YOU CAN NOT COPY ANY ARRAYS USING THE = OPERATOR
IF YOU'RE PASSING SOMETHING THAT SHOULDN'T BE CHANGED PASS AS `const`

C does not have classes/objects (only functions)

C does not support overloading

No `new delete` keywords, user `malloc()` and `free()`

C does not have reference variables, does have pointers

No bool or string in C

```
#include <stdio.h>
#define MAX 1000;
typedef char bigstring[MAX];
```

`/0` is the string terminating char

History:
1978 - Brian Kernighan and Dennis Ritchie at AT&T Bell Labs published a book describing a new high-level language with the simple name C

1985 - Bjarne Stroustrup published a book describing a C-based language called C++,

Data Types/Notes:
Int (%d) (limit ~ ∓2billion), double (%lf), char (%c)

Bits:
Unsigned char: 8 bits
Unsigned short: 16 bits
Unsigned long: 32 bits
Unsigned int: min 16 bits, max 32 bits?????
Unsigned long long 64 bits

When printing doubles, use `lf` (long float). Ex: `printf("%lf", 1.0);`
As a not, printf rounds. Ex: `printf("%.2lf", 1.1264);` will print 1.13
pi is most commonly written as 3.141593

Write a decimal using the following format 0.01 = 1.0e-2 || 100.0 = 1.0e2

`scanf("%s", str)` - reads up to the FIRST SPACE EVERY TIME
`fgets(str, num, stdin)` - reads in num number of chars WITH SPACES AND THE END CHAR (\n)

```
fgets(userText, USER_TEXT_LIMIT, stdin);
```

Math functions:
`sqrt(x); fabs(float) || abs(int); pow(base, exp);`
Because of division by 0, the expression `100 % (1 / 2)` results in an error
`3.0 / 2` gets converted into `3.0 / 2.0`

gcc a.c -o a.out → ./a.out

DESGIN PRINCIPLES

MIPS CHEAT SHEET

Strings

`#include <string.h>`

String literals are written as `char name[length] = "John";`

- `scanf("%s", name)` *NOTE: do not use the & symbol for this!!!!*
- *`int l = strlen(str);`*
- `strcpy("new", old); strncpy("new", old, #chars);`

You can't REassign a string with = (name = "John" will NOT work), you'll have to use

- `strcpy(name, "John");`

To print a special char, use %char. For example: `printf("%%")` → "%"

Note: the length of the string is the number pf chars + the terminating char (\0)

You can't compare two strings with **==**, use `strcmp(str1, str2) == 0` instead

Note: when using < or >, the program will go along each char and compare

- `strcmp(str1, str2) > 0` will check if str1 is > than str2

Note: doing `a = str[out of bounds]` will result in error, not blank as it may overwrite another var's memory space

To add to a string use `strcat(name, "extension");` Ex: `strcar(var, "!");`

Note: this will ONLY work with strings, not chars; Ex: `strcat(name, '!') == Error`

`isalpha(c), isdigit(c), isspace(c), ispunct(c), isxdigit(c)` - hex digit?;
`isprint(c)` - printable?;
`iscntrl(c)` - control char (not printable)
`toupper(c), tolower(c)` DO NOT change the original str, only return the char
`isalnum(c)` - is c alphabetic or a numeric digit?
`strchr(str, search` **`CHAR`**`);` - returns either NULL or address of 1st occurrence
`strlen(str)` - returns the length of the string
JUSTIFY STRING: `printf("%-30s", str);` will print the string 30 - len over

STRCHR REFERENCE - ALL RETURN A POINTER

Replacing a pointer to a place in a string will replace the char

`strchr` will check for a char in a str, `strstr` will check for a substr in a str

Adding 7 chars will look like: `strlen(str)+8` to account for the terminating char

Specifiers/Formatting (MORE INFO):

Floats:

Width: `printf("Value: %7.2f", myFloat);`
Precision: `printf("%+f", myFloat);`
Flags: `printf("%+f", myFloat);`
-: Left aligns the output given the specified width, padding the output with spaces.
+: Prints a preceding + sign for positive values. Negative numbers are always printed with the -

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

0: Pads the output with 0's when the formatted value has fewer characters than the width.
space: Prints a preceding space for positive value. `frintf("0[total].[dec]f", floatV)`
`printf("%3.4e", myFloat);` → `1.2340e+01` No # DEFAULTS TO 6 SPACES (6e)
`printf("Value: %7d", myInt);` will print: `Value:     301`
`printf("%08d", myInt);` will print `00000301`

Strings:
`printf("%20s String", str);` adds up to 20 spaces (- the length of str). Does not truncate
`printf("%.6s", myString);` truncates the string at index 6 ("Hello World" → "Hello ")
`printf("%-20s String", str);` left-aligns the string and adds spaces as needed
`fgets(userText, USER_TEXT_LIMIT, stdin);`

Booleans:
```
#include <stdbool.h>
bool name;
```
`rand() % n` will return a number between 0 and n - 1
Use `srand(seed)` or `srand((int)time(0));`

Arrays:
`type name[size]`
Multi-dimensional Arrays:
`type name[size1][size2][size3].....`
Ex: `int nums[12]; nums[0] = 12`
Ex: `int nums2d[12][5]; nums[0][0] = 12;`
Formatting Note: type name[num_rows][num_cols] and row # 1 is index #0
The null char can be overwritten, but this will make a printf() function print incorrectly

Naming Conventions:
const int NAME_HERE = 12;
int nameHere

Branches and Trees:
In a program, a branch is a sequence of statements only executed under a certain condition. *Ex: A hotel may discount a price only for people over age 65. An if branch is a branch taken only IF an expression is true.*
Note: due to "impercisions" do not compare floating-point types URL
`myVar = (condition) ? exprTrue : exprFalse;`
Note: can ONLY RETURN, so `(condition)?(x = 1):(y = 2);` won't work

MIPS CHEAT SHEET

```
switch(a) {
      case 0:
            fprint("ZERO\n");
            break;

      case 1:
            fprint("ONE\n");
            break;
      default:
            print("UNKNOWN\n");
            break;
}
```

Dynamic Allocation:

```
char *reverse( char *s ) {
      char &buf;
      int i, len;
      len = srtlen(s);
}
//Allocate memory for len+1 for null term
buf = (char) malloc()
???????????????????? Check lecture 2 notes?
```

Enums:

```
enum Colors { RED = 0, PURPLE = 1, BLUE = 2, GREEN = 3 };
enum old_name new_name //Create a new enum
var = RED;
```

REFERENCES:

`function name(int* varname) { *varname = 12; } → name(&varnameOG);`
OR `int* name = &other_var;`
OR `function(char* arr) { arr[0] = 'A'; } //Not *arr[0] = 'A';`

Programming Styles:

- **Modular development** is the process of dividing a program into separate modules that can be developed and tested separately and then integrated into a single program.
- **Incremental development** is a process in which a programmer writes, compiles, and tests a small amount of code, then writes, compiles, and tests a small amount more (an incremental amount), and so on.

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

Unit testing (Functions):

Unit testing is the process of individually testing a small part or unit of a program, typically a function

*Note: A **Testbench** is a program created specifically to test another program*

A **function stub** is a function definition whose statements have not yet been written.

- Ex: `int a() {printf("FIXME: Calculate MPG\n"); return 0; }`
- *Note: function declaration specifies the interface, while function definition specifies the contents*

Includes:

```
#include <thing> OR #include "file.h"
#ifndef FILENAME_H
#define FILENAME_H
// Header file contents
#endif
```

Memory Management:

`type *calloc(num_elements, element-size);` → `int* a = calloc(12, sizeof(int));`

`free(type *ptr) -> deallocates the memory in that pointer`

Files

```
FILE *fp = fopen(file_name, "r"); // read mode
char c[2];
fgets(c, 2, fp);
while (c != EOF) { printf("%c", ch); fgets(c, 2, fp); }
fclose(fp);
if( fscanf(inFile, "%d", &fileNum) == 1 )
```

`feof()` returns 1 if the previous read operation reached the end of the file

Binary Files:

```
fwrite(variablePointer, sizeof(type), numElements, outputFile)
fread(variablePointer, sizeof(type), numElements, inputFile)

FILE* inFile = fopen("myfile.bin", "rb");
long numBytesRead = fread(&fileNum, sizeof(int), 1, inFile);
while (numBytesRead != 0) {
  printf("num: %d\n", fileNum);
  numBytesRead = fread(&varName, sizeof(int), 1, inFile);
}
fclose(inFile);
```

MIPS CHEAT SHEET

Structs:

```
typedef struct{
  double** data;
  int rows;
  int cols;
} name;
```

`struct1 = struct2;` assigns all variables in struct to the other
`struct structName;`

.h files and Including them:

Include the *h* file in both the definition *c* file and the main *c* file

The struct definition appears in the *h* file and the functions appear in the *c* file

Pointers:

Create a pointer with `type* name = &var;`

Print the MEMORY ADDRESS of a var using `printf("%p", (void*) &name);`

`(type*)malloc(sizeof(type))` will allocate memory "dynamically"

`free(ptr);`

Bits to bytes is 8/1. For example: an int is 32 bits, so `sizeof(int)` returns 4 (bytes)

`ptr = (type*)realloc(ptr, numElements*sizeof(type));` resizes the memory for `ptr`

Logical Operators:

NOT: ¬

AND: Λ

OR: V

IF…THEN: →

ADT:

Abstract Data Types

- Uses Information hiding (hiding the underlying implementation, but NOT the data)
- Examples include `vector`

Vector:

- `vector vec;`
- `create_vector(&name, num_elements);`
- `type varname = *vector_at(&vec, index);`

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

- `*vector_at(&vec, index) = value;`
- `vector_size(&vec);`
- `vector_destroy(&nums);`
- `vector_resize(vec, newSize);`
- `vector_push_back(&vec, val);`
- `vector_erase(&vec, ind);`

Memory:

Mem limit: process id 65536?

A function's locals vars are located in the stack while it is executed

```
fprintf(stdout, "WORDS");
fprintf(stdout, "%d", numSeats);
fscanf(stdin, "%d", &maxEntries);
```

Recursion:

If the algorithm requires more than ~50 guesses/recursions, try re-writing it

- A binary search algorithm begins at the midpoint of the range and halves the range after each guess

Errors:

- Stack overflow - the stack frame for a function call extends past the end of the stack's memory. (Ex: too many recurssions)

Searches/sorts:

- Binary
- Linear
- Selection

Selection sort:

is a sorting algorithm that treats the input as two parts, a sorted part and an unsorted part, and repeatedly selects the proper next value to move from the unsorted part to the end of the sorted part. The outer loop will run N-1 times. The numbe rof comparisons will be $O(N^2)$

How many times longer will it take to sort a list with a elements vs a list with b elements $(\frac{a}{b})^2$

MIPS CHEAT SHEET

Insertion Sort:

Is a sorting algorithm that treats the input as two parts, a sorted part and an unsorted part, and repeatedly inserts the next value from the unsorted part into the correct location in the sorted part.

Quicksort:

A sorting algorithm that repeatedly partitions the input into low and high parts (each part unsorted), and then recursively sorts each of those parts. To partition the input, quicksort chooses a pivot to divide the data into low and high parts. The pivot can be any value within the array being sorted, commonly the value of the middle array element. Ex: For the list {4 34 10 25 1}, the middle element is located at index 2 (the middle of indices 0..4) and has a value of 10

`midpoint =` $i + \frac{k-i}{2}$

The midpoint is 1, so the pivot value is the element at `numbers[midpoint]`.

There are $\log_2$(N)...or.....In the worst case, for N elements, there will be `N-1` levels and `levels*N` comparisons????

Merge-sort:

divides a list into two halves, recursively sorts each half, and then merges the sorted halves to produce a sorted list. Keeps going until there's 1 item in each list. Creates a new list large enough to hold both lists (`2*N`)?

