
The Final Lab

Announcements

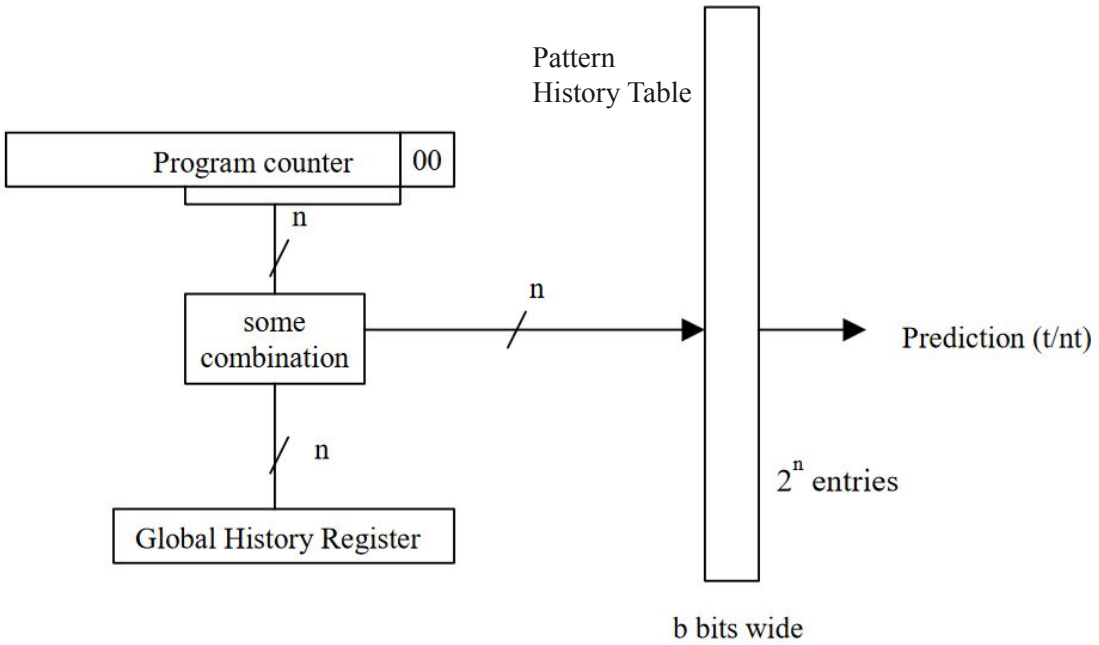
- There was another patch
  - More configuration options
  - Everything is now clock-dependent
  - Better Ventilator (it has banked mem)
- The competition begins!
  - Deadline on April 29th
  - Required 20 pts of Adv features
- What we're talking about today
  - Superscalar
  - Branch Prediction
    - Early Branch Recovery
  - Memory subsystem
  - RISC-V Extensions

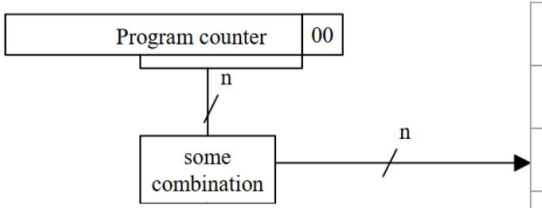
Branch Predictors (BPs)

Why are advanced predictors good?

- Flush penalty is very high
- We have to flush anyways (static non taken)
- Let’s try to do it less often now

BP + BTB General Structure





Index	Jump Address
‘0	Some PC
…	Some PC
‘1	Some PC

Integration into Processor

- Predict next PC
  - Take BTB address if BTB hit and predictor says take
    - If no BTB hit, then you can't take anyways
- Update BTB
  - Update on commit or after finishing address calculation?
  - Update only when the branch is taken or whenever an address is calculated?
    - Avoid BTB congestion
- Update Predictor
  - When you commit or calculate comparison result?
  - Carry the prediction and/or other relevant information needed to update BP throughout your processor along with instruction
    - 2-ports or arbiter if you want to read PHT when updating predictor

Some stuffs to consider

- Experiment with different predictors types and BTB sizes
- Larger predictors means larger SRAMs, but read/update still need to occur together
  - Ports scales almost linearly with area for SRAMs
  - Can use arbiter and multiple SRAM banks for predictor

Early Branch Recovery

Your Pipeline is Long

- For a misprediction to be recognized, instruction has to make it to end of ROB
  - In mp_pipeline, we could flush from execute - why not now?
- Lots of cycles that we don’t want to wait - how to make this faster?

Early Branch Recovery

- We ideally do something similar to MP Pipeline
  - Earliest time that branch is recognized as mispredicted is when it is in execute
  - Want to be able to flush from the execute stage
- There are complications here
  - How to know that the instruction that is mispredicted is *also not mispredicted*??
  - Need to be able to reset the physical state - re-free regs, roll back translation
  - ROB needs to discard all instructions after the mispredict - doing this can be expensive
  - Any instructions dependent on the mispredict should be evicted from the reservation stations
  - Complexity becomes exacerbated by the fact that there can be *transitive* dependencies on the mispredict

