CS-204

Design Document : RISC-V 32I Simulator

Phase - 3

Group Number : 3

Ayush Verma 2019CSB1147
Bhumika 2019CSB1152
Keshav Krishna 2019CSB1224
Rishabh Jain 2019CSB1286
Vishawam Datta 2019CSB1305

- **Two-Level Memory**

  A two level (Cache+Main Memory) is implemented which has a Set-Associative Instruction and data cache. It is realized on the basis of the inputs- Cache size, Block size and Associativity and has the components-Tag Array and Set Array.

  The *ProcessorMemoryInterface* class contains the text segment and data segment modules, each containing their own main memory module and cache module. This class receives data requests from the processor and calls appropriate methods of the TwoLevelMemory class to return data.

  The class *TwoLevelMemory* contains the main memory and cache module.

  The cache module, along with interface methods, contains the following objects-

  Tag Array: It is designed in the form of a 2D array, with sets along the rows and set elements being the columns. Thus, each block element corresponds to the tag element (i, j). Each element, in that sense, is initialized to -1 to indicate an empty tag initially.

  Set Array: It consists of the set objects. Each set object further contains a collection of cache block objects.

  Cache Set: This class contains a list of cache block objects. Along with acting as an interface for accessing the blocks in a certain set, it also has the

update_state_hit and update_state_miss functions, which are a manifestation of the LRU policy, as explained in the next section.

Cache Block: An individual cache block is represented by a class, with the functionality to read and write from using block offset and number of bytes (byte addressable) as input. The cache block contains a list holding individual bytes of size block_size. Any data that is written or read from the block is in the form of a list of individual bytes. An exception is raised when the input/requested data is not word aligned. An attribute pref_count(initially -1) is also maintained to indicate the history of use for LRU implementation. It also has the boolean valid to indicate if the block contains any data.

- **LRU**

  The update_state_hit function in the Cache iterates over all the vald blocks in the set and decrements their pref_count if it is greater than the pref_count of the block which was just updated, which in turn is now assigned the maximum pref_count.

  In case of a miss, the update_state_miss function returns the index of the block to write to, and also updates the pref_counts of the blocks. When there is a miss, there can be two cases:

  a) Set is not full: In this scenario, the index of the first empty block is returned and the counters of the non-empty blocks are decremented. The selected index block is assigned the highest pref_count, indicating it to be the most recently used block.

  b) Set is full: Here, the least frequently used block(i.e. pref_count == 0) is selected to be the victim. It is then reassigned the highest pref_count while the counters of the remaining valid blocks are decremented.

- **Reading from Cache**

  The main memory is a hash map , as in the previous phases, while the cache is implemented as explained above. To read from the cache, the first step is to generate the address, based on the input parameters.

  Cache Address Generation: It takes the base address of the requested data as the input and returns the tag, index and block offset as:

Tag = Base Address - Base Address % Cache Block Size

Index = (Tag / Cache Block Size) % (Cache Size/ Block Size)

Block Offset = Base Address % Cache Block Size

The function checkHit in the cache module checks for a hit by comparing the tag of the block in the tag array. If there is a match, the data is returned as a list of the corresponding bytes, based on the input block offset and number of bytes by the readDatafromCache function. Otherwise, if it is not a hit, the block containing the requested data is fetched from the main memory. This is then loaded into the cache with the help of writeWhenNotHit function. This function calls the update_state_miss function to get the location to write to, in the block.

Calling all of these procedures according to hit/miss is done in the GetUnsignedValueAtAddress / GetSignedValueAtAddress functions. After the list of bytes is received from memory, the data is converted into a word of data,sign extended and returned to the processor.

- **Write Policy**

  For write requests, the principles of Write Through with Write Allocate are followed.
  Similar to the read requests, write functionality is driven by the WriteValueAtAddress function. Firstly, the presence of the block is checked in the cache. If it is a hit in the cache, then the block is updated with the new data using the writeDatatoCache function. Then, the data is written in the main memory. If not hit,the data is written in the main memory, and then the block is fetched from the memory and written in the cache using the writeWhenNotHit function as in the case of an unsuccessful read.

