COM SCI M151B Week 4

Aidan Jan

October 24, 2024

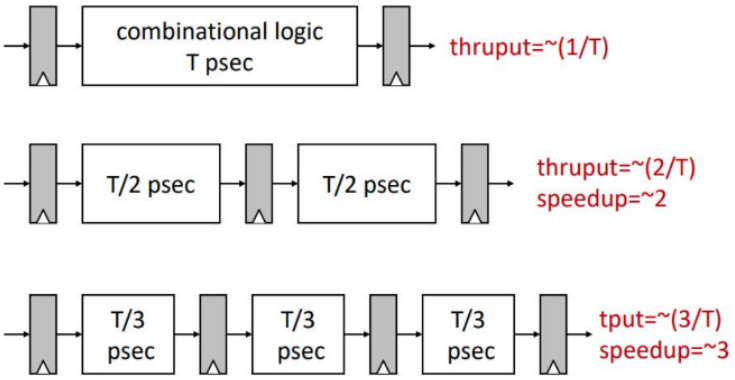
Review

ISA

- First step in the design process
- The ISA (instruction set architecture) is essentially the assembly code. Ex. RISC-V.
- ISA converts to machine code using a standard table.

The Iron Law of Processor Performance

- For single cycle design:
- CPU Time = $InstructionCount \times CyclePerInstruction \times CycleTime$
- Allowing for parallelism (overlaps)

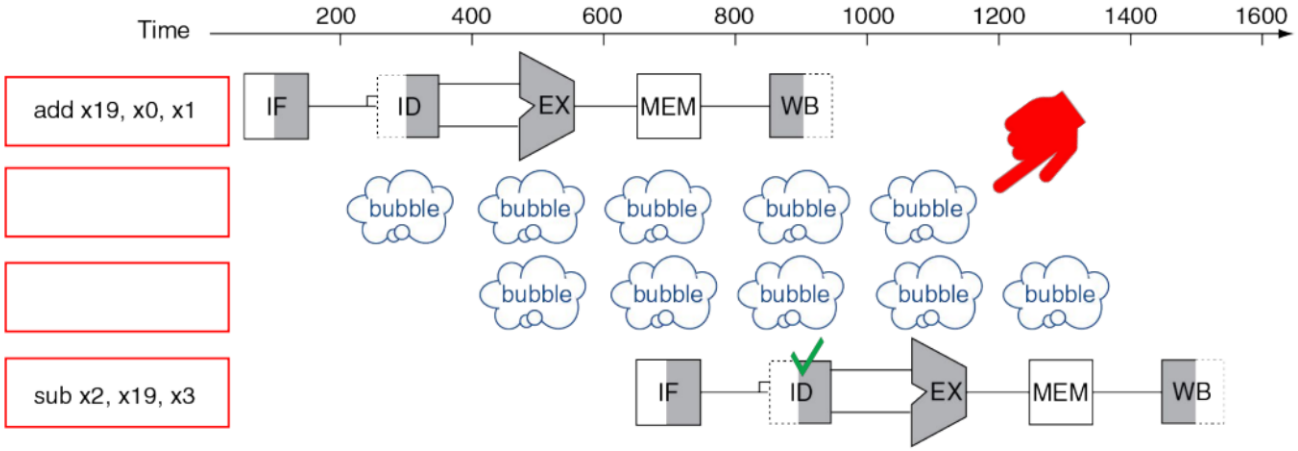


Hazards

Data Hazard - Read after write (RAW)

