Register transferring :-

```
#include <stdio.h>
int main() {
   // Define two variables
   int a = 5;
   int b;

   // Transfer data from variable 'a' to variable 'b' using a register
   b = a;

   // Print the result
   printf("Value of b after register transfer: %d\n", b);

   return 0;
}
```

OUTPUT:

Value of b after register transfer: 5

SINGLE BUS ORGANISATION:-

```
#include <stdio.h>
// Structure representing a single bus
typedef struct {
   int data;
   int address;
} Bus;
// Structure representing a CPU
typedef struct {
    Bus *bus;
} CPU;

// Structure representing a Memory
typedef struct {
   Bus *bus;
   int data[100]; // For simplicity, assuming memory size of 100 locations
} Memory;

// Function to read data from memory
int memory_read(Memory *mem, int address) {
   return mem->data[address];
}

// Function to write data to memory
```

```
void memory_write(Memory *mem, int address, int data) {
    mem->data[address] = data;
}

// Function to perform CPU operation (read from memory)
int cpu_operation(CPU *cpu, int address) {
    return memory_read(cpu->bus, address);
}

int main() {
    // Initialize bus, CPU, and memory
    Bus system_bus;
    CPU cpu;
    Memory memory;
    cpu.bus = &system_bus;
    memory.bus = &system_bus;

    // Write data to memory
    memory_write(&memory, 0, 10); // Writing value 10 at address 0

    // CPU reads data from memory
    int data_read = cpu_operation(&cpu, 0);
    printf("Data read by CPU: %d\n", data_read);

    return 0;
}
```

OUTPUT:

MULTIPLE BUS ORGANISATION:-

```
#include <stdio.h>

// Structure representing a bus
typedef struct {
   int data;
   int address;
} Bus;

// Structure representing a CPU
typedef struct {
   Bus *bus;
} CPU;
```

```
// Structure representing a Memory
typedef struct {
    Bus *bus;
    int data[100]; // For simplicity, assuming memory size of 100 locations
} Memory;

// Structure representing an I/O Device
typedef struct {
    Bus *bus;
} IO_Device;

// Function to read data from memory
int memory_read(Memory *mem, int address) {
    return mem->data[address];
}

// Function to write data to memory
void memory_write(Memory *mem, int address, int data) {
    mem->data[address] = data;
}

// Function to perform CPU operation (read from memory)
int cpu_operation(CPU *cpu, Memory *mem, int address) {
    return memory_read(mem, address);
}

// Function to perform I/O device operation (write to memory)
void io_device_operation(IO_Device *device, Memory *mem, int address, int data) {
```

```c
    memory_write(mem, address, data);
}

int main() {
    // Initialize buses, CPU, memory, and I/O device
    Bus data_bus;
    Bus io_bus;
    CPU cpu;
    Memory memory;
    IO_Device io_device;

    // Set bus pointers
    cpu.bus = &data_bus;
    memory.bus = &data_bus;
    io_device.bus = &io_bus;

    // Write data to memory
    memory_write(&memory, 0, 10); // Writing value 10 at address 0

    // CPU reads data from memory
    int data_read = cpu_operation(&cpu, &memory, 0);
    printf("Data read by CPU: %d\n", data_read);

    // I/O device writes data to memory
    io_device_operation(&io_device, &memory, 1, 20); // Writing value 20 at address 1

    // CPU reads updated data from memory
    data_read = cpu_operation(&cpu, &memory, 1);
    printf("Data read by CPU after I/O operation: %d\n", data_read);
```

```
    return 0;
}
```

OUTPUT:

Data read by CPU: 10

Data read by CPU after I/O operation: 20

TWO STAGE PIPELINING:-

```
#include <stdio.h>

// Structure representing an instruction
typedef struct {
    int opcode;
    int operand1;
    int operand2;
} Instruction;

// Function to simulate instruction fetch stage
void fetch_stage(int *instruction_count, Instruction *instruction_buffer) {
    // Simulating fetching instructions from memory
    // Increment instruction count
```

```
    (*instruction_count)++;
    // Simulating instruction decoding and filling instruction buffer
    instruction_buffer->opcode = (*instruction_count) % 3;  // Example: alternating opcodes
    instruction_buffer->operand1 = (*instruction_count) * 2;
    instruction_buffer->operand2 = (*instruction_count) * 2 + 1;
}

// Function to simulate instruction execution stage
void execute_stage(Instruction *instruction_buffer, int *result) {
    // Simulating instruction execution
    switch (instruction_buffer->opcode) {
        case 0:
            *result = instruction_buffer->operand1 + instruction_buffer->operand2;
            break;
        case 1:
            *result = instruction_buffer->operand1 - instruction_buffer->operand2;
            break;
        case 2:
            *result = instruction_buffer->operand1 * instruction_buffer->operand2;
            break;
        default:
            printf("Invalid opcode\n");
            break;
    }
}

int main() {
    int instruction_count = 0;
    Instruction current_instruction;
```

```
    int execution_result;

    // Perform multiple cycles of instruction fetch and execution
    for (int i = 0; i < 5; i++) { // Example: 5 cycles
        // Instruction fetch stage
        fetch_stage(&instruction_count, &current_instruction);

        // Instruction execution stage
        execute_stage(&current_instruction, &execution_result);

        // Output the result of the executed instruction
        printf("Cycle %d: Result = %d\n", i + 1, execution_result);
    }

    return 0;
}
```

