SE 3GA3 Summary

Chapter 1

Size of frame buffer (in bytes) = bits/pixel × 1byte/8bits × resolution

Benchmarking: determines the performance of a given system; take the best of 5 to 10 tests, since the computer runs other processes simultaneously, such as mouse moving, graphics, etc. that influence this and result in different times for every process; do not use the average time.

- CPU has control path and data path
- Cache is fast memory, limited because of cost
- Volatile memory forgets when powered down
- Vacuum tube is the old school transistor
- Even if your program isn't designed with parallelism in mind, there are some compilers that can allow it to run in parallel, anyways

- Elapsed time: total response time; determines system performance
- CPU Time: time spent processing a given job
- $\text{CPU Time} = \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$
- Clock Cycles = Instruction Count × Cycles per Instruction (CPI)
- $\therefore \text{CPU Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$
- $\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}$
- Number of instructions doesn't change the CPI
- Halving the instructions halves the Execution time
- Performance varies:
  - Algorithm
  - Language
  - Machine
  - Concurrent tasks
- $P = C \times V^2 \times f$
  - Power × 30
  - Voltage: 5V → 1V
  - Frequency: ×1000

Instructions that wait for memory access have high cache miss rates and reduce performance.

Benchmarking each computer can inform you which machine is better for a given computation.

Some servers still require power when they are running 0 computations, so it is useful to investigate turning them off
Size of frame buffer: bits/pixel × resolution
Performance = Execution time^{-1}
It takes $\frac{\text{size of file}}{\text{internet speed}}$ to send a file

Law of diminishing returns:

Amdahl's Law: performance enhancement is possible with a given improvement and is limited by the amount that the improved feature is used

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

$$T_{\text{previous}} = T_{\text{affected}} + T_{\text{unaffected}}$$

MIPS: Millions of Instructions per second

$$\begin{aligned}\text{MIPS} &= \frac{\text{Instruction count}}{\text{Execution time} \times 10^6} \\ &= \frac{\text{Instruction count}}{\frac{\text{Instruction count} \times \text{CPI}}{\text{Clock Rate}} \times 10^6} \\ &= \frac{\text{Clock rate}}{\text{CPI} \times 10^6}\end{aligned}$$

Leave spaces between every two hex digits because each group represents a byte (8 bits).

When writing hexadecimal numbers, write 0x before the number because that is the standard 0xAFC4

Unsigned integers are assumed to be positive

uint32 = $2^{32} - 1$

For signed 16 bit integers, the first of the 16 bits represents the sign.

Global CPI: weighted average CPI when you have multiple cores/programs running

Geometric mean: $\sqrt[n]{\prod_{i=0}^{n} \text{Execution Time Ratio}_{(i)}}$

Yield: percentage of good dies in a wafer

Chapter 2

- "Hello world!" has 4×4 bytes "`Hell,o wo,rld!,Ø` " since end of string must have a null operator
- indexing is in terms of 4-bit words (in 32-bit system), so it ends in the word, FFFC
- Register file has registers each with 32bits in them from 1-31…oh wait, actually also a 0, but it is filled with 32 nulls

CISC: Complex Instruction Set (e.g. x86)

in MIPS, all instructions are 32-bits long. No shorter, no longer
Stanford MIPS by MIPS Technology

C: f = (g+h) ←assignment
Assembly:
```
add t0, g, h # t0 = g + h
sub f, t0, t1 # f = t0 – t1
```

Each instruction is 3 terms long. Symantic:
operation Reg_{assign}$, Reg_{reference1}$, Reg_{reference2}$
Reg_{assign}$:= Reg_{reference1}$ [operation] Reg_{reference2}$

Assign:: = RegId '=' RegId Operator RegId

word: on MIPS, 32-bit data or 4-bytes

Assembler names:
`$t` variables are temporary variables
`$s` variables are saved variables

Preserved	Not preserved
Saved registers: `$s0–$s7`	Temporary registers: `$t0–$t9`
Stack pointer register: `$sp`	Argument registers: `$a0–$a3`
Return address register: `$ra`	Return value registers: `$v0–$v1`
Stack above the stack pointer	Stack below the stack pointer

