Project 1 Documentation
CPU Simulation

Brysen Pfingsten, Nathaniel Savoury, David Anthony Fields

September 22, 2025

CSAS 3111, Seton Hall University

Contents

1 Outline

1.1 Problem Statement

In this project, you will develop a Computer System Simulator that simulates key components of a basic computer system. The purpose of this assignment is to help you understand the fundamental concepts of computer systems (from chapters 1 and 2). You will simulate components such as the CPU, memory, interrupt handling, and Direct Memory Access (DMA).

The goal is to design and simulate a fully functioning computer system by implementing all the components listed below. Your project 1 will cover five modules, each focusing on the simulation of specific system components.

- Module 1: Central Processing Unit
- Module 2: Memory Management System
- Module 3: Instructions Set Architecture
- Module 4: Interrupts
- Module 5: Direct Memory Access

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 **??**. Finally, the simulation of the described computer system is be found in Section 7.

2 Module 1: Central Processing Unit

2.1 Problem Statement

Implement the core components of the CPU including the Program Counter (`PC`), Accumulator (`ACC`), and Instruction Register (`IR`). The CPU will follow the fetch-decode-execute cycle, fetching instructions from memory, decoding them, and performing operations.

Simulate the `PC`, which stores the address of the next instruction to be executed. Implement the `ACC` for performing arithmetic and logic operations, and the IR to hold the current instruction. The Status Register will manage flags like zero, carry, and overflow. The CPU must execute a fetch-decode-execute cycle, where it fetches an instruction from memory, decodes the opcode, and executes it, updating the register values. For instance, the CPU might fetch an `ADD` instruction, decode it, and then update the `ACC` with the result. Implement error handling for invalid instructions, and ensure the `PC` updates correctly after each cycle. Your program execution should simulate real-time instruction processing and register updates, handling valid instructions and interrupts efficiently. It must handle instructions such as `ADD`, `SUB`, `LOAD`, and `STORE`. Also, ensure that the `PC` is updated correctly after each instruction, and implement error handling for invalid instructions.

2.2 Implementation

To implement the core components of the CPU, we first chose to represent the CPU as a structure:

```
typedef struct {
  word PC;
  word ACC;
  word IR;
  Flags flags;
} Cpu;^^I
```

where the `PC`, `ACC` and, `IR` are all represented as `word`s and the flags is a structure:

```
typedef struct {
  int ZERO;
  int CARRY;
  int OVERFLOW;
  int INTERRUPT;
} Flags;
```

The main function for the implementation of the CPU is:

```
// Runs the fetch-execution cycle program_size times
// or until a halt is encountered
void cpu_run(const int program_size, word* mem) {
  for (int i = 0; i < program_size && CPU.PC != CPU_HALT; i++) {
    printf("=== Cycle %d ===\n", i + 1);

    if (CPU.PC == CPU_HALT) {
      printf("CPU Halted!\n");
      break;
    }
    fetch();
    execute();
    check_for_interrupt();

    cpu_print_state();
  }
}
```

The design idea behind this function leverages the fact that the CPU runs for at most `program_size` amount of times unless a halt is encounter. Since the CPU cycles a variable amount of times, we decided to use a for-loop. Within the body of the for-loop we print the cycle number, fetch the instruction using `fetch`, execute the instruction using `execute`, check for and handle system interrupts using `check_for_interrupt`, and finally printing the CPU state using `cpu_print_state`. Within `fetch`, we read the instruction from the `PC` and load it into the `IR` then increment the `PC` by 1. For `execute`, we decode the instruction by extracting the opcode and operand then execute the instruction using the extracted opcode and operand. After the instruction has been executed, we check for interrupts and handle any interrupts accordingly.[1] To close the cycle, print the CPU state where we show the values of the `PC`, `ACC`, `IR`, and the flags.

[1]For more information on how we execute instructions and check for interrupts, see Section 4.2 and Section 5.2 respectively.

3 Module 2: Memory Management System

3.1 Problem Statement

Simulate a hierarchical memory system, including RAM as the main memory for storing data and programs, and cache (L1 and L2) for faster memory access. The memory system should allow the CPU to perform read and write operations. Implement the logic cache hits and misses, where the CPU first checks the cache for data and falls back on RAM if necessary.

