CMSC411 Epic Cheat Sheet

Draft Summer 2014

Table of Contents (Locations in 5thEd)

ISAs and MIPS, Appendix B

Most of this information and related info is found in Appendix B, also Worksheet 2

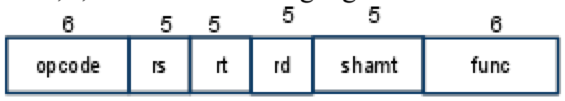
Classifications B.2

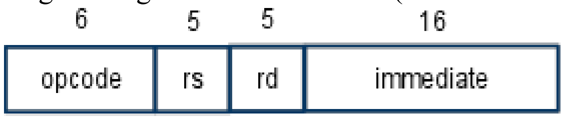
- MIPS is a **general purpose register** (GPR) architecture, has only explicit operands which are either registers or memory locations
  - Advantages: registers are faster, and memory traffic is reduced (keeping operands in registers instead of in memory)
- Two classes of register computers: register-memory like IA32 (can access memory as part of any instruction), or **load-store like MIPS** (can only access memory in load or store instructions.)
- Classified as (m,n): m is maximum # memory addresses, n is required # operands in an ALU instruction
- **MIPS is (0,3)** known as "**pure RISC**" architecture. Emphasis on simple instruction set and pipelining efficiency
- An ISA is said to be **orthogonal** if it lacks redundancy (i.e. there is only a single instruction that can be used to accomplish a given task); guarantees that one instruction has no side-effect affecting any other instruction
- MIPS supports 3 addressing modes: Register, Immediate, and Displacement
- MIPS64 has 32 64-bit General Purpose or Integer registers (R0--R31) and 32 64-bit Floating Point registers (F0--F31). R0 is set to zero.
- Assembly language (human readable) format is:

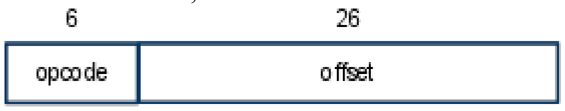
  ALU Instructions: OPerator destination, source, source

  Non-Jump Instructions: OPerator destination, source

Encodings B.3

- **Endianness** - "big endian" and "little endian". The LSB (least significant byte) in the HIGHEST memory address, and LSB in the LOWEST memory address respectively. Do not describe using "left and right" as it doesn't tell you which is the higher or lower address.
- **Byte Alignment** - accesses to memory objects larger than 1 byte must be *aligned*: given object with size *s* at memory address *A*, A mod s = 0.
  - Misalignment might require multiple memory accesses which may waste some space but runs faster.
- Effective address is not known until runtime (dynamic) and is computed at runtime by adding an offset (an immediate operand) to the contents of a register-- absolute locations in memory are the exception, not the norm.
- 3 instruction types: R, I, J in Machine Language
  - **R-type**: 

    - Register-register ALU functions (rd is destination)
  - **I-type**: 

    - loads and stores, conditional branches
  - **J-type**: 

    - jumps, exceptions/trapping. offset relative to program counter

MIPS B.9

- Working with MIPS64 - has 32 64-bit "general purpose" registers (integers) R_i and 32 floating point registers F_i.
  - **R0 is always 0**. This is **not true of F0**, F0 can hold any floating point value.
- Pages B36-B38 have tables with **examples of MIPS instructions** and their effects
- Immediate instructions end in the suffix 'I' and unsigned end in 'U'.
- Notice that with stores, we write to the second 'argument', with loads we write to the first
- There are special instructions to deal with moving between GPR and FPRs - MFC1, MTC1, DMFC1, DMTC1 move GPR data into FPRs. From there the data must be converted (CVT.x.y where converts from type x to type y; types L, W, D, or S). You will always have the MIPS instruction with the exam.
-

Quantitative CPU Analysis Ch. 1.8-1.9

The following CPU characteristics work interdependently in overall performance

- Clock Cycle Time (**CC_T**) - depends on **hardware** technology & organization
- Clock Rate (**CR) –** frequency of clock measured in hertz
- Clocks per Instruction (**CPI**) - **hardware & organization *AND* (ISA**)
- Instruction Count (**IC**) - instruction set architecture (ISA) & compiler technology
- Clock cycle time and clock rate are always inverses:
  - $Time = \frac{1}{Frequency}$
  - $CC_T = \frac{1}{CR}$
- **`Note:`** `1/execution time = rate or performance; 1/execution time = throughput`
- *Total* CC (Clock Cycles) and CPI for *multiple* programs to run:
  - $CC_{Num} = \sum_i (IC_i \cdot CPI_i)$
  - $CPI = \frac{CC_{Num}}{IC_{Total}}$
- **Note**: Pay attention to the summation and that the IC and CPI are for a *particular* program *i*
- CPU Execution time = CC * IC * CPI = (CPI*IC)/CR (because CC and CR are inverses)
- **Quantitative measures** of overall CPU performance with regard to speed
  - MIPS (millions of instructions per second)

$$MIPS = \frac{IC}{CPU_T * 10^6} = \frac{CR}{CPI * 10^6}$$

  - MFLOPS (millions of floating point operations per second)

$$MFLOPS = \frac{Num\ of\ floating\ point\ operations}{CPU_T * 10^6}$$

- Note: FLOPS are used because division cannot be pipelined

Amdahl's Law

$$Speedup = \frac{1}{(1-FE)+\frac{FE}{SE}}$$

- Where FE = Fraction Enhanced, FU = Fraction Unenhanced (= 1-FE) and SE = Speedup of the enhanced portion
- Speedup should be how much faster something is when a change is made in the system. Also it has no units.
- **Speedup** = Old CPU execution time / New CPU execution time;
- **Speedup** = New CPU performance / Old CPU performance

