EAST WEST UNIVERSITY

Department of CSE

Course Code
CSE360

Course Title
Computer Architecture

Project Title

Multiplication Game Problem

Section : 01 **Semester : Spring2025**

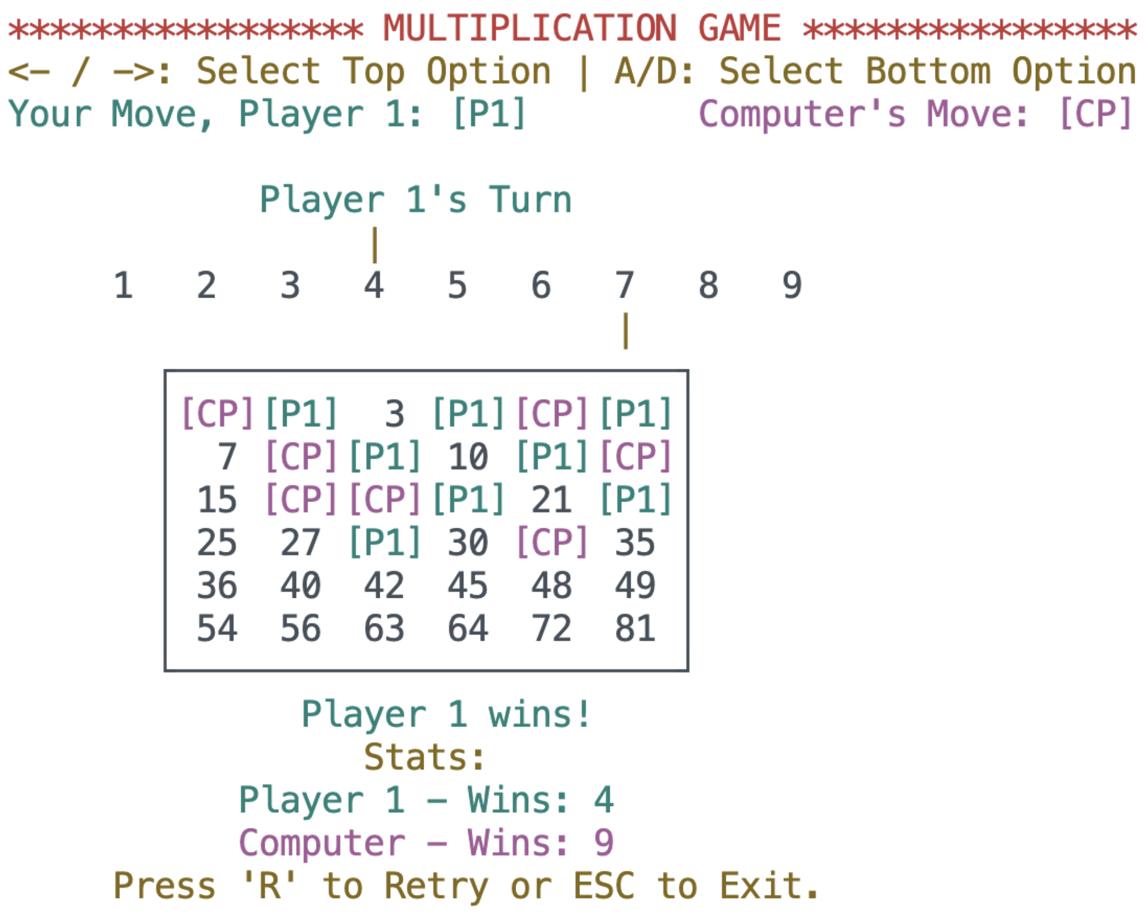
Submitted By

Name	ID	Roll
Abrar Hossain Zahin	2022-2-60-040	25

Submitted To

Dr. Nawab Yousuf Ali
Professor
Department of Computer Science and Engineering

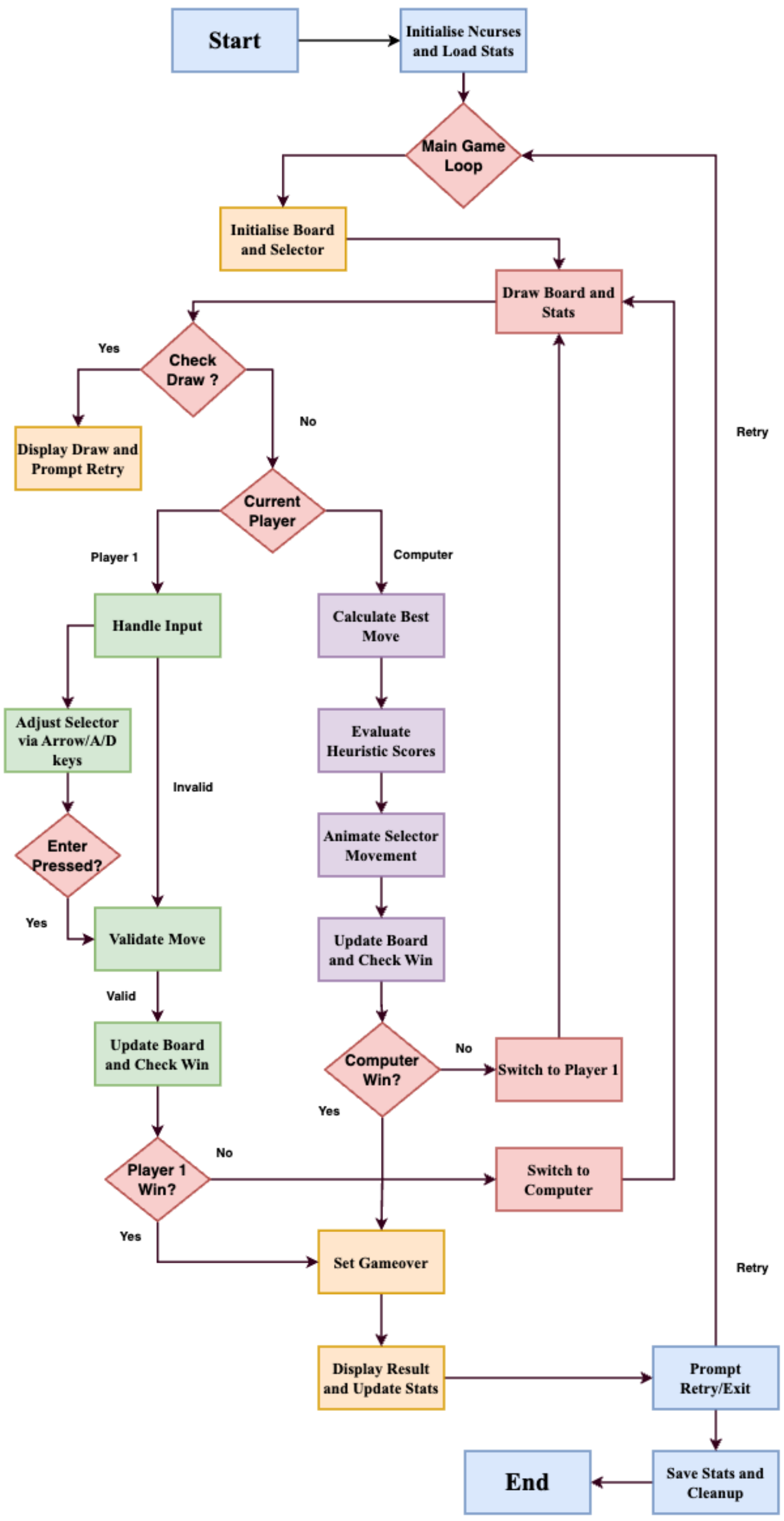
Date of Report Submitted : 22 May, 2025



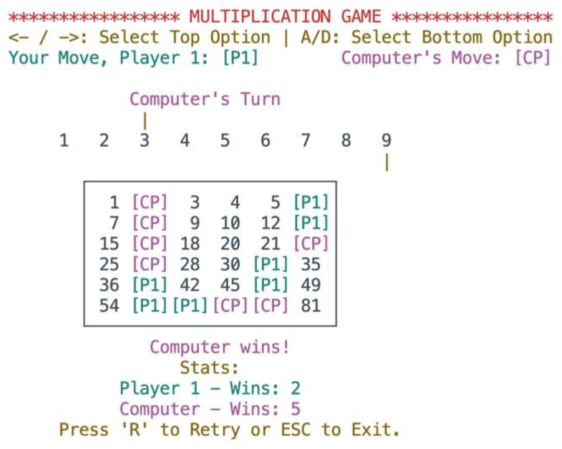
```
***************** MULTIPLICATION GAME ****************
<- / ->: Select Top Option | A/D: Select Bottom Option
Your Move, Player 1: [P1]        Computer's Move: [CP]

            Player 1's Turn
                   |
     1   2   3   4   5   6   7   8   9
                               |
       ┌─────────────────────────┐
       │[CP][P1]  3 [P1][CP][P1] │
       │  7 [CP][P1] 10 [P1][CP] │
       │ 15 [CP][CP][P1] 21 [P1] │
       │ 25  27 [P1] 30 [CP] 35  │
       │ 36  40  42  45  48  49  │
       │ 54  56  63  64  72  81  │
       └─────────────────────────┘
