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 Intro 1
2 Pipelining 2
  2.1 Data Hazards 2
    2.1.1 Forwarding 3
  2.2 Branch Hazards 3
    2.2.1 Branch Predictor (BP) 4
3 Caches 6
  3.1 Address Translation 7
  3.2 Cache Organization 7
    3.2.1 Writes 8
    3.2.2 Replacement Policies 9
    3.2.3 Lockup-Free/Non-Blocking Caches 9
    3.2.4 Multiported Caches 10
4 Decode 10
  4.1 x86 Decoding 10
5 Allocation 11
  5.1 Register Renaming 11
    5.1.1 Register Renaming via the Re-order Buffer 11
    5.1.2 Register Renaming via the Rename Buffer 12
    5.1.3 Merged Register File 12
    5.1.4 Register File Reads 12
6 Issue Stage 12

1 Intro

- Von Neumann Architecture - Instructions and Data are treated the same.
  - most basic computer architecture, LC-3 is based off of a classical Von Neumann Architecture
  - also has microsequencers
- Harvard Architecture - Separate instruction and data memories
  - used in DSPs today
- Dataflow Architecture - less "sequential", instructions are sent for execution the moment their operands are ready.
  - while a strict dataflow architecture isn't commonly used today, it rears its head through dynamic scheduling/out-of-ordered execution.
  - requires a different programming paradigm - more graphical programs, with operations being the nodes
- A typical CPU follows Fetch ⟶ Decode ⟶ Execute ⟶ Memory ⟶ Writeback

Instruction Set Architecture (ISA) - the contract/interface between hardware and software

- A good ISA is portable, general, provides convenient functionality to higher levels, allows an efficient implementation in lower levels
- Semantic Gap: How close are instructions, data types, addressing modes to the high-level language?
  - Who is the burden being placed upon? The hardware designer or the software designer?
  - You can somewhat "virtualize" ISA - having separate ISAs for the hardware and software, and a translator between the two. Ex: Rosetta on Apple Silicon
- Gray areas exist, x86 is so dominant that it can get around nightmare-ish decode logic. Legacy ISAs still exist because the cost of migrating to newer ISAs outweights the benefits.
- Using a licensed ISA is often expensive (cough cough arm), but often more commercially viable than using custom ISAs.
- RISC vs CISC - RISC allows for simpler hardware implementations (generally), CISC can have better code density.
- ISAs don't have to be strictly register-register (although register-register is pretty much exclusively used today)–you can have accumulator/stack-based architectures.
- Number of (Architectural) Registers is entirely dependent on ISA
  - More registers allow better register allocation, less compiler tricks, fewer saves/restores
  - More registers means larger instruction size, and larger area penalties
- Indirection: We can do a lot of funky stuff at the hardware level, and as long as the interface (ISA) is being upheld, its all good.
  - Therein lies the difference between ISA and Microarchitecture.

    - E.g. `add` instruction vs. adder implementation (CLA, CRA, hierarchical, etc.)

Big Endian vs Little Endian
- Big Endian: MSB at lowest address
- LIttle Endian: LSB at lowest address
- RISC-V is Little Endian.

How do we quantify performance?
- In general, performance is given by:

$$\frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}} \tag{1}$$

Different terms in this equations are goverened by different levels of abstraction. $\frac{\text{instructions}}{\text{program}}$ is dependent on the programmer. $\frac{\text{cycles}}{\text{instruction}}$ is dependent on microarchitecture and system architecture. $\frac{\text{seconds}}{\text{cycle}}$ is dependent on microarchitecture, pipeline depth, circuit design, semiconductor/VLSI technologies.

Multicycle vs Single-Cycle Processoors
- f_{clk} of a single-cycle processor is limited by the longest instruction (often load/stores)
- f_{clk} of a multi-cycle processor will often be a lot faster, although the time taken for the longest instruction will often be longer (because of additional register overheads).
- Although multi-cycle processor is a lot better, it suffers from hardware underutilization and throughput issues. If only there was a way to solve both of these issues…

2 Pipelining
- improves throughput, hardware utilization, latency of instructions stays (mostly) the same.
- potential speedup correlated with the number of pipeline stages
    - with more pipeline stages, the number of cycles required to saturate the pipeline increases.
- each instruction takes a uniform number of cycles to pass through a pipelined processor.
- Pipelining introduces a whole class of issues: **hazards**

There are three types of hazards
- Structural Hazard: a hardware resource is currently busy
- Data Hazard: Need to wait for a previous instruction to read/write
    - RAW (read-after-write) is a true data dependency
    - WAW/WAR are false dependencies
- Control Hazards: branches, jumps

2.1 Data Hazards
- One method of elimiminating data hazards in a pipelined processor is to increase the distance between dependent instructions, e.g. by inserting `nop`s between them.
    - introduces performance, portability, code density penalties

    - programmer/compiler needs to be aware of the exact microarchitecture so they can insert the correct number of nops, can we really trust them?