- Note that the speedup is always less than 100 percent. This is because **there will always be a fraction that cannot be enhanced**
  - **Example:** consider a task that takes one minute to perform but that must be repeated 60 times. For a single person, this will take 60 minutes. The task can be divvied up to 60 people with a goal of completion in one minute, but time is required to divvy up the work

Derivation

$$x \coloneqq \text{unenhanceable}, y \coloneqq \text{enhanceable} \therefore x+y=\text{execution time}_{old}$$

$$\frac{y}{SE}+x=\text{execution time}_{new}$$

$$\frac{\text{execution time}_{old}}{\text{execution time}_{new}}=\frac{x+y}{x+\frac{y}{SE}}=\left(\frac{x+y}{x+\frac{y}{SE}}\right)\cdot\left(\frac{\frac{1}{x+y}}{\frac{1}{x+y}}\right)=\frac{\frac{x+y}{x+y}}{\left(x+\frac{y}{SE}\right)\frac{1}{x+y}}=\frac{1}{\frac{x}{x+y}+\frac{y}{SE(x+y)}}=$$

$$\text{Since } \frac{x}{x+y}=\text{FU=1-FE and } \frac{y}{x+y}=\text{FE}$$

$$\frac{\text{execution time}_{old}}{\text{execution time}_{new}}=\frac{1}{FU+\frac{1}{SE}FE}=\frac{1}{(1-FE)+\frac{FE}{SE}}$$

CPU Architecture Appendix A, Chapter 2

Distinguishing hardware and software

It is critical to distinguish work that is done by the CPU at run-time, and work that is done before-hand by the compiler. Ultimately, the processor only executes the output from the compiler. Efficiency optimizations can be made by the compiler before code ever executes however there is a natural limit to this since many basic things like address in memory are not known until run-time. Most of what is discussed herein will be properties of the CPU architecture, but some properties also inherent to code. Properties affecting performance such as instruction count are purely dependent on the ISA; but cost per instruction is significantly affected by how the CPU is organized and therefore how long it takes to execute instructions

Pipelining A.1

- Consider doing laundry: gather clothes, wash, dry, fold. Doing many loads one step at a time will take forever because only one function is being done at a time.
- Now consider that while your first load is in the washer, you gather your second load of clothes. So when the washer is completed, you can have clothes in the dryer, and have clothes in the washer; etc.
- The idea is to keep each functional area (e.g. washing, drying) full in each interval of work. Call this pipelining.
- Like with laundry, pipelining increases CPU throughput, but it doesn't reduce the amount of time to perform an individual instruction.
- Processor must execute in such a way as to be consistent with executing compiler code sequentially; i.e. **in compiler order**.
- **Speedup** = CPI unpipelined / (1 + pipeline stall instructions per cycle); CPI unpipelined / pipeline stages

Dependences and hazards A.2

There are three kinds of hazards. Structural hazards result from resource contention. Control hazards result from the presence of branches--condition or unconditional (jumps) in the code. Data Hazards result from executing instructions in a way that is legal for the hardware, but violates the behavior expected of the same code executing on a sequential machine.
Dependences in the code can lead to data hazards.

A dependence occurs when two instructions use the same resource, typically a register.

A data dependence is present when one instruction requires the result of an earlier instruction to achieve correct execution.

A name dependence is present when two instructions use the same register, but are independent computationally.

Dependences

Name (2.1)

Name dependences occur because there are limited registers available to a CPU and result from the way the compiler creates instructions. Name dependences are so name because they can be resolved by renaming. There are two kinds of name dependences.

- **Anti-dependences** are potential **WAR** (Write After Read) hazards. Consider this code:

  S.D F4, 0(R1)
  DADD R1, R1, #-8

  - If the DADD writes to R1 before S.D has the chance to read it, then S.D might have the wrong value.
  - This can be resolved by having the DADD write to a different register which is why it's a name dependence
- **Output dependences** are potential **WAW** (Write After Write). Consider this code:

  L.D R1, R4, R3
  L.D R1, R5, R6
  DADD R8, R1, R2

  - If the first load executes before the second load, the add instruction may be reading the wrong value of R1.
  - This can be resolved by having the second load write to a different register
  - WAW hazards occur incredibly rarely because the compiler can make attempts to avoid this from happening.