- **Constraints and Word Alignment**

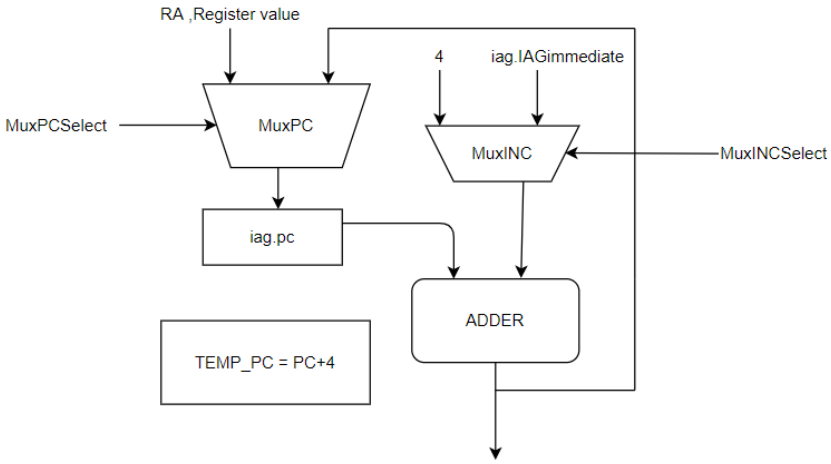
  Input specifications for cache configuration-

  1. Cache size size must be a power of 2 and of size >=4 bytes.
  2. Block size size must be a power of 2 and of size >=4 bytes. Cache size must never be less than block size.
  3. Associativity has to be a power of 2, including 1.

  Data access-
  1. Data to be read/written to memory must always be word aligned.

The following documentation is from the past reports-

IAG Module(Including BTB)



Inputs:-

- MuxINCselect - Control signal of MuxINC
- MuxPCselect - Control signal of MuxPC
- Boffset - To set the IAG value.

Output:-

- PC_temp - To store this value for jal and jalr statement as return address in a register.

Important Variables:-

- PC - SImulates the PC register in IAG.
- PC_temp- Simulates the PC_temp register in IAG
- IAGimmediate- Simulates the immediate input given to MuxINC
- BTB.inst_address = the address of the control instruction entered in the BTB
- BTB.target_address = the address of the branch that the instruction may take
- BTB.take = this is 1 if the unit predicts that the branch should be taken or else 0.

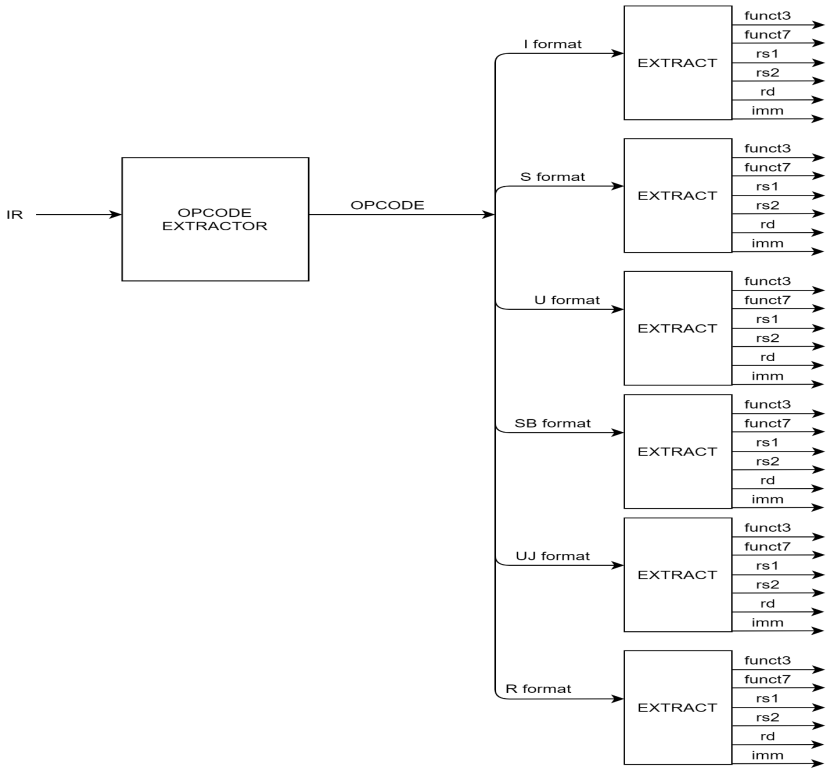
Methods:-

- ReadPC - Used to get the current value of PC
- WritePC - Used to update/ write the value of PC
- PCTempUpdate - Used to set the value of PC_temp to PC + 4
- PCset - Used to select , based on control signals whether to put PC as RA (register input in case of Jalr) or let PC remain PC.
- SetBranchOffset - Used to update the immediate value that is connected to MuxINC
- PCUpdate- Based on the control signals that are given as arguments , this function decides whether to set PC as PC + 4 or PC + immediate.
- BTB_check - check whether this instruction (PC) already exists in the table or not

