Contents

[CHEAT SHEET 4](#_Toc525302171)

[Performance 4](#_Toc525302172)

[CPU Cycles & Instructions 4](#_Toc525302173)

[Slide 1: Intro 7](#_Toc525302174)

[1 – Computer Abstract & Technology 9](#_Toc525302175)

[1.1 – Introduction 9](#_Toc525302176)

[1.2 – 8 Great Ideas 9](#_Toc525302177)

[1.3 – Below Your Program 10](#_Toc525302178)

[1.4 – Under the Covers 11](#_Toc525302179)

[1.6 Performance 14](#_Toc525302180)

[The Clock 16](#_Toc525302181)

[CPU Performance and Its Factors 16](#_Toc525302182)

[Example 1.6.2 - Improving Performance. 17](#_Toc525302183)

[Instruction Performance 18](#_Toc525302184)

[Classic CPU Performance Equation 19](#_Toc525302185)

[1.7 – The Power Wall 20](#_Toc525302186)

[1.8 – The Sea Change: Switch from uniprocessors to multiprocessors 22](#_Toc525302187)

[2 - Instructions 23](#_Toc525302188)

[2.1 – Introduction 23](#_Toc525302189)

[2.2 – Sign and Unsigned Numbers 24](#_Toc525302190)

[2.6 Representing Instructions in the Computer 25](#_Toc525302191)

[Types of Instruction Formats 25](#_Toc525302192)

[Summary of Instruction Format 28](#_Toc525302193)

[2.7 – Logical Operations 29](#_Toc525302194)

[2.8 – Instructions for Making Decision 31](#_Toc525302195)

[2.9 – Addressing for Wide Immediates and Addresses 33](#_Toc525302196)

[Wide Immediate Operations 33](#_Toc525302197)

[Addressing in Branches 34](#_Toc525302198)

[Conditional Branching to far Away 35](#_Toc525302199)

[LEGv8 Addressing Mode Summary 36](#_Toc525302200)

[Decoding Machine Code 37](#_Toc525302201)

[LEGv8 Instruction Formats via Type 37](#_Toc525302202)

[HELP 38](#_Toc525302203)

[2.9 - Parallelism and Instructions: Synchronization 39](#_Toc525302204)

[Hardware Primitive Example: Used to Build Basic Synchronization Primitive. 39](#_Toc525302205)

[An advantage of the load/store exclusive mechanism is that it can be used to build other synchronization primitives, such as atomic compare and swap or atomic fetch-and-increment, which are used in some parallel programming models.2.11 – Translating and Starting a Program 40](#_Toc525302206)

[Compiler 41](#_Toc525302207)

[Assembler 41](#_Toc525302208)

[Linker 42](#_Toc525302209)

[Loader 44](#_Toc525302210)

[Dynamically Linked Libraries 45](#_Toc525302211)

[Java 46](#_Toc525302212)

[2.12 – Supporting Procedures in Computer Hardware 47](#_Toc525302213)

[Using More Registers 47](#_Toc525302214)

[Example – Compiling a C Procedure 48](#_Toc525302215)

[Nested Procedures 50](#_Toc525302216)

[Allocating Space for New Data on the Stack 50](#_Toc525302217)

[Allocating Space for Data on the Heap 50](#_Toc525302218)

[3 – More Arithmetic for Computers 51](#_Toc525302219)

[3.1 – Introduction 51](#_Toc525302220)

[3.2 – Multiplication 52](#_Toc525302221)

[3.3 – Division 53](#_Toc525302222)

[3.4 – Floating Point 56](#_Toc525302223)

[3.5 – Parallelism and Computer Arithmetic 57](#_Toc525302224)

[4 – The Processor 58](#_Toc525302225)

[4.1 – Introduction 58](#_Toc525302226)

[Basic Implimentation of LEGv8 Subset 59](#_Toc525302227)

[4.2 – Logic Design Conventions 60](#_Toc525302228)

[Clocking Methedology 60](#_Toc525302229)

[4.3 – Building a Datapath 62](#_Toc525302230)

[4.4 – Implimentation Scheme 63](#_Toc525302231)

