CDA 5155 Fall 2016 – Project II

Tomasulo Algorithm with Branch Prediction

Assigned: Oct. 10, 2016

Due Date for On-Campus students: 11:55pm, Nov. 26, 2016

Due Date for EDGE students: 11:55pm, Nov. 27, 2016

Introduction

In this project, you are required to implement advanced pipeline using Tomasulo algorithm with out-of-order execution and in-order commit along with a Branch Predictor using the Branch Target Buffer for a processor which executes MIPS32 instructions as defined in Part I of the project. *For CDA4102 students, you need not implement BTB, instead, you stall the pipeline until Branch / Jump is committed.*

Basically, you need to implement the Tomasulo Algorithm as given in Figure 3.14 with two simplifications in handling store-load memory dependences:

1. The memory addresses for all loads and stores are generated in order.
2. When a load is data dependent on an early store (with a matching address), the load stalls until the parent store commits and the data is stored into memory. If a match is found with multiple early stores, the parent is the latest store.

IMPORTANT NOTE: In your simulator, ALL operations in a particular cycle complete at the END of the cycle. All the dependent operations cannot start until the subsequent cycle. For example, a source instruction in the write-result cycle cannot serve the dependent instruction (in RS) if the dependent instruction is issued in the same cycle.

Description of Processor Pipeline:

The pipeline has five stages. Instruction Fetch (IF), Decode and Issue (Issue), Execute (Execute), Write Result (WriteResult) and Commit (Commit). The detailed description of each stage along with the data structure is given next. Note, the simulator implements Tomasulo Algorithm with Reorder Buffer (ROB) similar to the description in Figure 3.14.

(1) Instruction Fetch (IF):

IF fetches one instruction per cycle based on the instruction address in PC and writes the instruction into the instruction queue (IQ) with unlimited size. PC is initialized to address 600.

For CDA5155 students: In the IF stage, a 16-entry, fully-associative BTB is used for the branch prediction. *Both taken and not-taken branches enter the BTB* for future branch prediction. Each entry in the BTB consists of three records: instruction address, branch target address, and a 1-bit predictor. When the current instruction PC is found in BTB with a taken prediction, the target address is loaded into the PC at the end of IF cycle. If the prediction is not-taken, then update PC=PC+4. The PC which is absent from the BTB is treated as non-branch instruction, hence PC=PC+4. The 1-bit predictor is updated

accordingly after the branch is resolved in the Execute stage. The BTB is empty initially. When a branch is not found in BTB, the new branch enters the BTB to replace the oldest entry using the LRU replacement policy. In this case the 1-bit predictor is initialized to '0' and '1' respectively for not-taken and taken branch after the branch is resolved in the Execute stage.

Note, if a branch is not found in BTB, It is treated as predict not-taken. In case the outcome is taken, it will be treated as mis-prediction and will be handled in the commit stage.

Jump also enters the BTB with its target address and the 1-bit predictor is always '1' (predicted taken). Note that you can stop fetching new instructions when the PC is greater than 716 (It should not happen if your program is correct).

For CDA4102 students: You do not need to implement BTB. When a Branch is decoded (in-order) in the 2nd stage, the previous fetched instruction (from PC+4) is canceled and the pipeline stalls to fetch any more instruction. When the Branch is committed, the target address or the fall-through address is used to update the PC and to resume the IF from the correct path in the following cycle.

(2) Decode and Issue (Issue):

Get an instruction from the instruction queue per cycle and decode the instruction (Decode/Issue is in-order). Issue the instruction if there is an empty Reservation Station (RS) and an empty slot in the Reorder Buffer (ROB). Fetch the operands to the RS entry if they are available in either the register file or the ROB. Update the flags to indicate the RS/ROB entry is now in use. The ID of the ROB entry allocated for the result is also sent to the RS, so that the ID can be used to match the result when the result is broadcasted on the CDB. If either the RS or the ROB is full, instruction issue is stalled until both have available entries.

There are 6 slots in the ROB. Instructions are issued and entered ROB in order. The records in each entry are defined as in Figure 3.13. In addition, there is one additional record in ROB for saving the address of the load and store instructions in order to detect store-load dependences (described later).

There are 10 ALU RS for integer ALU instructions, for address generation of load/store, and for branches. (You need not worry about FP instructions.)