Data (A.2 page A-17+)

Data dependences are potential **RAW** (Read After Write) hazards because there is information being passed between two instructions with one instruction inherently relying on another. Only data dependences have to do with needing to pass information between instructions. Take for example this code which loads data from memory and then adds it:

L.D R1, 4(R4)
ADD R3 R1 R4

The add instruction inherently relies on the data loaded from memory. These instructions must execute in order or else the ADD will be using the wrong values, and this cannot be fixed by changing names.

RAWs can be addressed either by waiting until the depending instruction is complete (**stall**), or by **forwarding**: above, as soon as the load gets the value it needs, the add can read that value directly instead of waiting for the load to write to R1.

Structural pg A-13

The CPU has resources which at any point in time is in use. A hazard happens when multiple instructions contend for the same resource. In the laundry example, there is only one washer. This can be overcome by having multiple washers, but with obvious tradeoffs in complexity of maintenance and expense. The circuits for data transfer are also structural hazards; imagine two Meeshs trying to fit through the same door ;)

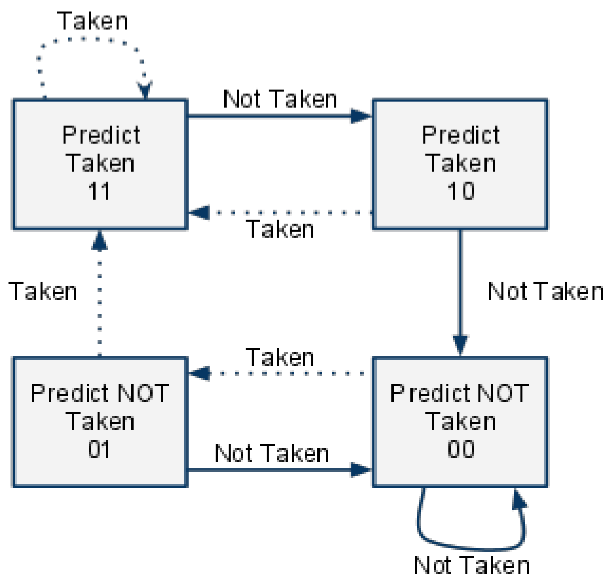
One structural hazard solution for instructions is to use a split cache - a different memory for instructions and data, so that retrieving an instruction does not contend with retrieving data (**Harvard Architecture**). Also the **number of read/write ports** for memories can be increased (similar to adding new functional units). Finally a **split clock cycle** avoids the hazard of a fetch stage and writeback stage at the same time - have reads done in the first half of the cycle, and writes in the second.

Control (A.2 page A-21+, 2.3)

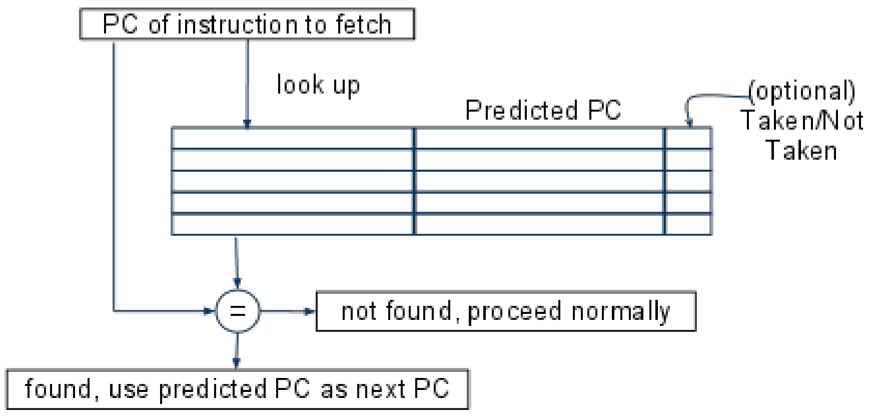
Occurs when an instruction depends on a branch instruction; it is unclear whether the next instruction should be executed or not and whether the PC must change until the branch is resolved.

Mitigations (2.3):

- Stalling
- Compiler can do **static scheduling** at **compile time** to take advantage of **branch delay slots** - in the classic pipeline, the instructions following a branch can be chosen in a way to either be independent of the branch or are likely to follow the branch target.
- Predictions (**dynamic** at **run time,** however not necesarily on dynamically scheduled processors):
    - The idea is we can make a decision on a branch correctly before its outcome is known with higher probabilities if we consider the result of previous branches. We can check these mechanisms at the same time as instruction fetch, even before we know it's a branch, so the prediction is ready.
    - **Branch Prediction Buffer** (**BPB**) - create a table that is indexed by some lower order bits in the memory address of an instruction. The entry in that table will hold a bit that predicts whether or not to take the branch. If the prediction is wrong, the bit will be flipped. Most branches will be wrong twice, so we can fix this using 2 bits (called 2-bit saturating predictor), according to this automaton. n-bit predictors are only marginally better than 2-bit and do not justify the extra cost/complexity



