ECE 411

Spring 2026

Review Notes for ECE 411 (Computer Organization and Design)

These notes aren't fully comprehensive.

Aniketh Tarikonda (aniketh8@illinois.edu)

Notes based off of:

- Lectures by Prof. Kumar
- *Computer Organization and Design: The Hardware/Software Interface, 5th Edition, David Patterson and John Hennessy*
- *Computer Architecture: A Quantitative Approach, 5th Edition, David Patterson and John Hennessy*
- *Processor Microarchitecture: An Implementation Perspective*

Contents

1 Caches

- Important aspect of memory hierarchy; hides long DRAM latencies due to program memory generally exhibiting **spatial and temporal locality**
- Organized in hierarchies (Level 1 to Level 3, usually)
- Lower level caches are closer to core, have lower associativity, store less data, and have the lowest nominal latency; L2/L3 caches are larger, hold both instructions/program data
- Program data/instruction addresses are usually **virtual addresses**. You can have virtually-indexed, virtually-tagged (VIVT) caches, but they suffer from aliasing (multiple processes sharing virtual addresses)
- To avoid aliasing, we physically tag addresses (VIPT/PIPT)
- In general, instruction cache (icache) access is simpler than data cache (dcache) because of memory disambiguation logic necessary with dcache.

1.1 Address Translation

How do we map back and forth between physical and virtual addresses?

- The processor has a paging mechanism to virtualize the linear physical address space, which involves dividing the address space into pages (can lie in main memory or external storage), and it keeps a mapping of virtual to physical pages in the page table (main memory).
- Instead of multiple memory accesses to decode addresses, we use a **translation lookaside buffer (TLB)**, which is effectively a page table cache.
  - ~ 1 cycle access time, indexed by page number of virtual address, returns physical page number
  - associativity and size varies (can have tens or hundreds of entries)

1.2 Cache Organization

- consists of a tag array and a data array (implementation wise, data array is a big SRAM block, tag array can be SRAM or flip-flops)
- cache consists of sets, each set having N blocks where N is the degree of associativity
  - direct-mapped cache means $N = 1$
  - tag array has the same logical organization as data array
  - K least-significant bits ($K = \log_2(Q)$ where Q is the block size) are used to identify specified bytes within each block
  - next M bits ($M = \log_2(S)$ where S is the number of sets) is the index.
- Different addresses can map to the same data array, so we need to maintain the tag array in order to map entries back to memory addresses.
- Cache access can often be the critical path in a processor design–there's value in pipelining it (separate into address calculation/disambiguation/access stages)
  - in this pipelined design, we can either access tag array/data array in parallel or sequentially (in different pipeline stages)
  - parallel implementation has higher power consumption, lower max clock freq., but is also one clock cycle faster

- use parallel for in-order processors which can't hide latency well, and serial for out-of-order (OoO) processors which can.
- Direct-mapped caches are fast, but suffer more conflict misses than associative caches. Higher associativity means less conflict misses, but also larger muxes and more access latency.
- Data caches should be non-blocking, instruction caches should be blocking and single-ported.

1.3 Lockup-Free/Non-Blocking Caches

- Blocking caches will stall the processor until a request is serviced.
- Executing instructions while cache misses are serviced requires dependency-tracking mechanisms, such as those in OoO processors.
- **Lockup-Free/Non-Blocking** caches allow load/store operations to be issued, despite pending cache misses; one method of implementing this is through **Miss Status Holding Registers (MSHRs)**
  - MSHRs hold information about pending misses

1.3.1 Implicitly vs. Explicitly Addressed MSHRs

- Implicitly-Addressed MSHRs
  - simplest design (low area/power consumption)
  - each MSHR holds data array block address and a valid bit (set on a primary/compulsory miss)
  - comparator to match future (secondary) misses of a block to the same MSHR
  - each cache block consists of N 32-bit or 64-bit words, so the MSHR contains N entries to record miss information (e.g. destination register of the instruction, size of load instruction)
  - can only support **one** outstanding miss per word because the block offset is implied by the position within the MSHR
- Explicitly-Addressed MSHRs
  - add block offset to each MSHR field–position within MSHR no longer implies address
  - we can have an arbitrarily high number of outstanding misses for a single block, without causing structural stalls

1.3.2 In-Cache MSHRs

- instead of allocating separate SRAM/register arrays for MSHRs, we make use of pre-existing cache resources
- requires one more bit to be added to the tag array–the transient bit–to indicate that a block is currently being fetched
- when in transient mode, the tag holds the block address, and the data array holds MSHR information
- can either be implicitly or explicitly addressed