              Player 1 wins!
                 Stats:
           Player 1 - Wins: 4
           Computer - Wins: 9
     Press 'R' to Retry or ESC to Exit.
```

Flowchart


Start
Initialise Ncurses and Load Stats
Main Game Loop
Initialise Board and Selector
Draw Board and Stats
Check Draw ?
Yes
No
Display Draw and Prompt Retry
Current Player
Player 1
Computer
Handle Input
Adjust Selector via Arrow/A/D keys
Enter Pressed?
Yes
Invalid
Validate Move
Valid
Update Board and Check Win
Player 1 Win?
No
Yes
Calculate Best Move
Evaluate Heuristic Scores
Animate Selector Movement
Update Board and Check Win
Computer Win?
No
Yes
Switch to Player 1
Switch to Computer
Set Gameover
Display Result and Update Stats
Prompt Retry/Exit
Retry
Retry
Save Stats and Cleanup
End

Graphical Visualization



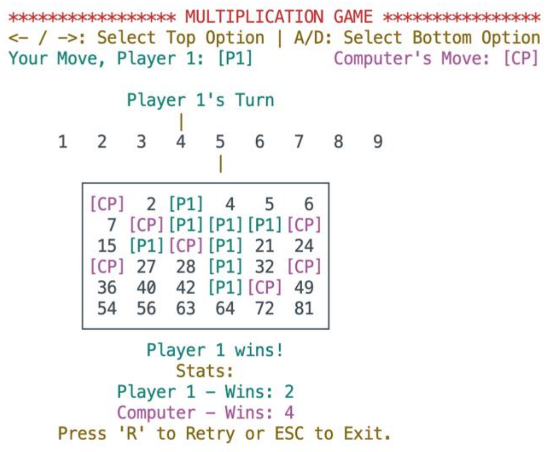
```
***************** MULTIPLICATION GAME *****************
<- / ->: Select Top Option | A/D: Select Bottom Option
Your Move, Player 1: [P1]        Computer's Move: [CP]

            Computer's Turn
              |
      1   2   3   4   5   6   7   8   9