Preserved variables *can* be altered, but are expected to have the same variable at the end, so it's better if you don't change them.

databus is usually 1-2 words wide. It is the communication bridge between the CPU to the memory.

each memory address is an 8-bit byte

```
lw $t0, 32($s3)      #load word
sw $t0, 48($s3)      #stores word
```

Assignment or **becomes** is represented by the following notation: ":="

```
beq $s1, $s2, L1     #branch if equal; if s1 == s2, goto L1
bne $s1, $s2, L1     #branch if not equal
```

```
bne $s0, $s1, Else       #if they aren't equal, code B
<<Code A here>>          #if they are equal, code A
j Exit                   #j is an instruction meaning jump to
Else: <<Code B here>>
Exit:
```

Memory accesses occur when doing a data transfer. The instructions are also stored in the memory, so each instruction requires an additional memory access, regardless whether or not it is for data.

By definition, operations coined **data transfer instructions** change both the cache *and* the memory (i.e. transfer data between memory and registers). Each data transfer instruction has 2 memory accesses: one to access the instruction and a second to actually access the data.

Arrays

```
lw $t0, $s0($s5)
```

s0 stores the memory location or **base address** of the start of the array. The number you put before your base address is your **offset**. So 0 would access the 1st element or element 0. Since offset is bitwise, each element (of byte size) is a multiple of 4. Thus, 1 would access the 2nd element.
If the array is {1,5,7,8,3}, the code would assign the temporary variable, t_0 to the 2nd element of the array, 5.

If you have an array and you want to find the value of the array at a point that is not a constant, you cannot simply refer to it as an offset. You need to:

```
sll index, index, 2          # i=4×i, since words increment by 4bits
add address, base, index     # address[base]+index = address[index]
lw $t0, 0(address)           # get the value from the array cell
```

`sll` shifts your byte by adding a 0 at the end. Thus, shifting twice moves your registers by a byte. This is used when accessing an array.

Isotonic:

Immediate commands are 16-bits

Think of memory as word array, where indices are addresses

eix crossdev

```
j     $31   #jump to register 31
```

MIPS has a delayed jump. It jumps after the line after it. Only worry about it when reading real code. However, if there's nothing you want to execute after your jump, you can either move your last line to after the jump, or use the `nop` command (no operation)

In MIPS, when doing operations on values that are not in registers or memory, you are dealing with **immediate values** or **constants**.

```
addi $s3,$s1,4 # $s3 = $s1 + 4
```

Zero is such a frequently used constant, that it is hard-wired as `$zero`.

Don't forget the two's complement stuff!

Sign extension: when numbers are not 32-bits long and need to be saved in 2's-complement, 32-bits for certain operations (excluding `or`, where the other bits are assumed to be 0, regardless of the sign), you sign-extend the number

1. Assume your left-most bit represents the sign
2. Insert the bits. The value of all added bits is the sign before you extended it.

folding: find the significant group of bits
unfolding: expanding from a byte to sign-extension form; the extra digits are meant to cancel each other out (page 90)

The **program counter** (PC) is a register that keeps track of the current instruction number.
PC relative addressing: jumps and if statements that refer to the next line by adding the number of lines (in words) to the next instruction

Address mode: whether you give a register number or an actual address

Chapter 3

Float to binary is number mod 2^{32}.

$$d = a + b \equiv d_{10} = \sum_{i=0}^{n} \left(a_i +_{10} b_i \right) \cdot 2^i$$

c_{-1}:=0
for i ∈ 0..n−1:
    C::= 1 iff at least two of a_i, b_i, c_{i-1} are 1
    d::= 1 iff odd many of a_i, b_i, c_{i-1} are 1
$d_n := c_n$

11× abc ← base 10
$11 \times abc = (1\times10^1 + 1\times10^0)\times(a\times10^2 + b\times10^1 + c\times10^0)$
$= a\times10^3 + b\times10^2 + c\times10^1 + a\times10^2 + b\times10^1 + c\times10^0$
= a(a+b)(b+c)c ← gather the terms with same 10power and each bracket is its own digit

