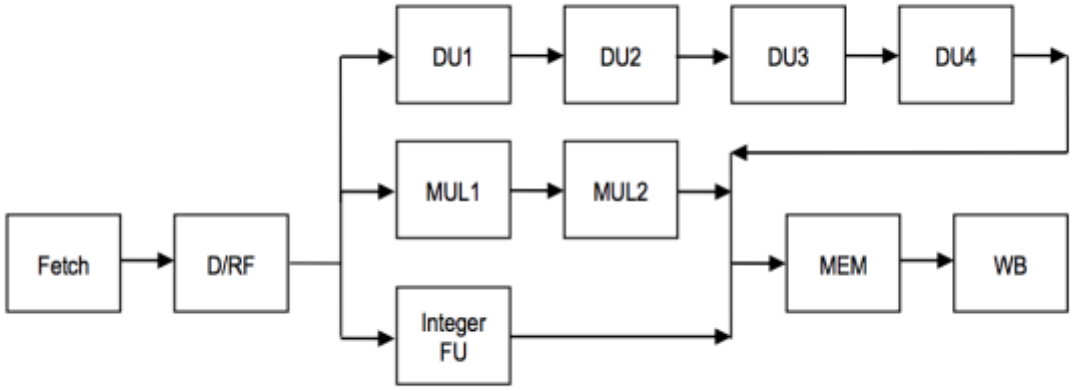
APEX PIPELINE SIMULATION-2

Design Document

[Gopal Nagarajan]

*[*B00675008*]*

APEX pipeline stages:



Implementation details:

Java Programming language is used to develop the simulator.
16 Registers are used- R0 to R15. ArrayList data structure used to implement register logic.
one additional PSW register to maintain the zero-flag status.
ArrayList data structure is used as a memory component. It ranges from 0 to 3999.
HashMap is used to maintain pipeline stages.
And the instructions available in the input file read line by line and store it in a List as an instruction set. Every index of this list is considered as four bytes instruction.
The Program counter value is initialized with 4000. Each instruction is accessed by increasing the program counter by 4bytes.
On looking in to each instruction, it's going to be segregated with opcode (to represent what kind of operation) and operands. Operands number can vary depends on the opcode. So, based on the opcode split the number of the operands including register, literal.

Code level explanation:
A RegisterHandler POJO is used to manage the 16 Register values and the operations like setting up the value, retrieving the value.
An apex-simulator-helper used to initialize values to register, memory, print results of every clock cycle along with stages, registers and memory level.
A hashmap is used to hold the status of every stage details. This one helps in manage available dependencies.

Approach:
For a single clock cycle the instructions available in every stage should be processed. I am calling the executing methods in the order like writeback, memory, DIV4, DIV3, DIV2, DIV1,

MUL2, MUL1, Integer FU, Decode, Fetch stages. It is the reverse order to original pipeline stage wise execution. This is very handy in detect the dependency and stalling the registers.

My approach is like starting from last stage writeback to check any unexecuted instructions available. Suppose if it's available it should be executed in a clock cycle. Once there are no instructions available in the writeback stage it check the memory stage for any pending instructions and moved to writeback. Then it goes to DIV4, DIV3, DIV2, DIV1, MUL1, MUL2, Integer FU, decode, fetch. At every stage pipeline status, corresponding stage is set up with the instructions and the previous stage is setup with null to represent the instructions available in previous stage moved to next stage and it currently free to hold next instructions.
For Example:

In Decode stage, the instructions from fetch stage is moved to decode and set fetch as null. This will help to set up new instructions in fetch stage in the next clock cycle.

This will happen in every stage like moving the output of one stage to the input of next stage for further processing.

In the case of multiply operation, the execution stage is of two cycles MUL1 and MUL2 as per design diagram. Once MUL opcode is figured it moved to MUL1 stage from decode and doing no operation. It is like a hold back stage for a cycle. And in the next cycle instruction moved to MUL2 where the actual computation happens. Then moved to Memory stage.

In the case of Divide operation, the execution stage is of four cycles. Once DIV opcode is figured it moved to DIV1, DIV2, DIV3 and the actual divide operation executed in DIV4. Apart from DIV4 all other DIV units are holdback.

Interlocking logic:

The dependencies can be identified in the decode stage. Each register is having status flag and this will set true once the instructions loaded in decode stage. This status remains true until the writeback stage write the result to the corresponding register. And there will be a forward flag status in Instruction structure to represent the corresponding source registers are forwarded one. So, whenever a new instruction arrives to decode stage it check for the source register status is whether it is occupied by any other ongoing computation and check the source register content is forwarded by instruction in three available execution stages.

In forwarding the execution unit set the forwarded data in temp variable and status to set up in the associated instructions available in decode stage. In the decode stage those instructions checked and set up the forwarded value to the source registers. Once all the dependency is resolved it moves to the next stage. Otherwise it will have stalled in decode.

At output dependency, the instruction in decode stage is getting stalled up to the dependent instruction enter in WB stage.

Apart from Divide and Multiply instructions all other instructions pass through the Integer FU stage as Execution stage in one clock cycle.
Divide has the highest priority followed by Multiply and then other operations.

In writeback stage the PSW zero-flag register value set to 1 if its arithmetic operation. Otherwise set zero in PSW zero-flag.
In the last unit of the execution stage the PSW flag value forwarded to decode stage.
BNZ and BZ depends on the PSW zero-flag value. Forwarding happens only for the arithmetic operation.

Once BNZ or BZ instruction encountered in instruction parsing phase it check for any previous instruction in execution. If it is the case then set those instructions register set as one of the source for this branch instruction. This will help to be in stall state at decode stage when there are any ongoing arithmetic operation takes place. Suppose if the arithmetic operation is already passed from execution to memory and the branch instructions enters in to decode it will stall up to the same instructions to writeback stage.

Jump instructions adds the register value and literal value resulted in address of the next instructions to be executed. This takes place by resetting the PC value. And flush the current fetch and decode stages.
JAL, Jump and link is used to store the next instruction address in a register. It will compute the jump address based on the sum of one source register and literal.

Once Halt instruction is encountered fetch and decode stages are stopped and cleared. No more instructions are going to fetch after this. This Halt passed to all Div stages through writeback.

Instruction Set:

1) Register-to-register instructions: ADD, SUB, MOVC, MUL, AND, OR, EX-OR
   ADD → `ADD <dest> <src1> <src2>`
   SUB → `SUB <dest> <src1> <src2>`
   MUL → `MUL <dest> <src1> <src2>`
   AND → `AND <dest> <src1> <src2>`
   MOVC → `MOVC <dest> <literal>`
   OR → `OR <dest> <src1> <src2>`
   EXOR → `EXOR <dest> <src1> <src2>`

2) LOAD, STORE
   LOAD → LOAD <dest> <src1> <literal>
   STORE → STORE <src1> <src2> <literal>

3) BZ, BNZ, JUMP, HALT

BZ → `BZ #literal`

BNZ → `BNZ #literal`

JUMP → `JUMP <src>` #literal

JAL → `JAL <dest> <src>` #literal

HALT → `Stops execution`