                                      |

          1 [CP]  3   4   5 [P1]
          7 [CP]  9  10  12 [P1]
         15 [CP] 18  20  21 [CP]
         25 [CP] 28  30 [P1] 35
         36 [P1] 42  45 [P1] 49
         54 [P1][P1][CP][CP] 81

              Computer wins!
                 Stats:
            Player 1 - Wins: 2
            Computer - Wins: 5
      Press 'R' to Retry or ESC to Exit.
```

Computer Win By Vertical



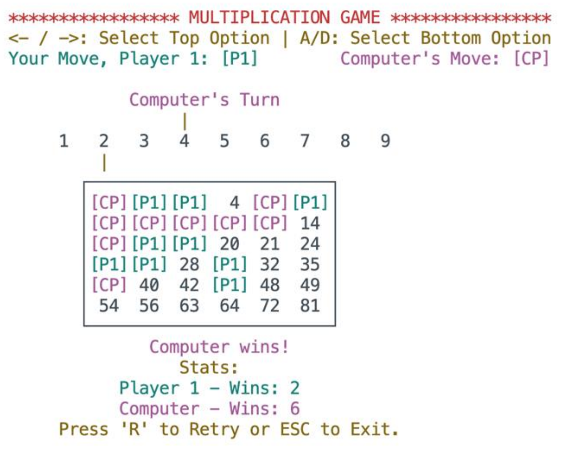
```
***************** MULTIPLICATION GAME *****************
<- / ->: Select Top Option | A/D: Select Bottom Option
Your Move, Player 1: [P1]        Computer's Move: [CP]

            Player 1's Turn
                  |
      1   2   3   4   5   6   7   8   9
                      |

        [CP]  2 [P1]  4   5   6
          7 [CP][P1][P1][P1][CP]
         15 [P1][CP][P1] 21  24
        [CP] 27  28 [P1] 32 [CP]
         36  40  42 [P1][CP] 49
         54  56  63  64  72  81

              Player 1 wins!
                 Stats:
            Player 1 - Wins: 2
            Computer - Wins: 4
      Press 'R' to Retry or ESC to Exit.
```

