Faculty of Media Engineering and Technology
Computer Systems Architecture – CSEN 601
Spring 2025

Project Report

Team Number: 26
Package Number and Name: Package 2 – "Fillet-O-Neumann with moves on the side"

Team Members:
Mennatallah Mohamed Gharieb – [58-10109] – [T 11]
Hoor Haytham Anas Mohamed – [58-10710] – [T 9]
Mariam Mohamed Barakat – [58-9679] – [T 10]
Nada Selim – [58-17829] – [T 22]
Rokia Houssam – [58-15838] – [T 22]
Aeisha Khalifa – [58-12534] – [T 22]
Rahma Issa – [58-9381] – [T 11]

Date: May 19, 2025
For: Review of CPU Simulator Implementation

Contents

1 Introduction **2**

2 Codebase Overview **2**

3 File Structure **2**

4 Key Components **3**
4.1 Instruction Set Architecture (ISA) 3
4.2 Hardware Definitions 3
4.3 Pipeline Stages 3
4.4 Data Structures 4

5 Detailed Functionality **4**
5.1 Parser (parser.c, parser.h) 4
5.2 Hardware (hardware.c, hardware.h) 4
5.3 Datapath (datapath.c, datapath.h) 4
5.4 Executor (executor.c, executor.h) 5
5.5 Main (main.c) 5

6 Pipeline Execution Flow **5**

7 Key Features **5**

8 Limitations and Potential Improvements **5**

9 Conclusion **6**

A Source Code **6**
A.1 hardware.h 6
A.2 hardware.c 6
A.3 datapath.h 6
A.4 datapath.c 8
A.5 executor.h 16
A.6 executor.c 16
A.7 main.c 20
A.8 parser.h 20
A.9 parser.c 21

1 Introduction

This report provides a comprehensive overview of a pipelined CPU simulator implemented in C, designed to execute instructions from a custom instruction set architecture (ISA). The codebase models a five-stage pipeline (Fetch, Decode, Execute, Memory, Writeback) and supports R-type, I-type, and J-type instructions. The simulator reads instructions from a text file, encodes them into a 32-bit binary format, and executes them in a pipelined manner, handling control hazards and arithmetic flags.

The report is structured to detail the codebase's purpose, file organization, key components, functionality, and areas for improvement. The complete source code is included in Appendix A for reference. This document serves as a technical reference for developers, educators, or researchers analyzing the simulator.

2 Codebase Overview

The CPU simulator emulates a simplified processor with a custom ISA comprising 12 instructions. It features a 2048-word memory (divided into instruction and data regions), 33 registers (including a program counter), and a pipelined execution model. The primary objectives of the codebase are:

- To parse and encode instructions from a text file.
- To execute instructions through a five-stage pipeline.
- To manage control hazards (e.g., jumps) via instruction dropping.
- To track arithmetic flags (carry, overflow) for operations like addition and multiplication.

The simulator is modular, with distinct files handling parsing, hardware definitions, pipeline stages, and execution control.

3 File Structure

The codebase is organized into several files, each with a specific role. The complete source code for these files is provided in Appendix A:

- **parser.c, parser.h**: Parses instructions from `dummy_instructions.txt` and encodes them into 32-bit binary format.
- **hardware.c, hardware.h**: Defines memory (2048 words), registers (33), and counters for instruction and data storage.
- **datapath.c, datapath.h**: Implements the pipeline stages (Fetch, Decode, Execute, Memory, Writeback) and instruction execution logic.
- **executor.c, executor.h**: Manages pipelined execution across clock cycles, handling instruction tracking and dropping.
- **main.c**: Entry point; initiates pipeline execution and prints final register and memory states.

4 Key Components

4.1 Instruction Set Architecture (ISA)

The ISA includes 12 instructions, categorized into three types:

- **R-type**: Register-based (e.g., ADD, SUB, MUL, AND, LSL, LSR).
- **I-type**: Immediate-based (e.g., MOVI, JEQ, XORI, MOVR, MOVM).
- **J-type**: Jump-based (e.g., JMP).

Each instruction is encoded as a 32-bit word with the following formats:

- **R-type**: `[opcode:4][r1:5][r2:5][r3:5][shamt:13]`
- **I-type**: `[opcode:4][r1:5][r2:5][immediate:18]`
- **J-type**: `[opcode:4][address:28]`

The instruction set is defined in `parser.c` with opcodes ranging from 0 (ADD) to 11 (MOVM).

4.2 Hardware Definitions

The hardware configuration is defined in `hardware.c` and `hardware.h`:

- **Memory**: `memory[2048]` stores instructions (0–1023) and data (1024–2047).
- **Registers**: `reg_array[33]` includes:
  - `R0`: Zero register (read-only).
  - `R1--R31`: General-purpose registers.
  - `R32`: Program Counter (PC).
- **Counters**:
  - `memoryInstructionCounter`: Tracks loaded instructions.
  - `memoryDataCounter`: Fixed at 1024 for data offset.

4.3 Pipeline Stages

The simulator implements a five-stage pipeline, with some stages split into two cycles:

1. **Fetch**: Retrieves instruction from `memory[PC]`, increments PC.
2. **Decode**:
   - **Cycle 1**: Extracts opcode, registers, immediate, and address.
   - **Cycle 2**: Assigns instruction type (R, I, J).
3. **Execute**:
   - **Cycle 1**: Validates instruction and extracts opcode.
   - **Cycle 2**: Performs operations (e.g., ADD, JMP).
4. **Memory**: Handles read (MOVR) or write (MOVM) operations.
5. **Writeback**: Writes results to registers.