1.4 Multiported Caches

- Can we do >1 load/store instruction per cycle?
- The naive implementation: doubling all control/data paths (double way muxes, double tag comparators, double aligners, etc.)
  - significantly increases cache cycle time
  - doubling tag/data arrays means changes/accesses to one cache needs to be broadcast to the other
- Virtual Multiporting involves issuing a memory operation on both the rising and falling edge of a clock cycle
  - does not scale well to higher frequencies
- **Multibanking**
  - divide cache into smaller banks, each of which are single ported.
  - multiple instructions can be issued per cycle assuming they access separate banks (otherwise, you end up with bank conflicts)
  - does not require tag/data arrays to have multiple ports
  - probably the best method of emulating multiported caches today

2 Instruction Fetch

- High-performance cores need to sustain 1 instruction fetched per cycle
  - this seems alright, until we consider branch/jmp/return from subroutine instructions
  - these instructions require special prediction units/speculative execution
- Fetch address prediction works in two steps
  - predicting direction of branch (taken/not-taken) - this is done with a **Branch Predictor**
  - predicting target address of branch - this is done with a **Branch Target Buffer (BTB)**
  - some processors treat return from subroutine as a special case, keeping track of these using a **return address stack (RAS)**

2.1 Instruction Cache

- Usually VIPT or PIPT
- Usually set-associative
- Superscalar designs generally read consecutive bytes from the same cache line
  - decoding logic is trivial for fixed-size RISC instructions
- Conventionally, instructions are stored in the same order as in the binary
  - trace caches store instructions in dynamic order

2.2 Branch Target Buffer (BTB)

- Most branch instructions have PC-relative offsets, which means we could just calculate the target address with an adder (PC + offset)
  - we would have to then have a separate pipeline stage (within fetch unit) for this adder
  - for high-performance, high-frequency processors this would incur a 1-cycle bubble every time a branch is taken (bad for performance)

- Thus, there is value in predicting target addresses with a BTB
- The BTB is normally implemented as a cache; it consists of a table indexed by fetch address, only containing a target address if the specified entry is a branch instruction
- The prediction is the target address as it was executed previously.

2.3 Return Address Stack (RAS)

- BTB could predict returns from subroutine pretty well, but RAS is simpler and has even more accuracy.
- RAS is a stack structure (LIFO); when a subroutine call is fetched, the address of the next instruction is pushed into the stack.
- When a return instruction is fetched, the most recent (top of stack) entry is popped and used as the target address.
- RAS is small, only consisting of a few tens of entries.
  - when the RAS is full, the oldest (bottom of stack) entry is overwritten
  - $\text{subroutine nesting level} \leq \text{RAS size}$ ...generally (the exception when we have deep recursive subroutines)

2.4 Branch Predictors (BPs)

- Flushing pipelines can incur a significant performance penalty, so we want really good BPs
- Static branch prediction has the smallest hardware overhead, but has the worst performance and requires profiling the code beforehand to see high-level trends in branch direction.
- All modern processors have Dynamic BPs instead.

2.4.1 Local BP

- Consists of a table containing 2^N entries and 2 bits per entry, where N is chosen as the N least-significant bits of the instruction address.
- The 2 bits represent the four states of a saturating counter FSM (Strongly NT $\longrightarrow$ Weakly NT $\longrightarrow$ Weakly Taken $\longrightarrow$ Strongly Taken)
- Called 'local' as the decision of one branch doesn't effect the other.
- Large programs can have aliasing (multiple addresses mapping to one entry in the BP) causing degradation in performance.
- Sometimes adequate, often times not. Accuracy generally ranges from 80% to 99%

2.4.2 Correlating BP and gshare

- Similar to local BP, but add a register (Branch Global History) that stores the outcome of the most recent branch instructions (10-20 past outcomes suffices, usually).
- This Branch Global History (BGH) is combined with PC through a hashing function, generating an index for a table consisting of 2-bit saturating counters.
  - this approach is called **gshare**
  - its been found that bitwise XOR between BGH and the least-significant bits of PC is a simple/effective hashing method which minimizes aliasing.

2.4.3 Hybrid BP

- Local BP has a shorter warm-up time than Correlating BP/gshare. Thus, in certain scenarios (e.g. right after a context switch), it may be beneficial to initially use Local BP, and later swap to a Correlating BP.
- Implementation-wise, consists of two branch predictors (local and correlating) and a selector, itself behaving like a branch predictor.
  - the select can be implemented as a table of 2-bit saturating counters where the MSB determines which BP to use.
  - selector drives a MUX, which selects the choice of BP to use.