- BTB_insert - This will insert the PC and the destination instruction address (if the branch is taken) in the BTB and the prediction corresponding to it , whether the branch is to be taken or not.The values in this table is in the form of a BTB_entry class variable which contains the variables ->inst_address, target_address and take (which signifies whether to take the branch or not , hence is a boolean value)

  Decode Unit :-
  Input: The input to the Decode unit is IR (instruction in hexadecimal format) and PC.
  The opcode of the machine instruction is calculated using bit masking.
  If the opcode equals “0x11” (17), the program gets terminated.
  Based on the opcode of the machine instruction, following functions are called which extract the respective fields (rs1, rs2, rd, function 3, function 7 and immediate value) from the respective instruction format :
    - decodeR # for R format instruction (add, and, or, sll, slt, sra, srl, sub, xor, mul, div, rem)
    - decodeI # for I format instruction (addi, andi, ori, lb, lh, lw, jalr)
    - decodeS # for S format instruction (sb, sw, sh)
    - decodeSB # for SB format instruction (beq, bne, bge, blt)
    - decodeU # for U format instruction (auipc, lui)
    - decodeUJ # for UJ format instruction (jal)

  In each of the above functions, bit masking is used to extract the fields from the machine code.
  The immediate field is taken special care, as it is a signed value. Signed value is ensured by use of ‘&’ and ‘|’ logical operators.
  Output: Control signal generator function is called by the decode unit after extracting the fields from the machine instructions.



Control signals

1. Control signals for the instructions will be generated in the decode stage and saved(enqueued) in the respective control signal queues of each of the stages.
2. Each stage will dequeue the control signals from the queue before execution.
3. The queues will be initialized with an appropriate offset since it takes 5 cycles for all the stages to become active in pipelining.
4. Appropriate NOPs will be passed for introducing bubbles in the pipeline.
5. Exclusive for Fetch- If queue==empty, then operate. If not empty, then NOP.
6. Exclusive and Decode- These stages will have a queue each. If queue==empty, then operate. If not empty, pop the element and read the code. It will contain the hazard code [21-23] for performing M to E forwarding with stalling, 0 for NOP, and -2 for pipeline flush, since it is in these cases that the output of Decode is don't care.
7. In data forwarding, a control signal is pushed in the stage to take data forwarding from a particular module when it is enqueued.

Control signal queues-

1. For fetch stage
    - fetch_operation : If this queue is empty, execute fetch stage. Else, do not execute fetch stage.
2. For decode stage

- decode_operation: If queue==empty, then operate. If not empty, pop the element and read the code. It will contain the hazard code [21-23] for performing M to E forwarding with stalling, 0 for NOP, and -2 for pipeline flush, since it is in these cases that the output of Decode is don't care.

When the decode stage is called, control signals are popped from the queues and checked.

3. Execute stage
    - exe_opcode: contains opcode of the instructions.
    - exe_funct3: contains funct3 of the instructions.
    - exe_funct7: contains funct7 of the instructions.
    - exe_ALUOp: contains ALUOp
    - exe_ALUcontrol: contains ALU control signal
    - Exe_operation: contains operation code. If the popped value is False, then don't execute the operation stage.

    When the execute stage is called, control signals are popped from the queues and saved in temporary variables inside the control module.

4. Memory stage:
    - mem_MemRead- contains control signal for reading memory
    - mem_MemWrite- contains control signal for writing to memory
    - mem_BytesToAccess- contains control signal for number of bytes to be accessed
    - mem_MuxYSelect- contains MuxY select code
    - mem_RMqueue- Contains data to be written in memory. If MemWrite==0, the popped value is don't care
    - mem_RAqueue- Contains return address. If MuxYSelect!=2, the popped value is don't care
    - mem_ForwardingQueue- Contains the code for MtoM forwarding. If the queue is empty, then forwarding is not performed. If not empty, then if the code popped is 0, the memory unit assigns the output to it's input to perform MtoM forwarding. read_from_rm boolean is set to true so that the memory unit does not read from the control signal queue for RM, but instead reads from the forwarding register.
    - mem_operation- contains operation code. If the popped value is False, then don't execute the operation stage.