# CHEAT SHEET

## Performance

Performance = 1 / execution time

## CPU Cycles & Instructions

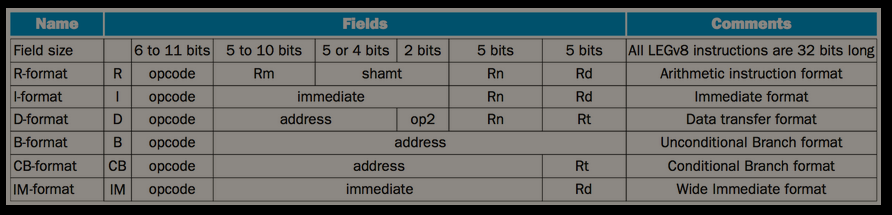
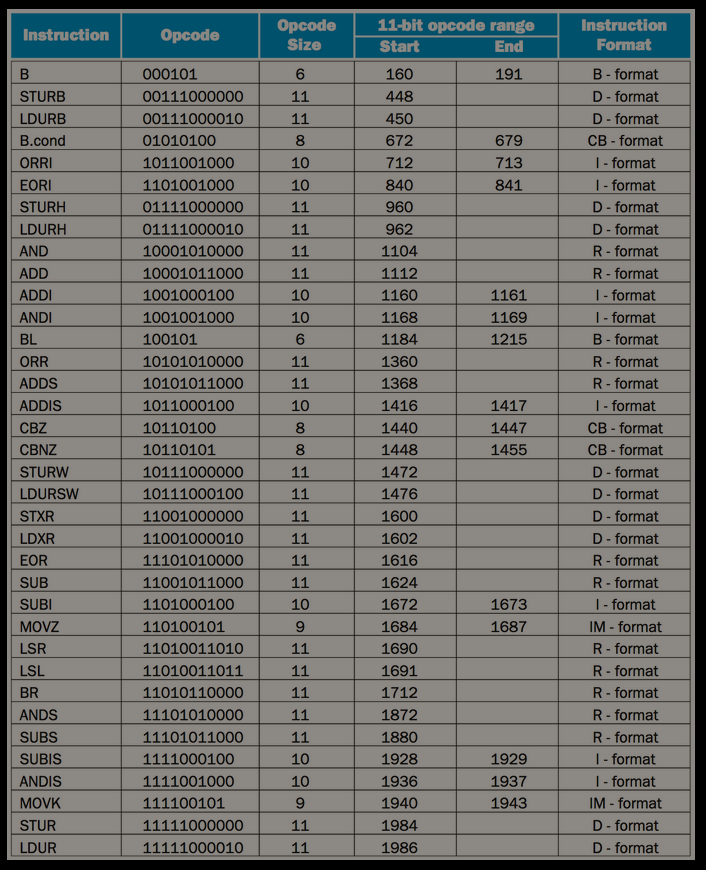
CPU execution time for a program = CPU clock cycles for a program x clock cycle time

CPU execution time for a program = CPU clock cycles for a program / clock rate

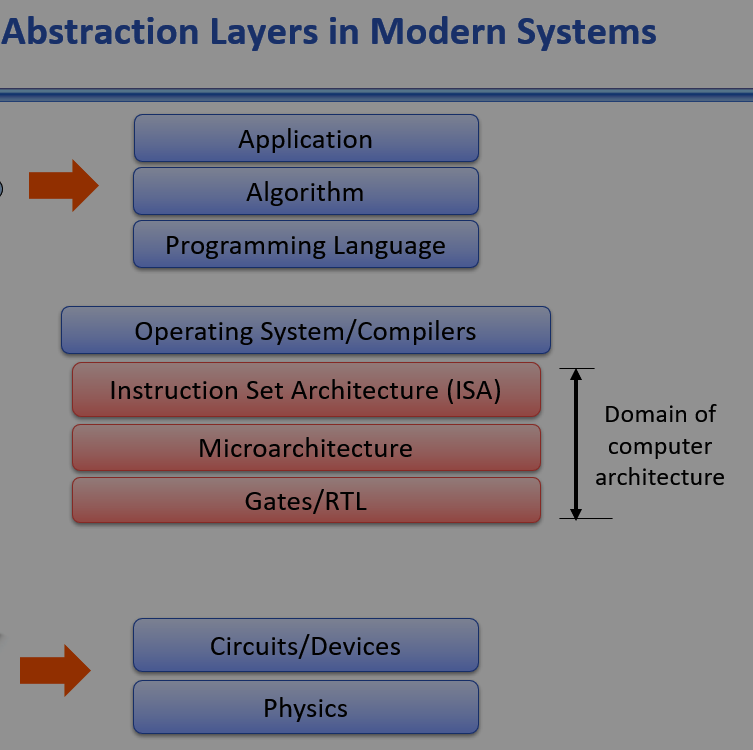
Execution time = number of instructions x average time per instruction

CPU clock cycles = # of instructions in program x average clock cycles per instruction

## Instruction Formats



# Slide 1: Intro



Phase 1: Mechanical Gears

Phase 2: Vacuum Tubes

ENIAC, EDVAC (Von Neumann), EDSAC (r/w memory).