Player 1 Win By Vertical



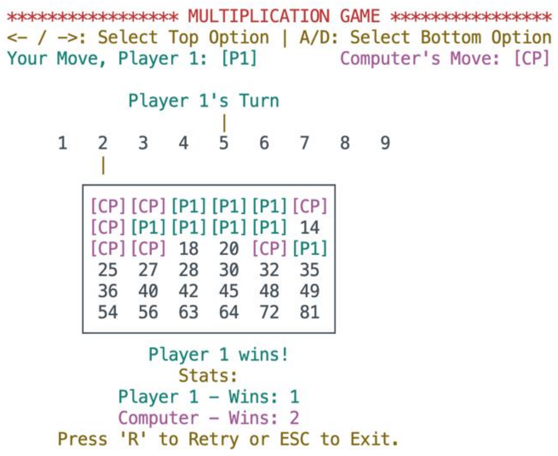
```
***************** MULTIPLICATION GAME *****************
<- / ->: Select Top Option | A/D: Select Bottom Option
Your Move, Player 1: [P1]        Computer's Move: [CP]

            Computer's Turn
                  |
      1   2   3   4   5   6   7   8   9
          |

        [CP][P1][P1]  4 [CP][P1]
        [CP][CP][CP][CP][CP] 14
        [CP][P1][P1] 20  21  24
        [P1][P1] 28 [P1] 32  35
        [CP] 40  42 [P1] 48  49
         54  56  63  64  72  81

              Computer wins!
                 Stats:
            Player 1 - Wins: 2
            Computer - Wins: 6
      Press 'R' to Retry or ESC to Exit.
```

Computer Win By Horizontal



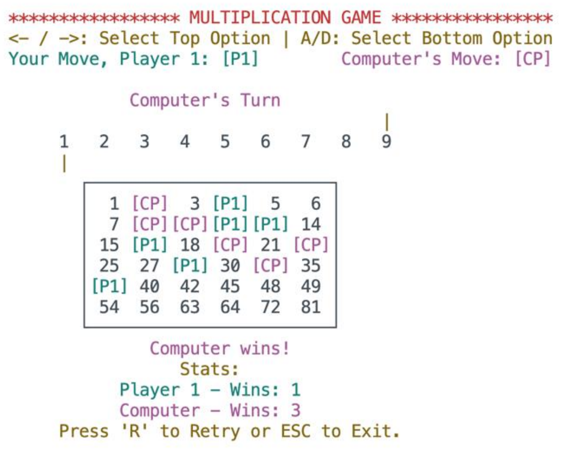
```
***************** MULTIPLICATION GAME *****************
<- / ->: Select Top Option | A/D: Select Bottom Option
Your Move, Player 1: [P1]        Computer's Move: [CP]

            Player 1's Turn
                      |
      1   2   3   4   5   6   7   8   9
          |

        [CP][CP][P1][P1][P1][CP]
        [CP][P1][P1][P1][P1] 14
        [CP][CP] 18  20 [CP][P1]
         25  27  28  30  32  35
         36  40  42  45  48  49
         54  56  63  64  72  81

              Player 1 wins!
                 Stats:
            Player 1 - Wins: 1
            Computer - Wins: 2
      Press 'R' to Retry or ESC to Exit.
```