4.4 Data Structures

Key data structures include:

- `decodedInstruction`: Stores decoded instruction details (opcode, registers, etc.).
- `mem_reg_destinationInfo`: Represents execution or memory operation results (type, index, value).
- `storeOutput`: Tracks instruction state across pipeline stages.
- `InstructionInfo`: Maps mnemonics to opcodes and types for parsing.

5 Detailed Functionality

5.1 Parser (parser.c, parser.h)

- **readFile()**: Reads `dummy_instructions.txt` and parses each line.
- **parseInstruction()**: Tokenizes lines into mnemonic and arguments (e.g., "ADD R1 R2 R3").
- **encodeInstruction()**: Converts instructions to 32-bit binary, handling two's complement for negative immediates.
- Stores encoded instructions in `memory[0--1023]`.

5.2 Hardware (hardware.c, hardware.h)

- Initializes `memory[2048]` to zero, with `memoryDataCounter` set to 1024.
- Defines `reg_array[33]` for registers and PC.
- Includes a redundant `R0` variable (not used, as `reg_array[0]` is managed directly).

5.3 Datapath (datapath.c, datapath.h)

- **Fetch**: Validates PC and retrieves instructions.
- **Decode**: Extracts fields using bit shifts and assigns types.
- **Execute**: Implements instructions (e.g., ADD sets carry/overflow flags, JEQ/JMP update PC).
- **Memory**: Reads/writes data for MOVR/MOVM.
- **Writeback**: Updates registers (skips R0, R32).
- **Flags**:
  - `flag`: Triggers instruction dropping for jumps.
  - `carry_flag`: Set for unsigned overflow/borrow.
  - `overflow_flag`: Set for signed arithmetic overflow.

5.4 Executor (executor.c, executor.h)

- **pipelineExecution()**: Manages pipeline execution:
  - **Odd cycles**: Fetch, Decode Cycle 2, Execute Cycle 2, Writeback.
  - **Even cycles**: Decode Cycle 1, Execute Cycle 1, Memory.
- Tracks instructions in `storeOutput` array, handling dropping for jumps.
- Terminates when all instructions are completed or dropped.

5.5 Main (main.c)

- Initiates `pipelineExecution()`.
- Prints final register and memory states (memory[1024–1099]).
- Includes commented-out test functions for overflow/carry flags.

6 Pipeline Execution Flow

1. **Initialization**: Loads instructions via `readFile()`, resets flags.
2. **Clock Cycles**:
   - **Odd**: Fetch new instructions, process later stages, drop instructions if `flag` is set.
   - **Even**: Process early stages, update `dropFlagCount` to reset `flag`.
3. **Termination**: Stops when all instructions are processed.

7 Key Features

- **Instruction Dropping**: Handles control hazards (JEQ, JMP) by dropping in-flight instructions for two cycles.
- **Sign Extension**: Extends 18-bit immediates to 32 bits using two's complement.
- **Arithmetic Flags**: Tracks carry and overflow for ADD, SUB, MUL.
- **Modular Design**: Separates parsing, hardware, datapath, and execution logic.

8 Limitations and Potential Improvements

- **Error Handling**: Limited validation for invalid instructions or memory accesses.
  - *Improvement*: Add robust checks in `parser.c` and `datapath.c`.
- **Pipeline Hazards**: No support for data hazards.
  - *Improvement*: Implement forwarding or stalling mechanisms.
- **Redundant Code**: `R0` variable and commented test functions.
  - *Improvement*: Remove `R0`, integrate tests into a testing framework.
- **Documentation**: Sparse inline comments.
  - *Improvement*: Add detailed comments for complex logic (e.g., flag handling).

9 Conclusion

The pipelined CPU simulator is a robust educational tool for studying CPU architecture and pipeline execution. Its modular design, clear ISA, and effective handling of control hazards make it suitable for academic or developmental use. Addressing limitations such as error handling and hazard management could enhance its reliability and functionality. This codebase serves as a valuable resource for understanding the intricacies of CPU design and instruction execution.

A Source Code

A.1 hardware.h

```
#ifndef HARDWARE_H
#define HARDWARE_H

// Memory size and register count constants
#define MEMORY_SIZE 2048
#define REGISTER_COUNT 33

// External memory and register declarations
extern int memory[MEMORY_SIZE];  // -01023: instructions, -10242047: data
extern int memoryInstructionCounter;  // starts at 0 till 1023: counter for
    storing instructions
extern int memoryDataCounter; //starts at 1024: counter for storing data
extern int reg_array[REGISTER_COUNT];  // 0: R0, -131: -R1R31, 32: PC

#endif // HARDWARE_H
```

Listing 1: hardware.h: Hardware Definitions

A.2 hardware.c

```
#include "stdio.h"
#include "hardware.h"

int memory[MEMORY_SIZE] = {0}; // 0-1023 stores instructions.
                          // 1024-2047 stores data.

int memoryInstructionCounter = 0; //counter for storing instructions in
    memory
int memoryDataCounter = 1024; //counter for storig data in memory

int reg_array[REGISTER_COUNT]; // Index 0: the zero register R0.
                               // Indices 1-31: general purpose registers
                                  R1-R31.
                               // Index 32: PC register.

int R0 = 0; //variable to store the values stored in R0 so we don't
    overwrite it.
```