You don't need to remember the carries for this method of addition (grouping)
0110 + 0111
$= (0\times2^3 + 1\times2^2 + 1\times2^1 + 0\times2^0) + (0\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0)$
$= (1+1)\times2^2 + (1+1)\times2^1 + 1\times2^0$
$= 1\times2^3 + 1\times2^2 + 1\times2^0$
= 1101

Sign and magnitude: $\underbrace{(-1)}_{\text{sign}}\left(1+\underbrace{\text{fraction}}_{0..1}\right)^{\overbrace{\text{exponent}-\text{bias}}^{\text{original exponent}}}$

Fraction is everything after the decimal point (put the first 1 before the decimal)

Underflow: number smaller than possible representable by float

Single precision: 32-bit rounding errors or overflow or underflow, so your bias is 127, so $2^{-1} = 2^{126-127}$ and your saved exponent is 126

Guard and Round: equal sig figs, but same decimal point, then make sure both have same number of digits after the decimal by inserting 0's to allow the result to include the potential to round up. after, round up to the number of sig figs of the number with the least sig figs

Denormal numbers: numbers with leading zeros in the fraction that allows for a certain degree of underflow even after the exponent is 0

Chapter 4

Bubble:

Forwarding (a.k.a. bypassing):

- Implemented in hardware

Hyperthreading: treating one core as multiple cores; this is useful because you have shared registers; it is also useful for using delays meant to avoid hazards for useful purposes

Very Long Instruction Word (VLIW): a group of instructions that can be issued on a single cycle; not used by GPGPUs [General Purpose Graphics Processing Unit] (useful for speeding up certain floating point calculations)

Dynamic Scheduling: Allow the CPU to execute instructions out of order to avoid stalls

Register renaming: moving values of registers between cores

SIMD instructions:

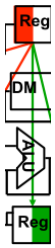
Pipelining

Pipelining:

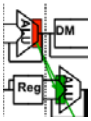
- think of the laundry example−every time you start the laundry machine is the start of a new pipeline
- more importantly, it's a way of using different parts of the processor simultaneously. Since each instruction has multiple parts, such as instruction fetch, register read, ALU

operation, Data access, register write, etc., you can have different instructions run on different parts at the same time

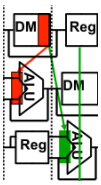
- Since some instructions have different times that they take for each step, pipelining may result in the next instruction having to wait until the current instruction finishes the current step. Thus, **state registers** are used to store the values from the end of each state in order to delay the next instruction, until the current instruction is complete
- This comes with problems, known as **pipeline hazards**
    - **Structural hazard**: If multiple instructions try to do the same step simultaneously (since each step in the pipeline takes different lengths of time to compute)
    - **Data hazards**: when you store something in a register during one instruction that you're supposed to call in the next instruction
        - Avoided by hyperthreading
        - You may read and write on the same cycle if you write on the negative edge of the clock cycle (first half) and read on the negative edge (second half), as long as there are instructions between the read and write instructions (which can be done by hyper-threading)



        - If the instructions are side-by-side, you need to do **stalling** or inserting bubbles. This stops the stacking process to let the register values to catch up
        - **Types**:
            - **RAW**: Read After Write
            - **WAW**: Write After Write
            - **WAR**: Write After Read
        - **Data forwarding**: Forwarding the value of a register to a different pipeline step:
            - e.g. if you know your laundry machine sucks, why dry it, fold it, and put it away before washing it again? Just restart the cycle (write happens before read, so might require an extra space)
            - **ALU/ALU**: for arithmetic operations, since the data access and register write steps do not change the value of the register, you could even put the value of one ALU operation straight into the next ALU operation without having to spend time saving it and whatnot



            - **EX/MEM**: ALU forwarding does not work as well if you have lots of lw/sw's, so you send the result after the DM stage to the ALU stage