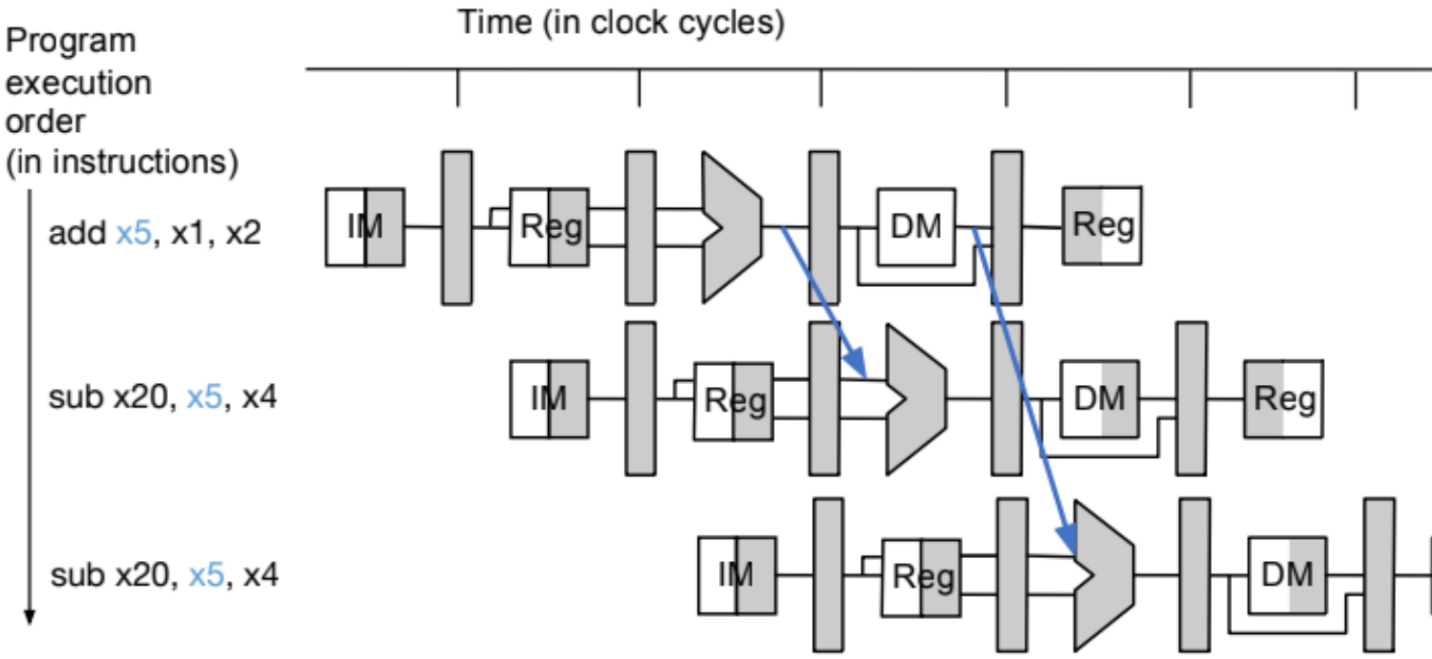
- Writing to a register (rd) and using it (rs1 or rs2) **before** the writing is finished (i.e., rd reaches to the WB stage.)
- To fix this, we either **stall** or **forward**.

Stalling:



- We add bubble instructions (In RISC-V, this is the NOP instruction, which does nothing).
- Alternatively, we can add other, unrelated instructions (e.g., instructions that need to be run that don't involve any of the same registers)

Forwarding:



- Instead of writing the resultant value to the register first, pass it to the ALU directly.
- Forwarding helps prevent stalls!
- See Week 3 Notes for how to detect when to forward.

Branches

- If we get hazards in a branch, we have two options:

- Stalling
- Speculation (not forwarding!)

Stalling:

- We must stall for 3 cycles!
  - When: End of ID stage
  - Where: End of EX or Beg. of Mem stage
  - Whether: End of Mem stage

Speculation/Prediction

- We predict which branch to take. To do this, we predict one as always taken and the other as never taken
- Not taken is easier to check for, because:
  - We don't need branch addresses; not taken means the next address is PC + 4.
  - Most instructions are not branch so they are not taken too!
- If we guess the branch incorrectly, we have to flush!
- The penalty for flushing is 3 cycles, unless we can resolve the branch during the DE stage. This reduces the flush to 1 cycle.

Branch Prediction

How can we improve branch prediction?

- Branch Miss Penalty
  - Resolving branches sooner has reduced this penalty
- Miss rate?
  - Predicting always not-taken has only 30% accuracy.
  - What if we predict always taken?
    - We need nextPC at Fetch stage! (we need to be able to run the next instructions down that branch to save time).

Guessing Always Taken

- Idea: keep track of previous targets and use that to guess!
- If we see a branch instruction before, we know where it jumped. Remember this if we see the same branch again!
- How frequent is this? Is this always correct?