Listing 2: hardware.c: Hardware Implementation

A.3 datapath.h

```c
#ifndef DATAPATH_H
#define DATAPATH_H

#include <stdbool.h>
#include "parser.h"

extern bool flag;
extern bool overflow_flag;
extern bool carry_flag;
typedef struct {
    int shamt;   // Shift amount
    int r3;    // Destination or third register
    int r2;    // Second source register
    int r1;   // First source register
    int opcode;    // Operation code
} RFormat;

typedef struct {
    int immediate;  // Immediate value (can be signed)
    int r2;   // Second register or unused for some ops
    int r1;   // Destination register
    int opcode;   // Operation code
} IFormat;

typedef struct {
    int address;   // Jump target address
    int opcode;    // Operation code
} JFormat;

typedef struct {
    InstrType type;  // Tag to identify which format is valid
    bool isValid;

    union {
        RFormat r_format;
        IFormat i_format;
        JFormat j_format;
    } format;
} Instruction;

typedef struct {
    InstrType type;
    int opcode;
    int r1;
    int r2;
    int r3;
    int shamt;
    int immediate;
    int address;
    bool isValid;
} decodedInstruction;

typedef struct {
    char type; // store 'M' for memory, 'R' for register, or 'B' for both
    int index; // indicate the index of memory or reg to accessed
    int value; // value to be stored or loc used to read from memory.
    bool isValid; // to flag if an error occurs
```

```
    bool dropFlag; // to flag if executing/decoding instructions need to be dropped due to a jmp
} mem_reg_destinationInfo;

int sign_extend(int value, int bits);

int fetch();

decodedInstruction decode_cycle1(int instruction);
decodedInstruction decode_cycle2(decodedInstruction instData);

int execute_cycle1(decodedInstruction instr);
mem_reg_destinationInfo execute_cycle2(int opcode, decodedInstruction instr);

// ---------- Instruction Implementations ----------
mem_reg_destinationInfo add(int r1, int r2, int r3);
mem_reg_destinationInfo sub(int r1, int r2, int r3);
mem_reg_destinationInfo mul(int r1, int r2, int r3);
mem_reg_destinationInfo movi(int r1, int imm);
mem_reg_destinationInfo jeq(int r1, int r2, int offset);
mem_reg_destinationInfo and(int r1, int r2, int r3);
mem_reg_destinationInfo xori(int r1, int r2, int imm);
mem_reg_destinationInfo jmp(int address);
mem_reg_destinationInfo lsl(int r1, int r2, int shamt);
mem_reg_destinationInfo lsr(int r1, int r2, int shamt);
mem_reg_destinationInfo movr(int r1, int r2, int offset);
mem_reg_destinationInfo movm(int r1, int r2, int offset);

// ---------- Memory and Writeback ----------
mem_reg_destinationInfo MEM(mem_reg_destinationInfo dest);
bool writeBack(mem_reg_destinationInfo dest);

#endif // DATAPATH_H
```