Main issues: space, storage, reliability (tubes & memory), heat, power.

Phase 3: Transistors

Mid 50s to mid 60s.

Intro to high level languages.

Main Issues

Focus on ISA

Machines getting more complex.

Need to hide complexity from programmer.

Memory

Phase 4: Integrated Circuits and VLSI

Mid 60s to 2000s

Kilby and Noyce

Semiconductor memory

Parallel Architectures

Main issues

Memory capacity and speed (cache and VM)

Extensive ISAs

ILP: functional units, pipelining

# 1 – Computer Abstract & Technology

## 1.1 – Introduction

Taking over from the conventional server is Cloud Computing, which relies upon giant datacenters that are now known as Warehouse Scale Computers (WSCs). Companies like Amazon and Google build these WSCs containing 100,000 servers and then let companies rent portions of them so that they can provide software services to PMDs without having to build WSCs of their own. Indeed, Software as a Service (SaaS) deployed via the Cloud is revolutionizing the software industry just as PMDs and WSCs are revolutionizing the hardware industry. Today's software developers will often have a portion of their application that runs on the PMD and a portion that runs in the Cloud.

Programmers interested in performance now need to understand the issues that have replaced the simple memory model of the 1960s: the parallel nature of processors and the hierarchical nature of memories. We demonstrate the importance of this understanding in COD Chapters 3 (Arithmetic for Computers) to 6 (Parallel Processors from Client to Cloud) by showing how to improve performance of a C program by a factor of 200. Moreover, as we explain in COD Section 1.7 (The power wall), today's programmers need to worry about energy efficiency of their programs running either on the PMD or in the Cloud, which also requires understanding what is below your code. Programmers who seek to build competitive versions of software will therefore need to increase their knowledge of computer organization.

## 1.2 – 8 Great Ideas

- Design for Moore’s Law.

- Abstraction to simplify design.

- Common case fast.

- Parallelism.

- Prediction.

- Hierarchy of memories.

- Redundancy.

## 1.3 – Below Your Program

Systems software: Software that provides services that are commonly useful, including operating systems, compilers, loaders, and assemblers.

Operating system: Supervising program that manages the resources of a computer for the benefit of the programs that run on that computer.

Using the computer to help program the computer, the pioneers invented software to translate from symbolic notation to binary. The first of these programs was named an assembler. This program translates a symbolic version of an instruction into the binary version. For example, the programmer would write

ADD A,B

and the assembler would translate this notation into

1000110010100000

This instruction tells the computer to add the two numbers A and B. The name coined for this symbolic language, still used today, is assembly language. In contrast, the binary language that the machine understands is the machine language.

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.

## 1.4 – Under the Covers

The underlying hardware in any computer performs the same basic functions: inputting data, outputting data, processing data, and storing data.

The five classic components of a computer are input, output, memory, datapath, and control, with the last two sometimes combined and called the processor.

Liquid Crystal Displays

The LCD is not the source of light; instead, it controls the transmission of light. A typical LCD includes rod-shaped molecules in a liquid that form a twisting helix that bends light entering the display. The rods straighten out when a current is applied and no longer bend the light. Since the liquid crystal material is between two screens polarized at 90 degrees, the light cannot pass through unless it is bent. Today, most LCD displays use an active matrix that has a tiny transistor switch at each pixel to control current precisely and make sharper images.

Liquid crystal display: A display technology using a thin layer of liquid polymers that can be used to transmit or block light according to whether a charge is applied.

Active matrix display: A liquid crystal display using a transistor to control the transmission of light at each individual pixel.

Pixel: The smallest individual picture element. Screens are composed of hundreds of thousands to millions of pixels, organized in a matrix.

The computer hardware support for graphics consists mainly of a raster refresh buffer, or frame buffer, to store the bit map. The image to be represented onscreen is stored in the frame buffer, and the bit pattern per pixel is read out to the graphics display at the refresh rate.

Integrated circuits (chips): device that combines dozens to millions of transistors.

Central processor unit (CPU): Also called processor. The active part of the computer, which contains the datapath and control and which adds numbers, tests numbers, signals I/O devices to activate, and so on.

It is the active part of the computer, following the instructions of a program to the letter. It adds numbers, tests numbers, signals I/O devices to activate, and so on. Occasionally, people call the processor the CPU, for the more bureaucratic-sounding central processor unit.

Datapath: The component of the processor that performs arithmetic operations.

Control: The component of the processor that commands the datapath, memory, and I/O devices according to the instructions of the program.