    When the Memory stage is called, control signals are popped from the queues and saved in temporary variables inside the control module.

5. Writeback stage:
    - reg_RegWrite: contains booleans for writing to the register file or not.
    - reg_rd: contains rd address
    - reg_operation: contains operation code. If the popped value is False, then don't execute the operation stage.

    When the Writeback stage is called, control signals are popped from the queues and saved in temporary variables inside the control module.

Hazard Unit:-

This file(hazard.py) basically contains the hazard unit, which basically detects any form of data hazard with previous instructions and informs the control unit about what data forwarding to do , how many stalls etc.All this unit takes as input is the opcode,rd ,rs1 ,rs2 ,funct3,funct7 of the current instruction.
The previous two instructions are stored in the queue. The current instruction dependency (Control hazard or data hazard) is checked with each of the previous two instructions.
The decision-maker function calls check_dependence_function which detects the dependency and returns two integers in a list according to the type of dependency between current instruction and previous-to-previous instruction and with previous instruction respectively.

Important Methods:-
Based on whether the forward knob is set or not two functions are called which are:

- Data_forwarding(i1,i2)
    - ➢ Arguments -> 2 integers i1 and i2.i1 tells the type of dependence b/w current instruction and previous to previous instruction, i2 tells the type of dependence b/w current instruction and previous instruction.
    - ➢ Return Value-> Now based on the values of i1 and i2, 2 Integers would be returned [a,b] a->encoding of forwarding b/w i1 and i3
      b->encoding of forwarding b/w i2 and i3

      This encoding gives information to the control circuit about what kind of forwarding is to be performed from the previous 2 instructions to the current instructions if any , and also which stage is to be stalled if any stalls are needed.

      -1: No forwarding
      0: M to M (No stall) every time forward to rs2
      11: M to E (No stall) to rs1 only
      12: M to E (No stall) to rs2 only
      13: M to E (No stall) to both rs1 and rs2
      21: M to E (1 stall of E)to rs1 only
      22: M to E (1 stall of E) to rs2 only
      23: M to E (1 stall of E)to both rs1 and rs2
      31: E to E (No stall) to rs1 only
      32: E to E (No stall) to rs2 only
      33: E to E (No stall) to both rs1 and rs2
      404 : D is stalled 1 time (404, -1)
      505: D is stalled 2 times (-1,505)
      310: M to M & E to E

- data_stalling(i1,i2)

- ➢ Arguments -> 2 integers i1 and i2.i1 tells the type of dependence b/w current instruction and previous to the previous instruction, i2 tells the type of dependence b/w current instruction and previous instruction.
- ➢ Return Value-> Now based on the values of i1 and i2, 1 Integer would be returned which tells information about how many cycles should the decode stage be stalled.

  The encoding of the return is as follows:

  -1:No stalling
  1: Stall D for 1 cycles
  2: Stall D for 2 cycles

- decision_maker(opcode,funct3,rs1,rs2,rd,forwarding_knob)
  Based on whether the forwarding knob is 1 (which signifies data forwarding is enabled) or 0 (which signifies data forwarding is disabled) this function makes the appropriate calls to either data_forwarding() (this returns a tuple of encoding telling what kind of action is to be performed, i.e type of data forwarding and stalling) function or data_stalling() function (how many stalls), this is done after first identifying what kind of dependence exists between the current instruction and previous instruction or previous to previous instruction or both if any and then returns the decision made to the control unit which then would proceed to perform these forwarding and stalling operations.

Static Branch Predictor

The BTB(in the IAG) stores the PC and the target address of the control instructions. The entries are created the first time that they are encountered. The static predictor-Always Taken is implemented.