Listing 3: datapath.h: Datapath Definitions

A.4 datapath.c

```
#include "stdio.h"
#include "stdbool.h"
#include "datapath.h"
#include "hardware.h"
#include <stdint.h>  // for int64_t, INT32_MAX, INT32_MIN
#include <limits.h>  // for INT32_MAX, INT32_MIN

bool flag = false;
int dropFlagCount = 0;
bool carry_flag = false;
bool overflow_flag = false;

int sign_extend(int value, int bits) {
    int mask = 1 << (bits - 1);          // Gets the sign bit (e.g. bit 17 for 18-bit)
    return (value ^ mask) - mask;        // Applies 'twos complement extension
}
```

```
17
18 int fetch(){
19     int pc = reg_array[32];
20
21     if(pc<0 || pc>1023 || pc >= memoryInstructionCounter){ //check that pc's value is valid
22         return -1;
23     }
24
25     int instruction = memory[pc];
26
27     printf("Fetched instruction at PC=%d: 0x%08X\n\n", pc, memory[pc]);
28
29     pc++;
30     reg_array[32] = pc;
31
32     return instruction;
33 }
34
35 decodedInstruction decode_cycle1(int instruction){
36     decodedInstruction res;
37
38     if((flag && dropFlagCount < 2) || instruction == -1){
39         res.isValid = false;
40         return res;
41     }
42
43     res.opcode = (instruction >> 28) & 0xF;
44     res.r1 = (instruction >> 23) & 0x1F; // R-type & I-type instructions
45     res.r2 = (instruction >> 18) & 0x1F; // R-type & I-type instructions
46     res.r3 = (instruction >> 13) & 0x1F; // R-type instructions
47     res.shamt = instruction & 0x1FFF; // R-type instructions
48
49     int tmp = instruction & 0b11111111111111111; //18 bits
50     int signedVal = sign_extend(tmp, 18);
51
52     res.immediate = signedVal; // I-type instructions
53     res.address = instruction & 0b11111111111111111111111111; // J-type instructions
54     res.isValid = true; // no error occured and the instrucion hasn't been dropped
55
56     return res;
57 }
58
59 decodedInstruction decode_cycle2(decodedInstruction instData){
60     int opcode = instData.opcode;
61
62     if((flag && dropFlagCount ==2) || instData.isValid == false){
63         return instData;
64     }
65
66     switch(opcode){
67         case 0b0000: //ADD
68             instData.type = R_TYPE;
69             break;
70         case 0b0001: //SUB
71             instData.type = R_TYPE;
```

```
            break;
        case 0b0010: //MUL
            instData.type = R_TYPE;
            break;
        case 0b0011: //MOVI
            instData.type = I_TYPE;
            instData.r2 = 0;
            break;
        case 0b0100: //JEQ
            instData.type = I_TYPE;
            break;
        case 0b0101: //AND
            instData.type = R_TYPE;
            break;
        case 0b0110: //XORI
            instData.type = I_TYPE;
            break;
        case 0b0111: //JMP
            instData.type = J_TYPE;
            break;
        case 0b1000: //LSL
            instData.type = R_TYPE;
            break;
        case 0b1001: //LSR
            instData.type = R_TYPE;
            break;
        case 0b1010: //MOVR
            instData.type = I_TYPE;
            break;
        case 0b1011: //MOVM
            instData.type = I_TYPE;
            break;
        default: // for invalid instructions
            printf("An error occurred in decode_cycle2!");
            instData.isValid = false;
    }

    return instData;
}

int execute_cycle1(decodedInstruction instr){
    if((flag && dropFlagCount < 2) || instr.isValid == false){
        return -1;
    }

    return instr.opcode;
}

mem_reg_destinationInfo execute_cycle2(int opcode, decodedInstruction instr
    ){
    mem_reg_destinationInfo res;

    if((flag && dropFlagCount == 2) || opcode == -1){
        printf("entered null. flag = %s, opcode = %d \n", flag ? "true" : "
            false", opcode);
        res.isValid = false;
        res.dropFlag = false;
        return res;
```

```
128     }
129
130     if (opcode < 0 || opcode > 11) {
131         printf("ERROR: Invalid opcode in execute_cycle2: %d\n", opcode);
132         res.isValid = false;
133         return res;
134     }
135
136     carry_flag = false;
137     overflow_flag = false;
138
139     switch (opcode) {
140     case 0b0000: //ADD
141             res = add(instr.r1,instr.r2,instr.r3);
142             break;
143     case 0b0001: //SUB
144             res = sub(instr.r1,instr.r2,instr.r3);
145             break;
146     case 0b0010: //MUL
147             res = mul(instr.r1,instr.r2,instr.r3);
148             break;
149     case 0b0011: //MOVI
150             res = movi(instr.r1,instr.immediate);
151             break;
152     case 0b0100: //JEQ
153             res = jeq(instr.r1,instr.r2,instr.immediate);
154             break;
155     case 0b0105: //AND
156             res = and(instr.r1,instr.r2,instr.r3);
157             break;
158     case 0b0110: //XORI
159             res = xori(instr.r1,instr.r2,instr.immediate);
160             break;
161     case 0b0111: //JMP
162             res = jmp(instr.address);
163             break;
164     case 0b1000: //LSL
165             res = lsl(instr.r1,instr.r2,instr.shamt);
166             break;
167     case 0b1001: //LSR
168             res = lsr(instr.r1,instr.r2,instr.shamt);
169             break;
170     case 0b1010: //MOVR
171             res = movr(instr.r1,instr.r2,instr.immediate);
172             break;
173     case 0b1011: //MOVM
174             res = movm(instr.r1,instr.r2,instr.immediate);
175             break;
176     }
177
178     return res;
179 }
180
181 mem_reg_destinationInfo add(int r1, int r2, int r3) {
182     int32_t a = reg_array[r2];
183     int32_t b = reg_array[r3];
184
185     // Use 64-bit to capture overflow bit
```

```
186     int64_t signed_result = (int64_t)a + (int64_t)b;
187     uint64_t unsigned_result = (uint64_t)(uint32_t)a + (uint64_t)(uint32_t)
            b;
188 
189     // Carry: check bit 32 (for 32-bit unsigned)
190     carry_flag = (unsigned_result >> 32) & 1;
191 
192     // Overflow: XOR of carry-in to MSB and carry-out of MSB
193     int sign_a = (a >> 31) & 1;
194     int sign_b = (b >> 31) & 1;
195     int sign_res = (int32_t)signed_result >> 31 & 1;
196 
197     overflow_flag = (sign_a == sign_b && sign_a != sign_res);
198 
199     mem_reg_destinationInfo res;
200     res.type = 'R';
201     res.index = r1;
202     res.value = (int32_t) signed_result;  // Truncate to 32-bit
203     res.isValid = true;
204     res.dropFlag = false;
205 
206     printf("ADD: a=%d, b=%d, result=%d | carry=%d, overflow=%d\n", a, b,