In contrast to sequential access memories, such as magnetic tapes, the RAM portion of the term DRAM means that memory accesses take basically the same amount of time no matter what portion of the memory is read.

Cache memory: A small, fast memory that acts as a buffer for a slower, larger memory.

Static random access memory (SRAM): Also memory built as an integrated circuit, but faster and less dense than DRAM.

One of the most important abstractions is the interface between the hardware and the lowest-level software. Because of its importance, it is given a special name: the instruction set architecture, or simply architecture, of a computer.

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.

Implementation: Hardware that obeys the architecture abstraction.

The Big Picture: Both hardware and software consist of hierarchical layers using abstraction, with each lower layer hiding details from the level above. One key interface between the levels of abstraction is the instruction set architecture—the interface between the hardware and low-level software. This abstract interface enables many implementations of varying cost and performance to run identical software.

Memory – A Safe Place for Data

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.

Because of their size and form factor, PMDs use flash memory, a nonvolatile semiconductor memory, instead of disks. Flash memory is slower than DRAM, but it is cheaper in addition to being nonvolatile. Although it costs more per bit than disks, its smaller, comes in smaller capacities, more rugged, and more power-efficient than disks. Therefore flash memory is standard for PMDs.

Unlike disks and DRAM, flash memory bits wear out after 100,000 – 1,000,000 writes.

Thus files systems keep track of the number of writes.

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 2012 was $0.05 to $0.10.

Flash memory: A nonvolatile semiconductor memory. It is cheaper and slower than DRAM but more expensive per bit and faster than magnetic disks. Access times are about 5 to 50 microseconds and cost per gigabyte in 2012 was $0.75 to $1.00.

## 1.6 Performance

To measure performance, we must first define performance.

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.

Throughput: Also called bandwidth. Another measure of performance, it is the number of tasks completed per unit time.

Performance: bigger is better!

We know A is n times as fast as B if…

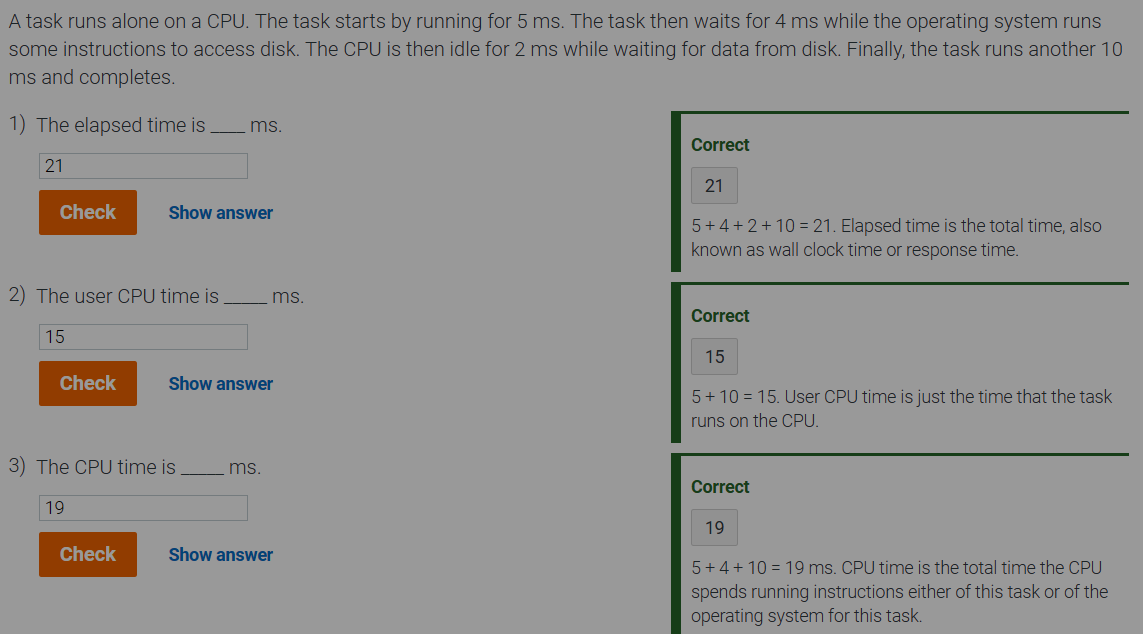
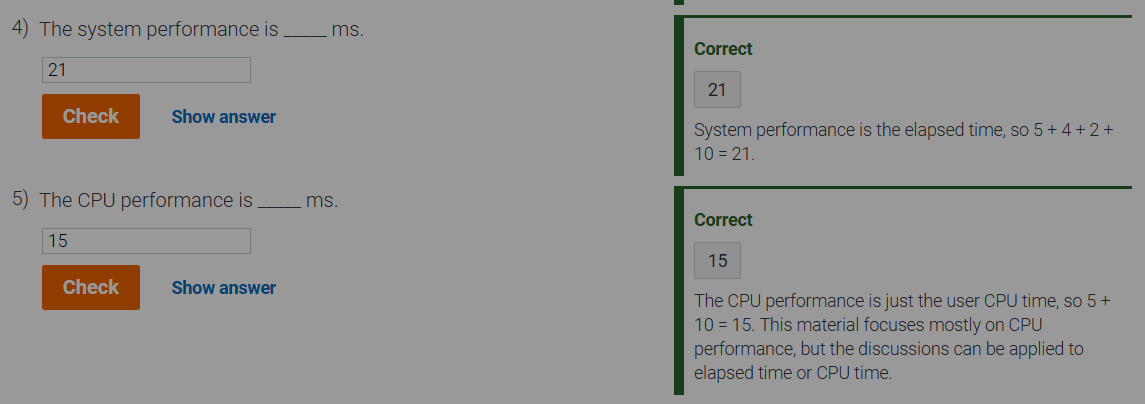
CPU execution time: Also called CPU time. The actual time the CPU spends computing for a specific task.