Player 1 Win By Horizontal



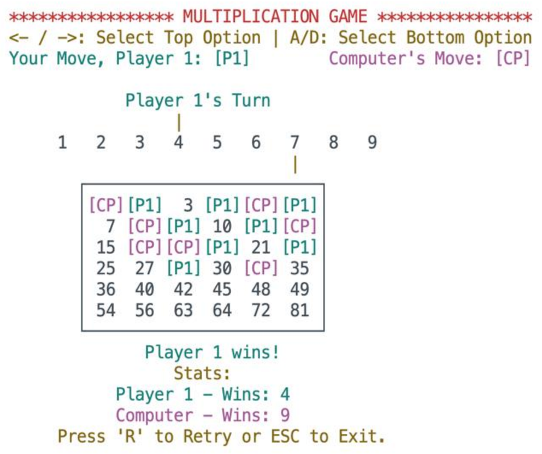
```
***************** MULTIPLICATION GAME *****************
<- / ->: Select Top Option | A/D: Select Bottom Option
Your Move, Player 1: [P1]        Computer's Move: [CP]

            Computer's Turn
                                      |
      1   2   3   4   5   6   7   8   9
      |

          1 [CP]  3 [P1]  5   6
          7 [CP][CP][P1][P1] 14
         15 [P1] 18 [CP] 21 [CP]
         25  27 [P1] 30 [CP] 35
        [P1] 40  42  45  48  49
         54  56  63  64  72  81

              Computer wins!
                 Stats:
            Player 1 - Wins: 1
            Computer - Wins: 3
      Press 'R' to Retry or ESC to Exit.
```

Computer Win By Diagonal



```
***************** MULTIPLICATION GAME *****************
<- / ->: Select Top Option | A/D: Select Bottom Option
Your Move, Player 1: [P1]        Computer's Move: [CP]

            Player 1's Turn
                  |
      1   2   3   4   5   6   7   8   9
                              |

        [CP][P1]  3 [P1][CP][P1]
          7 [CP][P1] 10 [P1][CP]
         15 [CP][CP][P1] 21 [P1]
         25  27 [P1] 30 [CP] 35
         36  40  42  45  48  49
         54  56  63  64  72  81

              Player 1 wins!
                 Stats:
            Player 1 - Wins: 4
            Computer - Wins: 9
      Press 'R' to Retry or ESC to Exit.
```

Player 1 Win By Diagonal

Multiplication Game Overview

The Multiplication Game is two-player (Human vs Computer) connect-four–style game on a 6×6 grid of multiplication products. The grid (board [6][6]) is pre-filled with integer products (e.g. 1,2,3,…28,32…,81 as given in the code). Players take turns choosing two factors: one from the “top” selector (using left/right arrow keys) and one from the “bottom” selector (using ‘a’/‘d’). When the user presses Enter, the chosen product (top*bottom) is checked against the board. If a matching board cell is empty, that cell is marked for the current player. The game cheaks for four in a row **horizontally** or **vertically** or **diagonally** to detect a win. On the computer’s turn, a simple AI scans all valid empty products and uses a heuristic (based on extending or blocking lines) to pick the best move. The interface is drawn in the terminal ASCII graphics using the NCURSES library: key presses (getch()) control the selectors, and printw/refresh update the ASCII grid. Player scores and moves are tracked between games. NCURSES is “a library of functions that manages an application’s display on terminals”, which the program uses for text-based I/O.

1. Architectural Principles Applied

- **Register and ALU Simulation:** The code uses a RegisterSet struct (Reg.A, Reg.B, Reg.RES, Reg.FLAG) to model CPU registers and an arithmetic logic unit. For Instance, in make_move() it loads the board value into Reg.A and the chosen value into reg.B, then checks if (reg.A = = reg.B), writing the player id into reg.RES and storing it to memory. Board values and scores can be loaded into "register" and processed (e.g., cheaking regs[0] + regs[1] == 4 to detect win). This imitate an ALU comparing register operands and storing a result. In hardware, a CPU's ALU "performs arithmetic and bitwise operations on integer binary numbers" using its registers. Here, setting Reg.FLAG during the win check similarly mimic a CPU's condition flag. In fact, processor registers are at the top of the memory hierarchy (the fastest storage), so using a struct to hold recent values is conceptually like keeping data in registers for rapid operations.