            res.value, carry_flag, overflow_flag);
207 
208     return res;
209 }
210 
211 mem_reg_destinationInfo sub(int r1, int r2, int r3) {
212     int32_t a = reg_array[r2];
213     int32_t b = reg_array[r3];
214 
215     int64_t signed_result = (int64_t)a - (int64_t)b;
216     uint64_t unsigned_result = (uint64_t)(uint32_t)a - (uint64_t)(uint32_t)
            b;
217 
218     // Carry (unsigned borrow): if a < b
219     carry_flag = (uint32_t)a < (uint32_t)b;
220 
221     // Overflow: a and b have opposite signs, and result sign is different
            from a
222     int sign_a = (a >> 31) & 1;
223     int sign_b = (b >> 31) & 1;
224     int sign_res = (int32_t)signed_result >> 31 & 1;
225 
226     overflow_flag = (sign_a != sign_b && sign_a != sign_res);
227 
228     mem_reg_destinationInfo res;
229     res.type = 'R';
230     res.index = r1;
231     res.value = (int32_t) signed_result;
232     res.isValid = true;
233     res.dropFlag = false;
234 
235     printf("SUB: a=%d, b=%d, result=%d | carry=%d, overflow=%d\n", a, b,
            res.value, carry_flag, overflow_flag);
236 
237     return res;
238 }
```

```
239
240 mem_reg_destinationInfo mul(int r1, int r2, int r3) {
241     int32_t a = reg_array[r2];
242     int32_t b = reg_array[r3];
243
244     int64_t wide_result = (int64_t)a * (int64_t)b;
245
246     // Overflow: if result doesn't fit in 32-bit signed range
247     overflow_flag = (wide_result > INT32_MAX || wide_result < INT32_MIN);
248     carry_flag = false;  // Carry flag is undefined in signed MUL
249
250     mem_reg_destinationInfo res;
251     res.type = 'R';
252     res.index = r1;
253     res.value = (int32_t) wide_result;  // Truncate
254     res.isValid = true;
255     res.dropFlag = false;
256
257     printf("MUL: a=%d, b=%d, result=%d | overflow=%d\n", a, b, res.value,
        overflow_flag);
258
259     return res;
260 }
261
262 mem_reg_destinationInfo movi(int r1, int imm) {
263     mem_reg_destinationInfo res;
264     res.type = 'R';
265     res.index = r1;
266     res.value = imm;
267     res.isValid = true;
268     res.dropFlag = false;
269
270     return res;
271 }
272
273 mem_reg_destinationInfo jeq(int r1, int r2, int offset) {
274     mem_reg_destinationInfo res;
275
276     if (reg_array[r1] == reg_array[r2]) {
277         reg_array[32] = offset + reg_array[32];
278         printf("    REG: PC --> %d\n", reg_array[32]);
279         res.index = 32;
280         res.isValid = true;
281         flag = true;
282     } else{
283         res.isValid = false;
284         res.dropFlag = false;
285     }
286
287     return res;
288 }
289
290 mem_reg_destinationInfo and(int r1, int r2, int r3) {
291     mem_reg_destinationInfo res;
292     res.type = 'R';
293     res.index = r1;
294     res.value = reg_array[r2] & reg_array[r3];
295     res.isValid = true;
```

```
296     res.dropFlag = false;
297
298     return res;
299 }
300
301 mem_reg_destinationInfo xori(int r1, int r2, int imm) {
302     mem_reg_destinationInfo res;
303     res.type = 'R';
304     res.index = r1;
305     res.value = reg_array[r2] ^ imm;
306     res.isValid = true;
307     res.dropFlag = false;
308
309     return res;
310 }
311
312 mem_reg_destinationInfo jmp(int address) {
313     mem_reg_destinationInfo res;
314     reg_array[32] = address;
315     printf("    REG: PC --> %d\n", reg_array[32]);
316     res.index = 32;
317     res.isValid = true;
318     flag = true;
319     return res;
320 }
321
322 mem_reg_destinationInfo lsl(int r1, int r2, int shamt) {
323     mem_reg_destinationInfo res;
324     res.type = 'R';
325     res.index = r1;
326     res.value = reg_array[r2] << shamt;
327     res.isValid = true;
328     res.dropFlag = false;
329
330     return res;
331 }
332
333 mem_reg_destinationInfo lsr(int r1, int r2, int shamt) {
334     mem_reg_destinationInfo res;
335     res.type = 'R';
336     res.index = r1;
337     res.value = reg_array[r2] >> shamt;
338     res.isValid = true;
339     res.dropFlag = false;
340
341     return res;
342 }
343
344 mem_reg_destinationInfo movr(int r1, int r2, int offset) {
345     int addr = reg_array[r2] + offset;
346
347     mem_reg_destinationInfo res;
348     res.type = 'B';
349     res.index = r1;
350     res.value = addr; // overwrite the value for the address in the memory
        function to get memory[addr]
351     res.isValid = true;
352     res.dropFlag = false;
```

```
353
354     return res;
355 }
356
357 mem_reg_destinationInfo movm(int r1, int r2, int offset) {
358     int addr = reg_array[r2] + offset;
359
360     mem_reg_destinationInfo res;
361     res.type = 'M';
362     res.index = addr;
363     res.value = reg_array[r1];
364     res.isValid = true;
365     res.dropFlag = false;
366
367     return res;
368 }
369
370 mem_reg_destinationInfo MEM(mem_reg_destinationInfo dest){
371     mem_reg_destinationInfo res;
372
373     if((flag && dropFlagCount < 2) || dest.isValid == false){
374         res.isValid = false;
375         res.dropFlag = false;
376         return res;
377     }
378
379     if(dest.type == 'B'){
380         res.type = 'R';
381         res.index = dest.index;
382         res.value = memory[dest.value+memoryDataCounter];
383         printf("    Output: MEM: Read from memory[%d] = %d to be written in
                R%d\n", dest.value, res.value, res.index);
384         res.isValid = true;
385         res.dropFlag = false;
386     } else if(dest.type == 'M'){
387         memory[dest.index + memoryDataCounter] = dest.value;
388         printf("    Output: MEM: Stored %d --> memory[%d]\n", dest.value,
               dest.index);
389         res.type = dest.type;
390         res.index = dest.index;
391         res.value = dest.type;
392         res.isValid = false;
393         res.dropFlag = false;
394     } else{
395         printf("    Output: nothing read or written into memory\n", dest.
               value, dest.index);
396         return dest;
397     }
398
399     return res;
400 }
401
402 bool writeBack(mem_reg_destinationInfo dest){
403     if(dest.isValid == false || dest.type != 'R'){
404         printf("    Output: REG: nothing was loaded into a register\n");
405         return false;
406     }
407     if(dest.index != 0 && dest.index != 32 && dest.type == 'R'){
```

```
408         reg_array[dest.index] = dest.value;
409     }
410 
411     printf("    Output: REG: Loaded R%d --> %d\n", dest.index, dest.value);
412 
413     if(dest.dropFlag == true){
414         flag = true;
415     }
416 
417     return true;
418 }
```