User CPU time: The CPU time spent in a *program* itself.

System CPU time (CPU performance): The CPU time spent in the *operating system* performing tasks on behalf of the program.

CPU performance (system CPU time): user CPU time. Time spent in a *program* itself.

System performance: elapsed time (all inclusive – start to finish) on an unloaded system.



### The Clock

Clock cycle: *Also called tick, clock tick, clock period, clock, or cycle*. The time for one clock period, usually of the processor clock, which runs at a constant rate.

Clock period: The length of each clock cycle. Equal to inverse of clock rate.

1GHz = 1 billion cycles per second.

1 ns = one billionth of a second.

Clock period = inverse of clock rate.

Clock rate = 1 billion cycles/second = 10^9 = 1 GHz

Clock period = ( 1 / (1 billion cycles/second) ) = 1 / 1 billion = 10^-9 = 1 ns

### CPU Performance and Its Factors

Recall: clock rate = inverse of clock period.

CPU execution time for a program = CPU clock cycles for a program x clock cycle time

CPU execution time for a program = CPU clock cycles for a program / clock rate

This formula makes it clear that a hardware designer can improve performance by reducing the number of clock cycles required for a program… or the length of a clock cycle (clock cycle time).

### Example 1.6.2 - Improving Performance.

Computer A can run a program in 10 seconds with a 2 GHz clock. We are helping a computer designer build a computer, B, that will run this program in 6 seconds. The designer has determined that a substantial increase in clock rate is possible, but this affects the rest of the CPU design, causing computer B to require 1.2 times as many clock cycles as computer A for this program.

What clock rate should we tell the designer to target?

First, we find the number of clock cycles required for A to complete the program.

CPU Time A = CPU clock cycles for a program / clock rate

10 sec = CPU clocks cycles for program(A) / 2 \* 109 (cycles/sec)

10 sec = clockCycles(A) / 2 \* 109 (cycles/sec)

clockCycles(A) = 10 sec \* 2 \* 109 (cycles/sec)

clockCycles(A) = 20 \* 109 cycles

Now, we use this to find the desired clock rate for B.

CPU Time B = CPU clock cycles for a program / clock rate

6 sec = 1.2 \* clockCycles(A) / clock rate

6 sec = 1.2 \* 20 \* 109 cycles/sec / clockRate(B)

clockRate(B) = 24 \* 109 cycles/sec / 6 sec

clockRate(B) = 4 \* 109 cycles / sec = 4 GHz

Therefore the clock rate of B to finish the program in 6 seconds is **4 GHz**, or twice the clock rate of A.

### Instruction Performance

The compiler generated instructions to execute, the computer had to execute the instructions to run the program. Therefore the execution time must depend on the number of instructions in a program.

Execution time = number of instructions x average time per instruction

CPU clock cycles = # of instructions in program x average clock cycles per instruction

CPI (Clock Cycles Per Instruction): average number of clock cycles each instruction takes to execute.

Example 1.6.3) Using Performance Equation

Suppose we have two different implementations of the same ISA. Computer A has a clock cycle time of 250 ps and a CPI of 2.0. Computer B has a clock cycle time of 500 ps and a CPI of 1.2 for the same program.

Which computer is faster, and by how much?