- **Memory Organization:** The 6*6 board and marks arrays are stored contiguously in row-major order in memory. In C program, a 2D array is laid out row by row with no gaps. Conceptually, the flattened board in memory looks like:

  Memory (board array, row-major):

  ```
  Index:  0  1  2  3  4  5  6  7  8  9 .......... 35
  Value:  1  2  3  4  5…..28 30 32 …….. 81
  ```

  Each integer occupies 4 bytes, so consecutive indices map to consecutive memory addresses. Using contiguous static arrays aids cache performance and makes indexing predictable. All global data (board, marks, player structs) reside in the program's data segment. Local variables and function call state reside on the stack, implementing the usual call/return flow.

- **Instruction-Level Logic:** The program's C control flow (loops, conditionals, function calls) directly translates to CPU instructions. For instance, the loop that checks for a win iterates through rows/columns, using for loops and if statements. In hardware this corresponds to repeated load-comapre-branch instructions. When the code does if (reg.A != player) reg.FLAG = 0;, it simulates a CPU comparing a register to a value and clearing the flag flag bit on inequality. The sequential execution model mirrors the fetch–decode-execute the next one is started" in a simple implementation. Branching on check_winner results corresponds to how processors handle conditional jumps.

- **I/O Simulation:** The Use of NCURSES abstracts terminal I/O but can be viewed as an I/O device interface. The program writes charecters to a screen buffer and calls refresh(), analogous to memory-mapped display where writing to a buffer eventually updates the physical screen. Keyboard input via getch() is processed in a loop; in hardware terms this is akin to handling keyboard interrupts or polled input from an I/O port. In summary, the interface code simulates reading/writing I/O devices in a text-based environment.

- **Performance-Aware Design (Heuristic AI):** The computer player uses a heuristic evalution to choose moves, reflectling low-level design for efficiency. The grid is small (36 cells), so all scanning and updates are very fast. The win-cheak logic is $O(n^3)$ nested loops over the 6×6 grid. A minor optimization is using bitwise shift to compute scores: the heuristic adds (1 << count) which is an efficient way to compute powers of two (a single CPU shift instruction). This avoids slower operations like loops or library pow functions.

No heavy optimizations (caching results or parallelism) were needed given the scale, but the code's use of static arrays and simple loops maps well onto hardware caches and ALU pipelines.

2. Development Challenges and Solutions

- **Terminal I/O and NCURSES Setup:** Getting the ncurses interface right was tricky. The program must initialize ncurses (initscr(), cbreak(), noecho(), etc.) and handle special keys. We solved this by carefully configuring ncurses (enabling keypad() and non-blocking input modes). Animating the computer's choice requried repeatedly redrawing the board while adjusting top_selector and bottom_selector, which we managed with delays (usleep) and screen refreshes.

- **Tracking game State:** It was challenging to remember the last chosen factors between turns. We introduced fiels in the player struct (prev_top, prev_bottom) to store the last-used selectors for each player. This allowed the next turn start from the correct values, improving usability.

- **Valid Move Detection:** Ensuring only valid unsed products were chosen requried carefully cheaking. We Wrote an is_valid_cell() function that scans the board to see if a product is both on the board and unmarked. This prevents illegal moves and simplified move logic.

- **AI Heuristic Implimentation:** Designing a basic yet effective computer opponent was challenging. We implemented a heuristic that counts possible lines (own vs opponent) in each direction. Using bitwise shifts to weight these counts (1 << player_count) gave an exponentially higher score for longer lines, effectively prioritizing winning moves. Turning this heuristic required trial and error, but it worked well for a simple AI.

- **File I/O and State Persistence :** For extra credit, saving game stats across runs was requried. Implimenting file reading/writing introduced I/O edge cases. We solved issues like file-not-found by initializing defualt vlaues and creating a new stats file on first run. Ensuring the file is written upon program exit.