- Rather than inserting nops, you can also re-order code to increase distance between dependent instructions. This is annoying to do however, both for the compiler and the programmer.

- We can try *stalling* once a data dependency is detected. This is better in some aspects than manually inserting nops, but still has poor performance.
    - Method 1: Register Dependency Table
        - When an instruction modifies a particular `rd` , keep track of it in a dependency table (flip a used bit to 1).
        - Subsequent instructions look at the table, stall if one of their registers is currently being used.
        - When an instruction commits, flip the `rd` 's used bit to 0
        - Issue: enforces naming hazards, sometimes instructions that don't have strict data dependencies are forced to stall because of shared `rd`

- The better approach to data hazards: **Forwarding**

2.1.1 Forwarding

- also called data bypassing, forwarding the result of a previous instruction to an earlier stage.
    - ignore values read from register file - they contain values from a previous archiectural state

- Implementationally, we add muxes to the EX-stage, WB-stage[1].
    - Control logic for the muxes are generated by a forwarding unit.

- Forwarding unit can be implemented with a priority encoder
    - Pipeline stages closer to EX given priority over those further away.
    - If `rd` matches select forwarded data from that particular pipeline stage.

- Extra considerations from mp_pipeline:
    - Memory instructions require stalls & forwarding.
    - Load-Use Hazards: when forwarding from the Data Memory stage, data has to travel from SRAM array of Data Cache through forwarding logic, through ALU until it hits the next register. **This is often a critical path**
        - Solution: insert a bubble when a load-use is detected, use the WB forwarding path instead.
        - Detect load-use early, in the decode stage.

2.2 Branch Hazards

- Naive solution: stall until branch resolution before issuing instructions

[1]not always necessary, depends on implementation

- By shifting branch resolution earlier in the pipeline, you can reduce branch-related penalities.
- Speculative Execution: What if we speculatively execute instructions based on predicted branch direction, and flush if we happen to make an incorrect guess?
- 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)**

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.

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
  - subroutine nesting level $\leq$ RAS size ...generally (the exception when we have deep recursive subroutines)

2.2.1 Branch Predictor (BP)
- 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.
  - ex: mp_pipeline, static NT leads to best-case IPC on CoreMark of ~ 0.7

- All modern processors have Dynamic BPs instead.

2.2.1.1 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 ⟶ Weakly NT ⟶ Weakly Taken ⟶ 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%

2.2.1.2 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.

2.2.1.3 Two-Level BP

- Keep both a Branch History Table (BHT) and a Prediction History Table (PHT)
- PC can directly index into BHT or we can have some intermediate hashing.
- BHT keeps track of previous branch results, PHT is indexed by BHT, contains predictions of branch outcomes.

2.2.1.4 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.

2.2.1.5 Challenges to Branch Prediction

- Aliasing: can be destructive, or constructive (such as with multithreading)
- Training Time: Some branch prediction schemes require a long warm-up time.
- Context Switches
- Timing: In order to meet timing, the branch predictor itself needs to be thoroughly pipelined.

Instruction Cache Considerations
- 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

3 Caches

- Important aspect of memory hierarchy; hides long DRAM latencies due to program memory generally exhibiting **spatial and temporal locality**
  - Spatial Locality: If I access address A, I'm likely to access address its neighbors ($A + 4$, $A + 8$, etc.)
  - Temporal Locality: If I access an address repeatedly, I'm likely going to access it again.
- 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
  - L2/L3 caches can be 16, 32+ way set associative because we want to minimize **conflict misses**
  - Serial vs. Parallel Accesses of Cache Levels: We can speculatively access L2/L3 when accessing L1, thus reducing latency on misses. However, this adds unnecessary accesses to next levels on hits.
  - Over a long time, different caches exploit different memory access patterns
- In general, instruction cache (icache) access is simpler than data cache (dcache) because of memory disambiguation logic necessary with data cache.
- How do we evaluate latency of these memory systems?
  - **AMAT**, or Average Memory Access Time

$$\text{AMAT} = (\text{Hit }\%)(\text{Hit Latency}) + (\text{Miss }\%)(\text{Miss Latency}) \tag{2}$$

As we can see with the formula, assuming our lower-level caches (L1, L2) don't have absymal hit rates, higher-level caches suffer from diminishing returns. This is why we don't really see L5, L6 caches.

