i + 1);

        // Set BX register to point to our string in memory
        THE_CPU.registers[BX] = proc1_mem;

        // Create and execute INT_PUTS interrupt instruction
        dword puts_instruction = (INTR << 24) | INT_PUTS;
        execute_instruction(INTR, puts_instruction);

        usleep(500000); // 0.5 seconds delay
    }

    // Store a completion marker in memory
    write_mem(proc1_mem + 50, 0xC0FFEE);
    printf("[Process 1] Stored completion marker at address 0x%X\n", proc1_mem + 50);

    // Free memory
    liberate(1);

    process1_done = 1;
    printf("[Process 1 Completed]\n\n");
    return NULL;
}

// ============ PROCESS 2: Complex Arithmetic Operations ============
void* process2_arithmetic(void* arg) {
    printf("\n[Process 2 Started] - Performing arithmetic operations\n");
    printf("[Process 2] Goal: Add to ACC until > 1,000,000, divide by 100, repeat 10 tim

    // Allocate memory for this process
    dword proc2_mem = mallocate(2, 100);

    // Create a local CPU state for this process
    Cpu local_cpu;
    local_cpu.registers[ACC] = 0;
    local_cpu.registers[AX] = 1000000; // Target value
    local_cpu.registers[BX] = 100;     // Divisor
    local_cpu.registers[CX] = 0;       // Iteration counter
```

```c
printf("[Process 2] Starting with ACC = %u\n", local_cpu.registers[ACC]);

// Repeat 10 times
for (int iteration = 0; iteration < 10; iteration++) {
    printf("[Process 2] === Iteration %d/10 ===\n", iteration + 1);

    // Add to ACC until it's greater than 1,000,000
    int add_count = 0;
    while (local_cpu.registers[ACC] <= local_cpu.registers[AX]) {
        local_cpu.registers[ACC] += 12345; // Add increments
        add_count++;
    }

    printf("[Process 2] Added %d times, ACC = %u\n", add_count, local_cpu.registers[

    // Divide by 100
    local_cpu.registers[ACC] /= local_cpu.registers[BX];
    printf("[Process 2] After division by 100: ACC = %u\n", local_cpu.registers[ACC]

    usleep(300000); // 0.3 seconds delay
}

// Multiply final result by 2
local_cpu.registers[ACC] *= 2;
printf("\n[Process 2] Final ACC value after multiplying by 2: %u\n", local_cpu.regis

// Store the result in memory
dword storage_addr = proc2_mem + 10;
write_mem(storage_addr, local_cpu.registers[ACC]);
printf("[Process 2] Stored final result %u at memory address 0x%X\n",
       local_cpu.registers[ACC], storage_addr);

// Verify the stored value
dword verify = read_mem(storage_addr);
printf("[Process 2] Verification: Read back value %u from memory\n", verify);

// Free memory
liberate(2);

process2_done = 1;
printf("[Process 2 Completed]\n\n");
```

```c
    return NULL;
}

// ============ PROCESS 3: DMA Transfer from SSD/HDD ============
void* process3_dma_transfer(void* arg) {
    printf("\n[Process 3 Started] - Waiting for DMA transfer from SSD/HDD\n");

    // Allocate memory for this process
    dword proc3_mem = mallocate(3, 200);

    printf("[Process 3] Simulating data on SSD and HDD...\n");

    // Prepare data on SSD
    for (int i = 0; i < 10; i++) {
        SSD[i] = 0xAA00 + i;
    }
    printf("[Process 3] Prepared 10 words on SSD\n");

    // Prepare data on HDD
    for (int i = 0; i < 15; i++) {
        HDD[i] = 0xBB00 + i;
    }
    printf("[Process 3] Prepared 15 words on HDD\n");

    // Simulate waiting for I/O
    printf("[Process 3] Waiting for I/O operations...\n");
    usleep(1000000); // 1 second delay

    // DMA Transfer from SSD to RAM
    printf("[Process 3] Initiating DMA transfer from SSD to RAM...\n");
    initiateDMA(SSD, &RAM[proc3_mem], 10);
    printf("[Process 3] DMA transfer from SSD complete\n");

    // Verify SSD transfer
    printf("[Process 3] Verifying SSD data in RAM:\n");
    for (int i = 0; i < 10; i++) {
        printf("  RAM[%d] = 0x%X (expected 0x%X)\n",
               proc3_mem + i, RAM[proc3_mem + i], 0xAA00 + i);
    }

    usleep(500000); // 0.5 seconds delay
    // DMA Transfer from HDD to RAM
    printf("[Process 3] Initiating DMA transfer from HDD to RAM...\n");
    initiateDMA(HDD, &RAM[proc3_mem + 20], 15);
    printf("[Process 3] DMA transfer from HDD complete\n");