Big O:

O(5) → constant time complexity

O(N log N) → log-linear

O(N + N^2) → quadratic

merge/quick sort → O(N log N)

Selection-sort → O(N^2)

Command Line Arguments:

```
int main(int argc, char* argv[]){ }
```

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

Classes of Computers:

Personal computers:

- General purpose, variety of software
- Subject to cost/performance tradeoff

Server Computers:

- Network-based
- High capacity, performance, reliability
- Range from small servers to building-sized

Supercomputers

- Type of server
- High-end scientific and engineering calculations
- Highest capability but represent a small fraction of the overall computer market

Embedded computers

- Hidden as components of systems
- Stringent power/performance/cost constraints

The PostPC Era:

Personal mobile device

- battery operated
- Connects to the internet
- Hundreds of dollars
- Smartphones, tablets, electronic glasses

Cloud computing

- Warehouse Scale Computers (WSC)
- Software as a Service (SaaS)
- A portion of software run on a PMD and a portion run in the cloud
- Amazon and Google leading

Understanding Performance:

Algorithm

- Determines number of operations executed

Programming language, compiler, architecture

- Determine number of machine instructions executed per operation

MIPS CHEAT SHEET

Processor and memory system

- Determine how fast the instructions are executed

I/O system (including OS)

- Determines how fast I/O operations are executed

Seven Great Ideas:

- Use **abstraction** to simplify design
- Make the **common case fast**
- Performance via **parallelism**
- Performance via **pipelining**
- Performance vie **production**
- **Hierarchy** of memories
- **Dependability** via redundancy

Below Your Program:

Application Software

- Written in high-level language

System Software

- Compiler: translates HLL code to machine code
- Operating system: service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks and sharing resources

Hardware

- Processor, memory, I/O controllers

Levels of Program Code:

High-level language

- Level of abstraction closer to problem domain
- Provides productivity and portability

Assembly language

- Textual representation of instructions

Hardware Representation

- Binary digits (bits)

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

- Encoded instructions and data

Components of a Computer:

Same components for all kinds of computers

- Desktop, server, embedded

input/output:

- User-interface devices ???????
- ??????? (slide switched)

Liquid Crystal Display (LCD) Screen: picture elements (pixels)

- Mirrors content of frame buffer memory through the frame buffer
- Doesn't actually generate light, but rather either passes or blocks light

Active Matrix display is an LCD that uses a transistor to control whether light passes through each pixel

Inside the Processor:

Datapath

- Performs the operations of data

Control

- Sequences datapath, memory, ….

Cache memory

- Small fast SRAM memory for immediate access to data

Abstractions: (Get Notes for this)

Abstraction helps us deal with complexity

- Hide lower-level detail

Instruction set architecture (ISA)

- The hardware/software

Application binary Interface

- The ISA plus system software interface

Implementation

- The details underlying and interface

MIPS CHEAT SHEET

A safe place for data:

Volatile memory

- Loses instructions on power off

Non-volatile secondary memory

- Magnetic disk
- Flash memory
- Optical disk (CDROM, DVD)

Networks:

- Communication, resource sharing, nonlocal access
- Local area network (LAN): Ethernet
- Wide area network (WAN): the Internet
- Wireless network: WiFi, Bluetooth

Semiconductor Technology:

- A substance that does not conduct electricity well
- Silicon is a semiconductor

Add materials to transform properties

- Conductors
- Insulators
- Switch

Integrated Circuit Cost:

Cost per die $= \frac{cost\ per\ wafer}{Dies\ per\ wafer * Yield}$

Dies per wafer = wafer area/Die area

Yeld = 1/(1+(Defects per area * (Die area/2)))^2

Response Time and Throughput:

- Response time
    - Also called execution time
    - The total time required for the computer to complete a task, including disk accesses, memory accesses, I/O activities, operating system overhead, CPU execution time, and so on.
    -

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

- Throughput
    - Also called bandwidth
    - Another measure of performance, it is the number of tasks completed per unit time.
- How are response time and throughput affected by:
    - Replacing the processor with a faster version
    - Adding more processors

Relative Performance:

- Define Performance = 1/Execution time
- "X is n time faster than Y"
    - $Performance_x / Performance_y = Execution\ time_y / Execution\ time_x = n$
- Example: time taken to run a program
    - 10s on A, 15s on B
    - $Execution\ time_b / Execution\ time_a = 15s/10s = 1.5$
    - So A is 1.5 times faster than B

Measuring Execution Time:

- Elapsed Time
    - Total response time, including all aspects
        - Discounts I/O time, other jobs' shares
    - Determines System Performance
- CPU Time
    - Time spent processing a given job
        - Discounts I/O time, other jobs' shares
    - Compromises user CPU time and system CPU time
    - Different programs are affected differently by CPU and system performance

CPU Clocking:

- Operation of digital hardware governed by a constant-rate clock
- Clock period: duration of a clock cycle
    - Eg., 250ps = 0.25ns = $250*10^{-12}$s
- Clock frequency (rate): cycles per second
    - Eg., 4GHz = 4000MHz = $4.0*10^{9}$Hz

MIPS CHEAT SHEET

PERFORMANCE MEASUREMEMNT UNITS

CPU Time:

CPU Time = CPU Clock cycles * clock cycle time = $\frac{CPU\ clock\ cycles}{Clock\ Rate} = \frac{instruction\ time}{Clock\ Rate}$

CPI in more detail:

- If different instruction classes take different number of cycles
  - $clock\ cycles = \sum_{i=1}^{n} (CPI_i * Instruction\ Count_i)$
- Weighted Average CPI
  - $CPI = \frac{Clock\ cycles}{Instruction\ Count} = \sum_{i=1}^{n} (CPI_i * \frac{Instruction\ Count_i}{Instruction\ Count})$

$IPS = \frac{Clock\ Rate}{CPI}$

Performance Summary:

$$\text{CPU Time} = \frac{Instructions}{Program} * \frac{Clock\ Cycles}{Instruction} * \frac{seconds}{Clock\ Cycle}$$

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c

Reducing Power:

- Suppose a new CPU has
  - 85% of capacitive load of the old CPU
  - 15% voltage and 15% frequency reduction
  - $\frac{P_{new}}{P_{old}} = \frac{C_{old} * 0.85 * (V_{old} * 0.85)^2 * F_{old} * 0.85}{C_{old} * V_{old} * F_{old}} = 0.85^4 = 0.52$
  - $\frac{P_{new}}{P_{old}} = \frac{(Cap\ load * 0.85) * (voltage * 0.85)^2 * (Frequency\ switched * 0.85)}{Capacitative\ load * voltage^2 * Frequency\ Switched}$
- The power wall
  - We can't reduce the voltage further
  - We can't remove more heat
- How else can we improve performance

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

Energy = capacitative load * voltage^2

Power = $\frac{1}{2}$ Capacitiative load * Voltage^2 * Frequency switched

Increase in power given voltage, old/new frequency, and old power

$P_{new} = \frac{P_{old}}{}$

Multiprocessors:

- Multicore microprocessors
    - More than one processor chip
- Requires explicitly parallel programming
    - Compare with instruction level parallelism
        - Hardware executes multiple instructions at once
        - Hidden from programmer
    - Hard to do
        - Programming for performance
        - Load balancing
        - Optimizing communication and synchronization

SPEC CPU Benchmark:

- Programs used to measure performance
    - Supposedly typical of actual workload
- Standard Performance Evaluation Corp
    - Develops benchmarks for CPU, I/O, Web, …
- SPEC CPU2006
    - Elapsed time to execute a selection of programs
        - Negligible I/O, so focuses on CPU performance
    - Normalize relative to reference machine
    - Summarize as geometric mean of performance ratios
    - CINT2006 (integer) and CFP2006 (floating-point)

SPEC Power Benchmark:

- Power consumption of server at different workload levels
    - Performance: ssj_ops/sec
    - Power: Watts (Joules/sec)

- Overall ssj_ops per Watt $= (\sum_{i=0}^{10} ssj_{ops_i}) \backslash (\sum_{i=0}^{10} power_i)$

SPEC Ratio:

SPEC Ratio = reference time / runtime per benchmark (new)

Pitfall: Amdahl's Law

- Improving an aspect of a computer and expecting a proportional improvement in overall performance
- $T_{improved} = \frac{T_{affected}}{improvement\ factor} + T_{unaffected}$
- Example: multiply accounts for 80s/100s
  - How much improvement in multiply performance to get 5x overall?
  - $20 = \frac{80}{n} + 20$ AKA can't be done!
- Corollary: make the common case fast

Fallacy: Low Power at Idle

- Look back at i7 power benchmark
  - At 100% load: 258W
  - At 50% load: 170W (66%)
  - At 10% load: 121W (47%)
- Google Data Center
  - Mostly operates at 10% - 50% load
  - At 100% load less than 1% of the time
- Consider designing processors to make power proportional to load

Pitfall: MIPS as a Performance Metric

- MIPS: Millions of Instructions Per Second
  - Doesn't account for
    - Differences in ISAs between computers
    - Differences in complexity between instructions
    - $MIPS = \frac{instruction\ \ count}{Execution\ count * 10^6} = \frac{Clock\ rate}{CPI * 10^6}$
  - CPI varies between programs on a given CPU

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

NOTE: computer advances have led to emphasis instead on parallel processors and hierarchical memory, which will be introduced later

Decimal	Abbreviation	Value	Binary term	Abbreviation	Value	% Larger
kilobyte	KB	1000^1	kibibyte	KiB	2^{10}	2%
megabyte	MB	1000^2	mebibyte	MiB	2^{20}	5%
gigabyte	GB	1000^3	gibibyte	GiB	2^{30}	7%
terabyte	TB	1000^4	tebibyte	TiB	2^{40}	10%
petabyte	PB	1000^5	pebibyte	PiB	2^{50}	13%
exabyte	EB	1000^6	exbibyte	EiB	2^{60}	15%
zettabyte	ZB	1000^7	zebibyte	ZiB	2^{70}	18%
yottabyte	YB	1000^8	yobibyte	YiB	2^{80}	21%
ronnabyte	RB	1000^9	robibyte	RiB	2^{90}	24%
queccabyte	QB	1000^{10}	quebibyte	QiB	2^{100}	27%

Bti is short for Binary Digit

Assembler: A program that translates a symbolic version of instructions into the binary version.
Assembly language: A symbolic representation of machine instructions.
Machine language: A binary representation of machine instructions.
Input device: A mechanism through which the computer is fed information, such as a keyboard.
Output device: A mechanism that conveys the result of a computation to a user, such as a display, or to another computer.
Datapath: is where data is transformed via computations like addition or subtraction
An **integrated circuit** is often called a chip
The CPU (processor) contains the datapath and control and which adds numbers, tests numbers, signals I/O devices to activate.

Note: The CPU (processor) contains the datapath and control and which adds numbers, tests numbers, signals I/O devices to activate.

MIPS CHEAT SHEET

RAM:

- DRAM - Large memory where most data is stored (most expensive)
- SRAM - A faster memory technology than DRAM, but using more area to store a bit.
- Cache - A small memory that keeps a copy of data from larger memory.

Volatile memory:

- Storage, such as DRAM, that retains data only if it is receiving power

Nonvolatile memory:

- A form of memory that retains data even in the absence of a power source and that is used to store programs between runs
- A DVD disk is nonvolatile.

Memory Hierarchy:

Main memory:

- Also called primary memory
- Memory used to hold programs while they are running
- typically consists of DRAM in today's computers.

Secondary memory:

- Nonvolatile memory used to store programs and data between runs
- typically consists of flash memory in PMDs and magnetic disks in servers.

Magnetic disk:

- Also called hard disk.
- A form of nonvolatile secondary memory composed of rotating platters coated with a magnetic recording material.
- Because they are rotating mechanical devices, access times are about 5 to 20 milliseconds and cost per gigabyte in 2020 was $0.01 to $0.02.