First, we find the total CPU clock cycles required by each computer to complete the program.

totalClockCycles(A) = I x CPI(A) = I x 2.0

totalClockCycles(B) = I x CPI(B) = I x 1.2

Then, we compute the CPU time for each computer.

cpuTime(A) = totalClockCycles(A) x cycleTime(A)

= I x 2.0 x 250 ps = I x 500 ps

cpuTime(B) = totalClockCycles(B) x cycleTime(B)

= I x 1.2 x 500 ps = I x 600 ps

Clearly A is faster. But by how much?

n = ( cpuPerf(A) / cpuPerf(B) ) = ( execTime(B) / execTime(A) ) =

= ( I x 600 ps ) / ( I x 500 ps) = 1.2

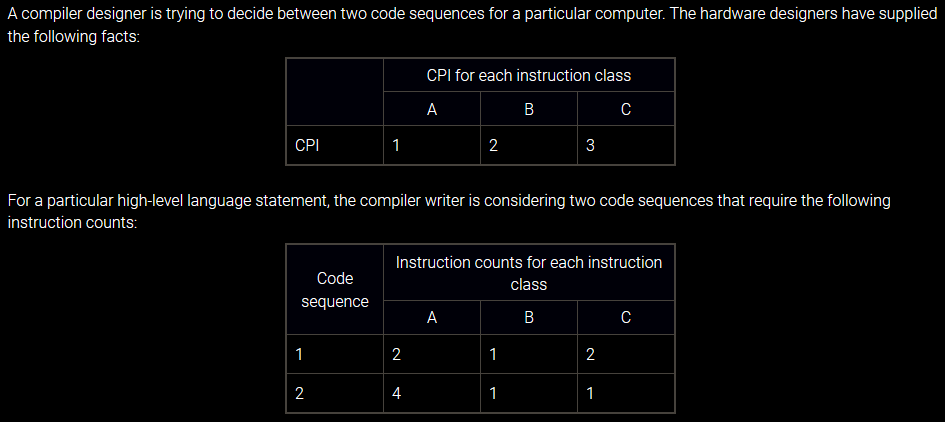
Computer A is 1.2 times as fast as computer B for this program.

### Classic CPU Performance Equation

CPU time = Instruction count x CPI x Clock cycle time

or

CPU time = Instruction count x CPI / clock rate



Which code sequence is faster? What is the CPI for each sequence?

Sequence 1 has a total of ( 2 + 1 + 2 ) = 5 instructions.

Total clock cycles for S1 = ( 2 x 1 ) + ( 1 x 2 ) + ( 2 x 3 ) = 10 cycles

Sequence 2 has a total of ( 4 + 1 + 1 ) = 6 instructions.

Total clock cycles for S2 = ( 4 x 1 ) + ( 1 x 2 ) + ( 1 x 3 ) = 9 cycles

So sequence 2 is faster – completing the same amount of work in fewer cycles. But what is the CPI for each sequence?

