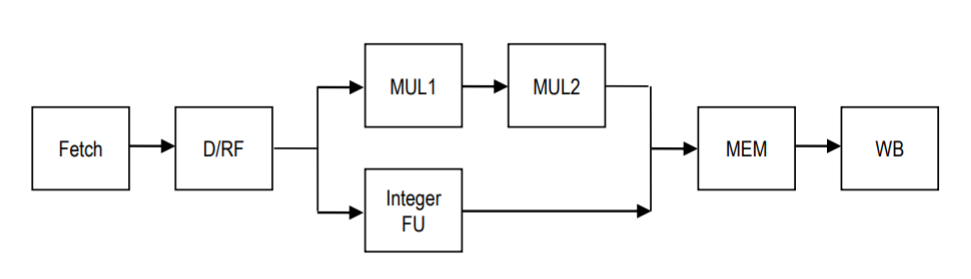
**DESIGN DOCUMENT FOR APEX PIPELINE**

This project implement a cycle-by-cycle simulator for an in-order APEX pipeline with two different function units, as shown below:

****

**Module 1 :-**

**Initialization**

After reading the set of Instructions from the text file all the instructions with its PC address are stored in following strut :-

struct instruction

{

char instruction\_string[100];

int line\_number;

int address;

};

Then stage\_info for all the stages, Register File, Code Memory, Data Memory base address and data fields are initialized.

struct flags

{

char zero[10];

char carry;

char Negative;

char psw\_flag[10];

};

struct stats

{

int cycle;

};

struct instruction

{

char instruction\_string[100];

int line\_number;

int address;

};

struct register1

{

int value;

char status[10];

};

struct register\_file

{

char registers[4];

struct register1 regtr;

};

struct stage

{

struct instruction\_info input\_instruction;

struct instruction\_info output\_instruction;

char stalled[10];

};

struct data\_memory {

int mem\_address;

int mem\_value;

char mem\_char;

};

**Module 2 :-**

**Simulation**

In the simulator loop stages are called in following order :-

fetch();

decode();

execute\_mul1();

execute\_mul2();

execute();

memory();

writeback();

Instruction will stop following to fetch stage till the instruction counter is reached.

After calling all the stages, cycle is incremented in simulator method after every loop. After incrementing the cycle all values which are in the input stage are transferred in output stage for that particular stage, so that in next cycle the next stage can pick all these values from output of previous stage and stored in the input buffer of current stage and can process the same.

**Fetch Stage :-**

Fetches the first instruction to start the Apex pipeline implementation. Next instruction will keep on coming in next cycle as along as fetch stage is not stalled. Fetch Stage will be stalled if decoded and execution stages are stalled.

**Decode Stage :-**

Decode Input instruction fetches the first instruction in second cycle from Output of fetch instruction and it keep doing the same in every cycle as along as Decode is not stalled. In decode stage the instruction string which is fetched from fetch stage is decoded and Source Register, Destination Register, Literal Value. Op code all are stored in following Struct :-

struct stage

{

struct instruction\_info input\_instruction;

struct instruction\_info output\_instruction;

char stalled[10];

};

struct instruction\_info

{

int pc\_value;

char instruction\_str[100];

int source\_reg1;

int source\_reg2;

int dest\_reg1;

int dest\_reg2;

int target\_memory\_addr;

int target\_memory\_data;

int source\_reg1\_value;

int source\_reg2\_value;

int dest\_reg1\_value;

int dest\_reg2\_value;

int dest\_reg\_value;

int literal\_value;

char opcode[20];

};

Only for the HALT instruction execution is done in decode while for all other stages only values are decoded in these stage. HALT instruction in decode stage make NOP for previous stage i.e Fetch Stage so that no next instruction can be processed after HALT. After making NOP for previous stage HALT moves into execution stage.

HALT will not get executed in decode only in one scenario where BZ or BNZ is in execution stage and branching condition is satisfied so following scenario is also handled in decode stage for HALT.

BZ and BNZ halts at decode stage till the PSW flag is valid. BNZ and BZ waits in decode to get PSW flag valid it gets valid once all the arithmetic instruction before BZ or BNZ are in Writeback stage.

MOVE, STORE, ADD, SUB, MUL, LOAD, AND, OR, EX-OR, JUMP are stalled in decode stage till source registers of there instruction are valid to avoid true dependency (RAW – Read after write). Register status is stored in following structs:-

struct register1

{

int value;

char status[10];

};

**Integer FU :-**

All the instruction other than Multiplication are executed here, arithmetic calculations for ADD, LOAD, STORE, SUBTRACT, AND, OR, EX-OR are done in these stage. Output for the arithmetic instruction are stored in input of current stage it doesn’t directly updates the destination register, it is done in Writeback Stage.

**Implementation of BZ :-**

BZ checks if the Zero flag is TRUE if it is TRUE it takes branch, only when ZERO flag is TRUE it does branching otherwise BZ is passed from execution without performing any operation. Branching Instruction is decided after adding the PC value of current stage to the literal value of the instruction.

**Implementation of BNZ :-**

BNZ checks if the Zero flag is FALSE if it is FALSE it takes branch, only when ZERO flag is FALSE it does branching otherwise BNZ is passed from execution without performing any operation. Branching Instruction is decided after adding the PC value of current stage to the literal value of the instruction.

**Implementation of JUMP :-**

PC value is calculated by adding the literal value and the value of one source register. This PC value calculated is the address where instruction should get jumped.

Execution in Integer FU is stalled if there is Instruction in Multiply FU 2 because priority is given to multiplication.

**Multiplication FU 1 & 2:-**

In following 2 stage arithmetic calculation is done for Multiplication. Output value is not directly updated in Register file it is kept in input of following stage and passed to next stage. It will reflect the register in Write back stage.

If there is a instruction in Multiply FU 2 then flag is set by Multiply FU to notify Integer FU stage, so that execution for Execution will be stalled for Integer FU and also for decode and fetch if the instruction in decode is other that Multiply.

Stalled for following stages are released in Memory stage when instruction from Multiply FU 2 moves to Memory stage in next cycle.

**Memory Stage :-**

Memory Address calculated for STORE in previous stage is directly updated in data memory. Following is the Data Memory where values are stored.

struct data\_memory {

int mem\_address;

int mem\_value;

char mem\_char;

};

Nothing will be done for STORE in writeback stage.

For the memory address which is calculated in previous stage for LOAD is used to retrieve the Memory value for that address so that same can be updated in Writeback for that destination Register.

**Write back stage :-**

All the output values calculated for the register are updated to Register data structure and also register status is made valid after updating the values.

Now the stage which are stalled are also released so that execution can start from next cycle as following register are no longer invalid and corrected values will be used to calculate the result in execution stages these avoid the true dependency.

Following are the ways through which interlocking mechanism is handled.

**Module 3 :-**

**Display**

Following is the register structure and memory structure where values for register are updated are writeback stage and displayed when display module is selected.

Also the cycle information for which user enters is shown in following module.

struct register1

{

int value;

char status[10];

};

struct register\_file

{

char registers[4];

struct register1 regtr;

};

struct data\_memory {

int mem\_address;

int mem\_value;

char mem\_char;

};