    // Verify HDD transfer
    printf("[Process 3] Verifying HDD data in RAM:\n");
    for (int i = 0; i < 15; i++) {
        printf("  RAM[%d] = 0x%X (expected 0x%X)\n",
               proc3_mem + 20 + i, RAM[proc3_mem + 20 + i], 0xBB00 + i);
    }

    // Process the transferred data
    dword sum = 0;
    for (int i = 0; i < 10; i++) {
        sum += RAM[proc3_mem + i];
    }
    printf("[Process 3] Sum of SSD data: 0x%X\n", sum);

    sum = 0;
    for (int i = 0; i < 15; i++) {
        sum += RAM[proc3_mem + 20 + i];
    }
    printf("[Process 3] Sum of HDD data: 0x%X\n", sum);

    // Free memory
    liberate(3);

    process3_done = 1;
    printf("[Process 3 Completed]\n\n");
    return NULL;
}

// ============ INTERRUPT GENERATORS ============
void* timer_interrupt_thread(void* arg) {
    int count = 0;
    while (!process1_done || !process2_done || !process3_done) {
        sleep(2);
        add_interrupt(SAY_HI, 5);
        count++;
        if (count >= 3) break; // Limit interrupts
    }
    return NULL;
}

void* io_interrupt_thread(void* arg) {
    int count = 0;
    while (!process1_done || !process2_done || !process3_done) {
        sleep(3);
        add_interrupt(SAY_GOODBYE, 3);
        count++;
        if (count >= 2) break; // Limit interrupts
    }
    return NULL;
}

// ============ DEMO CPU PROGRAM ============
void load_demo_program() {
    printf("\n[System] Loading demo CPU program into memory...\n");

    // Simple program that demonstrates the instruction set
    int addr = 0;

    // Initialize some values in memory
    write_mem(0x100, 42);
    write_mem(0x101, 10);

    // Program: Load, Add, Multiply, Store
    // LOAD AX, 0x100
    RAM[addr++] = (LOAD << 24) | (AX << 20) | 0x100;

    // ADD AX, AX, #8 (immediate)
    RAM[addr++] = (ADD << 24) | (AX << 20) | (AX << 16) | (1 << 12) | 8;

    // MUL BX, AX, #2 (immediate)
    RAM[addr++] = (MUL << 24) | (BX << 20) | (AX << 16) | (1 << 12) | 2;

    // STORE BX, 0x102
    RAM[addr++] = (STORE << 24) | (BX << 20) | 0x102;

    // LOAD CX, 0x101
    RAM[addr++] = (LOAD << 24) | (CX << 20) | 0x101;

    // SUB DX, BX, CX
    RAM[addr++] = (SUB << 24) | (DX << 20) | (BX << 16) | (CX);

    // STORE DX, 0x103
    RAM[addr++] = (STORE << 24) | (DX << 20) | 0x103;

    // HALT instruction - use INT_HALT interrupt
    RAM[addr++] = (INTR << 24) | INT_HALT;

    printf("[System] Demo program loaded (%d instructions)\n\n", addr);
}

// ============ MAIN ============
int main() {
    printf("\n");
    printf("╔══════════════════════════════════════════════════════════════╗\n");
    printf("║        Advanced Operating System Simulator Demo              ║\n");
    printf("║              Project 2 - Fall 2025                           ║\n");
    printf("╚══════════════════════════════════════════════════════════════╝\n");

    // ========== INITIALIZATION ==========
    printf("\n[INITIALIZATION PHASE]\n");
    printf("═══════════════════════\n");

    init_cache(&L1, L1CACHE_SIZE);
    init_cache(&L2, L2CACHE_SIZE);
    init_ram(RAM_SIZE);
    init_HDD(HDD_SIZE);
    init_SSD(SSD_SIZE);
    init_interrupt_controller();
    init_cpu(&THE_CPU);
    init_processes();

    printf("\n✓ All systems initialized successfully\n");

    // ========== LOAD DEMO PROGRAM ==========
    load_demo_program();

    // ========== CREATE PROCESS THREADS ==========
    printf("\n[PROCESS EXECUTION PHASE]\n");
    printf("===========================\n");
    printf("\nStarting 3 concurrent processes...\n");

    pthread_t proc1_thread, proc2_thread, proc3_thread;
    pthread_t timer_thread, io_thread;

    // Launch interrupt generators
    pthread_create(&timer_thread, NULL, timer_interrupt_thread, NULL);
    pthread_create(&io_thread, NULL, io_interrupt_thread, NULL);

    // Launch process threads
    pthread_create(&proc1_thread, NULL, process1_hello_professor, NULL);
    pthread_create(&proc2_thread, NULL, process2_arithmetic, NULL);
    pthread_create(&proc3_thread, NULL, process3_dma_transfer, NULL);