Flash memory:

- A nonvolatile semiconductor memory.
- It is cheaper and slower than DRAM but more expensive per bit and faster than magnetic disks.

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

- Access times are about 5 to 50 microseconds and the cost per gigabyte in 2020 was $0.06 to $0.12.

Instruction set architecture:

- Also called architecture
- An abstract interface between the hardware and the lowest-level software that encompasses all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory access, I/O, and so on.

Application binary interface (ABI):

- The user portion of the instruction set plus the operating system interfaces used by application programmers
- It defines a standard for binary portability across computers.

Networks/Communication Between Computers:

Local area network (LAN):

- A network designed to carry data within a geographically confined area, typically within a single building.

Wide area network (WAN):

- A network extended over hundreds of kilometers that can span a continent.

Note: IEEE 802.11 is a widely-used local wireless networking technology, also known as "WiFi"

Transistor:

- An on/off switch controlled by an electric signal.

Very-large-scale integrated (VLSI) circuit:

- A device containing hundreds of thousands to millions of transistors.

Chip Manufacturing:

Silicon crystal ingot: A rod composed of a silicon crystal that is between 8 and 12 inches in diameter and about 12 to 24 inches long.

Wafer: A slice from a silicon ingot no more than 0.1 inches thick, used to create chips.

Defect: A microscopic flaw in a wafer or in patterning steps that can result in the failure of the die containing that defect.

MIPS CHEAT SHEET

Die: The individual rectangular sections that are cut from a wafer, more informally known as chips.

Yield: The percentage of good dies from the total number of dies on the wafer.

- After being sliced from the silicon ingot, blank wafers are put through 20 to 40 steps to create patterned wafers.
- These patterned wafers are then tested with a wafer tester, and a map of the good parts is made.
- Then, the wafers are diced into dies.
- These good dies are then bonded into packages and tested one more time before shipping the packaged parts to customers. One bad packaged part was found in this final test.

Cost per die $= \frac{cost\ per\ wafer}{Dies\ per\ wafer\ *\ yield}$

Dies per wafer $\approx \frac{Wafer\ area}{Die\ area}$

Yield $= \frac{1}{(1+(Defects\ per\ area\ *\ Die\ area/2))^2}$

What affects what

Instructions:

Instruction set: The vocabulary of commands understood by a given architecture.

- Can be represented as numbers
- Is the language, not the program written in the language

VOCAB

- **Word:** The natural unit of access in a computer, usually a group of 32 bits; corresponds to the size of a register in the MIPS architecture.
- **Data transfer instruction:** A command that moves data between memory and registers.
- **Address :** A value used to delineate the location of a specific data element within a memory array.

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

3 popular Instruction Sets:

1. ARMv7 is similar to MIPS. More than 100 billion chips with ARM processors will be manufactured in between 2017 and 2020, making it the most popular instruction set in the world.
2. The second example is the Intel x86, which powers both the PC and the cloud of the PostPC Era.
3. The third example is ARMv8, which extends the address size of the ARMv7 from 32 bits to 64 bits. Ironically, as we shall see, this 2013 instruction set is closer to MIPS than it is to ARMv7.

MIPS:

Instructions:

- Add
  - *add x, y, z* → will add *y* and *z*, then put the resulting value in *x*
  - *add a, a, b* → will add *a* and *b*, then put the resulting value in *a*
- Sub
  - sub a, b, c → will subtract b from c, then put the result in a

Comments: #comment here

Operands:

Name	Example	Comments
32 registers	`$s0..$s7, $t0..$t9, $zero, $a0..$a3, $v0..$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory [4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

Note: the memory words is NOT the same as a register!!!

Every register is 32 bits wide

More registers means a slower clock frequency, as it takes more time to cover them all

MIPS INSTRUCTION CHEAT SHEET PRINT THIS!!!!!!!

Example: compiling a C program using registers:

MIPS CHEAT SHEET

Q:

```
f = (g + h) - (i + j);
```

The variables f, g, h, i, and j are assigned to the registers $s0, $s1, $s2, $s3, and $s4, respectively. What is the compiled MIPS code?

A:

1. Add 2 temporary registers called t0 and t1
2. Compute:

```
add $t0, $s1, $s2  #register $t0 contains g + h
add $t1, $s3, $s4  #register $t1 contains i + j
sub $s0, $t0, $t1  #f gets $t0 - $t1, which is (g + h) - (i + j)
```

Load: The data transfer instruction that copies data from memory to a register

In an **lw** *instruction, a* ***base address*** *is the starting address of an array in memory, a* ***base register*** *is a register that holds an array's base address, and an* ***offset*** *is a constant value added to a base address to locate a particular array element.*

To copy the value in index 8 of an array in MIPS, use the following formula:

`$s3` = array (`A`)'s base address

`$t0` = temporary register

```
lw $t0, 8($s3)
```

This sets `$t0` to `A[8]`

Alignment restriction: A requirement that data be aligned in memory on natural boundaries.

NOTE: In MIPS, words must start at addresses that are multiples of 4. This requirement is called an alignment restriction, and many architectures have it. (COD Chapter 4 (The Processor) suggests why alignment leads to faster data transfers).

WARNING: Words MUST be a multiple of 4 and thus the following is not valid:

Assuming `$s3` has `5000`, is the following an acceptable instruction?

`lw $t0, 3($s3)` → ***NO*** because 3 is not a multiple of 4

`Sw` → "Store Word", stores the word in a register

Ex: `sw $t0, 48($s3)` will store whatever is in `$t0` into the 12th index of `$s3` (12 * 4 = 48)

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

BUT: If `$s3` has 900, `sw $t0, 20($s3)` will save to address **920**

NOTES

To add a constant to a memory address, use the `addi` command.

Ex: `addi $s3, $s3, 4    # $s3 = $s3 + 4`

To distiguish between base 10 and binary systems, binary systems are denoted as $\#_{two}$ and base 10 numbers are denoted as $\#_{10}$

Binary digit: Also called a bit. One of the two numbers in base 2 (0 or 1) that are the components of information.

Add/subtract the base 10 numbers as you toggle the binary digits *(the largest 32-bit # is 4 **bill**)*

Least significant bit: The rightmost bit in a MIPS word.
Most significant bit: The leftmost bit in a MIPS word.

Overflow: when the results of an operation are larger than can be represented in a register.
sign and magnitude representation: a signed number representation where a single bit is used to represent the sign, and the remaining bits represent the magnitude

Note: To represent a number as negative, simply flip the 0s and 1s, MOVING THE START 1 OVER.
Ex:
2_{ten} = 0000 0000 0000 0000 0000 0000 0000 0010_{two}
-2_{ten} = 1111 1111 1111 1111 1111 1111 1111 1110_{two}

TWO'S COMPLIMENT
$-2{,}147{,}483{,}648_{ten}$ DOES NOT EXIST IN TWO'S COMPLIMENT

MIPS CHEAT SHEET

Two's Compliment Overflow occurs when the leftmost retained bit of the binary bit pattern is not the same as the infinite number of digits to the left (the sign bit is incorrect): a 0 on the left of the bit pattern when the number is negative or a 1 when the number is positive (see THIS LINK)
One's Compliment
Biased Notation

Hexadecimal: Numbers in base 16 (written as $\#_{hex}$)

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

MIPS Fields:

- *op* : Basic operation of the instruction, traditionally called the *opcode*.
- *rs*: The first register source operand.
- *rt*: The second register source operand.
- *rd*: The register destination operand. It gets the result of the operation.
- *shamt*: Shift amount. (COD Section 2.6 (Logical operations) explains shift instructions and this term; it will not be used until then, and hence the field contains zero in this section.)
- *funct*: Function. This field, often called the function code, selects the specific variant of the operation in the op field.

Opcode: The field that denotes the operation and format of an instruction.

I-format:

destination register: a register that receives the result of an operation.

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

MIPS architecture:

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

MIPS Instructions:

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

OR: A logical bit-by-bit operation with two operands that calculates a 1 if there is a 1 in either operand.

NOT: A logical bit-by-bit operation with one operand that inverts the bits; that is, it replaces every 1 with a 0, and every 0 with a 1.

NOR: A logical bit-by-bit operation with two operands that calculates the NOT of the OR of the two operands. That is, it calculates a 1 only if there is a 0 in both operands.

Note: Logical AND immediate and logical OR immediate put 0s into the upper 16 bits to form a 32-bit constant, unlike add immediate, which does sign extension.

Instructions for making decisions:

```
beq register1, register2, L1
```

MIPS CHEAT SHEET

This instruction means go to the statement labeled `L1` if the value in `register1` equals the value in `register2`. The mnemonic `beq` stands for *branch if equal*.

```
bne register1, register2, L1
```

It means go to the statement labeled `L1` if the value in `register1` does *not* equal the value in `register2`. The mnemonic `bne` stands for *branch if not equal*. These two instructions are traditionally called *conditional branches*.

Conditional branch: An instruction that requires the comparison of two values and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the comparison.

Unconditional branch: an instruction that always follows the branch (EXAMPLE)

Loops:

```
While (x == y) {
      \\Loop body
}
```

```
Loop: bne $s0, $s1, Exit
      # Loop body
      j Loop
Exit:
```

Basic block: A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning).

slt: "Set on less than" - sets register 3 to 1 if register 1 is < than register 2, else sets reg 3 to 0
Ex: `slt $t0, $s3, $s4 # t0 = 1 if $s3 < $s4`
Ex: `slti $t0, $s2, 10 # t0 = 1 if $s2 < 10`

sltu: "Set less than unsigned" - unsigned checking version of `slt`
This can also become `sltiu`

Case/Switch Statement:

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

Jump address table: Also called **jump table**. A table of addresses of alternative instruction sequences.

OPERATOR NOTATION NOTE

- **Unresolved reference:** A reference that requires more information from an outside source to be complete.
- **Linker** (Also called a **Link editor**): A systems program that combines independently assembled machine language programs and resolves all undefined labels into an executable file.
- **Assembler directive:** An operation that tells the assembler how to translate a program but does not produce machine instructions; always begins with a period.
- **Source language:** The high-level language in which a program is originally written.
- **External label:** Also called **global label**. A label referring to an object that can be referenced from files other than the one in which it is defined.
- **Local label:** A label referring to an object that can be used only within the file in which it is defined.
- **Forward reference:** A label that is used before it is defined.
- **Symbol table:** A table that matches names of labels to the addresses of the memory words that instructions occupy.

An assembler's first pass reads each line of an assembly file and breaks it into its component pieces. These pieces, which are called **lexemes**, are individual words, numbers, and punctuation characters.

Backpatching: A method for translating from assembly language to machine instructions in which the assembler builds a (possibly incomplete) binary representation of every instruction in one pass over a program and then returns to fill in previously undefined labels.

Object file format:
A UNIX assembler produces an object file with six distinct sections.

MIPS CHEAT SHEET

- The object file header describes the size and position of the other pieces of the file.
- The text segment contains the machine language code for routines in the source file. These routines may be unexecutable because of unresolved references.
- The data segment contains a binary representation of the data in the source file. The data also may be incomplete because of unresolved references to labels in other files.
- The relocation information identifies instructions and data words that depend on absolute addresses. These references must change if portions of the program are moved in memory.
- The symbol table associates addresses with external labels in the source file and lists unresolved references.
- The debugging information contains a concise description of the way the program was compiled, so a debugger can find which instruction addresses correspond to lines in a source file and print the data structures in readable form.

`.asciiz "The sum from 0 .. 100 is %d\n"` will store the string in memory
NOTE:

In the execution of a procedure, the program must follow these six steps:

- Put parameters in a place where the procedure can access them.
- Transfer control to the procedure.
- Acquire the storage resources needed for the procedure.
- Perform the desired task.
- Put the result value in a place where the calling program can access it.
- Return control to the point of origin, since a procedure can be called from several points in a program.