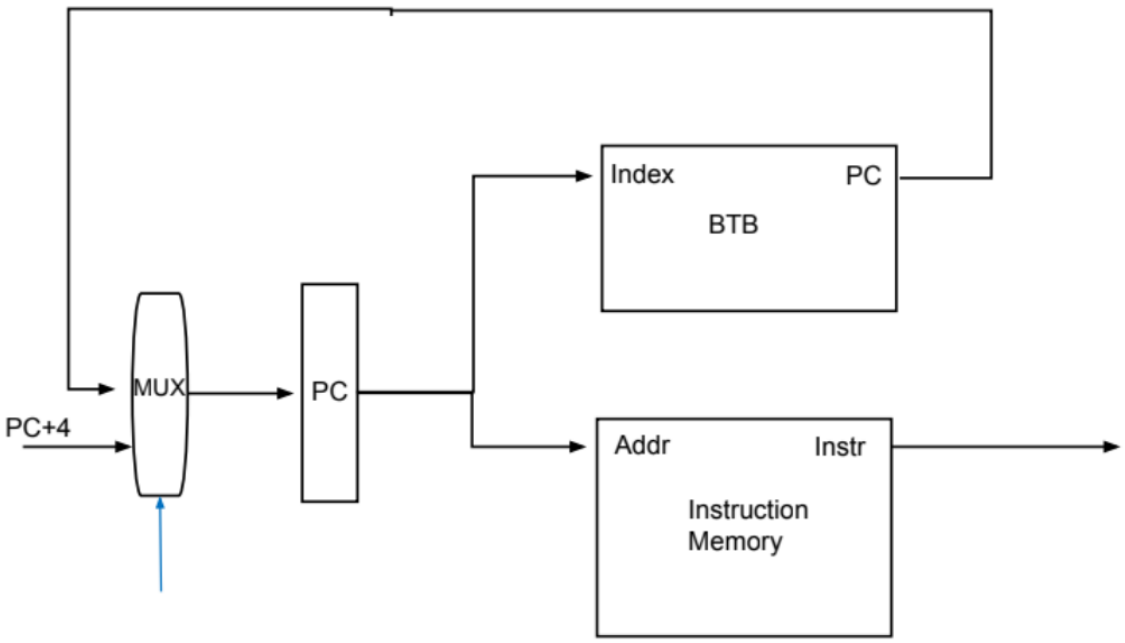
Branch Predictor - Where?

Branch Target Buffer (BTB)

- A table that stores *target addresses.*
- Entries are indexed by PC.
  - The size? Lower bits of PC (to utilize *locality*).

Algorithm for BTB

- For a new PC, record the target address in the table (using the PC as the index).
- Next time (a recurring PC), look up the table (i.e., by using the same index) and predict (i.e., use the stored value as the next PC).



Branch Predictor - When?

- How do we know that this is a control flow instruction?
  - Since we only put control-flow instructions in the table, if the current PC matches with one entry, it is guaranteed that this instruction is a control-flow.
- Who updates the BTB?
  - Once the instruction is **decoded** (i.e., we realize that this is a control-flow instruction), AND the target address is known, we can update the BTB (i.e., we save both the PC and the target address.)

Quick Recap

- Miss Penalty
  - Move the outcome to DE
  - Always not taken!
- Miss Rate
  - Always not taken is only 30% accurate, so let's do Taken.
  - For that we need nextPC in FE.
  - For that we need a table (aka BTB)!

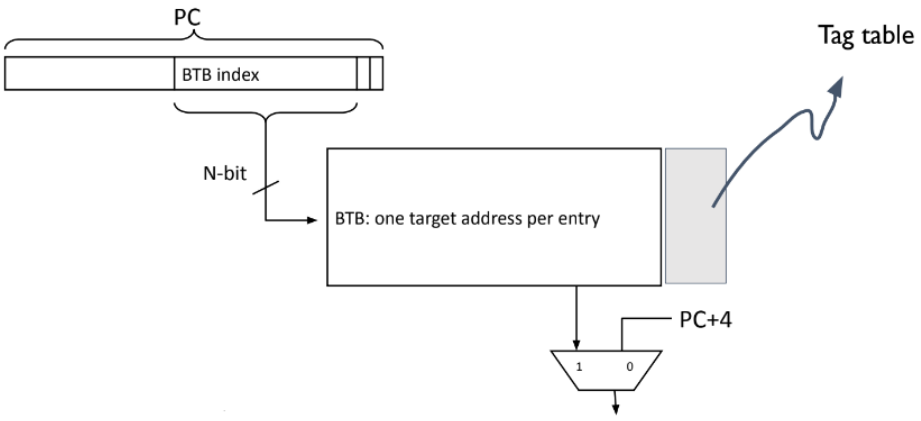
Branch vs. Non-Branch

- Non-Branch (we can ignore this; next instruction is PC + 4)
- Branch
  - Seen before (look up table)
  - Not seen before

Why would this work?