3. Insights on Computer Architecture

Implimenting this game deepened our understanding of the CPU's internal model. Using the RegisterSet struct made it clear how a CPU loads operands into registers and produces a result: we explicitly moved values into Reg.A/Reg.B and computed Reg.RES. We realized how condition codes are used: for example, setting reg.FLAG in the win check emulates the zero/compare flag that a real CPU would set after an operation. Working with arrays highlighted memory layouts: we saw that the 2D board is actually a contiguous block of memory, acccessed via calculated addresses row-major indexing). The project also illustrated the instruction cycle in action – our C loops and branches represent what a CPU does each clock cycle. For instance, the for-loop combined with if conditions is essentially a sequence of fetch–decode-execute steps, similar to the sequential model "each instruction being processed before the next one.
Another insight was about performance at the hardware level. We noted that bitwise shifts and simple integer math ALU instruction. Since registers and caches are much faster than main memory, storing frequently accessed data (like the current selectors or recent move) in register-like variables is ideal. In our small game, everything comfortably fits in cache, but this reinforced the idea of a memory hierarchy (registers → cache → RAM) and how data locality matters.

4. Low-Level Simulation and Optimizations

Several low-level design choices simulate CPU behavior or optimize performance. The RegisterSet is a direct simulation of CPU's registers and ALU inputs. During move-making and win-checking, we only ever write to memory via reg.RES, as if executing a single CPU mov instruction for the result. Logical comparisons set reg.FLAG, similar to setting a CPU flag register used by conditional jumps.
Performance-wise, we used bitwise shifts for scoring (efficient ALU operations) rather than computing powers of two with loops. The code avoids expensive operations: there are no floating-point or complex data structures, only integer arithmetic and array indexing. Array accesses (board[i][j]) compile to simple address calculations (base + offset) in a single instruction or two. We also minimized branching: for example, the heuristic loop uses simple if-conditions without function calls, allowing potential CPU pipelining. In a more advanced setting, one could further optimize by precalculating all possible moves or using bitboards, but for this scale the straightforward approach sufficed.

5. Future Improvements

Future work could include extending the game and its architecture simulation. Introduce multiple difficulty levels by varying win condition. Implement networked multiplayer so player can challenge other online friends. Allowing users to choose between aggressive, defensive or balanced computer opponent. Provide detailed post-game analytics, including heat maps of chosen cells, move-by-move reply and avarage time.The AI could be improved with deeper search (minimax) or caching (memoization) of board states. One could simulate even lower-level details: for example, manually managing a bit-level representation of the board to use bitwise AND/OR for for faster win-detection. Profiling the program would help identify any bottlenecks on larger boards. Finally, porting the I/O to a graphics library or network interface would test memory and I/O design on different hardware paradigms.

1. Memory Layout(Static Arrays)

```
Static Data
board[6][6]  - 36 ints            <-- Pre-filled multiplication table
marks[6][6]  - 36 ints            <-- Stores player marks (0, 1, 2)
player1
  └── wins, moves, top, bottom
player2
  └── wins, moves, top, bottom
reg
  └── A, B, RES, FLAG
        ↓
Stack
Function call frames
Local variables
Return addresses
```

Memory Segment :

Code(Text)	
Static Data	<- global arrays, structs
Heap	<- dynamic alloc (unused here)
Stack	<- grows downward

2. Register Simulation Model

RegisterSet Reg	Example (in make_move):

A → Operand 1 (board[i][j]
B → Operand 2 (user input
RES → Result (player ID)
FLAG → Status (match found, win
	reg.A = board[i][j]
	reg.B = selected_value
	if (reg.A == reg.B) {
	reg.RES = current_player
	marks[i][j] = reg.RES }
	
	Emulates this CPU behavior:
	LOAD R1, board[i][j]
	LOAD R2, selected_value
	CMP R1, R2
	SET FLAG = (R1 == R2)
	MOV marks[i][j], current_player

3. Instruction Cycle (Game Turn Simulation):

Fetch	← Read input (keyboard or AI)
Decode	← Determine action (move selector, select cell)
Execute	← Multiply factors, validate move
Memory Access	← Update marks[i][j], stats, registers
Write Back	← Redraw board, update score, switch player
Check Interrupt	← ESC key, 'r', or win condition

Implimentation Walkthrough

The following provides a perfect overview of the source code,

- **Initilization:**

  - initscr(); start_color(); – Initializes the ncurses screen and color pairs.
  - cbreak(), noecho(), keypad(stdscr, TRUE) – Configures input (no line buffering, capture special keys).
  - load_stats(); – Reads previous game statistics from file.

