COL216 Assignment 10

Vishal Bindal
2018CS50425

Chirag Mohapatra
2018CS50403

To Run:

0. Set values of x (prob of MISS) and N (no of cycles in case of MISS) in global.cpp

(A) To give instructions in form of MIPS assembly code:

1. Set syntax_input = true in main.cpp (It has been set to true already while submitting)
2. Run '`make`' to create an executable 'main'
3. Enter the required instructions in 'input.txt' (one test case already filled in while submitting). The registers could be like $8, $9, etc or $t1, $t2, etc
4. Run '`./main`'

(B) To give instructions in form of 32-bit instructions directly:

1. Set syntax_input = false in main.cpp
2. Run '`make`' to create an executable 'main'
3. Enter the required instructions in 'binary_input.txt' (one test case already filled in while submitting)
4. Run '`./main`'

Output:

1. The state for each clock cycle is printed in the console. For each clock cycle, following is shown:
    a. Clock cycle no.
    b. Current instruction in each stage of the pipeline (Here instruction nos. refer to the line no in input_clean.txt, read more below)
    c. Details about any read/write from data memory (Memory address and value)
    d. Details about any write to the register file (Register no. and value)
    e. The status of the register file, i.e. values in each register
    f. Details about forwarding of any data (Reg whose data forwarded)
2. At the end, the following 2 statistics are printed:

    a. Total clock cycles
    b. IPC
3. We also print to the console if it's a HIT or MISS in the MEM stage, and then show <STALLED> in the pipeline stages for clock cycles till the time memory access is going on

Documentation

Differences from assignment 9

1. Memory access during MEM stage can now take 1 or N cycles, with prob x and (1-x) respectively
2. We create a function isHit() in Instruction.cpp, which generates a random no in the range (0,1) and checks if it is less than x. If yes, it's a hit else it's a miss. For randomising we use a different random seed each time
3. In case of a hit, memory access is done in 1 cycle as usual. In case of miss, stall() is called for (N-1) cycles, for all instructions in the pipeline. After the stall ends, the memory access also happens successfully.

Differences from assignment 8

1. We implemented forwarding to resolve data hazards where possible in order to speed up the processor
2. Forwarding
    a. In the function getData() in Instruction.cpp, we check if the register which we want to read data from (is **rs** or **rt** of current instruction) is equal to the (a) **rd** of an add/sub/sll/srl or (b) **rt** of an lw instruction already in the pipeline. If so, then it is a data hazard.
    b. Then we check if the data is okay to forward (i.e. if the prev instruction has passed the EX stage in case (a), or has passed the MEM stage in case (b)). using the function okay_to_forward()
    c. If yes then we forward the data using forward_data() and there is no stall, else we stall for a cycle.
3. We also print to the console, about any data forwarded during the cycle.

Implementation structure

1. If the input is in the form of assembly code, we first 'clean' the file by removing unnecessary spaces and newlines. We parse instructions into tokens (delimited by space, bracket, comma) and create a 'clean' input file input_clean.txt. **All the instruction nos. in the output refer to corresponding line nos. in input_clean.txt**

2. We then convert each instruction into 32-bit binary instructions, and output them to ‘binary_input.txt’. We also take care of labels for jump/branch instructions, by mapping each label to corresponding instruction no first and then encoding the required jump/branch value in the instruction.

3. We have represented each instruction as an object of a class ‘Instruction’.
    a. This object stores the instruction string, rd/rs/rt register nos, values read while decoding, result of alu operation, etc. Thus it acts as a proxy for the pipeline registers, while making it easy to organise and print data during the simulation.
    b. This object also has functions: instruction_decode(), execute(), memory_access() and write_back() which are run in the corresponding stage of the pipeline the instruction is in. Each of these functions is implemented in the form of switch-case, with the action depending on the type of instruction.

4. We have implemented stalling by creating a function stall(), which stalls instruction >=i (the ith instruction and instructions after it) for n cycles. This function is called in case of data hazards and for branch hazards accordingly, during the instruction decode stage. Data hazards is checked using a function ‘is_data_hazard()’, which checks register nos of instructions already in the pipeline.

5. For each clock cycle, a new instruction is fetched (if not stalled), and the instructions in the pipeline run their corresponding functions (depending on the pipeline stage they are in) (if not stalled)

Analysis for different x and N values:

(we have assumed the following initialisation of the register file:
reg_array[0] = 2
reg_array[1] = 1
reg_array[29] = 4095
Others: 0)

For showing the effect of changing x and N values, we have used the input file :

```
sll $4 , $0 , 1
sll $5 , $0 , 2
jal one
add $17 , $2 , $16
j end
one:
sub $29 , $29 , $0
sw $31 , 1($29)
sw $4 , 0($29)
bgtz $4 , two
add $29 , $29 , $0
jr $31
two:
sub $4 , $4 , $1
jal one
lw $4 , 0($29)
lw $31 , 1($29)
add $29 , $29 , $0
add $2 , $2 , $5
jr $31
end:
```

Keeping N fixed , as expected when x increases , the probability of HIT increases and so the clock cycles become gradually less .

x	N	Total clock cycles
0	10	236
0.2	10	201
0.8	10	97
1	10	79

With x = 1 , the perfect HIT system , the clock cycles are the same as in assignment 9 .

x	N	Total clock cycles
1	10	79
1	20	79
1	30	79
1	40	79

When x = 1 , the value of N is irrelevant which is as expected .

With x = 0 , increasing N by a constant amount , the clock cycles also increase by the same amount . (Here , N increases by 10 and cycles increase by 180)

x	N	Total clock cycles
0	10	236
0	20	416
0	30	596
0	40	776

With x not equal to 0 or 1 , the cycles have a higher probability of being higher, but there is no guarantee that they will increase . Although practically they almost always increase. We will get different values when we run the program multiple times .

x	N	Total clock cycles
0.2	10	201
0.2	20	323
0.2	30	510
0.2	40	699

x	N	Total clock cycles
0.8	10	97
0.8	20	136
0.8	30	165
0.8	40	311