- **Control hazards**: when jumping, it takes time to compute the destination, but to keep the pipeline moving, it computes the next step even if you were planning on skipping the next step
  - Avoided by nop
  - Avoided by hyperthreading
  - Avoided by **delayed branch**, which is where you put your second last step after the branch
  - Reduced by using **branch prediction**, which is where the processor guesses that it will not probably branch (since you don't know the address, until it's been calculated, so you might as well assume it doesn't branch) and will begin the instructions in the branch, until it has computed the inequality. Use probability to determine whether or not it is better to use beq or bne. The one with the higher probability should. However, if it guesses incorrectly this must be followed by making sure there will be no problems that will arise from this.
    - **Dynamic prediction** is where the processors look up address of instruction to see if branch was taken last time the instruction was executed
      - This is done by having a **branch history table** that saves the most recent value
      - **Branch target buffer**:
- Notation:
  - 1a: EX/MEM.RegisterRd = ID/EX.RegisterRs 1 line
  - 1b: EX/MEM.RegisterRd = ID/EX.RegisterRt 1 line load
  - 2a: MEM/WB.RegisterRd = ID/EX.RegisterRs 2 lines
  - 2b: MEM/WB.RegisterRd = ID/EX.RegisterRt 2 lines load
  - Rs: first read port
  - Rt: second read port

In pipeline diagrams, the right half of registers is highlighted when reading; left is highlighted when writing. Since there is only a single pipeline stage, the information from the previous instruction might write over the current value of the pipeline registers. To avoid this, the pipeline registers store the value of the register and the register number.

Cross-talk: when the logical AND of 2 control lines determine whether a certain operation occurs, you can get cross-talk, wh

Chapter 5

Can use frame pointer in MIPS
GNU

Each address has:

- Tag: identifier so you know what data you're dealing with; this is useful for identifying if you have found the correct data or not (cache miss/hit)
- Index: which block on the cache is the data
- Offset: where on the block is the data

Unroll: extend length of code (might make code faster)

Cache miss: calling cache not yet in cache
Cache hit: calling cache from caches

3 Types of misses: capacity misses, compulsory misses, and conflict misses
Capacity Miss: cache cannot contain all blocks
Compulsory Miss: first access to block is impossible to avoid that is relatively smaller for larger programs
Conflict miss: multiple memory locations mapped to the same cache location

Design change	Effect on miss rate	Negative performance effect
Increase cache size	Decrease capacity misses	May increase access time
Increase associativity	Decrease conflict misses	May increase access time
Increase block size	Decrease compulsory misses	Increases miss penalty. For very large block size, may increase miss rate due to pollution.

Increasing the cache size can also reduce conflict (or collision) misses.

Cache prefetch: the act of applying locality in the sense that you keep the stuff you need in the cache that you're going to use again

Miss penalty: amount of time it takes to bring stuff from memory to cache
Hit time: amount of time it takes to check if the memory is in the cache

Hit ratio: hits/accesses
Miss ratio: 1 – hit ratio

Data-cache miss:
Instruction-cache miss:

SPU: Synergistic Processing Unit

Since smaller amounts of cache is faster and larger is slower, they have introduced **multi-level cache**. If you miss the first level, then you go for the second level (i.e. hierarchy

Levels of cache: L-1 < L-2; each level is roughly 100× faster than the previous one; everything is copied−everything has to be in memory, but stuff can be copied into the higher levels

Penalty: Access time/miss rates
Extra Penalty: amount of time it takes to go from L-2 to L-1

Sometimes you can get algorithms that have lower amount of instructions, but still takes longer because it has varying amount of cache misses

Virtual Memory: memory on your hard drive, where each block is called a page (thus page file)
VM faults are called page faults; using your hard drive as RAM

Hybrid HDD/SSD: the cache is SS and the storage is HD

htop: Linux memory analyzer

Principle of Locality: it's easier to access a smaller amount of data (think about library analogy−easier to sign out book and bring it to your desk)
Temporal Locality: probability that if you use a book once, you'll use it again soon after
Spacial Locality: probability that if you use a book, you'll use the books around it

Struct: objects/categories

Amdahl's Law: speed up $= \frac{\text{old time}}{\text{new time}} = \frac{1}{\frac{t}{x} + (1-t)}$,
x = amount of improvement,
t = time unaffected by improvement
everything is generally a percentage
More on Amdahl's Law: https://www.youtube.com/watch?v=WdRiZEwBhsM

cache size in blocks > 2^m,
$x_n\, x_{n-1} \ldots x_1 x_0 \,\%\, b^m = x_{m-1} x_{m-2} \ldots x_1 x_0$ base b, $n \geq m$
assume the address is a string
"address" = "tag" "index" (the low order m-bits of the address)

Fault penalties:

Any virtual page can go anywhere in physical memory.