There is NO separate load/store unit (buffer). Both Load (LW) and Store (SW) need to allocate one ALU RS for address calculation/memory access and one entry in ROB. Once the memory address is computed, it is broadcasted to the ROB and RS. Note, to simplify the store/load memory dependence, all address calculations for STORE and LOAD instructions must be executed ***in order***.

Branch/Jump instruction needs one entry in the ALU RS and an entry in ROB. Note that Jump does not require any computation. For simplicity, however, it does enter the ALU RS and to be executed in the execution stage in one cycle to figure out the target address.

NOP and BREAK instructions only get one entry in ROB and ready to be committed right after being issued.

(3) Execute (Execute):

After issuing, if one or more of the operands is not yet available, monitor the CDB while waiting in the RS for the source operand(s) to be computed. This step checks for RAW hazards. When both operands

are available in RS, then execute the instruction. Note, all instructions take ONE cycle to execute except for load/store which takes one cycle for Execute (address calculation) and one cycle for memory access (more description later). Recall that there are 10 execution units (ALU) for Load/Store, ALU instruction, and branch/jump. Hence multiple instructions can be executed in parallel.

For Jump/Branch Instructions:

For CDA5155 students: Both Branches and Jump are resolved (for both outcome and target address) in the execution stage. The Branch outcome and the target address along with the predicted outcome are saved on ROB and mark Branch / Jump ready. Meanwhile, update the BTB of the Branch including its target address, and the 1-bit predictor according to the actual outcome.

For CDA4102 students: Basically the same as for CDA5155 students except that there is no update to the BTB.

For Load/Store Instructions:

a. The address generations of Load and Store instruction are executed in order (the order in ROB).

b. Load instruction takes two steps. The first step is address calculation (AC). When the basic register is ready, and all load/store ahead of this load have their addresses ready in the ROB, the load address takes one cycle to be calculated. The second step is memory access, which proceeds under the following conditions.

   i. The load AC has completed, and either of the following two conditions is satisfied:

   ii. There is no early store in ROB with the same address; or

   iii. The matched store in ROB (parent) is committed and the load can proceed to memory access in the following cycle.

   Note, Load waits in RS for AC and memory access. In case of (iii), commit of the parent Store triggers the waiting load in RS for memory access. (In case of matching more than one stores, the latest one is the parent.)

c. Store instruction also takes two steps. The first step is address calculation (AC). When the basic register is ready, and all load/store ahead of this store have their addresses ready in the ROB, the store address takes one cycle to be calculated. The second step is to store the source operands into memory which will take one cycle and occur at the commit stage. Store is marked ready in ROB when both address and source operand data is ready. Note, Store also waits in RS for AC and its source operand. When both are available, the Store in ROB is marked ready (to commit).

(4) Write Result (WriteResult):

When the result is available, write it into the ROB as well as to any RS waiting for this result through the CDB, and mark the operand in the RS as available. We assume CDB has unlimited bandwidth, and there is no CDB hazard at this stage. Instruction spends one cycle at this stage. Store, Jump, Branch, NOP and BREAK instructions bypass this stage. Note, WriteResult also updates the value in ROB and make instructions in ROB ready to commit.

For Load Instruction (WriteResult of address, and WriteResult of Load data):

The generated load address is broadcasted and saved in ROB and RS.

a. If there is no early store address match the load address (take no additional cycle), the load will move to the second step by taking another cycle to access the memory (one cycle), and then repeat the WriteResult stage (one cycle) to broadcast the load results to ROB and RS the same as other WriteResult.

   Therefore, Load takes a total of 3 cycles (if no memory dependence with a prior store), one cycle for address generation (specified in the Execute stage), ZERO cycle for WriteResult load address and detect memory dependence, one cycle for memory access, and one cycle for broadcast load data to RS/ROB. NOTE also that although it takes ZERO cycle for put the load address in ROB, the address generation for subsequent store or load will not start until the next cycle.

b. If there is one or more (the latest store should be used) store address matched the load address, the load will stall (waiting in RS) until the latest store with matched address commit (from ROB). Load can move to memory access in the next cycle.

For Store Instruction (WriteResult of address):

After store address is generated, Store instruction writes address into ROB and RS. If the source operand is also ready (from RS), Store is marked ready in ROB. The Store can be committed in the next cycle. The store WriteResult of address may trigger waiting load in RS in order to generate the load address in order. NOTE, to be consistent with the WriteResult of load address, store address WriteResult to ROB also takes ZERO cycle and the load address (if waiting) can be calculated in the next cycle.