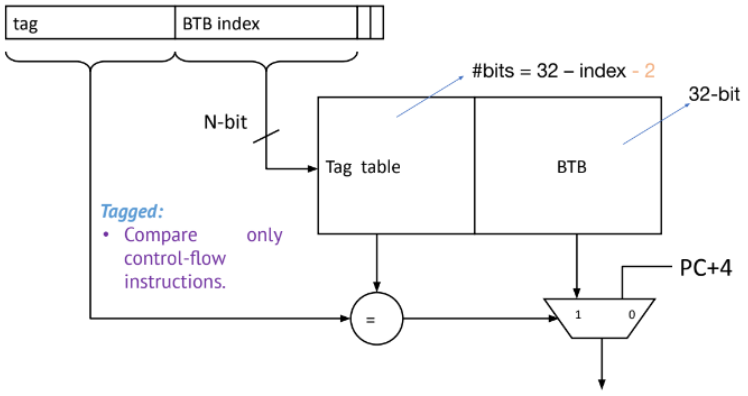
- Two rules: **temporal** and **spatial** locality
  - **Temporal Locality:** If you just did something, very likely you will do the same again **soon**!
  - **Spatial Locality:** If you used something, very likely you will need/use similar/related things.
- Programs are typically very predictable (90% of the time spent in only 10% of the code)

BTB



- The Tag table is implemented to make sure that the PC matches with the entry.
- The tag table uses the PC itself as the tag!
  - (PC, target address) pairs are unique.

Tagged BTB



- We now have a tag (PC value) that maps to the branch location.
- The N decides the number of entries we would have.
- The tag table saves a lot of memory. With the normal BTB, if we wanted to ensure it worked for the entire code, then we would need as many entries as lines of code. (which is a lot)
- Instead, we can tag only the branching statements, and store only the last N bits of the instruction as a tag. Ideally, two instructions would never have the same N lower bits, but if it does, then we pay the penalty of flushing incorrect instructions.
- We can optimize even more!

Types of Control-Flow Instructions

Control Flow Inst.	**Direction at fetch time**	**# possible next fetch**	**When next fetch addr. resolved?**
Conditional	Unknown	2	Decode (rs1 == rs2)
Unconditional	Always taken	1	Decode (PC + offset)
Call	Always taken	1	Decode (PC + offset)
Return	Always taken	Many	Decode (reg. dependent)
Indirect	Always taken	Many	Decode (reg. dependent)

BTB

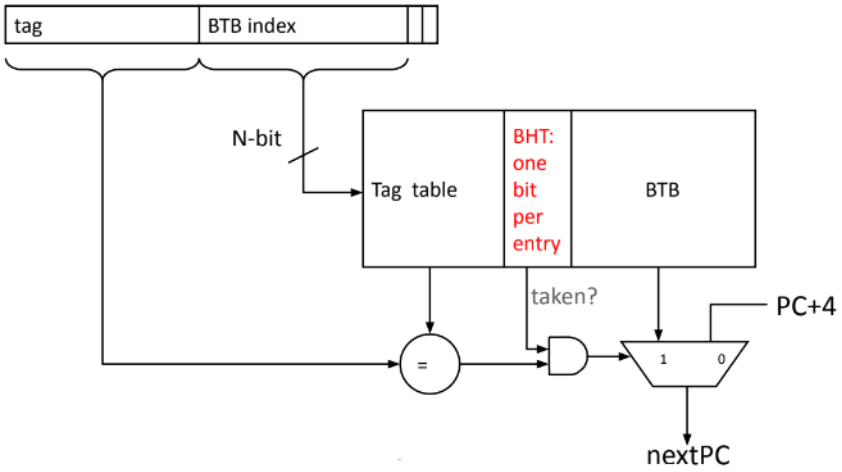
- if r-type, i-type, lw, sw:
  - PC = PC + 4 → PC won't match in BTB
- if jal
  - PC = PC + offset → this is what BTB would give us!
- if beq
  - PC = PC + offset → we still assume always-taken, but at least, we have the **address** now!
- if ret, call, jalr
  - PC = ? → many different possible outcomes. BTB won't be that helpful here (it can only store one of them).

For conditional branches we are doing always taken (70% accurate) which is a lot better than always not taken (30%). **Can we do better?**