3.2 Implementation

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;
}

//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;
  }
}
```

4 Module 3: Instructions Set Architecture

4.1 Problem Statement

Define a simple ISA with basic operations like `ADD`, `SUB`, `LOAD`, and `STORE`. Assign opcodes to each instruction, allowing the CPU to interpret and execute them. The ISA should ensure that the CPU can fetch, decode, and execute these instructions, modifying the values in registers or interacting with memory as necessary. Provide mechanisms for error handling in case of undefined opcodes.

4.2 Implementation

To implement our ISA, we define an enumeration type for the different opcodes and functions to handle each one. Our enum is defined as:

```
typedef enum op {
  OP_LOAD   = 0x1,
  OP_STORE  = 0x2,
  OP_ADD    = 0x5,
  OP_SUB    = 0x6,
  OP_HALT   = 0xF,
  OP_INTR   = 0x9,
  OP_ENDINT = 0xA,
} OP;
```

The opcodes are dispatched to their functions by the `execute_instruction` function. Invalid opcodes are treated as an error and the CPU is halted.

```
void execute_instruction(const OP opcode, const mem_addr operand)
{
  switch (opcode) {
    case OP_LOAD:  load(operand);       break;
    case OP_STORE: store(operand);      break;
    case OP_ADD:   add(operand);        break;
    case OP_SUB:   sub(operand);        break;
    case OP_INTR:  interrupt(operand);  break;
    case OP_HALT:  halt();              break;
    default:
      printf("ERROR: Invalid opcode %u (IR=0x%04X)\n", (unsigned)opcode,
      (unsigned)operand);
      CPU.PC = CPU_HALT;}}
```

All functions except for HALT take as input an operand which represents the address where the relevant data is to be found. The ISA functions are defined as follows:

```
// Loads the data at the given memory address into cpu's ACC register
void load(const mem_addr operand) {
  CPU.ACC = read_mem(operand);
  set_zero_flag(CPU.ACC);
}

// Stores the data in cpu's ACC register at the given memory address
void store(const mem_addr operand) {
  write_mem(operand, CPU.ACC);
}

// Adds the value in the cpu's ACC register with the given value
// The sum is stored in ACC and the appropriate flags are set
void add(const mem_addr operand) {
  word init = CPU.ACC;
  CPU.ACC += operand;
  set_add_flags(init, operand, CPU.ACC);
}

// Subtracts the value in the cpu's ACC register with the given value
// The difference is stored in ACC and the appropriate flags are set
void sub(const mem_addr operand) {
  word init = CPU.ACC;
  CPU.ACC -= operand;
  set_sub_flags(init, operand, CPU.ACC);
}

//Initiates and handles CPU interrupts
void interrupt(const mem_addr operand) {
  set_interrupt_flag(operand);
}

// Halts execution of the given cpu
void halt() {
  printf("HALT\n");
  CPU.PC = CPU_HALT;
}
```

5 Module 4: Interrupts

5.1 Problem Statement

Implement a mechanism for interrupts that allows the CPU to pause its current task, service an interrupt through an Interrupt handler, and then return to the previous task. This includes handling hardware interrupts (e.g., timer interrupts) and software interrupts (e.g., system calls). Ensure that the CPU can manage both synchronous and asynchronous events, simulating how real systems handle task prioritization through interrupts.

5.2 Implementation

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

```
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.

```
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.

```
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.

```
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 interupt and handle it
    switch(intrpt.irq) {
    case SAY_HI :
      printf("INTERRUPT: hello\n");
      set_interrupt_flag(false);
      reset_curr_interrupt();
      break;
    case SAY_GOODBYE :
```

```
      printf("INTERRUPT: goodbye\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++;
}
```

6 Module 5: Direct Memory Access

6.1 Problem Statement

Simulate a DMA system that allows for data transfer between memory and I/O devices without CPU intervention. Implement a DMA controller that manages data transfers between RAM and devices like disk storage or network interfaces. The CPU will initiate the DMA transfer, but the actual data movement should proceed without using CPU cycles, demonstrating how DMA improves system efficiency by freeing up the CPU for other tasks.

6.2 Implementation

Our DMA implementation revolves around two functions: `initiateDMA` and `dmaTransfer`. `initiateDMA` is invoked when a transfer between memory and I/O devices is required. Once the request has been initiated, `dmaTransfer` handles the actual transfer of values.

```
void dmaTransfer(word* source, word destination, int size) {
    for (int i = 0; i < size; i++) {
        //get the value source is pointing to
        word val = *source;
        write_mem(destination, val);
        destination++;
        source++;
    }
}

void initiateDMA(word* source, word destination, int size) {
    dmaTransfer(source, destination, size);
}
```