Four considerations when designing memory systems
- **Placement** - where do we keep the data within the structure?
- **Indexing** - how do we read data out of the structure?
- **Replacement** - cache size $\ll$ main memory size, so how do we replace old data?
- **Writes** - writes actually change architectural state

3.1 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)
  - TLBs have amazing hit rates (99-99.9% hit rates, usually)
- 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).
  - **Homonym problem** (one virtual address maps to different physical addresses)
  - **Synonym problem** (multiple virtual addresses map to the same physical address)
  - VIVT suffers from **both** problems.
- To avoid aliasing/homonym problem, we physically tag addresses (VIPT/PIPT)
  - while PIPT avoids homonyms, its also suffers from slow memory lookups as it needs to go through the TLB each time.

3.2 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 $N = 1$
  - 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.
  - remaining upper bits are reserved for tag
- 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.

- Modern processors have separate instruction and data caches.
    - Data size is often much larger than instructions, so a unified L1 data & instruction cache would be dominated by data.
    - Instructions often have more locality than data. e.g. consider large graph data structures.

- Data caches should be non-blocking, instruction caches should be blocking and single-ported.

Miss Classification
- **Compulsory Miss**: "cold start miss" - misses due to data not being cached yet.
- **Capacity Miss**: not enough space in cache to hold data. e.g. L1 cache is 32KB, you cannot physically fit any data structure larger than 32KB.
- **Conflict Miss**: data in cache misses because it was replaced. e.g. L1 cache is 32KB, you want two arrays of size 1KB to be both be cached, but they map to the same set, and so one replaces the other within the cache.
    - **Thrashing** - When frequently used data is consistently evicted $\rightarrow$ re-loaded into cache.

Direct Mapped Cache
- A memory value has a single corresponding location in the cache
- Rather simple to implement, but suffers from conflict misses

Fully-Associative Cache
- Memory value can be placed anywhere in the cache
- 0 indexing bits
- Power, Performance issues–we need comparators for each way in the fully-associative cache.

Set-Associative Cache
- A memory value can be placed in any location within its set.
- A good compromise between DMC and FAC, most cache systems nowadays are set-associative.

3.2.1 Writes
- Write-back: add dirty bit, if a line is evicted and marked as "dirty," write to memory.
- Write-through: on every write operation, write to main memory.

Imagine a system with L1/L2 cache hierarchy. Write operation misses in L1, hits in L2:
- Write Allocate Cache: on writes, load data into L1 cache and then modify it.
    - pairs well with write-back
- No Write Allocate: on writes, load data directly into processor, don't allocate anything in L1.

3.2.2 Replacement Policies

- For direct-mapped cache, its fairly trivial to determine what data to replace. Becomes less trivial with set-associative/fully-associative caches.
- LRU (least-recently used) tends to be the preferred replacement policy because it assumes that the recent past can predict near-future outcomes, pairing well with the idea of temporal locality.
  - not **always** optimal–e.g. streaming data with less re-use.

LRU Implementation:

- for each cache line, maintain a queue containing N entries, where N is the number of ways/ associativity. Each entry has size $\log N$
- on a memory access that hits in the set, push the selected way to the front of the queue. The back of the queue will be the LRU.
- this is viable in a 2-way set-associative cache (especially because you can do it with a single LRU bit)
- scales poorly for anything that isn't 2-way.

pLRU

- used in $\geq$ 2-way set-associative caches
- Use a binary tree where each node is a single bit determining which subtree is LRU.
- called "pLRU" because its not strictly LRU, but approximates it well.
- guarantees we won't evict MRU (most-recently used)
- For a N-way set associative cache, pLRU requires $N - 1$ bits. This scales a lot better than $N \log N$.

3.2.3 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

3.2.3.1 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

3.2.3.2 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

3.2.4 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

4 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.

4.1 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

5 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.

5.1 Register Renaming

- 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.

5.1.1 Register Renaming via the Re-order Buffer

- 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

5.1.2 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.

5.1.3 Merged Register File

- 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.

5.1.4 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.

6 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.