Translation-Lookaside Buffer (TLB): A cache that keeps track of recently used address mappings to try to avoid an access to the page table

Round Robin

Least-Recently Used (LRU):

tlbwr #puts EntryLo into TLB entry at Random

Average Memory Access Time (AMAT): hit latency [a.k.a. time for a hit; rounded up to highest cycle] + avg penalty [miss rate × miss penalty]
More aesthetic equation: AMAT = hit time + Miss rate × Miss penalty

AMAT × memory accesses = run time

Data bits: cache size (converted to bits)

$$\text{total sets} = \frac{\text{blocks}}{\text{words per block}}$$

$$\text{Tag field width} = \frac{\text{address space}}{\text{total sets}}$$

Tags: blocks × Tag field width

Valid bits = blocks = 256

Dirty bits = blocks = 256

$$\frac{\text{virtual address bits}}{\text{page size (bits)}} = \frac{\text{virtual address bits}}{\text{entries in page table} \times \text{ size per page table entry}}$$

Working set: set of popular pages in virtual memory

Memory-stall cycles/Instruction=Misses/Instruction×(Total miss latency–Overlapped miss latency)

Latency: the number of stages in pipeline or the number of stages between two instructions during execution

Chapter 6

MTTF: Mean Time To Failure
MTTR: Mean Time To Repair
MTBF: Mean Time Between Failures = MTTF + MTTR

$$\textbf{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

Rotational latency: (a.k.a. rotational delay) time, in milliseconds, until the desired sector of a disk rotates a part of a disk so that it is under the read-write head; on average, this is half the rotation time

$$\text{Average rotational latency} = \frac{0.5 \text{ rotations}}{\text{RPM} \times 60 \frac{\text{sec}}{\text{min}}}$$

$$\textbf{Transfer time} = \frac{\text{file size}}{\text{transfer rate } \left[\frac{\text{MB}}{\text{s}}\right]}$$

Disk Read Time: avrg seek time + avrg rotational delay + transfer time + controller overhead

2 Types of flash memory:

- NOR: used in RAM, embedded systems
- NAND: used in USB keys, cheaper, denser, block-at-a-time access

Polling: periodically checking the I/O status register; common in embedded systems due to low hardware cost, but wastes CPU time
Interrupts: device interrupts CPU when ready
DMA: Direct Memory Access, device runs autonomously, since the device has own processor

Chapter 7

Amdahl's Law doesn't apply to parallel computers, since we can achieve linear speedup, but only on applications with weak scaling

Unfortunately, increasing the amount of cores only benefits the execution time that is spent on tasks that can be run parallelly, since part has to be done sequentially and is thus, unaffected

$$\text{Speed-up} = \frac{\text{Execution time before}}{\left(\text{Execution time before} - \text{Execution time affected}\right) + \frac{\text{Execution time affected}}{100}}$$

$$= \frac{1}{\left(1 - \text{Fraction time affected}\right) + \frac{\text{Fraction time affected}}{100}}$$

Given the size of a problem is M and there are P processors:
Strong scaling: measuring speed-up while keeping fixed problem size, so memory per processor (M/P)
Weak scaling: program size grows proportionally to the number of processors (memory per processor is M)

Cluster
SIMD: Single-Instruction, Multiple Data

SMP: Shared memory multiprocessor

UMA: Uniform Memory Access time
NUMA: Non-UMA

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

SPMD: Single Program Multiple Data

Concurrency: generally software concept, where multiple threads are executed at the same time; can be implemented in sequential hardware by interleaving, or through parallelism (hardware)

Predicated:

Appendices

warp: a group of 8 SIMD threads
Usually, pipeline cycles through multiple warps, similar to multi-threading