As mentioned above, registers are the fastest place to hold data in a computer, so we want to use them as much as possible. MIPS software follows the following convention for procedure calling in allocating its 32 registers:

- $a0 - $a3: four argument registers in which to pass parameters
- $v0 - $v1: two value registers in which to return values

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

- $ra: one return address register to return to the point of origin

- **jump-and-link instruction:** An instruction that jumps to an address and simultaneously saves the address of the following instruction in a register ($ra in MIPS).
    - `jal ProcedureAddress`
- **Return address:** A link to the calling site that allows a procedure to return to the proper address; in MIPS it is stored in register $ra.
    - `jr $ra`
- **Caller:** The program that instigates a procedure and provides the necessary parameter values.
- **Callee:** A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.
- **Program counter *(PC)***: The register containing the address of the instruction in the program being executed.

- Stack: A data structure for spilling registers organized as a last-in- first-out queue.
- Stack pointer: A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found. In MIPS, it is register $sp.
- Push: Add element to stack.
- Pop: Remove element from stack.

➔ $t0 - $t9: temporary registers that are not preserved by the callee (called procedure) on a procedure call

➔ $s0 - $s7: saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)

- Global pointer: The register that is reserved to point to the static area.

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

MIPS CHEAT SHEET

- Procedure frame: Also called activation record. The segment of the stack containing a procedure's saved registers and local variables.
- Frame pointer: A value denoting the location of the saved registers and local variables for a given procedure.
- Text segment: The segment of a UNIX object file that contains the machine language code for routines in the source file.

Note: the heap grows upwards and the stack grows downwards

Name	Register number	Usage	Preserved on call?
`$zero`	0	The constant value 0	n.a.
`$v0–$v1`	2–3	Values for results and expression evaluation	no
`$a0–$a3`	4–7	Arguments	no
`$t0–$t7`	8–15	Temporaries	no
`$s0–$s7`	16–23	Saved	yes
`$t8–$t9`	24–25	More temporaries	no
`$gp`	28	Global pointer	yes
`$sp`	29	Stack pointer	yes
`$fp`	30	Frame pointer	yes
`$ra`	31	Return address	yes

`lb $t2, 0($t1)` → loads the

I skipped this pls review thx

https://learn.zybooks.com/zybook/RPICSCI2500KuzminFall2022/chapter/5/section/3

https://learn.zybooks.com/zybook/RPICSCI2500KuzminFall2022/chapter/8/section/3

https://learn.zybooks.com/zybook/RPICSCI2500KuzminFall2022/chapter/6/section/1

Asserted Signal: A signal that is (logically) true, or 1.

Deasserted signal: A signal that is (logically) false, or 0.

Combinational logic: A logic system whose blocks do not contain memory and hence compute the same output given the same input.

Sequential logic: A group of logic elements that contain memory and hence whose value depends on the input as well as the current contents of the memory.

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

Boolean Algebra:

The OR operator is + (A + B)

The AND operator is written as ● (A ● B)

- Identity law: $A + 0 = A$ and $A \cdot 1 = A$
- Zero and One laws: $A + 1 = 1$ and $A \cdot 0 = 0$
- Inverse laws: $A + \overline{A} = 1$ and $A \cdot \overline{A} = 0$
- Commutative laws: $A + B = B + A$ and $A \cdot B = B \cdot A$
- Associative laws: $A + (B + C) = (A + B) + C$ and $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- Distributive laws: $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ and $A + (B \cdot C) = (A + B) \cdot (A + C)$

Gate: A device that implements basic logic functions, such as AND or OR.

NOR gate: An inverted OR gate.

NAND gate: An inverted AND gate.

*Note: a **Multiplexer** might also be called a **Selector***

Selector value: Also called control value. The control signal is used to select one of the input values of a multiplexor as the output of the multiplexor.

Multiplexors can be created with an arbitrary number of data inputs. When there are only two inputs, the selector is a single signal that selects one of the inputs if it is true (1) and the other if it is false (0). If there are n data inputs, there will need to be $log_2(n)$ selector inputs. In this case, the multiplexor basically consists of three parts:

- A decoder that generates n signals, each indicating a different input value
- An array of n AND gates, each combining one of the inputs with a signal from the decoder
- A single large OR gate that incorporates the outputs of the AND gates

Sum of products: A form of logical representation that employs a logical sum (OR) of products (terms joined using the AND operator).

Note: MIPS addu, addiu, and subu CAN operate on either signed or unsigned operands, there is just no error on overflow

Programmable logic array (PLA): A structured-logic element composed of a set of inputs and corresponding input complements and two stages of logic: the first generates product terms of

the inputs and input complements, and the second generates sum terms of the product terms. Hence, PLAs implement logic functions as a sum of products.

MIPS Multiplication:

Multiply normally (mult each of the bottom to each of the top, adding a 0 at the start every time), then add the results together

Registers:

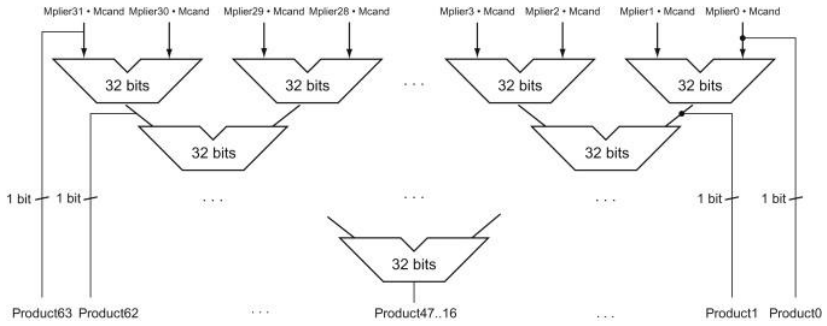
The Multiplier shifts bits to the right and is 32 bits wide

The Multiplicand shifts bits to the left and is 64 bits wide

The product register is 64 bits wide

Note: Each iteration of the multiplication algorithm consists of 3 basic steps

Faster Multiplication:



MIPS Division: REVISIT

Literally just normal long division.

1. See how many times the thing outside the house goes into the thing inside the house
2. Put the first thing * that under the number in the house then divide
3. Put the # of times the thing outside went in, or 0 if it didn't
4. Whatever's left at the end is…THE REMAINDER WOW!!!!!

Dividend: A number being divided.

Divisor: A number that the dividend is divided by.

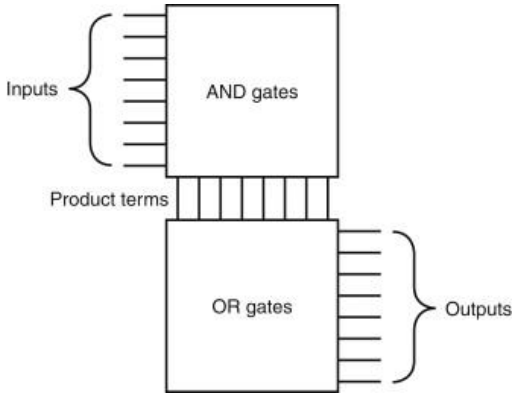
Quotient: The primary result of a division; a number that when multiplied by the divisor and added to the remainder produces the dividend.

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

Remainder: The secondary result of a division; a number that when added to the product of the quotient and the divisor produces the dividend.

Minterms: Also called product terms. A set of logic inputs joined by conjunction (AND operations); the product terms form the first logic stage of the programmable logic array (PLA).



ROMs

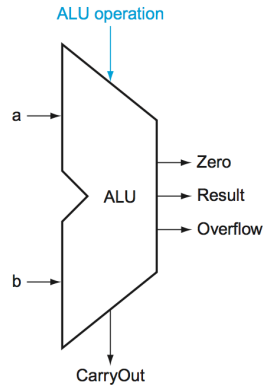
- **Read-only memory (ROM)**: A memory **whose contents are designated at creation time**, after which the contents can only be read. ROM is used as structured logic to implement a set of logic functions by using the terms in the logic functions as address inputs and the outputs as bits in each word of the memory.
- **Programmable ROM (PROM)**: A form of read-only memory that can be programmed when a designer knows its contents.
- **Don't Cares**: inputs that don't matter (AKA B in `if(A||B)`)

Arrays of logic elements:

- **Bus**: In logic design, a collection of data lines that is treated together as a single logical signal; also, a shared collection of lines with multiple sources and uses.
- **arithmetic logic unit (ALU):** The device that performs the arithmetic operations like addition and subtraction or logical operations like AND and OR

```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [31:0] A,B;
   output reg [31:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
   always @(ALUctl, A, B) begin //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1 : 0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0;
      endcase
    end
endmodule
```

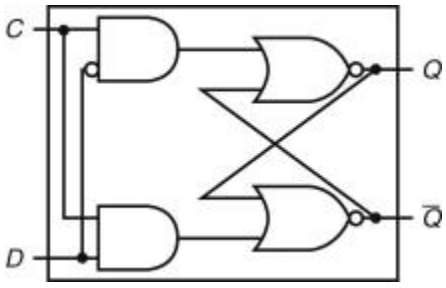
MIPS CHEAT SHEET



- **Flip-flop:** A memory element for which output is equal to the value of the stored state inside the element and for which the internal state is changed only on a clock edge.
- **Latch:** A memory element in which the output is equal to the value of the stored state inside the element and the state is changed whenever the appropriate inputs change and the clock is asserted.
- **D flip-flop:** A flip-flop with on data input that stores the value of that input signal in the internal memory when the clock edge occurs.

Interrupt: An exception that comes from outside of the processor. (Some architectures use the term *interrupt* for all exceptions.)

D Latch implemented with NOR gates



Here is a Verilog description of a module for a rising-edge D flip-flop, assuming that C is the clock input and D is the data input:

```
module DFF(clock, D, Q, Qbar);
   input clock, D;
   output reg Q;         // Q is a reg since it is assigned in an always block

   output Qbar;
```

Note: declaring a struct just declares the type and does NOT use any memory!

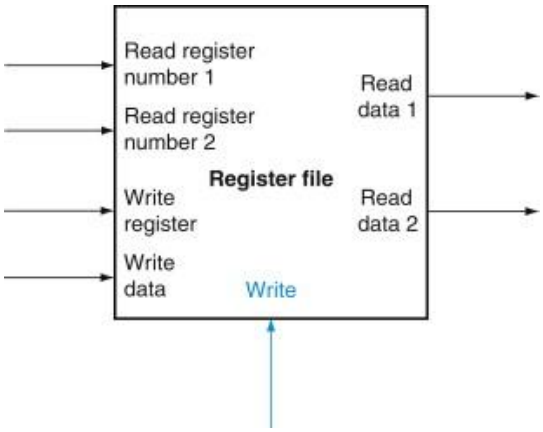
RETURN TO

```
   assign Qbar = ~ Q;    // Qbar is always just the inverse of Q

   always @(posedge clock)  // perform actions whenever the clock rises
      Q = D;
endmodule
```

Setup time: The minimum time that the input to a memory device must be valid before the clock edge.
Hold time: The minimum time during which the input must be valid after the clock edge.