CPI = CPU clock cycles / Instruction count

Sequence 1 CPI = 10 cycles / 5 instructions = 2.0

Sequence 2 CPI = 9 cycles / 6 instructions = 1.5

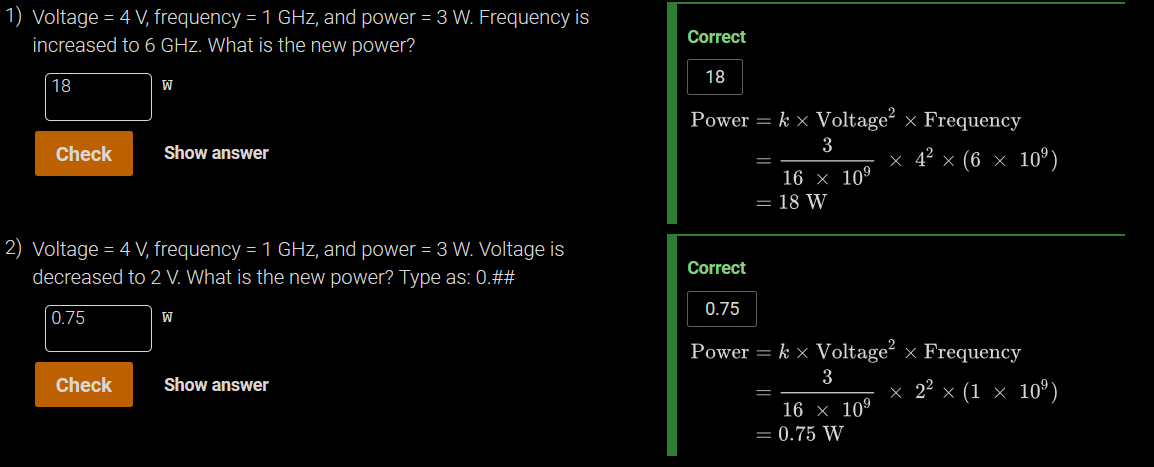
## 1.7 – The Power Wall

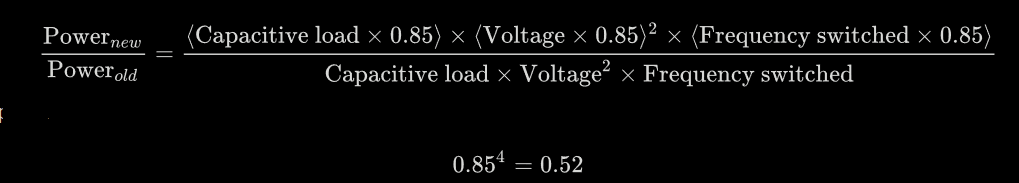
Complementary Metal Oxide Semiconductor (CMOS): dominant tech for integrated circuits. Primary source of energy is dynamic energy.

Energy = capLoad x volt2 ( full trans: 0 -> 1 -> 0 )

Energy = capLoad x volt2 ( one trans: 0 -> 1 )

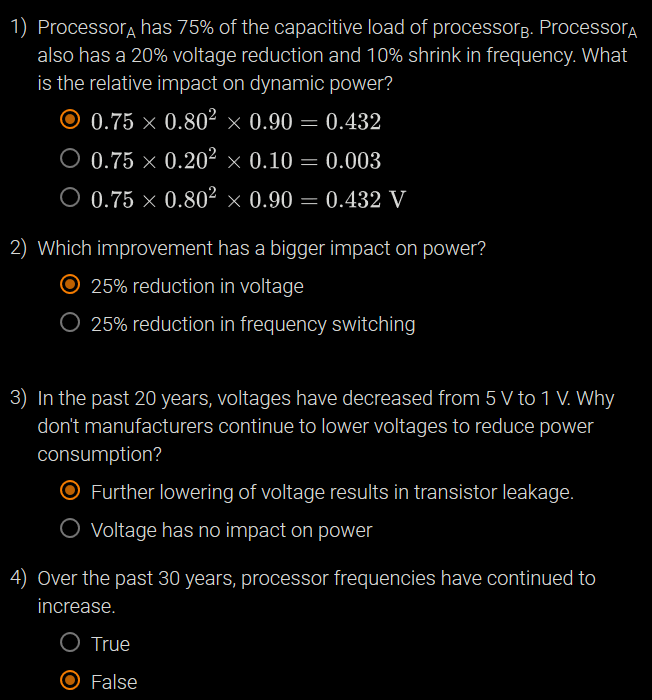
Power per transistor is just the product of energy of a transition and the frequency of transitions:

Power = capLoad x volt2 x freq

1.7.1 – Relative Power

Lower the voltage too much results in leaky transistors. About 40% of power consumption in servers. To address problem, designers have attached large devices to increase cooling, and turn off parts of chips not used during given clock cycle.

Increasing number of transistors increases power dissipation, even if transistors are off.



## 1.8 – The Sea Change: Switch from uniprocessors to multiprocessors

# 2 - Instructions

## 2.1 – Introduction

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

The language of computers are similar to one another. They all must provide the same type of services. Chosen instruction set is ARMv8. We are using a subset for learning is LEGv8.

To demonstrate how easy it is to pick up other instruction sets, we will take a quick look at three other popular instruction sets.

1. MIPS is an elegant example of the instruction sets designed since the 1980s.
2. ARMv7 is an older instruction set also from ARM Holdings plc, but with 32-bit addresses instead of ARMv8's 64 bits. More than 14 billion chips with ARM processors were manufactured in 2015, making them the most popular instruction sets in the world. Ironically, in the authors' opinion, and as shall be seen, ARMv8 is closer to MIPS than it is to ARMv7.
3. The final example is the Intel x86, which powers both the PC and the Cloud of the post-PC era.

## 2.2 – Sign and Unsigned Numbers

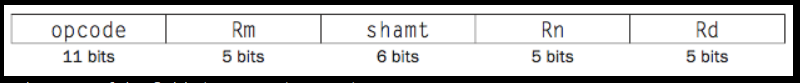
## 2.6 Representing Instructions in the Computer



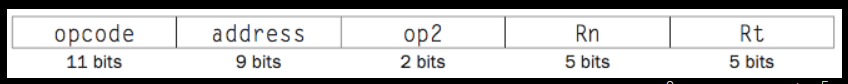
* *opcode* : Basic operation of the instruction, and this abbreviation is its traditional name.
* *Rm*: The second register source operand.
* *shamt*: Shift amount.
* *Rn*: The first register source operand.
* *Rd/Rt*: The register destination operand. It gets the result of the operation.