- **Branch target buffer (2.9)** - the BPB does not account for the fact that different branch instructions can map to the same prediction bits. This BTB stores the PC of an instruction to fetch and its predicted PC, with optional prediction bits on the end. If an entry for the instruction is not found and turns out to be a taken branch then it is added to the table. If it is a branch that ends up not being taken, remove from table. Has a penalty CPI = (P * hit rate * misprediction rate) + (P * miss rate * misprediction rate)where P = penalty in number of cycles which must be lost if branch predict fails


PC of instruction to fetch
look up
Predicted PC
(optional) Taken/Not Taken
=
not found, proceed normally
found, use predicted PC as next PC

- **Branch folding buffer** - an extension of the BTB, instead of keeping the predicted PC of the branch for unconditional branches (jumps), store the actual instruction so it can be executed directly. The PC lookup is used as a return address

CPU Organization

5 Stage Pipe-like Pipe, Appendix A.1-A.5

The simplest architecture, executes instructions sequentially through 5 processing stages: Instruction Fetch, Instruction Decode, Execute, Memory, Write-back.

- WAW hazards cannot occur because instructions always complete execution in sequential order.
- WAR hazards cannot occur because register reads always occur in the earlier ID stage so a later instruction cannot overwrite the source registers
- RAW hazards occur and are mitigated by stalling and forwarding as explained earlier
- Control hazards occur and can be mitigated statically by the compiler filling branch delay slots, or dynamically at run-time by making predictions about the branch direction.

The ability to do forwarding, as well as controlling the flow of instructions through the pipeline, is accomplished through pipeline registers, which are a set of registers that exist between each of the pipeline stages. Work is performed in the stages and then the results are written to the pipeline registers. The pipeline registers can access each other and this is how forwarding is implemented: the ID stage can check for dependent instructions in EX, MEM, and WB and if it finds them, can directly read the results. Glorious details in A.3

Floating Point (A.5)

The 5-stage (integer) pipe-like-pipe has an extension to handle floating point instructions. FP operations take more than 1 cycle to implement, and different FP instructions take different amounts of time to complete corresponding to **latency** and **initiation interval.** For example FP add takes 4 cycles to execute and FP divide takes 24.

Latency - the number of intervening cycles between an instruction that produces a result and an instruction that uses the result. It is essentially equal to 1 cycle less than the depth of the execution pipeline, which is the number of stages from the EX stage to the stage that produces the result.

Initiation Interval - number of cycles that must elapse between issuing two operations of a given type. With floating point divide, the FP divide FU must clear before a new instruction can be initiated and so IN THIS CASE the initiation interval is the number of pipeline stages + 1 (because initiation interval is 1-based whereas latency is 0-based).

As a result of the difference in latency and initiation interval among the different FP functional units, if we maintain the same '5-stage' architecture, despite instructions enter the pipeline and specifically the execution stage in pipeline order, they complete the execution stage out-of-order. This introduces structural hazards (e.g. with the unpipelined FP Divide, multiple instructions completing and wanting to write at once), WAW hazards, imprecise exception handling, and more frequent RAW hazards.

Structural hazard checks can be checked at ID to see when the instruction will need the write port and stall. Alternatively can stall in the MEM stage. WAW hazards are addressed by detecting the hazard and stamping out the result of the earlier write. Another is to stall until the preceding instruction enters the MEM stage

Also introduces complications in maintaining precise exception handling. However out of order completion of execution is not inherently a problem because dependences are enforced. Mitigated by a history buffer (previous values), and relaxing how precise the exception handling must be.

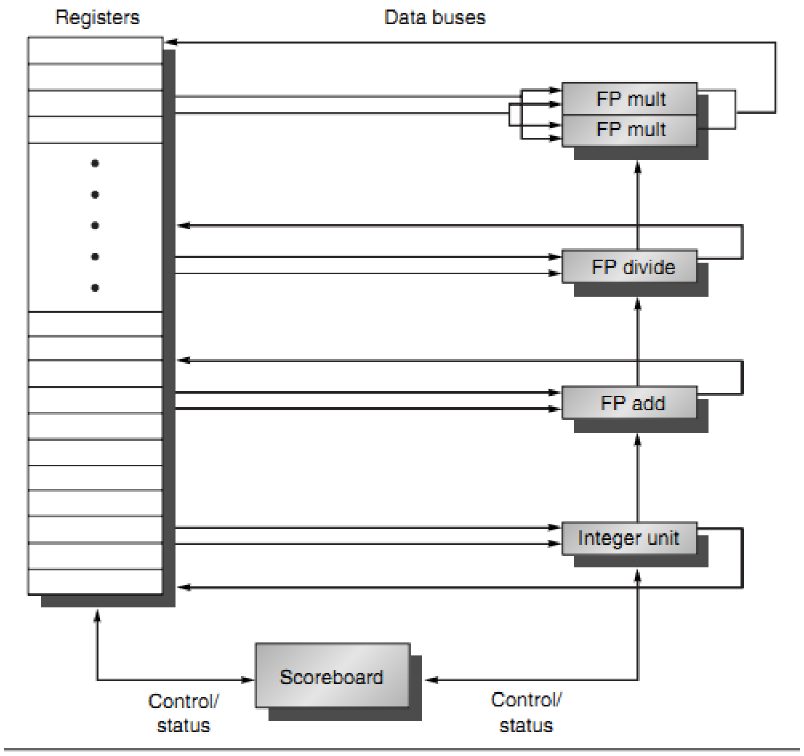
Scoreboarding A.3

The problem with the sequential 5-stage pipeline is that when an instruction is stalled because of a dependence, the rest of the pipeline is stalled as well, e.g. no new instructions can be fetched etc. Therefore stalls become a significant bottleneck and the idea is to come up with a method that allows instruction execution to continue, without violating any dependences and allowing code to execute correctly.