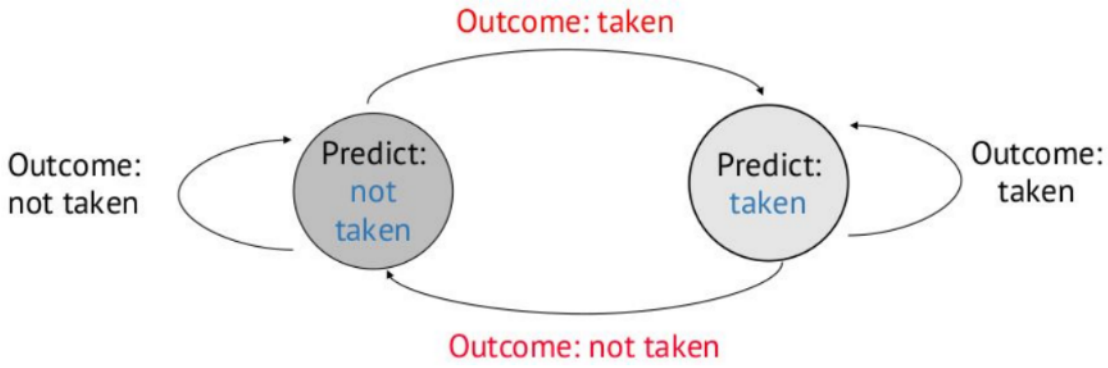
How to improve the prediction accuracy?

- Use a **branch predictor**
  - Predict the outcome of the branch *dynamically*!
- How? Use history!
  - Recurring branches (e.g., backward) repeat their pattern/behavior.
  - We can track previous branches using a table to store their outcomes; if the new branch exists in that table, use that outcome.

Branch History Table (BHT)



- For each control statement, we store a single bit (1 or 0) on whether or not the branch was taken.
- For all control statements other than conditional, we store a 1, because those are always taken.
- For conditionals, we update the last bit *based on* the last outcome of the branch.

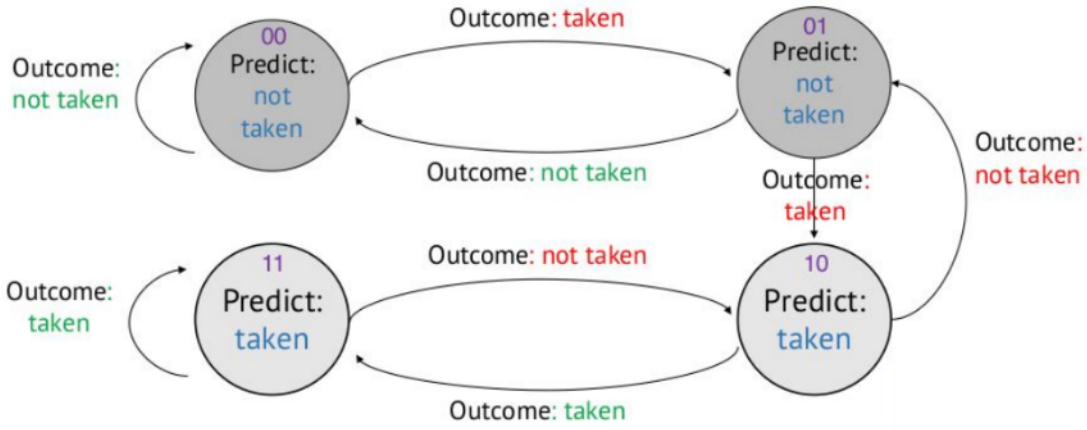


- Issue: they change too quickly!

```
Pattern:    TTTTTNTTTT
Prediction: TTTTTTNTTT
```

- Even for heavily biased patterns (i.e., easy to predict), we have only 80% accuracy.

Bimodal BHT (2-bit predictor)



- We now have better predictions!

```
Pattern:    TTTTTNTTTT
Prediction: TTTTTTTTTT
```

- About 85-90% accuracy for **many** programs with 2-bit counter based prediction.
- Need *two* bits for each entry, **more overhead!**

Ways to Improve BHT accuracy

- Use a more complex Branch Prediction
- We won't get further than this, but here are some ideas:
  - History of history
    - Two-level prediction: all (global) and grouped (local). Make the decision based on both patterns.
    - Tournament: alternatively choose between global and local.
  - Compiler/program directed
    - Use program information to train the predictor (or add some hints.)
  - Machine learning
  - Hybrid
    - Combine all these ideas, and dynamically pick the one that is most accurate at the time (e.g., TAGE)

Return Stack Buffer

- We have a way to deal with conditionals, unconditionals, and calls. What about return?
- This is an easy problem to solve. Keep a hardware "stack" of return addresses.
  - Every time we `call` a function, push the address of the call to the stack!
  - When return is called, we pop the last address from the stack and that will be the correct return address.
  - This is accurate 100% of the time!
- Minor changes to BTB is needed
- Typically, 4-8 entries is enough, but minor changes can be made to support tail recursion.

What about Indirects (JALR)?

- We need additional predictors for this.
- However, increasing complexity adds overhead.
- We usually just ignore this case and tank the loss.