<u>OUTPUT:</u>

```
Cycle 1: Result = -1
Cycle 2: Result = 20
Cycle 3: Result = 13
Cycle 4: Result = -1
Cycle 5: Result = 110
```

FOUR STAGE PIPELINING:-

```
#include <stdio.h>
// Structure representing an instruction
```

```
typedef struct {
    int opcode;
    int operand1;
    int operand2;
} Instruction;

// Structure representing the pipeline registers
typedef struct {
    Instruction instruction;
    int result;
} PipelineRegister;

// Function to simulate instruction fetch stage
void fetch_stage(int *instruction_count, Instruction *current_instruction) {
    // Simulating fetching instructions from memory
    // Increment instruction count
    (*instruction_count)++;
    // Simulating instruction decoding
    current_instruction->opcode = (*instruction_count) % 3;  // Example: alternating opcodes
    current_instruction->operand1 = (*instruction_count) * 2;
    current_instruction->operand2 = (*instruction_count) * 2 + 1;
}

// Function to simulate instruction decode stage
void decode_stage(Instruction *current_instruction, PipelineRegister *decode_reg) {
    // Transfer the instruction to the decode register
    decode_reg->instruction = *current_instruction;
}
```

```
// Function to simulate execute stage
void execute_stage(PipelineRegister *decode_reg, PipelineRegister *execute_reg) {
    // Simulating instruction execution
    switch (decode_reg->instruction.opcode) {
        case 0:
            execute_reg->result = decode_reg->instruction.operand1 + decode_reg->instruction.operand2;
            break;
        case 1:
            execute_reg->result = decode_reg->instruction.operand1 - decode_reg->instruction.operand2;
            break;
        case 2:
            execute_reg->result = decode_reg->instruction.operand1 * decode_reg->instruction.operand2;
            break;
        default:
            printf("Invalid opcode\n");
            break;
    }
}

// Function to simulate writeback stage
void writeback_stage(PipelineRegister *execute_reg) {
    // Printing the result obtained from the execution stage
    printf("Result: %d\n", execute_reg->result);
}

int main() {
    int instruction_count = 0;
```

```
    Instruction current_instruction;
    PipelineRegister decode_reg, execute_reg;

    // Perform multiple cycles of the pipeline stages
    for (int i = 0; i < 5; i++) { // Example: 5 cycles
        // Instruction fetch stage
        fetch_stage(&instruction_count, &current_instruction);

        // Instruction decode stage
        decode_stage(&current_instruction, &decode_reg);

        // Instruction execute stage
        execute_stage(&decode_reg, &execute_reg);

        // Instruction writeback stage
        writeback_stage(&execute_reg);

        // Output the current instruction being processed
        printf("Cycle %d: Instruction Opcode = %d, Operand1 = %d, Operand2 = %d\n",
               i + 1, current_instruction.opcode, current_instruction.operand1,
current_instruction.operand2);
    }

    return 0;
}
```

OUTPUT:

Result: -1

Cycle 1: Instruction Opcode = 1, Operand1 = 2, Operand2 = 3

Result: 20

Cycle 2: Instruction Opcode = 2, Operand1 = 4, Operand2 = 5

Result: 13

Cycle 3: Instruction Opcode = 0, Operand1 = 6, Operand2 = 7

Result: -1

Cycle 4: Instruction Opcode = 1, Operand1 = 8, Operand2 = 9

Result: 110

Cycle 5: Instruction Opcode = 2, Operand1 = 10, Operand2 = 11

STATIC PREDICTION:-

```
#include <stdio.h>
#define TAKEN 1
#define NOT_TAKEN 0

// Function to simulate static prediction
int static_prediction(int instruction_address) {
    // Implement a simple strategy based on the instruction address
    if (instruction_address % 2 == 0) {
        // Predict taken for even instruction addresses
        return TAKEN;
    } else {
        // Predict not taken for odd instruction addresses
        return NOT_TAKEN;
    }
```

```
}

int main() {
    // Example instruction addresses
    int instruction_addresses[] = {100, 101, 102, 103, 104};
    int num_instructions = sizeof(instruction_addresses) / sizeof(instruction_addresses[0]);

    printf("Static prediction results:\n");
    for (int i = 0; i < num_instructions; i++) {
        int prediction = static_prediction(instruction_addresses[i]);
        printf("Instruction at address %d: Prediction = %s\n", instruction_addresses[i],
               prediction == TAKEN ? "Taken" : "Not Taken");
    }

    return 0;
}
```