7 The Simulation

```
int main()
{
  //initialize the cache for use
  init_cache(&L1, L1CACHE_SIZE);
  init_cache(&L2, L2CACHE_SIZE);

  //initialize the ram
  init_ram(RAM_SIZE);
  init_HDD(HDD_SIZE);
  init_SSD(SSD_SIZE);

  //initialize the interrupt controller
  init_interrupt_controller();

  //initialize the cpu
  init_cpu(&CPU);

  //memory adresses for later
  write_mem(0x0100, 0x0100);

  //write some random instructions to memory
  write_mem(0x0, 0x5001);
  write_mem(0x1, 0x5002);
  write_mem(0x2, 0x5003);

  initiateDMA(RAM, *HDD, 100);

  write_mem(0x3, 0x5004);
  write_mem(0x4, 0x6004);
  write_mem(0x5, 0x6003);

  add_interrupt(0x0001, 5);

  write_mem(0x6, 0x6002);
  write_mem(0x7, 0x6001);

  initiateDMA(HDD, *SSD, 25);

  write_mem(0x8, 0x1100);
```

```
  write_mem(0x9, 0x5050);

  initiateDMA(RAM, *SSD, 50);

  write_mem(0xA, 0x200C);

  add_interrupt(0x0002, 1);

  initiateDMA(SSD, *HDD, 200);

  write_mem(0xB, 0xF000);


  cpu_run(20, RAM);
  print_cache_stats();
  printf("saved memory value == %X", read_mem(0x000C));

  free(L1.items);
  free(L2.items);
  free(RAM);
  return 0;
}
```

Results...

```
Initialized cache at -> '0x5dfa437b94a0' <- with size: 5
Initialized cache at -> '0x5dfa437b94c0' <- with size: 20
initialized ram with size: 500
initialized interrupt controller
Initialized the cpu!
CPU STATE
PC:  0
ACC: 0
IR:  FFFFFFFF
FLAGS:
  ZERO:      0
  CARRY:     0
  OVERFLOW:  0
  INTERRUPT: 0
```

```
=== Cycle 1 ===
INTERRUPT: goodbye
CPU STATE
PC:  1
ACC: 1
IR:  5001
FLAGS:
  ZERO:      0
  CARRY:     0
  OVERFLOW:  0
  INTERRUPT: 1


=== Cycle 2 ===
INTERRUPT: hello
CPU STATE
PC:  2
ACC: 3
IR:  5002
FLAGS:
  ZERO:      0
  CARRY:     0
  OVERFLOW:  0
  INTERRUPT: 0


=== Cycle 3 ===
CPU STATE
PC:  3
ACC: 6
IR:  5003
FLAGS:
  ZERO:      0
  CARRY:     0
  OVERFLOW:  0
  INTERRUPT: 0


=== Cycle 4 ===
CPU STATE
PC:  4
```

```
ACC: A
IR:  5004
FLAGS:
  ZERO:      0
  CARRY:     0
  OVERFLOW:  0
  INTERRUPT: 0


=== Cycle 5 ===
CPU STATE
PC:  5
ACC: 6
IR:  6004
FLAGS:
  ZERO:      0
  CARRY:     0
  OVERFLOW:  0
  INTERRUPT: 0


=== Cycle 6 ===
CPU STATE
PC:  6
ACC: 3
IR:  6003
FLAGS:
  ZERO:      0
  CARRY:     0
  OVERFLOW:  0
  INTERRUPT: 0


=== Cycle 7 ===
CPU STATE
PC:  7
ACC: 1
IR:  6002
FLAGS:
  ZERO:      0
  CARRY:     0
```

```
  OVERFLOW:  0
  INTERRUPT: 0


=== Cycle 8 ===
CPU STATE
PC:  8
ACC: 0
IR:  6001
FLAGS:
  ZERO:      1
  CARRY:     0
  OVERFLOW:  0
  INTERRUPT: 0


=== Cycle 9 ===
CPU STATE
PC:  9
ACC: 100
IR:  1100
FLAGS:
  ZERO:      0
  CARRY:     0
  OVERFLOW:  0
  INTERRUPT: 0


=== Cycle 10 ===
CPU STATE
PC:  A
ACC: 150
IR:  5050
FLAGS:
  ZERO:      0
  CARRY:     0
  OVERFLOW:  0
  INTERRUPT: 0


=== Cycle 11 ===
```

```
CPU STATE
PC:  B
ACC: 150
IR:  200C
FLAGS:
  ZERO:      0
  CARRY:     0
  OVERFLOW:  0
  INTERRUPT: 0


=== Cycle 12 ===
HALT
CPU STATE
PC:  FFFFFFFF
ACC: 150
IR:  FFFFF000
FLAGS:
  ZERO:      0
  CARRY:     0
  OVERFLOW:  0
  INTERRUPT: 0



Cache statistics:
L1 hits:   0
L1 misses: 13
L2 hits:   0
L2 misses: 13
saved memory value == 150
```