Pipeline buffers:

Buffers.py contains buffers and temporaries of the buffers that are used in program execution, buffer are updated with value of buffer temporaries at the end of cycle

- RA, RB and RAtemp and RBtemp
- RZ, RZtemp
- RY, RYtemp
- RM, RMtemp
- IRbuffer
- Fetch_output_PC_temp
- Decode_output_PC_temp
- Decode_input_PC
- Decode_input_branch_prediction

Buffer Update:

The above buffers are updated at the end of the cycle, based on the control signals. The output, during the cycle is stored in the temp buffers, and gets loaded into the corresponding output buffers in the buffer_update function. Further, this takes place depending on the stall signals and data forwarding signals, as mentioned below. The data forwarding signals lead to a data transfer between the corresponding input and output buffers
Buffer update control signals:

- Fetch_stall- update output of fetch only if false
- Decode_stall- update output of decode only if false
- Execute_stall- update output of execute only if false

The following are booleans are utilised for data forwarding and are discussed in the last section.

- MtoEtoRA
- MtoEtoRB
- EtoEtoRA
- EtoEtoRB
- MtoM

PC update mechanism-

In the fetch stage, the instruction pointed to by PC register in IAG module is loaded.
The fetch stage will save the PC it decodes in decode_input_PC buffer.
This is implemented for correcting branch mispredictions, for auipc. If a branch is mispredicted, the PC in IAG module has to be overwritten by the correct target PC, which would be PC+4 in case of NotTaken and PC+imm or RA+imm in the case of Taken. The PC will be obtained from decode_input_PC.since the PC register in IAG contains the incorrect target. The corrected PC is then stored in Decode_output_PC_temp.

We are introducing a fetch output PC buffer,Fetch_output_PC_temp which contains the target address predicted by the fetch stage, which is the target PC stored in BTB in the case that the branch is predicted as taken(i.e BTB contains the entry for the control instruction), or PC+4 in the case the branch is predicted as not taken(instruction is not a control instruction)
The Decode_input_branch_prediction buffer contains the branch prediction boolean from the fetch stage.
These are simply the output buffers of the fetch stage and need to update according to the fetch_stall control signal at the end of the cycle.

Updating PC buffer in IAG module- In case of a branch misprediction, Deocde_output_PC_temp will be used to overwrite the PC. Else, always Fetch_output_PC_temp will be used to overwrite the PC.

Stall/flush mechanism-

Whenever a stage is stalled, all stages behind it will also be stalled-> the buffers should not be updated at the end of the corresponding cycle. Whenever the pipeline is stalled,a NULL instruction will be pushed in the hazard table.

Stalling the first 2 stages- Fetch and Decode

- **Control hazard**- Push the control signals of the branch instruction in all the queues and update the target PC. Then, push NOPs each in decode, execute, memory, writeback and set stall booleans of decode stage to True. A bubble would be introduced in the pipeline.
- **In the case of RAW Data Hazard when stalling the decode stage is necessary-** To resolve data dependency, push 1 NOP in fetch, execute, memory and writeback and set buffer updation booleans of the fetch and decode to false. A bubble would be introduced in the pipeline. Push control signals of instruction in decode only when the RAW dependency is resolved.

Stalling the first 3 stages- Fetch , decode and execute(occurs in M to E forwarding with stalling)

- To be detected in the decode stage itself. Push 1 NOP each in fetch, MtoE forwarding code in decode control queue(mentioned above), 1 NOP and the control signals of the instruction that is decoded in execute, memory and register update stages queues. The decode stage will set stall booleans of the fetch, decode and execute to True in the next cycle when it dequeues the MtoE code in its signal queue.
  Thus, in the next cycle,the decode unit will stall and set the forwarding booleans and stalling booleans to true and a bubble would be introduced in the pipeline.

Forwarding mechanism:

Data Dependency is detected by the hazard unit when the instruction is decoded. According to this information, the following boolean control signals are generated, which direct data forwarding at the end of the cycle.

- MtoEtoRA- Directs RY to RA
- MtoEtoRB- Directs RY to RB
- EtoEtoRA- Directs RZ to RA
- EtoEtoRB- Directs RZ ot RB
- MtoM- Directs RY to RM

In the case for MtoE with stalling, the mechanism for the same is mentioned above with the stall mechanisms.

For instance, the boolean EtoEtoRA, directs the forwarding of data in the RZ buffer(output of execute stage) to the input buffer RA of the execute stage.
In cases where both stall+forwarding is required(including cases of branch misprediction), the following stall signals are also generated and corresponding NOPs are pushed.

- fetch_stall
- decode_stall
- execute_stall