Listing 4: datapath.c: Datapath Implementation

A.5 executor.h

```
1  #ifndef EXECUTOR_H
2  #define EXECUTOR_H
3  
4  #include "datapath.h"
5  
6  extern int dropFlagCount;
7  
8  typedef struct{
9      int instruction; // stores output of fetch
10     decodedInstruction d1; // stores output of decode cycle 1
11     decodedInstruction d2; // stores output of decode cycle 2
12     int opcode; // stores output of execute cycle 1
13     mem_reg_destinationInfo output; //stores output of execute cycle 2
14     mem_reg_destinationInfo mem; // stores output of memory
15     bool wb; // stores output of writeback
16     int count; // stores count of execution cycle
17     bool droped; // true if the instruction has been dropped
18 } storeOutput;
19 
20 const char* instrTypeToStr(InstrType type);
21 
22 // Function to start pipeline execution
23 void pipelineExecution();
24 
25 #endif // EXECUTOR_H
```

Listing 5: executor.h: Executor Definitions

A.6 executor.c

```
1  #include "stdio.h"
2  #include "stdbool.h"
3  #include "executor.h"
4  #include "datapath.h"
5  #include "hardware.h"
6  
7  const char* instrTypeToStr(InstrType type) {
8      switch (type) {
9          case R_TYPE: return "R_TYPE";
10         case I_TYPE: return "I_TYPE";
```

```
11          case J_TYPE: return "J_TYPE";
12          default: return "UNKNOWN";
13      }
14  }
15
16  void pipelineExecution(){
17      int clk = 1;
18      flag = false;
19      readFile();
20
21      storeOutput instArray[1024];
22      int instArrayPointer = 0;
23      int stopCount = 0;
24
25      do {
26          printf("\n");
27          printf("------------------- clk cycle = %d ---------------\n", clk)
                ;
28          printf("\n");
29
30          if(clk % 2 != 0){
31              int fetchOutput = fetch();
32
33              if(fetchOutput != -1){
34                  storeOutput tmp = {0};
35                  tmp.instruction = fetchOutput;
36                  tmp.count = 1;
37                  instArray[instArrayPointer++] = tmp;
38              }
39
40              if(flag){
41                  for(int j=0; j<instArrayPointer; j++){
42                      storeOutput tmp2 = instArray[j];
43                      if(flag && tmp2.count != 6 && tmp2.instruction !=
                            fetchOutput && tmp2.count != 7 && tmp2.droped ==
                            false){
44                          printf("Instruction 0x%08X has been droped\n", tmp2
                                .instruction);
45                          stopCount++;
46                          tmp2.droped = true;
47                      }
48                      instArray[j] = tmp2;
49                  }
50          }
51
52              for(int i=0; i<instArrayPointer; i++){
53                  storeOutput tmp2 = instArray[i];
54                  if(tmp2.count != 7 && tmp2.droped == false){
55                      if(tmp2.count == 2){
56                          printf("Instruction 0x%08X going through decode
                                cycle 2\n", tmp2.instruction);
57                          printf("    Decode cycle 2 input variable:\n
                                opcode = %d, r1 = %d, r2 = %d, r3 = %d, shamt =
                                %d, immediate = %d, address = %d\n", tmp2.d1.
                                opcode, tmp2.d1.r1, tmp2.d1.r2, tmp2.d1.r3, tmp2
                                .d1.shamt, tmp2.d1.immediate, tmp2.d1.address);
58                          tmp2.d2 = decode_cycle2(tmp2.d1);
```

```
59                      printf("    Output: type = %s\n", instrTypeToStr(
                            tmp2.d2.type));
60                      tmp2.count++;
61                  } else if(tmp2.count == 4){
62                      printf("Instruction 0x%08X going through execute
                            cycle 2\n", tmp2.instruction);
63                      printf("    Execute cycle 2 input variable:\n
                            opcode = %d, r1 = %d, r2 = %d, r3 = %d, shamt =
                            %d, immediate = %d, address = %d, type = %s\n",
                            tmp2.d2.opcode, tmp2.d2.r1, tmp2.d2.r2, tmp2.d2.
                            r3, tmp2.d2.shamt, tmp2.d2.immediate, tmp2.d2.
                            address, instrTypeToStr(tmp2.d2.type));
64                      tmp2.output = execute_cycle2(tmp2.opcode,tmp2.d2);
65                      if(tmp2.output.type == 'R'){
66                          printf("    Output: value %d will be written in
                                register %d\n", tmp2.output.value, tmp2.
                                output.index);
67                      } else if(tmp2.output.type == 'M'){
68                          printf("    Output: value %d will be written in
                                memory at location %d\n", tmp2.output.value
                                , tmp2.output.index);
69                      } else{
70                          printf("    Output: value to be written in
                                register %d will be read from memory
                                location %d\n", tmp2.output.index, tmp2.
                                output.value);
71                      }
72                      if(flag){
73                          dropFlagCount++;
74                      }
75                      tmp2.count++;
76                  } else if(tmp2.count == 6){
77                      printf("Instruction 0x%08X going through write back
                            \n", tmp2.instruction);
78                      if(tmp2.mem.type = 'R'){
79                          printf("    Write back input: value %d will be
                                written into register %d\n", tmp2.mem.value,
                                 tmp2.mem.index);
80                      } else{
81                          printf("    Write back input: nothing will be
                                written into a register\n");
82                      }
83                      tmp2.wb = writeBack(tmp2.mem);
84                      printf("    Instruction 0x%08X finished execution\n
                            ");
85                      tmp2.count++;
86                      stopCount++;
87                  }
88                  instArray[i] = tmp2;
89              }
90          }
91
92      } else{
93          if(flag && dropFlagCount == 2){
94              flag = false;
95              dropFlagCount = 0;
96          } else if(dropFlagCount == 1){
97              dropFlagCount++;
```

```
98              }
99              for(int i=0; i<instArrayPointer; i++){
100                 storeOutput tmp2 = instArray[i];
101
102                 if(tmp2.count != 7 && tmp2.droped == false){
103                     if(tmp2.count == 1){
104                         printf("Instruction 0x%08X going through decode
                            cycle 1\n", tmp2.instruction);
105                         printf("    Decode cycle 1 input variable:  \n
                            instruction = 0x%08x\n", tmp2.instruction);
106                         tmp2.d1 = decode_cycle1(tmp2.instruction);
107                         tmp2.count = tmp2.count + 1;
108                         printf("    Output variables: opcode = %d, r1 = %d,
                             r2 = %d, r3 = %d, shamt = %d, immediate = %d,
                            address = %d \n", tmp2.d1.opcode, tmp2.d1.r1,
                            tmp2.d1.r2, tmp2.d1.r3, tmp2.d1.shamt, tmp2.d1.
                            immediate, tmp2.d1.address);
109                     } else if(tmp2.count == 3){
110                         printf("Instruction 0x%08X going through execute
                            cycle 1\n", tmp2.instruction);
111                         printf("    Execute cycle 1 input variables: \n
                            type = %s\n", instrTypeToStr(tmp2.d2.type));
112                         tmp2.opcode = execute_cycle1(tmp2.d2);
113                         printf("    Output: opcode = %d\n", tmp2.opcode);
114                         tmp2.count++;
115                     } else if(tmp2.count == 5){
116                         printf("Instruction 0x%08X going through memory
                            read or write\n", tmp2.instruction);
117                         if(tmp2.output.type == 'M'){
118                             printf("    Memory read or write input:\n
                                value %d to be stored in memory at location
                                %d\n", tmp2.output.value, tmp2.output.index)
                                ;
119                         } else if(tmp2.output.type == 'B'){
120                             printf("    Memory read or write input:\n
                                value to be stored in register %d to be read
                                 from memory at location %d\n", tmp2.output.
                                index , tmp2.output.value);
121                         } else{
122                             printf("    Memory read or write input: nothing
                                 will be read or written in memory\n");
123                         }
124
125                         tmp2.mem = MEM(tmp2.output);
126                         tmp2.count++;
127                     }
128                 }
129
130                 instArray[i] = tmp2;
131             }
132         }
133
134         clk++;
135     } while (stopCount < instArrayPointer);
136
137     printf("\n");
138     printf("Pipeline finished execution at clk cycle %d\n", clk-1);
139 }
```