    // Wait for all processes to complete
    pthread_join(proc1_thread, NULL);
    pthread_join(proc2_thread, NULL);
    pthread_join(proc3_thread, NULL);

    printf("\n[CPU EXECUTION PHASE]\n");
    printf("=======================\n");
    printf("\nExecuting loaded CPU program...\n\n");

    // Run the CPU with the loaded program
    cpu_run(7);

    // ========== RESULTS AND STATISTICS ==========
    printf("\n[RESULTS AND STATISTICS]\n");
    printf("===========================\n");

    printf("\nFinal Memory Contents:\n");
    printf("  Address 0x100: 0x%X (initial value)\n", read_mem(0x100));
    printf("  Address 0x101: 0x%X (initial value)\n", read_mem(0x101));
    printf("  Address 0x102: 0x%X (result of computation)\n", read_mem(0x102));
    printf("  Address 0x103: 0x%X (result of subtraction)\n", read_mem(0x103));

    print_cache_stats();

    printf("\nCPU Final State:\n");
```

```c
    printf("  AX: 0x%X\n", THE_CPU.registers[AX]);
    printf("  BX: 0x%X\n", THE_CPU.registers[BX]);
    printf("  CX: 0x%X\n", THE_CPU.registers[CX]);
    printf("  DX: 0x%X\n", THE_CPU.registers[DX]);

    // ========== CLEANUP ==========
    printf("\n[CLEANUP PHASE]\n");
    printf("═══════════════\n");

    // Cancel interrupt threads
    pthread_cancel(timer_thread);
    pthread_cancel(io_thread);
    pthread_join(timer_thread, NULL);
    pthread_join(io_thread, NULL);

    // Free allocated resources
    free_interrupt_controller();
    free(L1.items);
    free(L2.items);
    free(RAM);
    free(HDD);
    free(SSD);

    printf("\n✓ All resources freed successfully\n");

    printf("\n");
    printf("╔══════════════════════════════════════════════════════════════╗\n");
    printf("║                   Demo Completed Successfully                ║\n");
    printf("╚══════════════════════════════════════════════════════════════╝\n");
    printf("\n");
    printf("Press Enter to exit...");
    getchar();

    return 0;
}
```

And the output

```
╔════════════════════════════════════════════════════════════╗
║       Advanced Operating System Simulator Demo             ║
║             Project 2 - Fall 2025                          ║
╚════════════════════════════════════════════════════════════╝
```

```
[INITIALIZATION PHASE]
═══════════════════════
Initialized cache at -> '0x55d0da1a5160' <- with size: 5
Initialized cache at -> '0x55d0da1a5140' <- with size: 20
initialized ram with size: 500
initialized memory table with size 500
Initialized interrupt controller.
Initialized the cpu!
CPU STATE
  AX: 0x0
  BX: 0x0
  CX: 0x0
  DX: 0x0
PC:  0
ACC: 0
IR:  FFFFFFFF
FLAGS:
  ZERO:     1
  OVERFLOW: 0
  CARRY:    0



✓ All systems initialized successfully

[System] Loading demo CPU program into memory...
[System] Demo program loaded (8 instructions)


[PROCESS EXECUTION PHASE]
═══════════════════════════

Starting 3 concurrent processes...

[Process 1 Started] - Printing 'Hello, Professor' using CPU interrupts
Process (PID 1) given (100 bytes) of memory from [0 --> 99]
[Process 1] Loaded message into memory at address 0x0
[Process 1] Iteration 1/5: Hello, Professor!

[Process 2 Started] - Performing arithmetic operations
```

```
[Process 2] Goal: Add to ACC until > 1,000,000, divide by 100, repeat 10 times, then mul

Process (PID 2) given (100 bytes) of memory from [100 --> 199]
[Process 2] Starting with ACC = 0
[Process 2] === Iteration 1/10 ===
[Process 2] Added 82 times, ACC = 1012290
[Process 2] After division by 100: ACC = 10122