The solution is to use **dynamic scheduling** at run-time and by the processor and CPU reorganization. The goal is to allow an instruction to execute as soon as its operands are available and without stalling subsequent instructions. Therefore since execution begins as soon as possible, we have out of order execution implying out of order completion. Scoreboarding allows instructions to execute out of order and maintain program dependences.

The scoreboard is a control unit in the CPU that acts as the brain, and decides how and when to proceed instructions by checking for potential hazards. Every instruction goes through the scoreboard.

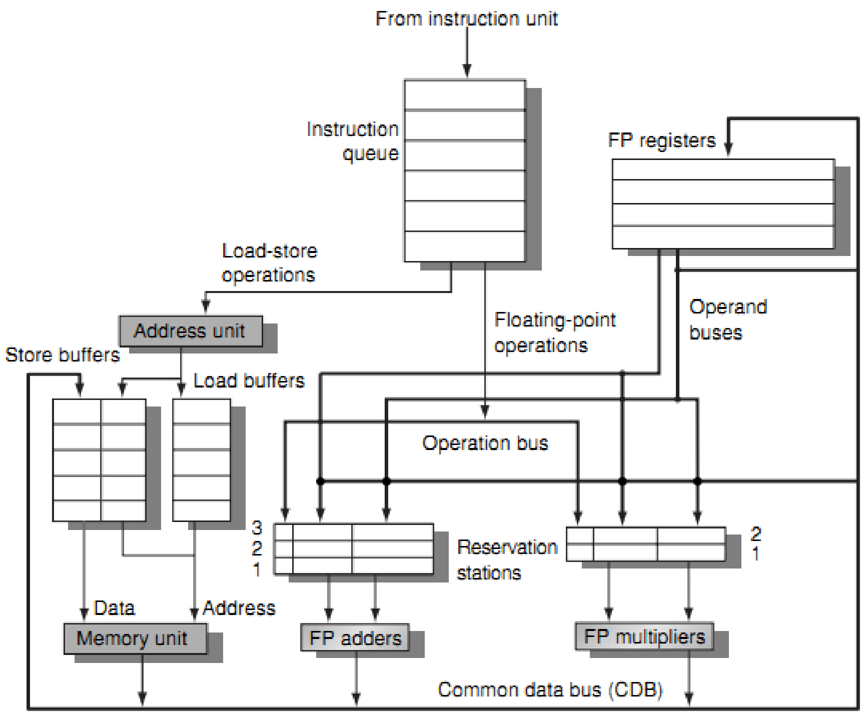
- Introduction of WAR and WAW hazards since execution can complete out of order, an instruction may attempt to write to a register before a preceding instruction has the chance to read/write it



- Instruction execution follows the following 4 stages (memory stage omitted). Note that waiting for operands happens inside the functional unit
    - Issue - if there is a free functional unit and there are no WAW hazards, then issue to a functional unit. There is a buffer between fetch and issue which can fill.
    - Read operands - in the functional unit, wait until there are no RAW hazards and then begin execution
    - Execution
    - Write result - but need to check for WAR hazards and stall if necessary.
- There are limited number of buses to/from the register files which introduces a structural hazard

Tomasulo's Algorithm (2.4-2.5)

Scoreboarding still has not resolved named dependences which can be a major bottleneck, as well, functional units are occupied even though an instruction may be waiting for operands. Tomasulo eliminates both of these problems through the use of reservation stations and register renaming.