A register file with two read ports and one write port has five inputs and two outputs



Register Files

- A set of registers that can be read and written by supplying a register number to be accessed
- Needs three inputs: a register number, the data to write, and a clock that controls the writing into the register

Hardware Schematic: a diagram that shows how the combinational gates should be connected to achieve a particular hardware functionality

Hardware Description Language: converts code for how a block should behave into a hardware schematic

VERILOG IS A HARDWARE DESCRIPTION LAGUAGE

Design Functionality Example (for a D-flip-flop):

- Clock should be an input to the flop
- If the active-low reset is 0, then the flop should reset
- If the active-low reset is 1, then the flop output ‘q’ should follow input ‘d’
- Output 'q' should get a new value only at the posedge of clock

Verification: Test the circuit by providing input stimuli to the design model

Note: All simulations performed by EDA (Electronic Design Automation) software tools and the Verilog RTL is placed inside something called a “testbench”, which…tests it with different stimuli

Sections of Verilog Code (in order):

All sections should be contained within the `module` and `endmodule` keywords

- Module definition and port list declaration
- List of input and output ports
- Declaration of other signals using allowed Verilog data types
- Design may depend on other Verilog modules and hence their instances are created by module instantiations
- The actual Verilog design for this module that describes its behavior

```
module [design_name] ( [port_list] );

    [list_of_input_ports]
    [list_of_output_ports]

    [declaration_of_other_signals]

    [other_module_instantiations_if_required]

    [behavioral_code_for_this_module]
endmodule
```

Ex:

```
module dff (input d, rstn, clk, output q);
      reg q; //Declare variable to store output values
```

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

```
	always @ (posedge clk) begin //This block is always executed at
	// the positive edge of clk 0->1
		if (!rstn)
			//At the posedge, if rstn is 0, then q should get 0
			q <= 0;
		else
			//At the posedge, if rstn is 1, then q should get d
			q <= d;
	end
endmodule
```

Testbench Code:

```
module tb;
	// declare inp/outp vars
	reg tb_clk;
	reg tb_d;
	reg tb_rstn;
	reg tb_q;

	//Create an instance of the design (instantiation)
	dff   dff0( .clk  (tb_clk),  //connect clock inp with TB signal
		    .d    (tb_d),   //connect data inp with TB signal
		    .rstn (tb_rstn) //connect reset inp with TB signal
		    .q    (tb_q)    //connect output q with TB signal

	//Drive signals tb_* with certain values (stimulus)
	//Since these tb_* signals are connected to design inputs,
	//The design will be driven with the values in tb_*
	initial begin
		tb_rsnt     <= 1'b0;
		tb_clk<= 1'b0;
		tb_d        <= 1'b0;
	end
endmodule
```

MIPS CHEAT SHEET

Memory Elements:

Static random access memory (SRAM): A memory where data is stored statically (as in flip-flops) rather than dynamically (as in DRAM). SRAMs are faster than DRAMs, but less dense and more expensive per bit.

The specs are written as follows: # of entries (in millions) ✖ width of each address
Ex: 4M ✖ 8 ⇒ 4M entries, each 8 bits wide. This means it will have 22 address lines (because 2^22 =~ 4 million), an 8-bit input line, and an 8-bit output line

Note: the number of addressable locations is often called the height, with the number of bits per unit called the width.
Large memory is implemented with a shared output line called a *bit line.*
To allow multiple sources to drive a single line, a three-state-buffer (or tristate buffer) is used.
A three-state buffer has two inputs—a data signal and an Output enable—and a single output, which is in one of three states: asserted, deasserted, or high impedance. The output of a tristate buffer is equal to the data input signal, either asserted or deasserted, if the Output enable is asserted, and is otherwise in a high-impedance state that allows another three-state buffer whose Output enable is asserted to determine the value of a shared output.

- The value stored in any cell is kept on a pair of inverting gates, and as long as power is applied, the value can be kept indefinitely
- require four to six transistors per bit

DRAMs

- the value kept in a cell is stored as a charge in a capacitor,
- A single transistor is then used to access this stored charge, either to read the value or to overwrite the charge stored there.
- Because DRAMs use only a single transistor per bit of storage, they are much denser and cheaper per bit
- use a two-level decoding structure, and this allows us to refresh an entire row (which shares a word line) with a read cycle followed immediately by a write cycle

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

- Typically, refresh operations consume 1% to 2% of the active cycles of the DRAM, leaving the remaining 98% to 99% of the cycles available for reading and writing data.

Error correction

- Because of the potential for data corruption in large memories, most computer systems use some sort of error-checking code to detect possible corruption of data. One simple code that is heavily used is a *parity code*. In a parity code the number of 1s in a word is counted; the word has odd parity if the number of 1s is odd and even otherwise. When a word is written into memory, the parity bit is also written (1 for odd, 0 for even). Then, when the word is read out, the parity bit is read and checked. If the parity of the memory word and the stored parity bit do not match, an error has occurred.
    - This cannot detect an even-number of errors

Error detection code: A code that enables the detection of an error in data, but not the precise location and, hence, correction of the error.

Finite State Machines (State Machines):

Consists of:

- A sequential logic function consisting of a set of inputs and outputs
- A next-state function that maps the current state and the inputs to a new state
- An output function that maps the current state and possibly the inputs to a set of asserted outputs.

THESE RUN SYNCRONOUSLY ON THE EDGE OF EVERY CLOCK CYCLE

Is used for:

- Sequential systems

Next-state function: A combinational function that, given the inputs and the current state, determines the next state of a finite-state machine.

Note: When a finite-state machine is used as a controller, the output function is often restricted to depend on just the current state. Such a finite-state machine is called a Moore machine. This is

the type of finite-state machine we use throughout this book. If the output function can depend on both the current state and the current input, the machine is called a Mealy machine.

Timing Methodologies:

Race: When the contents of a state element depend on the relative speed of different logic elements

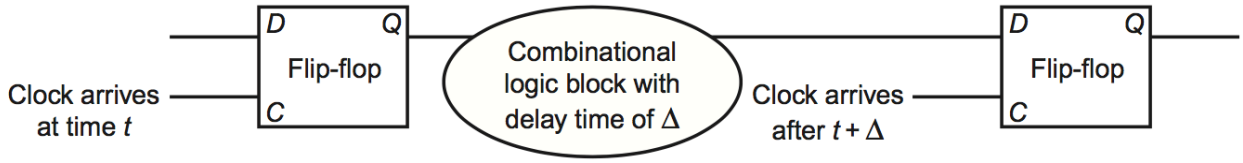
Such a system must allow for:

$t_{prop} + t_{combinational} + t_{setup}$

Where:

- t_{prop} is the time for a signal to propagate through a flip-flop; it is also sometimes called clock-to-Q.
- $t_{combinational}$ is the longest delay for any combinational logic (which by definition is surrounded by two flip-flops).
- t_{setup} is the time before the rising clock edge that the input to a flip-flop must be valid.

Clock skew: The difference in absolute time between the times when two state elements see a clock edge.

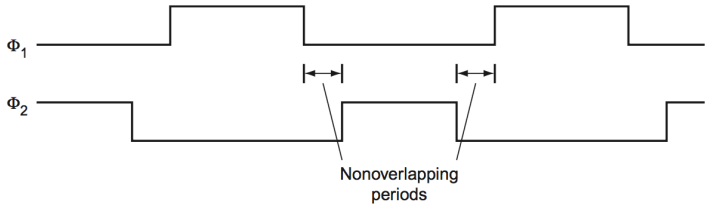


$t_{prop} + t_{combinational} + t_{setup} + t_{skew}$

level-sensitive clocking: A timing methodology in which state changes occur at either high or low clock levels but are not instantaneous as such changes are in edge-triggered designs.

Level-sensitive timing

Non-overlapping clock cycles: when the high/low status of both clocks are the same



Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

Asynchronous inputs and synchronizers

Synchronizer:

- changes async to sync
- inputs are the asynchronous signal and a clock and output is a signal synchronous with the input clock.

Metastability: A situation that occurs if a signal is sampled when it is not stable for the required setup and hold times, possibly causing the sampled value to fall in the indeterminate region between a high and low value.

Synchronizer failure: A situation in which a flip-flop enters a metastable state and where some logic blocks reading the output of the flip-flop see a 0 while others see a 1.

Note: the probability that the flip-flop will stay in the metastable state decreases exponentially, o it's never 0, and devs just estimate when it'll be very low

- **Scientific notation:** A notation that renders numbers with a single digit to the left of the decimal point.
- **Normalized:** A number in floating-point notation that has no leading 0s.
- **Floating point:** Computer arithmetic that represents numbers in which the binary point is not fixed.

Note: the format is as follows: $\#_{base} * base^{\#2}$. *Ex)* $1.1101_{two} * 2^{3}$ *or* $100.10 * 2^{-6}$

Floats can also be in the form: $(-1)^{S} * F * 2^{E}$

- **Fraction:** The value, generally between 0 and 1, placed in the fraction field. The fraction is also called the mantissa.
- **Exponent:** In the numerical representation system of floating-point arithmetic, the value that is placed in the exponent field.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	exponent								fraction																						
1 bit	8 bits								23 bits																						

- **Overflow (floating-point):** A situation in which a positive exponent becomes too large to fit in the exponent field.
- **Underflow (floating-point):** A situation in which a negative exponent becomes too large to fit in the exponent field.
- **Double precision:** A floating-point value represented in two 32-bit words.
- **Single precision:** A floating-point value represented in a single 32-bit word.

$(-1)^{S} * (1 + Fraction) * 2^{(exponent-bias)}$

For double precision, the bias is 1023

Float Addition:

To add two numbers, raise the lowest exponent to match the higher exponent, then add

ex) $9.99_{ten} * 10^{1} + 1.610_{ten} * 10^{-1} ==> 1.61 * 10^{-1} = 0.01610 * 10^{1} ==>$

$9.999_{ten} + 0.016_{ten} = 10.015 * 10^{1} ==> 1.0015 * 10^{2}$

Float Multiplication:

The new exponent is adding the two old exponents together

ex) $1.110_{ten} * 20^{10} * 9.200_{ten} * 10^{-5} ==> exp = 10 + (-5) = 5 ==>$

$1.110_{ten} * 9.200_{ten} = 10.212000_{ten} ==> 10.212000_{ten} * 10^{5}$ normalize/round: $1.0212_{ten} * 10^{6}$

MIPS Arithmetic Operations:

- Floating-point addition, single (add.s) and addition, double (add.d)
- Floating-point subtraction, single (sub.s) and subtraction, double (sub.d)
- Floating-point multiplication, single (mul.s) and multiplication, double (mul.d)
- Floating-point division, single (div.s) and division, double (div.d)
- Floating-point comparison, single (c.x.s) and comparison, double (c.x.d), where x may be equal (eq), not equal (neq), less than (lt), less than or equal (le), greater than (gt), or greater than or equal (ge)
- Floating-point branch, true (bc1t) and branch, false (bc1f)
- Load double (l.d)

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

- **Combinational element:** An operational element, such as an AND gate or an ALU.
- **State element:** A memory element, such as a register or a memory.
- **Clocking methodology:** The approach used to determine when data is valid and stable relative to the clock.
- **Edge-triggered clocking:** A clocking scheme in which all state changes occur on a clock edge.

Control signal: A signal used for multiplexor selection or for directing the operation of a functional unit; contrasts with a *data signal*, which contains information that is operated on by a functional unit.

- Asserted: The signal is logically high or true.
- Deasserted: The signal is logically low or false.