Listing 6: executor.c: Executor Implementation

A.7 main.c

```c
#include <stdio.h>
#include <stdbool.h>
#include "datapath.h"
#include "hardware.h"
#include "executor.h"
#include <stdint.h>  // For fixed-width types like int32_t
#include <limits.h>  //  This is what you need for INT32_MAX, INT32_MIN, and UINT32_MAX

void print_registers() {
    printf("=== Registers ===\n");
    for (int i = 0; i < 32; i++) {
        printf("R[%d] = %d\n", i, reg_array[i]);
    }
    printf("PC = %d\n", reg_array[32]);
}

void print_memory() {
    printf("\n=== Memory [1024+] ===\n");
    for (int i = 1024; i < memoryDataCounter; i++) {
        printf("mem[%d]=%d  ", i, memory[i]);
        if ((i - 1024 + 1) % 5 == 0)  // Print 5 memory cells per line
            printf("\n");
    }
   printf("\n");
}

int main(){
    pipelineExecution();

    // Results
    printf("\n=== Registers ===\n");
    for (int i = 0; i < 32; i++)
        printf("R[%d] = %d\n", i, reg_array[i]);
    printf("PC = %d\n", reg_array[32]);

    printf("\n=== Memory [1024+] ===\n");
    for (int i = 1024; i < 1100; i++)
        printf("mem[%d] = %d\n", i, memory[i]);

    return 0;
}
```

Listing 7: main.c: Main Program

A.8 parser.h

```c
#ifndef PARSER_H
#define PARSER_H

#include <stdio.h>
```

```
5
6  #define MAX_LINE_LENGTH 100
7
8  // Opcode enumeration
9  typedef enum {
10     ADD = 0,
11     SUB,
12     MUL,
13     MOVI,
14     JEQ,
15     AND,
16     XORI,
17     JMP,
18     LSL,
19     LSR,
20     MOVR,
21     MOVM
22 } Opcode;
23
24 // Instruction type enumeration
25 typedef enum {
26     R_TYPE,
27     I_TYPE,
28     J_TYPE
29 } InstrType;
30
31 // Instruction structure
32 typedef struct {
33     char mnemonic[6];
34     Opcode opcode;
35     InstrType type;
36 } InstructionInfo;
37
38 // Function declarations
39 Opcode getOpcode(const char *mnemonic);
40 InstrType getInstrType(const char *mnemonic);
41 void intToBinary(int value, int bits, char *output);
42 int getRegisterNumber(const char *reg);
43 void encodeInstruction(const char *mnemonic, char *arg1, char *arg2, char *
       arg3);
44 void parseInstruction(char *line);
45 void readFile();
46
47 #endif // PARSER_H
```