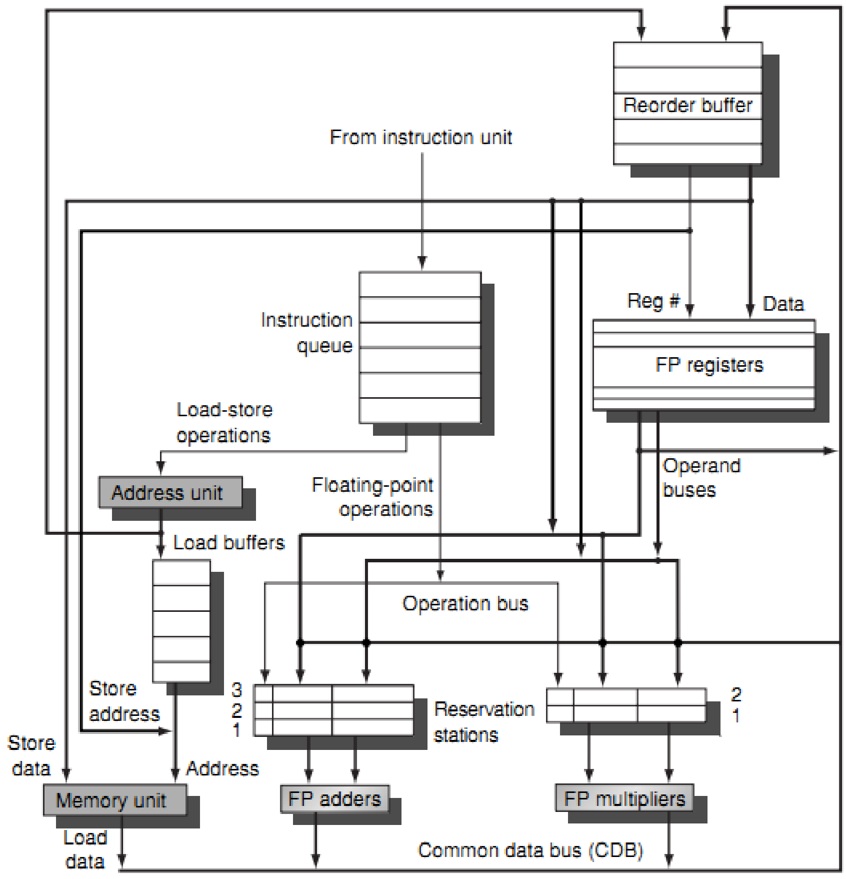


- store/load buffers and the connection between them - this is for **dynamic memory disambiguation** which prevents a store and load from using the same memory address
- **Reservation stations** where register renaming happens - the destination registers are explicitly renamed to the name of the reservation station, and there is 1-1 mapping between an instruction and its reservation station, so there are no named dependences
- **Common Data Bus (CDB)** which broadcasts the results of execution back to reservation stations who are waiting for them, and to the register files where the results are written.
    - Major structural hazard
- RAW hazards resolved by instructions waiting in the reservation stations until the source

operands become available through the CDB
- Precise exception behavior enforced because instructions in reservation stations are also not allowed to begin execution until all preceding branch instructions are complete.
- The picture above is misleading because all of this control logic is handled by a scoreboard which is not shown, but would be "below" inside in the picture.
- Stalls can still occur if there are not enough available reservation stations in the issue stage - consider that they are all occupied and all waiting for results.

Speculation (2.6)

Up till now dealing with control dependences, we would make predictions about which instructions we want to execute, but we never actually would execute them. Using speculation, we can still use the static (compiler) and dynamic (prediction) methods to decide which instructions should follow a branch, but now we actually execute the instructions as well. Clearly if our prediction ends up being incorrect then we must flush the whole rest of the pipeline and not write incorrect results to registers/memory. An addition to Tomasulo's organization can allow us to do this and it is the **reorder buffer (ROB).** A distinction occurs with instructions who have 'completed', vs. instructions that have 'committed' and actually write their results into memory/register file.



For an instruction to issue, in addition to needing a reservation station available, an entry in the ROB must also be available. Stores are also handled in the commit stage. An instruction commits from the reorder buffer if it does not depend on a control instruction, or if the depending control is validated; otherwise the entire pipeline is flushed. The ROB also allows for precise exception handling because there is a record of the order of instruction execution and their values. WAW and WAR hazards don't exist because commits happen in order.

Superscalar and Multiple Issue 2.7-2.8

The architectures up to this point correspond to an **ideal CPI of 1** and can issue only one instruction per CPU cycle. This can be extended to allow more than one instruction to issue in a single clock cycle and corresponds to an ideal CPI of **less than 1.** Frequently this is discussed with Tomasulo's algorithm with speculation: **SST** (superscalar speculative Tomasulo).
Another superscalar architecture is **VLIW** - Very Long Instruction Word, where a fixed number of instructions are formatted as one very large instruction packet and then decoded by the processor. This architecture is statically scheduled and uses multiple independent functional units.

Limitations

- In S.S. architectures particularly Tomasulo, FP operations cannot be issued in parallel due to structural hazards
- There is an inherent limit to **ILP** (instruction level parallelism) in the code
- Complexity in building the hardware and managing the control

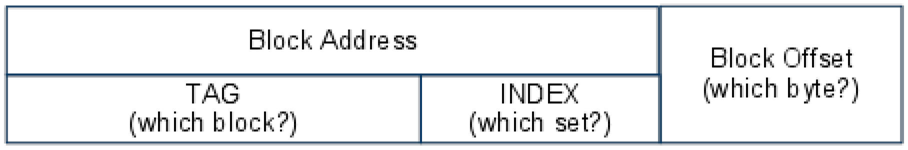
Memory, Cache, Virtual Memory (Appendix C, Chapter 5)