[Process 3 Started] - Waiting for DMA transfer from SSD/HDD
Process (PID 3) given (200 bytes) of memory from [200 --> 399]
[Process 3] Simulating data on SSD and HDD...
[Process 3] Prepared 10 words on SSD
[Process 3] Prepared 15 words on HDD
[Process 3] Waiting for I/O operations...
[Process 2] === Iteration 2/10 ===
[Process 2] Added 81 times, ACC = 1010067
[Process 2] After division by 100: ACC = 10100
[Process 1] Iteration 2/5: Hello, Professor!
[Process 2] === Iteration 3/10 ===
[Process 2] Added 81 times, ACC = 1010045
[Process 2] After division by 100: ACC = 10100
[Process 2] === Iteration 4/10 ===
[Process 2] Added 81 times, ACC = 1010045
[Process 2] After division by 100: ACC = 10100
[Process 3] Initiating DMA transfer from SSD to RAM...
[Process 3] DMA transfer from SSD complete
[Process 3] Verifying SSD data in RAM:
  RAM[200] = 0xAA00 (expected 0xAA00)
  RAM[201] = 0xAA01 (expected 0xAA01)
  RAM[202] = 0xAA02 (expected 0xAA02)
  RAM[203] = 0xAA03 (expected 0xAA03)
  RAM[204] = 0xAA04 (expected 0xAA04)
  RAM[205] = 0xAA05 (expected 0xAA05)
  RAM[206] = 0xAA06 (expected 0xAA06)
  RAM[207] = 0xAA07 (expected 0xAA07)
  RAM[208] = 0xAA08 (expected 0xAA08)
  RAM[209] = 0xAA09 (expected 0xAA09)
[Process 1] Iteration 3/5: Hello, Professor!
[Process 2] === Iteration 5/10 ===
[Process 2] Added 81 times, ACC = 1010045
[Process 2] After division by 100: ACC = 10100
```

```
[Process 1] Iteration 4/5: [Process 3] Initiating DMA transfer from HDD to RAM...
[Process 3] DMA transfer from HDD complete
[Process 3] Verifying HDD data in RAM:
  RAM[220] = 0xBB00 (expected 0xBB00)
  RAM[221] = 0xBB01 (expected 0xBB01)
  RAM[222] = 0xBB02 (expected 0xBB02)
  RAM[223] = 0xBB03 (expected 0xBB03)
  RAM[224] = 0xBB04 (expected 0xBB04)
  RAM[225] = 0xBB05 (expected 0xBB05)
  RAM[226] = 0xBB06 (expected 0xBB06)
  RAM[227] = 0xBB07 (expected 0xBB07)
  RAM[228] = 0xBB08 (expected 0xBB08)
  RAM[229] = 0xBB09 (expected 0xBB09)
  RAM[230] = 0xBB0A (expected 0xBB0A)
  RAM[231] = 0xBB0B (expected 0xBB0B)
  RAM[232] = 0xBB0C (expected 0xBB0C)
  RAM[233] = 0xBB0D (expected 0xBB0D)
  RAM[234] = 0xBB0E (expected 0xBB0E)
[Process 3] Sum of SSD data: 0x6A42D
[Process 3] Sum of HDD data: 0xAF569
Freed (PID 3) at memory [200 --> 399]
[Process 3 Completed]

Hello, Professor!
[Process 2] === Iteration 6/10 ===
[Process 2] Added 81 times, ACC = 1010045
[Process 2] After division by 100: ACC = 10100
[Process 2] === Iteration 7/10 ===
[Process 2] Added 81 times, ACC = 1010045
[Process 2] After division by 100: ACC = 10100
[INTERRUPT] Queued IRQ 1 (priority 5)
[Process 1] Iteration 5/5: Hello, Professor!
[Process 2] === Iteration 8/10 ===
[Process 2] Added 81 times, ACC = 1010045
[Process 2] After division by 100: ACC = 10100
[Process 2] === Iteration 9/10 ===
[Process 2] Added 81 times, ACC = 1010045
[Process 2] After division by 100: ACC = 10100
[Process 1] Stored completion marker at address 0x32
Freed (PID 1) at memory [0 --> 99]
[Process 1 Completed]
```

```
[Process 2] === Iteration 10/10 ===
[Process 2] Added 81 times, ACC = 1010045
[Process 2] After division by 100: ACC = 10100
[INTERRUPT] Queued IRQ 2 (priority 3)

[Process 2] Final ACC value after multiplying by 2: 20200
[Process 2] Stored final result 20200 at memory address 0x6E
[Process 2] Verification: Read back value 20200 from memory
Freed (PID 2) at memory [100 --> 199]
[Process 2 Completed]


[CPU EXECUTION PHASE]
=======================

Executing loaded CPU program...


[RESULTS AND STATISTICS]
===========================

Final Memory Contents:
  Address 0x100: 0x2A (initial value)
  Address 0x101: 0xA (initial value)
  Address 0x102: 0xFFFFFFFF (result of computation)
  Address 0x103: 0xFFFFFFFF (result of subtraction)

Cache statistics:
L1 hits:   0
L1 misses: 5
L2 hits:   0
L2 misses: 5

CPU Final State:
  AX: 0x4
  BX: 0x0
  CX: 0x0
  DX: 0x0

[CLEANUP PHASE]
```

```
════════════════

✓ All resources freed successfully

╔══════════════════════════════════════════════════════════════╗
║                 Demo Completed Successfully                  ║
╚══════════════════════════════════════════════════════════════╝
```