Divide and Conquer this thing

- What state do we need to capture granularly?
  - RAT state needs to be captured for every *mispredictable* control instruction
  - Freelist needs to also be backed up… but this can actually be simplified. How?
- ROB clearing can be as simple as resetting the tail pointer to the ROB index assigned to mispred
- Reservation station clearing *does not all need to be done at once*
  - If careful, we can tag each instruction with prior branch dependency, broadcast the mispredicted tags incrementally
  - Need to be careful to avoid ROB contention
- A simple implementation can avoid granular state altogether
  - Split Flush (or as I like to call it, DonkEBR)

Caches

Banked Cache

- mp_cache has one big SRAM
- What if you split that SRAM into multiple SRAMs?
    - ie banks of SRAM
- Each bank controls its own address space
- Have to arbitrate to bmem for all banks
- Multiple ports for read/write
    - Multiple inflight load/store instructions

Non-blocking Cache

- Kinda like competition memory (BMEM)
  - 32 bit data buses
  - 1 cycle transaction requests
- Keeps queue of incoming requests
- Does mem requests OoO optimally
- Outputs reads asynchronously with a valid/address signal

- Few ways to incorporate this with other advanced features
  - Fire and forget - need to rollback store mis-forwards
  - Memory Disambiguation - precommit store buffer shenanigans
  - Banked Cache
  - Ask your TA for the points awarded if you want to pursue this kinda stuff

Pipelined Cache

- Memory timing diagram is ordinary_memory from mp_pipeline
  - Or magic_memory with delayed response times
- For read-only cache:
  - Don’t go from check_tag -> idle -> check_tag
  - Want to check next request 1 clock cycle after hit
- Have to handle SRAM delay for write requests
  - Cache write forwarding

Memory Subsystem

Memory Disambiguation

- Simple store forwarding - Think MT2 question
  - Only forward SW instructions to previous load instructions with same address in LSQ
- Complex Store Forwarding
  - Builds on simple store forwarding. Now consider sb, sh, sw and [r/w]mask to do store to load forwarding
- Store buffer
  - Place all your stores into a secondary store buffer (FIFO) instead of D-cache
  - Store buffer takes care of doing write to memory while rest of processor moves on
  - For loads, check if data is in store buffer first then D-cache
  - Can be placed either pre-commit or post-commit

RISC-V Extenstions

RISC-V Extensions

- C-Extension
  - Must handle 8-bit, 16-bit instruction lengths
  - Instructions can be across multiple memory words
- M-extension
  - Wikipedia has good descriptions of advanced multipliers/divider
  - Use scripts to generate SV
  - Pipeline out the FU to handle multiple computations at once
- F-extension
  - Synopsys IPs: /software/Synopsys-2021_x86_64/icc/R-2020.09-SP4/dw
    - Look at examples to learn how to declare IP
- We will have some testcases compiled specially for each extension in competition testcases

Superscalar

There's always a bottleneck somewhere

- We've established static-taken is not great
- Your fetch could probably be better
- The memory unit could definitely be optimized
- *There's no way that's all we can do… right?*
    - Assuming that we have a good pipeline already, we want to **widen** it - more in, more out per cycle

The Widened Pipeline

- *Superscalar* refers to any OoO processor that is able to accommodate >1 instructions per cycle
- Ideally: better ROB usage, better FU utilization, better throughput
- Sounds cool

The Superscalar Tax is Non-Trivial

- More complex logic in nearly every stage
  - Aside from development time, can expose bugs in the rest of the processor
  - Logic complexity will lead to decreased FMAX
  - Can potentially add more registers and retime, but now your IPC can go down
- Branch prediction is harder to do well, can lead to lower pipeline utilization
- Opportunity cost is high
- Sounds cool?

The Big Things to Consider

- ***Instruction Grouping:*** Do we want to keep the instruction groups from fetch contiguous?
  - Maybe not: when do we separate the fetch packet?
  - Earlier separation can lead to better throughput past that point, but is also expensive logic
  - Expect lower frequencies in a stage after you separate a fetch packet
- ***Width:*** How many instructions can be fetched/committed at once?
  - Usually no. of fetched instructions will be relevant up until your *dispatch*
  - Fetch and commit width *can be different* - try different combinations!
  - Consider making both parametrized - the loop structures shouldn't be too different

How do each of your stages change?

- ***Fetch:*** needs to be able to pull multiple instructions at once
- ***Decode:*** needs to account for potential transitive dependencies between multiple instructions
- ***Dispatch:*** each issue queue needs to be able to enqueue multiple instructions at once
  - Potential for the entire fetch packet to not be dispatched at once - what to do in this case?
- ***Issue:*** No changes here!
- ***ROB:*** Needs to both *enqueue* and *dequeue* multiple instructions at once
  - Instruction grouping will play a big factor here - do I *need* to make the ROB one-wide?
  - For an N-wide ROB, internal fragmentation is now an issue - thanks 391!

Other effects of Superscalar

- Misaligned instruction fetches from the cache
  - Can be mitigated with a banked cache, or simply outlawed
- Need to find a way to keep accurate branch prediction - this is hard.
  - Can limit the number of branches per fetch packet
  - Can issue static predictions on some instructions
  - Can simply make the branch predictor *wider* - how to accommodate the update logic for this?
- For full utilization, need to add more functional units and reservation stations
  - This is hard to do in the case of the memory unit
  - ROB depth should be made deeper as well to take advantage of memory delays
  - Can lead to higher fanout in contested structures - consider grouping functional units (check BOOM)