Cache Types

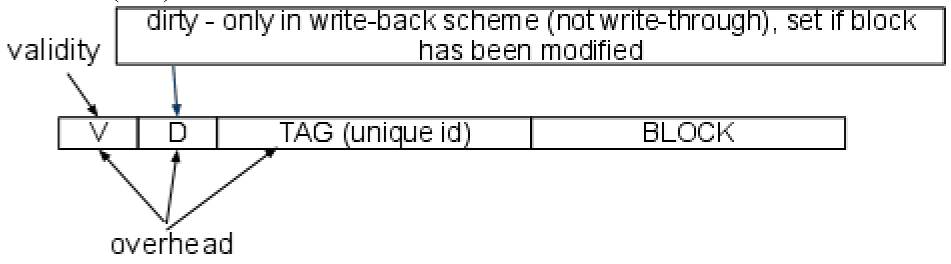
- Direct-Mapped (One way set associative) - one block per set, so each block can only ever be at one place
- Fully-Associative - one giant set, so any block can be anywhere
- Set-Associative (N-way) - N blocks per set, any of these blocks can go anywhere in the set
  $2^{num\ index\ bits} = \frac{cache\ size}{block\ size * n - way} = num\ of\ sets\ in\ cache$
    - where block size is the set size
    - and n-way is the number of blocks per set
- $MAT_{L1} = HT_{L1} + (MR_{L1} * MP_{L1})$
    - where MAT = Memory Access Time
    - HT = Hit Time
    - MR = Miss Rate
    - MP = Miss Penalty
        - re-curse into next lower-level, $= HT_{L2} + (MR_{L2} * MP_{L2})$

Cache Addressing

In instruction or data (address)



In cache (line)


dirty - only in write-back scheme (not write-through), set if block has been modified
validity
V
D
TAG (unique id)
BLOCK
overhead

- **Block Replacement**
    - Random
    - Least Recently Used (LRU)
    - Not Most Recently Used (NMRU)
- **Stall Sources**
    - Instruction fetches
    - Data accesses through load and store instructions

      $$IF\ \frac{stall\ cycles}{instruction} = \left(\frac{memory\ access}{instruction}\right) * miss\ rate * miss\ penalty(Data)$$

- **Write Strategy**
    - **Write-Back** - Blocks to be written are only written to at this level of cache; the write does not propagate to lower levels until the block leaves cache due to a miss; blocks that have been written have their dirty bit set to 1
        - **Write-Allocate** - Any block which is to be written much be brought to cache first - used with Write-Back
    - **Write-Through** - Blocks to be written are written in both cache (if it is already there) and to the next level in the memory hierarchy; does not require the use of dirty bits
        - **Write-No-Allocate** - Blocks to be written are written in main memory directly and do not need to loaded into cache - used with Write-Through
    - Notes
        - Write-Through is slower but easy to implement
        - Write-Back lowers memory bandwidth used, but is bad for consistency because I/O and other processes see old information in the memory hierarchy

- **Cache Misses**
    - **Compulsory** - occurs after a context switch, or when the machine is first turned on; the cache contains no valid blocks at all
    - **Capacity** (*Fully-Associative ONLY*) - occurs when there are no empty slots in the fully-associative cache
    - **Conflict** (Set-Associative & Direct Mapped ONLY) - occurs when the set to which a memory block is mapped has no empty lines

Average Memory Access Time Equations $MAT = HT + MR * MP$

- **Memory Stall Cycles** =
    - = IC * misses per instruction * MP
    - = IC * memory references per instruction * MR * MP
      $= (number\ of\ reads * readMR * readMP) + (number\ of\ writes * writeMR * writeMP)$
    - = IF stall cycles + data stall cycles
- **CPU Time** =
    - (# of CPU cycles + # of Memory Stall Cycles) * CCT
    - $= IC * (CPI_{exe} + \frac{memory\ stalls}{instruction}) * CC_T$
        - where CCT - clock cycle time
        - $\frac{memory\ stalls}{instruction} = MR_{instruction} * MP_{instruction} + (frequency(load/store)) * M$
          $CPI_{exe} = freq(load) * CPI(loads) + freq(stores) * CPI(stores)$
          $+ \mathrm{freq(others)} * \mathrm{CPI(others)}$

          ignoring stalls

Cache Optimizations

- **Reduce Miss Penalty**
    - Multi-level Caches
    - Giving Priority to Reads Over Writes
    - Critical Word First and Early Restart - these two strategies do not wait for an entire block to be loaded before sending the CPU its requested word and un-stalling the processor (works best with larger cache blocks)
        - Critical Word First - request the missed word first from memory and send it to the processor as soon as it arrives; let the processor execute while loading the rest of the block
        - Early Restart - fetch the word in normal order, but as soon as the missed word arrives, sent it to the processor and let execution continue while the block finishes loading
    - Merging Write Buffer - write-through and write-back both put updated blocks into a Write Buffer while they wait to be updated in main memory. Sometimes the same block will be modified twice before it gets flushed out of the Write Buffer and updated in main memory. (works best with write-through since it has a larger write buffer)
        - Merging the write buffer refers to combining all modifications of a block into one entry.
        - Buffer refers to combining all modifications of a block into one entry, instead of having multiple nearly identical copies of the same block
    - Victim Cache - holds blocks which have recently been discarded from cache, allowing them to be accessed quicker than if you had to reload from main memory. Reduces impact of conflict misses.
- **Reduce Miss Rate**
    - Way Prediction and Pseudoassociativity Caches
    - Compiler optimization
    - Larger Block Size (increases MP!) - fitting more data into each block means the CPU does not need to request new blocks as often (until it starts causing conflict/capacity misses)