Note that in project I, the data section has all "zero". However, in project II, store may change the content at certain address. Each 32 bits starting from address 716 are considered as a signed integer.

(5) <u>Commit</u>:

One instruction can be committed per cycle from the top of the ROB. Commit must be in-order. There are different sequences of actions at commit stage depending on whether the committing instruction is a "regular" ALU instruction, a load, a store, or a branch/jump. The regular commit occurs when an instruction reaches the head of the ROB and its result is present in the buffer; at this point, the processor updates the register with the result and removes the instruction from the ROB and RS. Load commit is the same as a regular ALU instruction. Committing a store is similar except that main memory is updated rather than a result register. Store updates memory takes one cycle at the commit stage. Any dependent load (with the same address as the store) waiting in the RS is waken up and moved to the memory access in the next cycle. Upon an instruction commits, its entry in the ROB and RS is reclaimed.

For Branch/Jump (CDA5155):

a. If Branch/Jump is predicted correctly, commit the branch and remove Branch from ROB and RS.

b. If branch outcome is taken, but predicted not taken, all instructions after the branch in the ROB are cancelled. Update the PC with target address and commit the Branch. Update BTB. Remove Branch from ROB/RS.

c. If the branch outcome is not taken, but the predicted outcome is taken, all instructions after the branch in the ROB are cancelled. Commit the Branch. Update the PC with PC+4 (note, old PC should goes with the branch for update the PC). Update BTB. Remove Branch from ROB/RS.

d. For Jump, if it missed the BTB (hence the false-through path was taken), update PC with the target address, insert jump into BTB with target address, and mark '1' (taken) in the 1-bit

predictor. All instructions after the Jump in the ROB are cancelled. Commit / Remove Jump from ROB/RS.

For Branch / Jump (CDA4102):

Based on the Branch outcome, update PC with target address or PC+4 which triggers IF of the correct instruction in the next cycle. Remove Branch / Jump from ROB/RS.

Pipeline Units

The pipeline has following function units.

(1) Instruction Queue (IQ): An **unlimited** buffer to hold instructions fetched by instruction fetch stage. The instruction stays in IQ until issue cycle.

(2) Reservation Station (RS) for Integer ALU: Instructions enters RS after being issued and waits in the corresponding RS until the functional unit and all the source operands (via register file or CDB) are available. There are 10 integer ALU RS entries (for 10 integer units), for ALU instructions, address generation for load and store instructions, and for branch and jump.

(3) Reorder Buffer (ROB): It contains 6 entries. All the issued instructions have to stay in ROB till finishing Commit stage.

(4) Register File: There are 32 integer registers. We assume the register file has unlimited read/write ports, so there will be no hardware hazard for register read/write.

(5) Branch Target Buffer (BTB) (for CDA5155 students): We use Branch Target Buffer in our project. There are 16 entries in BTB. The BTB is organized as fully-associative with LRU replacement policy, based on the PC address. Each entry of BTB records the corresponding PC address, the target PC, and a 1-bit predictor.

(6) Main Memory: We assume there are sufficient read/write ports to Main Memory, instruction fetch, data read and data write can happen at the same cycle. Instruction fetch and data read/write all take 1 cycle to finish.

Assumptions

As in part I, assume the program starts at memory location 600 (decimal). PC is initialized to this location for fetching the first instruction out of the memory. The data section begins at address “716”. Following that is a sequence of 32-bit 2’s complement signed integers for the program data up to the end of file. The instruction section won’t exceed “716”. The whole simulation ends when a “BREAK” instruction is committed.

Assume the effective address is the same as the physical memory address. Instruction issue is static and in-order. Instruction commit is in-order. Note that proper pipeline registers must be used to latch intermediate results between pipeline stages.

Guidelines

- Your output should match the sample output format.
- Your simulator should simulate the actual execution and produce the correct results for the given program.
- A program will be considered "complete" once the BREAK instruction leaves the Commit stage.

Command Line

Your simulator (MIPSsim) should provide the following options to users, **dis/sim option is omitted for this part.**

MIPSsim inputfilename outputfilename [-Tm:n]

- Inputfilename - The file name of the binary input file.
- Outputfilename - The file name for printing the output.
- -Tm:n - Optional argument to specify the start (m) and end (n) cycles of simulation output trace.
- -T0:0 indicates that no tracing is to be performed (just print the final state); eliminating the argument specifies that every cycle (complete execution) is to be traced.