- **Datapath element:** A unit used to operate on or hold data within a processor. In the MIPS implementation, the datapath elements include the instruction and data memories, the register file, the ALU, and adders.
- **Program counter (PC):** The register containing the address of the next instruction in the program to be executed.
- **Register file:** A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed
- **Sign-extend:** To increase the size of a data item by replicating the high-order sign bit of the original data item in the high-order bits of the larger, destination data item
- **Sign-extend:** To increase the size of a data item by replicating the high-order sign bit of the original data item in the high-order bits of the larger, destination data item
- **Branch taken:** A branch where the branch condition is satisfied and the program counter (PC) becomes the branch target. All unconditional branches are taken branches.
- **Branch not taken or (untaken branch):** A branch where the branch condition is false and the program counter (PC) becomes the address of the instruction that sequentially follows the branch.

https://learn.zybooks.com/zybook/RPICSCI2500KuzminFall2022/chapter/10/section/1
http://www.cburch.com/logisim/index.html

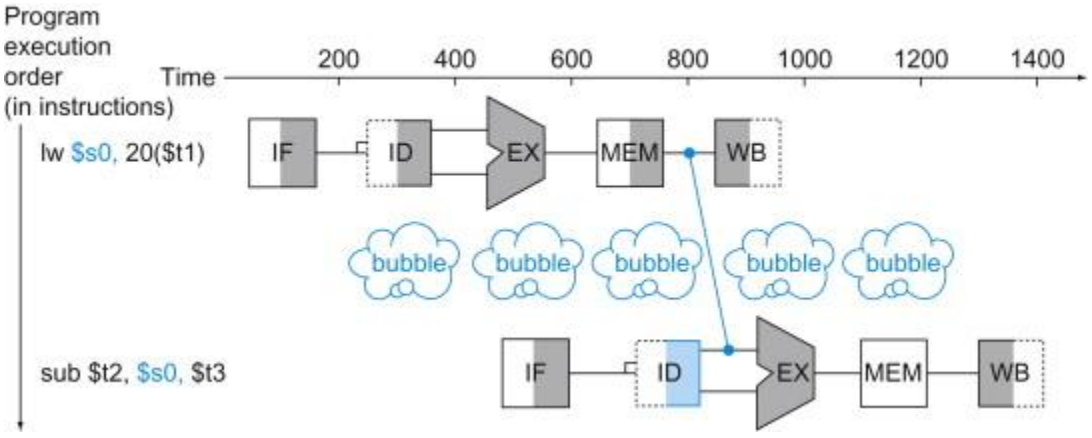
MIPS CHEAT SHEET

INSERT DATAPATH STUFF HERE

Pipelining: An implementation technique in which multiple instructions are overlapped in execution, much like an assembly line

Pipelining Hazards:

- **Structural hazard:** When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.
- **Data hazard:** Also called a pipeline data hazard. When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.
- **Forwarding:** Also called bypassing. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible registers or memory.
- **Load-use data hazard:** A specific form of data hazard in which the data being loaded by a load instruction has not yet become available when it is needed by another instruction.
- **Pipeline stall:** Also called bubble. A stall initiated in order to resolve a hazard.



Control hazard: Also called branch hazard. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

- **Latency (pipeline):** The number of stages in a pipeline or the number of stages between two instructions during execution.

Note: declaring a struct just declares the type and does NOT use any memory!

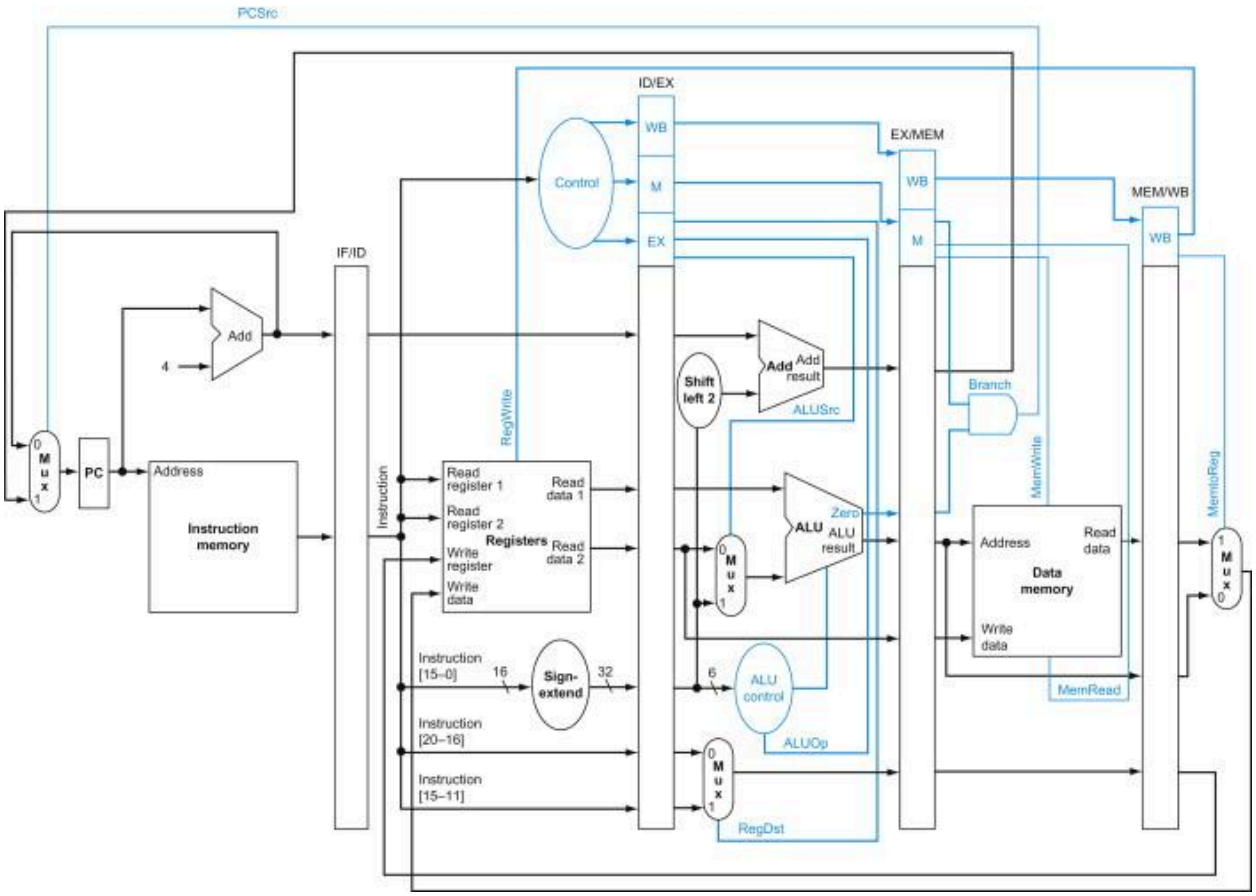
RETURN TO

Written to IF/ID register	CC 1 The write address is originally a field in the instruction, and is written into IF/ID during CC 1.
Written to ID/EX register	CC 2 The write address is read from IF/ID and passed along to the ID/EX register during CC 2.
Written to EX/MEM register	CC 3 The write address is read from ID/EX and passed along to the EX/MEM register during CC 3.
Written to MEM/WB register	CC 4 The write address is read from EX/MEM and passed along to the MEM/WB register during CC 4.
Used as the register file's write address	CC 5 The address is read from the MEM/WB register, and is then used to indicate where the data should be written in the register file, completing the load. Note that the address had to be passed along, unchanged, from one pipeline register to the next, so that the address would be available during stage 5 for the write to the register file.

Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

Note: A write instruction happens during the first half of the cycle and the read happens during the second

MIPS CHEAT SHEET



Data Hazard Register Conditions/Types:

- 1a: EX/MEM.RegisterRd = ID/EX.RegisterRs
- 1b: EX/MEM.RegisterRd = ID/EX.RegisterRt
- 2a: MEM/WB.RegisterRd = ID/EX.RegisterRs
- 2b: MEM/WB.RegisterRd = ID/EX.RegisterRt

Note: the format is (`dependant instruction) = (instruction it depends on)`

Register naming conventions?
https://web.archive.org/web/20121010181500/https://www.d.umn.edu/%7Egshute/spimsal/talref.html

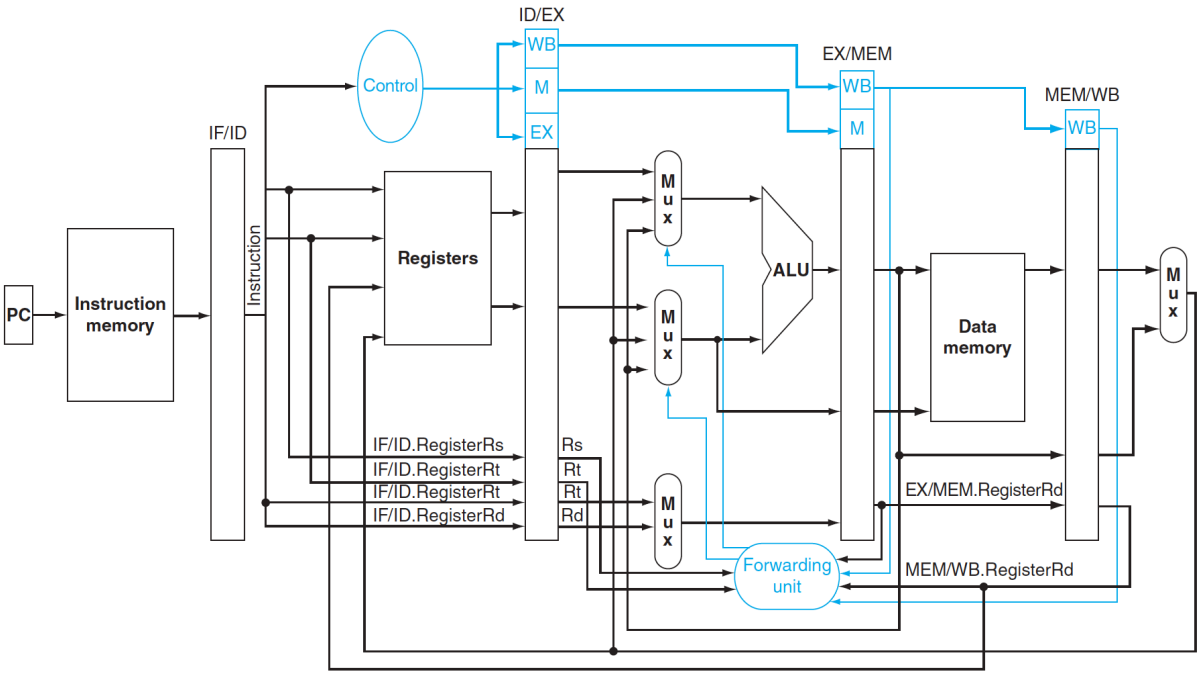
Alternative: https://en.wikipedia.org/wiki/MIPS_architecture#Versions