OUTPUT:

```
Static prediction results:
Instruction at address 100: Prediction = Taken
Instruction at address 101: Prediction = Not Taken
Instruction at address 102: Prediction = Taken
Instruction at address 103: Prediction = Not Taken
Instruction at address 104: Prediction = Taken
```

DYNAMIC PREDICTION:-

```c
#include <stdio.h>
#define TAKEN 1
#define NOT_TAKEN 0
#define STRONGLY_TAKEN 3
#define STRONGLY_NOT_TAKEN 0

// Structure representing a branch predictor
typedef struct {
    int state; // State of the predictor (2-bit saturating counter)
} BranchPredictor;

// Initialize the branch predictor
void init_predictor(BranchPredictor *predictor) {
    predictor->state = STRONGLY_NOT_TAKEN;
}

// Predict the outcome of a branch instruction
int predict(BranchPredictor *predictor) {
    if (predictor->state >= 2) {
        return TAKEN;
    } else {
        return NOT_TAKEN;
    }
}

// Update the branch predictor based on the actual outcome
void update_predictor(BranchPredictor *predictor, int actual_outcome) {
    if (actual_outcome == TAKEN) {
        if (predictor->state < STRONGLY_TAKEN) {
```

```
            predictor->state++;
        }
    } else {
        if (predictor->state > STRONGLY_NOT_TAKEN) {
            predictor->state--;
        }
    }
}

int main() {
    BranchPredictor predictor;
    init_predictor(&predictor);

    // Simulate branch prediction for a sequence of branch instructions
    int branch_outcomes[] = {TAKEN, TAKEN, NOT_TAKEN, TAKEN, NOT_TAKEN};
    int num_branches = sizeof(branch_outcomes) / sizeof(branch_outcomes[0]);

    printf("Branch prediction results:\n");
    for (int i = 0; i < num_branches; i++) {
        int prediction = predict(&predictor);
        printf("Branch %d: Prediction = %s\n", i + 1, prediction == TAKEN ? "Taken" : "Not
Taken");
        update_predictor(&predictor, branch_outcomes[i]);
    }

    return 0;
}
```

OUTPUT:

Branch prediction results:

Branch 1: Prediction = Not Taken

Branch 2: Prediction = Not Taken

Branch 3: Prediction = Taken

Branch 4: Prediction = Not Taken

Branch 5: Prediction = Taken

Data hazards:-

```
#include <stdio.h>
int main() {
    int a = 5;
    int b = 10;
    int c;

    // Instruction 1: Add 'a' and 'b' and store the result in 'c'
    c = a + b;

    // Instruction 2: Multiply 'c' by 2 and store the result in 'c'
    c = c * 2;

    // Instruction 3: Print the value of 'c'
    printf("Result: %d\n", c);

    return 0;
}
```

OUTPUT:

Result: 30

Instruction Hazards:-

```
#include <stdio.h>
int main() {
    int a = 5;
    int b = 10;
    int c;

    // Instruction 1: Load the value of 'a' into a register
    int temp_a = a;

    // Instruction 2: Load the value of 'b' into a register
    int temp_b = b;

    // Instruction 3: Add the values stored in the two registers and store the result in 'c'
    c = temp_a + temp_b;

    // Instruction 4: Multiply 'c' by 2 and store the result in 'c'
    c = c * 2;

    // Instruction 5: Print the value of 'c'
    printf("Result: %d\n", c);

    return 0;
}
```

OUTPUT:

Result: 30

Structure Hazards:-

```
#include <stdio.h>

int main() {

    int a = 5;

    int b = 10;

    int c;


    // Instruction 1: Compare 'a' and 'b' and set a flag if 'a' is greater than 'b'

    int flag = (a > b);


    // Instruction 2: If the flag is set, add 'a' and 'b' and store the result in 'c', else store 'b' in
'c'

    if (flag) {

        c = a + b;

    } else {

        c = b;

    }


    // Instruction 3: Multiply 'c' by 2 and store the result in 'c'

    c = c * 2;


    // Instruction 4: Print the value of 'c'

    printf("Result: %d\n", c);


    return 0;
```

```
}
```

OUTPUT:

Result: 20

SUPER SCALAR processing:-

```
#include <stdio.h>
// Function to perform addition
int add(int a, int b) {
    return a + b;
}

// Function to perform subtraction
int subtract(int a, int b) {
    return a - b;
}

int main() {
    int a = 5;
    int b = 10;
    int c, d;

    // Superscalar processing:
    // Execute multiple instructions in parallel if possible

    // Stage 1: Instruction Fetch
    // Instructions are fetched simultaneously
```

```c
    // Stage 2: Instruction Decode
    // Instructions are decoded in parallel

    // Stage 3: Execution
    // Instructions are executed in parallel
    c = add(a, b);
    d = subtract(a, b);

    // Stage 4: Writeback
    // Results are written back to registers

    // Print the results
    printf("Result of addition: %d\n", c);
    printf("Result of subtraction: %d\n", d);

    return 0;
}
```

OUTPUT:

Result of addition: 15

Result of subtraction: -5