3 Decode

- complexity of decode unit depends **heavily** on ISA
- RISC processors have simple decode units (usually single-cycle)
  - few control signals for the pipeline
  - as a result, decode is straightforward to implement for high-performance, high-frequency designs.

3.1 x86 Decoding

- x86 is CISC, so the decode unit is significantly more complex and multi-cycle
- large x86 instructions can be broken down into smaller, RISC-like instructions which can easily be passed into an out-of-ordered execution engine.
  - This process of breaking down instructions is called **Dynamic Translation**
  - First started with the AMD K5/Intel P6
  - nearly all x86 processors use dynamic translation today

4 Allocation

- This phase handles register renaming and instruction dispatch.
- Instruction Dispatch reserves resources that will be later used by the instruction include:
  - Entries in the Issue Queue
  - Entries in the Re-order Buffer
  - Entries in the Load/Store Queue
- In the case that there are multiple issue queues associated with multiple functional units (FUs), allocation should determine which FU to map an instruction to.
- Register Renaming is specific to OoO architectures, used as a method to take advantage of Instruction-Level Parallelism (ILP).
  - There are two types of dependencies between instructions: data dependencies (RAW) and name dependencies (WAR, WAW)
  - Data dependencies must be in-order (nothing we can do about that)
  - Name dependencies, on the other hand, are *false* dependencies–only an issue because of constrained storage.

  - Thus, a potential solution is to have (issued) instructions write in a different storage location
  - There are huge performance benefits associated with Register Renaming, due to the limited register file size of modern processors (e.g. 32 integer registers for RV32I) leading to a high prevalence of dependencies.
- Tomasulo implemented register renaming using the identifier of the reservation station (RS) of the destination of the instruction. This isn't really done today.

4.1 Register Renaming via the Re-order Buffer (ROB)

- ROB is basically a FIFO structure, with tail allocated for each new instruction, and head released on instruction commit.
- You have a Register Rename Table, **Architectural Register File (ARF, specified by ISA)**, and **ROB** (in a later backend pipeline stage)
- ROB contains results of uncommitted instructions, ARF contains latest committed results
- For each entry in the rename table, whose size is fixed with respect to ARF, we have information on whether latest definition is in ARF or ROB, as well as the position within ROB (latter case).
- During the commit stage, the value is copied ROB $\longrightarrow$ ARF

4.2 Register Renaming via the Rename Buffer

- Downside of Register Renaming in ROB is that instructions that don't produce a register result take up unnecessary space in the ROB.
- Separate structure (Rename Buffer) for results of in-flight (uncommitted) instructions. This way, only instructions which write back to registers take up storage space.
- Otherwise, implementation is similar to Register Renaming via ROB.

4.3 Merged Register File (ERR-coded)

- Single register file holding both committed and speculated values. Thus, size is larger than ARF.
- Each register is either free or allocated.
  - Free registers are stored in a free list (circular buffer storing all identifiers)
  - Allocated registers can either store committed or uncommitted/speculative data
- **Register Alias Table (RAT)/ Register Map Table** - maps between physical register file (PRF) and ARF.
- When an instruction is renamed, we look at the RAT to find the source operands; if the instruction writes to a register, we allocate a register on the free list–if this isn't possible, we need to stall.
  - The destination operand is renamed to this new free register, and the RAT is updated accordingly.
  - Physical registers are freed when the following instruction which shares the same **architectural** destination register commits.

4.4 Register File Reads

- We can either *read before issue* or *read after issue*.
- If we read before issue, we selectively read available values, and unavailable operands are marked as such in the issue queue, and later obtained via data bypassing.
  - Allows for a fewer number of ports in the register file, comes at the cost of larger issue queues (area/power)
- If we read after queue, issue queue stores identifiers of register source operands, and the register file is actually read once the instruction is issued to be executed. Once again, we use data bypassing for uncommitted results.
  - Requires more ports in the register file, but requires operands to be read only once.
- ROB/Rename Table-based renaming works well with read before issue. Merged Register File-based renaming works well with read after issue.

5 Issue Stage

- In the case of an in-order processor, a typical implementation of the issue stage would just use scoreboarding:
  - Data dependence table which keeps track if operands are unavailable, or available (can either be sent from some bypass level, or available in register file)
  - Resource availability table which keeps track if functional units are available.
  - Tables indexed using source register identifiers of the instruction to be issued.