    - Higher Set Associativity (increases HT, thus increasing MP of higher level cache!) - works by the 2:1 cache rule - the MR of a direct-mapped cache of size s is ~ the MR of a 2-way set associative cache of size s/2
    - Larger Cache (increases HT, cost, power consumption) - fitting more blocks into cache means they need to replaced less often
- **Reduce Hit Time**
    - Way Prediction
    - ***Avoiding Address Translation During Indexing - using the physical address of a block in memory as the address in cache, instead of turning it into a virtual address. Thus the CPU, which sees cache as a blob of physical memory doesn't need to translate addresses all the time
    - Small and Simple Caches - direct-mapping the L1 cache allows the tag-check and the data transmission to be overlapped. Also, smaller hardware can be faster, especially when the L2 cache can be fitted on the same chip as the processor and L1 cache
    - Trace Caches (only in Pentium 4 family) - contain a sequence of previously issued instructions in the actual order they were executed rather than as they were laid out in instruction memory. This lets branch prediction be folded into the cache, but requires it to be validated in order for the fetch to be considered valid
- **Reducing Cache Miss Penalty or Miss Rate vs. Parallelism**
    - Non-blocking Caches to Reduce Stalls on Cache Misses
    - Hardware Pre-fetching
    - Compiler-Controlled Pre-fetching

Virtual Memory

- Virtual Address (produced by CPU) => Physical Address (used to access main memory)
- **Classe**s of Virtual Memory Systems
    - **Page**: fixed-size blocks
    - **Segments**: variable-size blocks
- Techniques for **Fast Address Translation**
    - ○ Page Tables: one memory access to obtain the physical address and another access to get the data
    - TLB: keep address translations in a special cache
- **Page vs. Segment**

	Page	**Segment**
Words per Address	One	Two
Programmer Visible	Invisible	May be visible
Replace a Block	Trivial (same size)	Hard (different sizes)
Memory Use Inefficiency	Internal Fragmentation (unused prop. to page)	External Fragmentation (unused prop. to main mem)
Efficient Disk Traffic	Yes	Not Always

- Cache vs. Virtual Memory

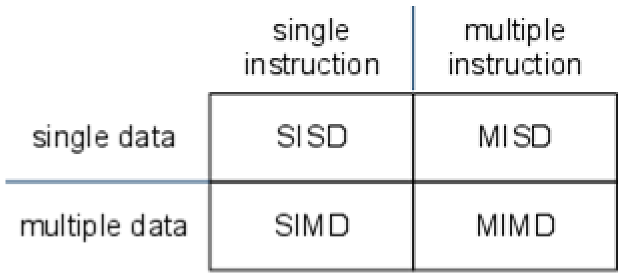
	Cache	Virtual Memory
Replacement By	Hardware	OS
Backing Media	Main Memory	Disk
Size	Variable	Determined Processor Address Size
Associativity	Variable	Fully Associative
Block Lookup	Tag/Index	Virtual Page Mapped in Page Table or TLB
Write Strategy	WT or WB	Always Write Back

○ (offset used by both to determine actual data location in the block(cache) or page (virtual memory))

- Rules of Thumb
    - **Amdahl/Case Rule**: A balanced computer system needs about 1MB of main memory capacity and 1 megabit per second of I/O bandwidth per MIPS of CPU performance
    - **90/10 Locality Rule**: A program executes about 90% of its instructions in 10% of its code
    - **85/60 Branch-Taken Rule**: About 85% of backward-going branches are taken while about 60% of forward-going branches are taken
    - ○ **2:1 Cache Rule**: The miss rate of a direct-mapped cache of size N is about the same as a two-way set-associative cache of size N/2
- **Precise Exception Handling**

Multi-processor

Flynn's Taxonomy



	single instruction	multiple instruction
single data	SISD	MISD
multiple data	SIMD	MIMD

- **SPMD** - single program, multiple data - is run on different data, but each processor can be at a different point in the program, unlike SIMD
- **MPMD** - multiple programs, multiple data - run on different processors using different data. Typically one program is the manager and farms out programs & data to the other processors
- **SISD** - individual CPU cores - no parallelism in execution or data streams, pipelining does not

count as parallelism
- **SIMD** - array processors, GPUs - runs the same instruction on multiple data streams in parallel
- **MISD** - space shuttle flight controller - runs multiple instructions in parallel on the same data, used for fault tolerance and not much else
- **MIMD** - super computers, distributed systems - runs different instructions on different data simultaneously