**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) {

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;

}

**OUTPUT:**

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:-**

#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

// 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

--------------------------------------------------------------------------------------------------------------------------