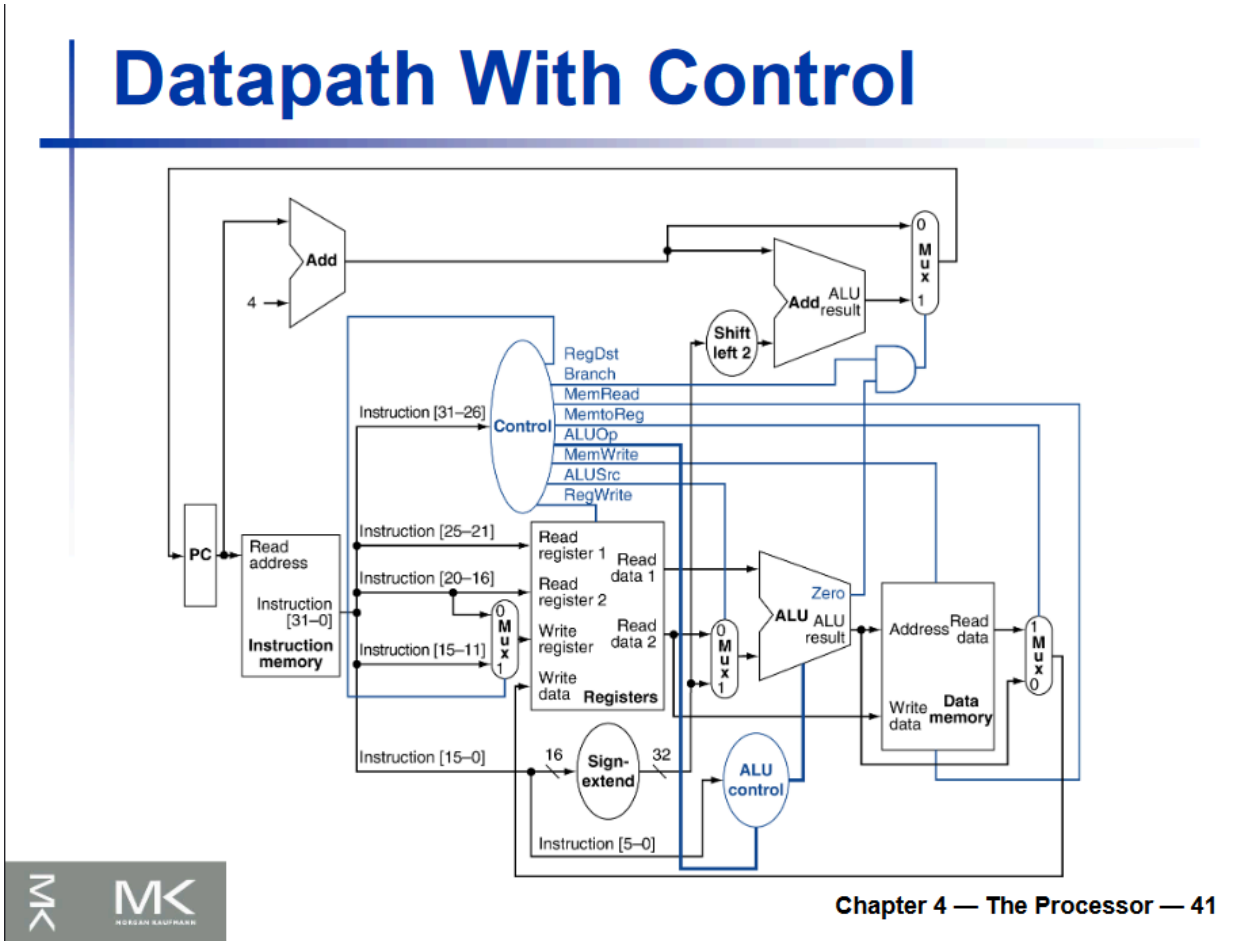
Rd: destination register

Rs: ?????????

Note: declaring a struct just declares the type and does NOT use any memory!

[RETURN TO]





Instruction Breakdown/Datapath Tutorial

- **nop:** An instruction that does no operation to change state.
- **Latency (pipeline):** The number of stages in a pipeline or the number of stages between two instructions during execution.

Flush: To discard instructions in a pipeline, usually due to an unexpected event.

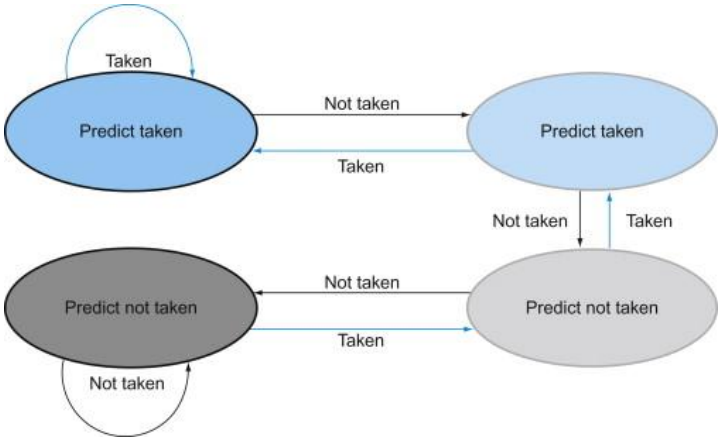
To discard instructions, we merely change the original control values to 0s, much as we did to stall for a load-use data hazard. The difference is that we must also change the three instructions in the IF, ID, and EX stages when the branch reaches the MEM stage; for load-use stalls, we just change control to 0 in the ID stage and let them percolate through the pipeline. Discarding instructions, then, means we must be able to flush instructions in the IF, ID, and EX stages of the pipeline.

- **Branch prediction:** A method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

Note: the standard approach is to assume all branches untaken, so it should proceed at max speed?

Note: A smart predictor uses branching history to make better predictions.

- **Dynamic branch prediction:** Prediction of branches at runtime using runtime information.
- **Branch prediction buffer:** Also called **branch history table**. A small memory that is indexed by the lower portion of the address of the branch instruction and that contains one or more bits indicating whether the branch was recently taken or not.



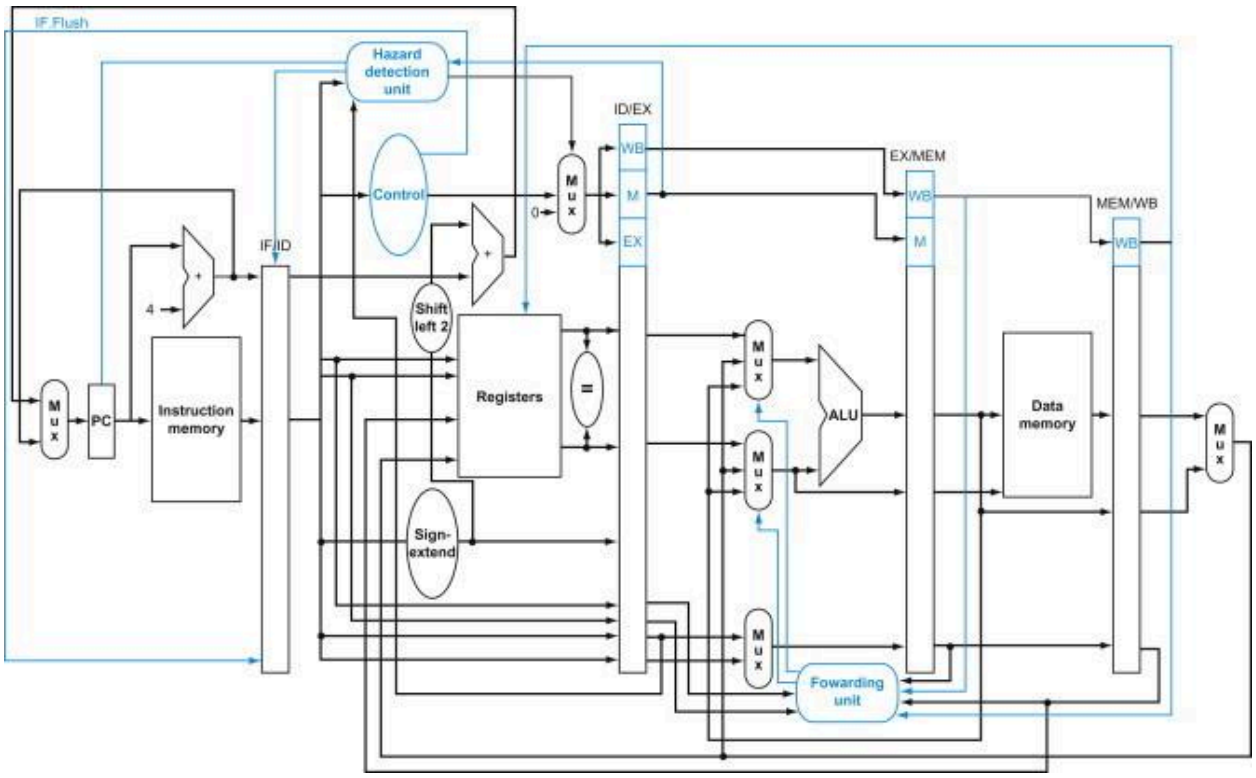
Two-Bit Prediction Scheme

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

- **Branch delay slot:** The slot directly after a delayed branch instruction, which in the MIPS architecture is filled by an instruction that does not affect the branch.
- **Branch target buffer:** A structure that caches the destination PC or destination instruction for a branch. It is usually organized as a cache with tags, making it more costly than a simple prediction buffer
- **Correlating predictor:** A branch predictor that combines local behavior of a particular branch and global information about the behavior of some recent number of executed branches.
- **Tournament branch predictor:** A branch predictor with multiple predictions for each branch and a selection mechanism that chooses which predictor to enable for a given branch.

Note: This is missing some components (see THIS LINK)



Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

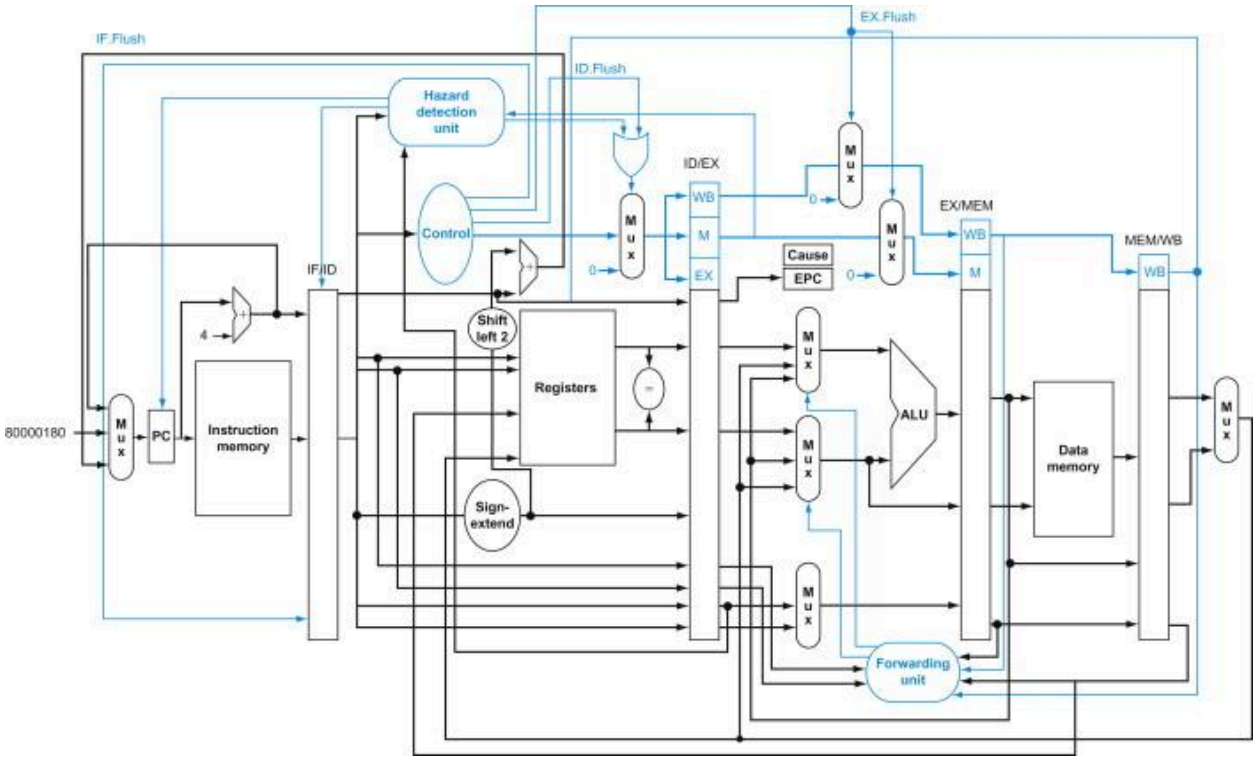
- **Exception:** Also called **interrupt**. An unscheduled event that disrupts program execution; used to detect overflow.
- **Interrupt:** An exception that comes from **outside** of the processor. (Some architectures use the term *interrupt* for all exceptions)
- **Vectored interrupt:** An interrupt for which the address to which control is transferred is determined by the cause of the exception.