Listing 8: parser.h: Parser Definitions

A.9 parser.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "parser.h"
5  #include "hardware.h"
6
7  InstructionInfo instructionSet[] = {
8      {"ADD", ADD, R_TYPE},
```

```
    {"SUB", SUB, R_TYPE},
    {"MUL", MUL, R_TYPE},
    {"MOVI", MOVI, I_TYPE},
    {"JEQ", JEQ, I_TYPE},
    {"AND", AND, R_TYPE},
    {"XORI", XORI, I_TYPE},
    {"JMP", JMP, J_TYPE},
    {"LSL", LSL, R_TYPE},
    {"LSR", LSR, R_TYPE},
    {"MOVR", MOVR, I_TYPE},
    {"MOVM", MOVM, I_TYPE}
};

Opcode getOpcode(const char *mnemonic) {
    for (int i = 0; i < 12; i++) {
        if (strcmp(mnemonic, instructionSet[i].mnemonic) == 0)
            return instructionSet[i].opcode;
    }
    return -1;
}

InstrType getInstrType(const char *mnemonic) {
    for (int i = 0; i < 12; i++) {
        if (strcmp(mnemonic, instructionSet[i].mnemonic) == 0)
            return instructionSet[i].type;
    }
    return -1;
}

void intToBinary(int value, int bits, char *output) {
    output[bits] = '\0';
    for (int i = bits - 1; i >= 0; i--) {
        output[i] = (value & 1) + '0';
        value >>= 1;
    }
}

int getRegisterNumber(const char *reg) {
    if (reg[0] == 'R')
        return atoi(reg + 1);
    return 0;
}

int binaryStringToInt(const char *binary) {
    int result = 0;
    while (*binary != '\0') {
        result = (result << 1) | (*binary - '0');
        binary++;
    }
    return result;
}

void encodeInstruction(const char *mnemonic, char *arg1, char *arg2, char *
    arg3) {
    char binary[33]; // 32 bits + null terminator
    Opcode opcode = getOpcode(mnemonic);
    InstrType type = getInstrType(mnemonic);

```

```
    char opBin[5], r1Bin[6], r2Bin[6], r3Bin[6], shamtBin[14], immBin[19],
        addrBin[29];

    intToBinary(opcode, 4, opBin);

    if (type == R_TYPE) {
        intToBinary(getRegisterNumber(arg1), 5, r1Bin);
        intToBinary(getRegisterNumber(arg2), 5, r2Bin);
        intToBinary(getRegisterNumber(arg3), 5, r3Bin);

        if (strcmp(mnemonic, "LSL") == 0 || strcmp(mnemonic, "LSR") == 0)
            intToBinary(atoi(arg3), 13, shamtBin);
        else
            intToBinary(0, 13, shamtBin);

        sprintf(binary, "%s%s%s%s%s", opBin, r1Bin, r2Bin, r3Bin, shamtBin)
            ;
    }

    else if (type == I_TYPE) {
        intToBinary(getRegisterNumber(arg1), 5, r1Bin);
        int imm;
        if (arg3 == NULL) {
            intToBinary(0, 5, r2Bin); // For MOVI
            imm = atoi(arg2);
        } else {
            intToBinary(getRegisterNumber(arg2), 5, r2Bin);
            imm = atoi(arg3);
        }

        if (imm < 0)
            imm = (1 << 18) + imm; // Two's complement for negative numbers

        intToBinary(imm, 18, immBin);
        sprintf(binary, "%s%s%s%s", opBin, r1Bin, r2Bin, immBin);
    }

    else if (type == J_TYPE) {
        int addr = atoi(arg1);
        if (addr < 0)
            addr = (1 << 28) + addr;

        intToBinary(addr, 28, addrBin);
        sprintf(binary, "%s%s", opBin, addrBin);
    }

    printf("Binary Encoding: %s\n", binary);

    memory[memoryInstructionCounter++] = binaryStringToInt(binary);
}

void parseInstruction(char *line) {
    char *mnemonic = strtok(line, " \n");
    char *arg1 = strtok(NULL, " \n");
    char *arg2 = strtok(NULL, " \n");
    char *arg3 = strtok(NULL, " \n");

    if (!mnemonic)
```

```
122         return;
123 
124     printf("Parsed Instruction: %s %s %s %s\n", mnemonic,
125            arg1 ? arg1 : "-", arg2 ? arg2 : "-", arg3 ? arg3 : "-");
126 
127     encodeInstruction(mnemonic, arg1, arg2, arg3);
128 }
129 
130 void readFile() {
131     FILE *file = fopen("dummy_instructions.txt", "r");
132     if (!file) {
133         perror("Could not open file");
134     }
135 
136     char line[MAX_LINE_LENGTH];
137     while (fgets(line, sizeof(line), file)) {
138         parseInstruction(line);
139     }
140 
141     fclose(file);
142 }
```

Listing 9: parser.c: Parser Implementation