- **Main Game Loop:**

```
while (true) {
   init_game();                          // Clears board state and resets variables.
   current_player = PLAYER1;
   game_over = 0;
   while (!game_over) {
      clear();                              // Clears the screen buffer.
      draw_board(current_player);           // Renders the board and UI.
      show_stats();                         // Displays scores.
      if (check_draw()) {
         mvprintw(..., "Game is a draw!"); getch(); break;
      }
      if (current_player == PLAYER1) {
         handle_input(&current_player, &game_over);
      } else {
         computer_move(&game_over);
      }
   }
   save_stats();                         // Update stats file if game ended.
   if (!prompt_retry())                  // Ask user Y/N to play again.
      break;                             // Exit if no retry.
}
endwin();                                // End ncurses mode.
```

This loop continues until the user quits. After each move, it checks for a win or draw. The draw_board() function prints the 6×6 grid and highlights the current selection with a cursor.

- **Input Handling**

  - Captures arrow key and Enter presses (getch()).
  - Updates top_selector/bottom_selector variables to move the selection box.
  - When Enter/Space is pressed, it calls make_move(board[top_selector][bottom_selector], PLAYER1).
  - The code then calls check_winner(PLAYER1) to see if the human won, setting game_over if so. Otherwise it switches current_player to 2.

- **Computer Move**

  - Iterates over all empty board cells. For each, it temporarily simulates placing PLAYER2 there and calls heuristic_score(i,j,PLAYER2).
  - heuristic_score() counts how many lines of 2, 3, or 4 the move creates for the computer and subtracts similar counts for the opponent (simple defense).
  - The move with the highest score is chosen. animate_computer_choice() briefly flashes the choice. Then it calls make_move(best_val, PLAYER2).
  - After the move, it checks check_winner(PLAYER2) and may set game_over.

- **Win/Draw Cheaking**

  - check_winner(player) scans the 2D marks array (0=empty,1=human,2=computer). It checks every possible 4-in-a-row (horizontal, vertical, diagonal). If found, returns true.
  - check_draw() simply checks if total moves equal 36. If true and no winner, it's a draw.

- **State Update**

  - make_move(val, player) updates the marks array and increments the player's moves count.
  - If a win is detected, it increments the player's win count, displays a message, and sets game_over.

- **Cleanup**

  - After exiting the main loop, endwin() restores the terminal to normal mode. This finishes the program.

Overall, the code flows from initialization → (play round → update state) → cleanup. Each user action or AI move corresponds to reading/writing the board (memory) and updating registers (variables), demonstrating low-level state changes.

User Manual

- **Compilation:** On Linux, MacOS, VS code with GCC and ncurses:

  ```
  gcc -o Multiplication Multiplication.c -lncurses
  ```

- **Running:**

  ```
  ./Multiplication
  ```

  This launches the game in the terminal. The program should be run in a terminal emulator that supports ncurses.

- **Game Controls:**
  - Use the arrow keys (← → ↑ ↓) to move the selection cursor around the board.
  - Press Enter (or Space) to select the highlighted number.
  - The board numbers are then marked for the current player.
  - The scores (wins and total moves) are shown below the board in real-time.

- **Game Rules**
  - Players alternate turns, selecting one board number each turn.
  - Once selected, that number is removed from future play.
  - The goal is to get four marks in a row horizontally, vertically, or diagonally.
  - If a player achieves four in a row, they immediately win that round.
  - If all 36 numbers are taken without a winner, the round is a draw.

- **Winning and Restart**

  When a round ends, a message "Player X wins!" or "Draw!" appears. After pressing any key, you are prompted to play again. Enter Y to start a new game or N to exit.

- **System Requirements**
  - **Operating System:** Linux or any OS with an ncurses-compatible terminal.
  - **Libraries:** Ncurses library (-lncurses) for terminal graphics.
  - **Processor/Memory:** Very low requirements; any PC is sufficient.
  - **Terminal:** A standard ANSI-compatible terminal window

Conclusion

The Multiplication Game effectively demontrates core computer architecture concepts through an interective, strategic and engaging terminal-based game. Through the use of arrays, structs, control flow, simulated registers, I/O handling using C and the neurses library, the project bridges the gap between low-level architectural principles and real-world software design. It also includes AI decision-making with heuristic evaluation and persistent storage using file I/O. Overall, this game project not only provides charming gameplay but also deepens understanding of how software can model hardware-level operations making it practical and educational practice in computer architecture