Exception type	Exception vector address (in hex)
Undefined instruction	$8000\ 0000_{hex}$
Arithmetic overflow	$8000\ 0180_{hex}$

Note: The exception addresses are separated by 32 bytes (AKA eight instructions)
Note: When the exception is not vectored, a single entry point for all exceptions can be used, and the operating system decodes the status register to find the cause

If we're not using Vectored Interrupt, then we can use a single entry point and the OS decodes the error signal from there. To do this we need:

- **exception program counter** (EPC): A 32-bit register used to hold the address of the affected instruction. (Such a register is needed even when exceptions are vectored)
- **Cause:** A register used to record the cause of the exception. In the MIPS architecture, this register is 32 bits, although some bits are currently unused



RETURN TO

*Note: The current address **+ 4** is saved when an exception is triggered*

- **Imprecise interrupt:** Also called imprecise exception. Interrupts or exceptions in pipelined computers that are not associated with the exact instruction that was the cause of the interrupt or exception
- **Precise interrupt:** Also called precise exception. An interrupt or exception that is always associated with the correct instruction in pipelined computers

Note: MIPS processors interrupt the earliest instruction first

Note: Although MIPS uses the exception entry address 8000 0180_{hex} for almost all exceptions, it uses the address 8000 0000_{hex} to improve performance of the exception handler for TLB-miss exceptions

Pipelining exploits the potential parallelism among instructions. This parallelism is called **instruction-level parallelism (ILP)**

- **Instruction-level parallelism:** The parallelism among instructions
- **Multiple issue:** A scheme whereby multiple instructions are launched in one clock cycle
- **Static multiple issue:** An approach to implementing a multiple-issue processor where many decisions are made by the compiler before execution
- **Dynamic multiple issue:** An approach to implementing a multiple-issue processor where many decisions are made during execution by the processor
- **Issue slots:** The positions from which instructions could issue in a given clock cycle; by analogy, these correspond to positions at the starting blocks for a sprint

Speculation: An approach whereby the compiler or processor guesses the outcome of an instruction to remove it as a dependence in executing other instructions

		Second task	
		Read	**Write**
First task	**Read**	Read after read (RAR) No dependency	Write after read (WAR) Antidependency
	Write	Read after write (RAW) True dependency	Write after write (WAW) Output dependency

MIPS CHEAT SHEET

There are two primary and distinct responsibilities that must be dealt with in a multiple-issue pipeline:

1. **Packaging instructions into issue slots**: how does the processor determine how many instructions and which instructions can be issued in a given clock cycle? In most static issue processors, this process is at least partially handled by the compiler; in dynamic issue designs, it is normally dealt with at runtime by the processor, although the compiler will often have already tried to help improve the issue rate by placing the instructions in a beneficial order.
2. **Dealing with data and control hazards**: in static issue processors, the compiler handles some or all of the consequences of data and control hazards statically. In contrast, most dynamic issue processors attempt to alleviate at least some classes of hazards using hardware techniques operating at execution time.

- **Issue packet:** The set of instructions that issues together in one clock cycle; the packet may be determined statically by the compiler or dynamically by the processor
- **Very Long Instruction Word (VLIW):** A style of instruction set architecture that launches many operations that are defined to be independent in a single wide instruction, typically with many separate opcode fields
- **Use latency:** Number of clock cycles between a load instruction and an instruction that can use the result of the load without stalling the pipeline
- **Loop unrolling:** A technique to get more performance from loops that access arrays, in which multiple copies of the loop body are made and instructions from different iterations are scheduled together

- **Register renaming:** The renaming of registers by the compiler or hardware to remove antidependences
- **Antidependence:** Also called name dependence. An ordering forced by the reuse of a name, typically a register, rather than by a true dependence that carries a value between two instructions
- **Superscalar:** An advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle by selecting them during execution

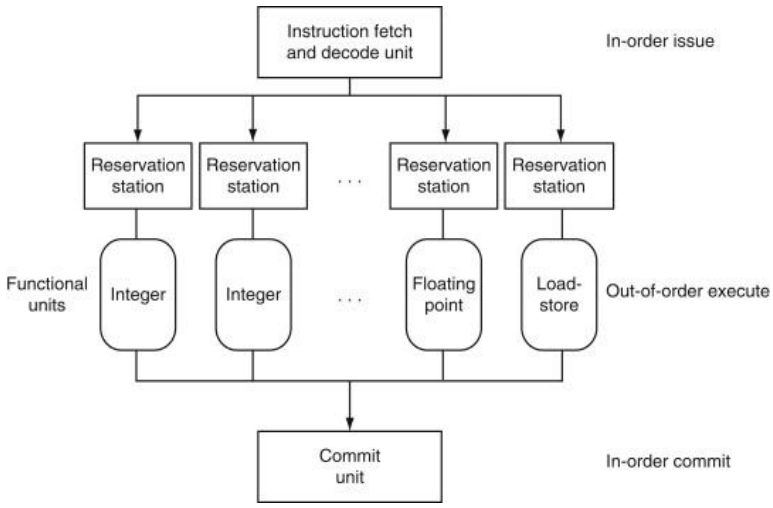
Note: Superscalar processors are also known as dynamic multiple issue processors. During runtime, superscalar processors can select and execute multiple instructions per clock cycle

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

- **Dynamic pipeline scheduling:** Hardware support for reordering the order of instruction execution so as to avoid stalls
- **Commit unit:** The unit in a dynamic or out-of-order execution pipeline that decides when it is safe to release the result of an operation to programmer-visible registers and memory
- **Reservation station:** A buffer within a functional unit that holds the operands and the operation
- **Reorder buffer:** The buffer that holds results in a dynamically scheduled processor until it is safe to store the results to memory or a register

The three primary units of a dynamically scheduled pipeline



- **Out-of-order execution:** A situation in pipelined execution when an instruction blocked from executing does not cause the following instructions to wait.
- **In-order commit:** A commit in which the results of pipelined execution are written to the programmer visible state in the same order that instructions are fetched

Note: To make programs behave as if they were running on a simple in-order pipeline, the instruction fetch and decode unit is required to issue instructions in order, which allows dependences to be tracked, and the commit unit is required to write results to registers and memory in program fetch order. This conservative mode is called in-order commit. Hence, if an exception occurs, the computer can point to the last instruction executed, and the only registers updated will be those written by instructions before the instruction causing the exception.

MIPS CHEAT SHEET

Although the front end (fetch and issue) and the back end (commit) of the pipeline run in order, the functional units are free to initiate execution whenever the data they need is available. Today, all dynamically scheduled pipelines use in-order commit.

UHHHHHHHHHHHHHHHHHHHH

Note: Unrolling nearly doubles performance. Optimizations for ***subword parallelism*** *and* ***instruction level parallelism*** *result in an overall speedup of 14.8 versus the DGEMM in COD Figure 3.21. Compared to the Python version in COD Chapter 1, it is 4600 times as fast* (see 12.8.1)

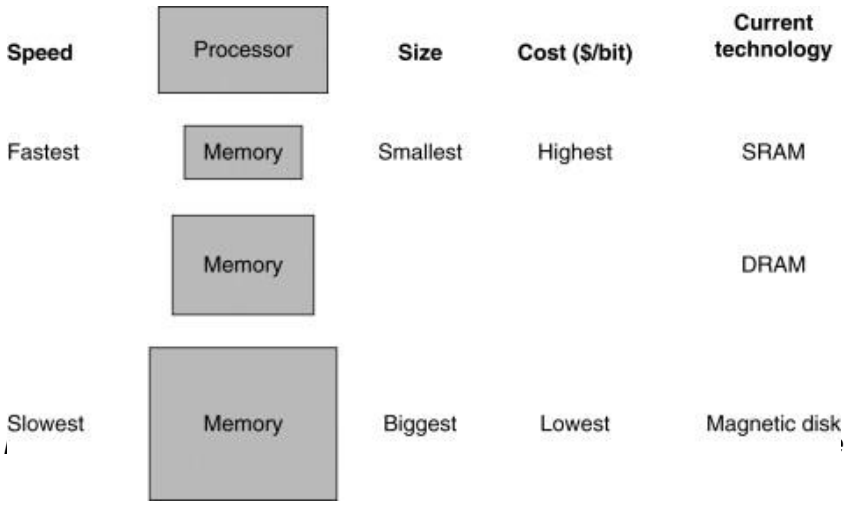
This shit

A link to instruction-level parallelism FAQ is HERE

- **Instruction latency:** The inherent execution time for an instruction

Capter 13 (oh god):

- **Temporal locality:** The locality principle stating that if a data location is referenced then it will tend to be referenced again soon.
- **Spatial locality:** The locality principle stating that if a data location is referenced, data locations with nearby addresses will tend to be referenced soon.
- **Memory hierarchy:** A structure that uses multiple levels of memories; as the distance from the processor increases, the size of the memories and the access time both increase while the cost per bit decreases



Basic Structure of a memory Hierarchy

any memory!

RETURN TO

- **Block (or line):** The minimum unit of information that can be either present or not present in a cache

Note: memory and data are stored with the smallest-first, as any smaller data is a "subset" of larger data below it (see THIS)????

Note: when copying data between levels, an entire block is usually copied (see THIS)

- **Hit rate:** The fraction of memory accesses found in a level of the memory hierarchy
- **Miss rate:** The fraction of memory accesses not found in a level of the memory hierarchy

Note: the `hit rate` can also be called the `hit ratio`. The `miss rate` is `1 - hit rate`

- **Hit time:** The time required to access a level of the memory hierarchy, including the time needed to determine whether the access is a hit or a miss
- **Miss penalty:** The time required to fetch a block into a level of the memory hierarchy from the lower level, including the time to access the block, transmit it from one level to the other, insert it in the level that experienced the miss, and then pass the block to the requestor

Summary: *Programs exhibit both temporal locality, the tendency to reuse recently accessed data items, and spatial locality, the tendency to reference data items that are close to other recently accessed items. Memory hierarchies take advantage of temporal locality by keeping more recently accessed data items closer to the processor. Memory hierarchies take advantage of spatial locality by moving blocks consisting of multiple contiguous words in memory to upper levels of the hierarchy. In most systems, the memory is a true hierarchy, meaning that data cannot be present in level i unless it is also present in level i + 1*

Memory technology	Typical access time	$ per GiB in 2020
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$3–$6
Flash semiconductor memory	5,000–50,000 ns	$0.06–$0.12
Magnetic disk	5,000,000–20,000,000 ns	$0.01–$0.02

How data is moved through memory

Memory Technologies:

MIPS CHEAT SHEET

SRAM:

- Data stored as an array with a single access port for read/write
- Has fixed time to access any datum, although read/write times may differ
- Doesn't need to refresh, so access time is very close to clock cycle time

DRAM:

- Data is stored as a charge in a capacitor
- Because DRAMs use only a single transistor per bit of storage, they are much denser and cheaper per bit than SRAM
- Because charge is stored on a capacitor, it must be periodically refreshed
    - Get around erasing all data by using a two-level decoding structure, AKA copying data to another row, refreshing the old row, then writing the data back

Internal Organization of DRAM:

- Organized into banks
    - Each bank is made up of rows
- Sending a PRE (precharge) command opens or closes a bank
- A row address is sent with an Act (activate), which transfers a row to a buffer
    - When in the buffer, a row can be transferred to successive column addresses at whatever the width of the DRAM is (usually 4,8,16 bits in DDR4), or by specifying a block transfer/starting address)
- *Note: each command, as well as block transfers, is synchronized with a clock*
- **SDRAM:** sychronized DRAM (has a clock in it), eliminates time when memory/processor syncronize

The fastest version is called Double Data Rate (DDR) SDRAM. The name means data transfers on both the rising and falling edge of the clock, thereby getting twice as much bandwidth as you might expect based on the clock rate and the data width. The latest version of this technology is called DDR4. A DDR4-3200 DRAM can do 3200 million transfers per second, which means it has a 1600-MHz clock.

- **Flash memory**: a type of *electrically erasable programmable read-only memory* (EEPROM).

Note: declaring a struct just declares the type and does NOT use any memory!

RETURN TO

- **Wear leveling:** The process by which a **controller** spreads writes by remapping blocks that have been written many times to less-trodden blocks in **flash memory** to avoid wear

- **Track:** One of thousands of concentric circles that make up the surface of a magnetic disk
- **Sector:** One of the segments that make up a track on a magnetic disk; a sector is the smallest amount of information that is read or written on a disk
- **Seek:** The process of positioning a read/write head over the proper track on a disk