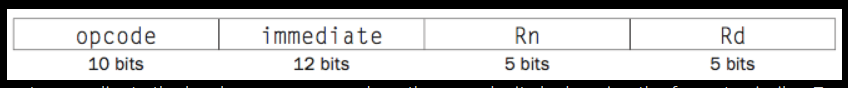
### Types of Instruction Formats

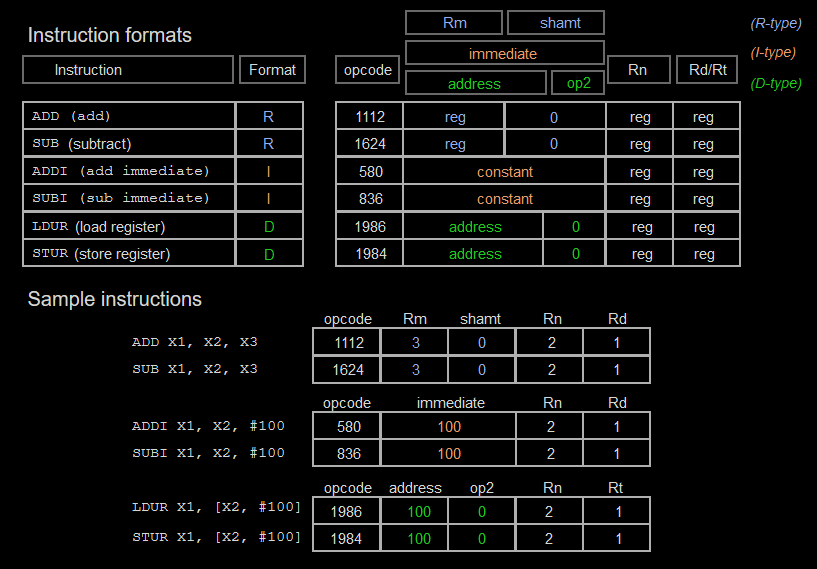
R-Format or Register Format

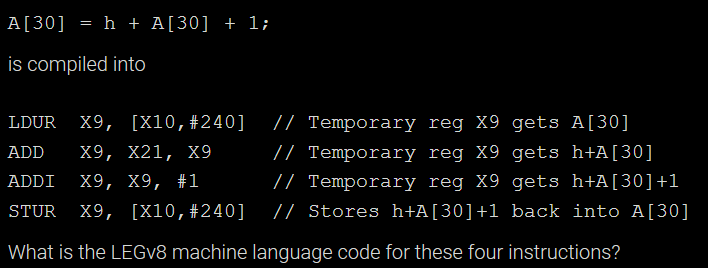


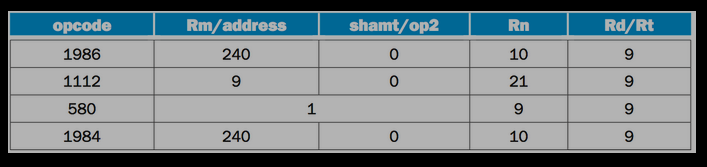
D-Format or Data-Transfer Format

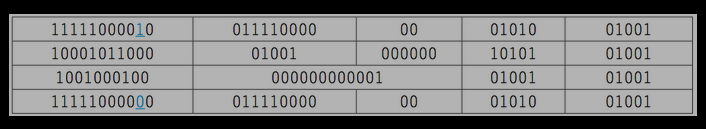


I-Format or Immediate Format









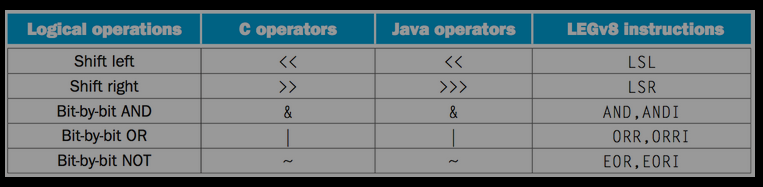
We only use ADDI instead of ADD for teaching purposes. Not required for ARMv8.

Unlike MIPS, LEGv8 immediate field in I-format is zero-extended. Thus, LEGv8 includes both ADDI and SUBI instructions, while MIPS has just ADDI and both positive and negative immediates.

Any increase in the number of registers would result in an increase in instruction size.

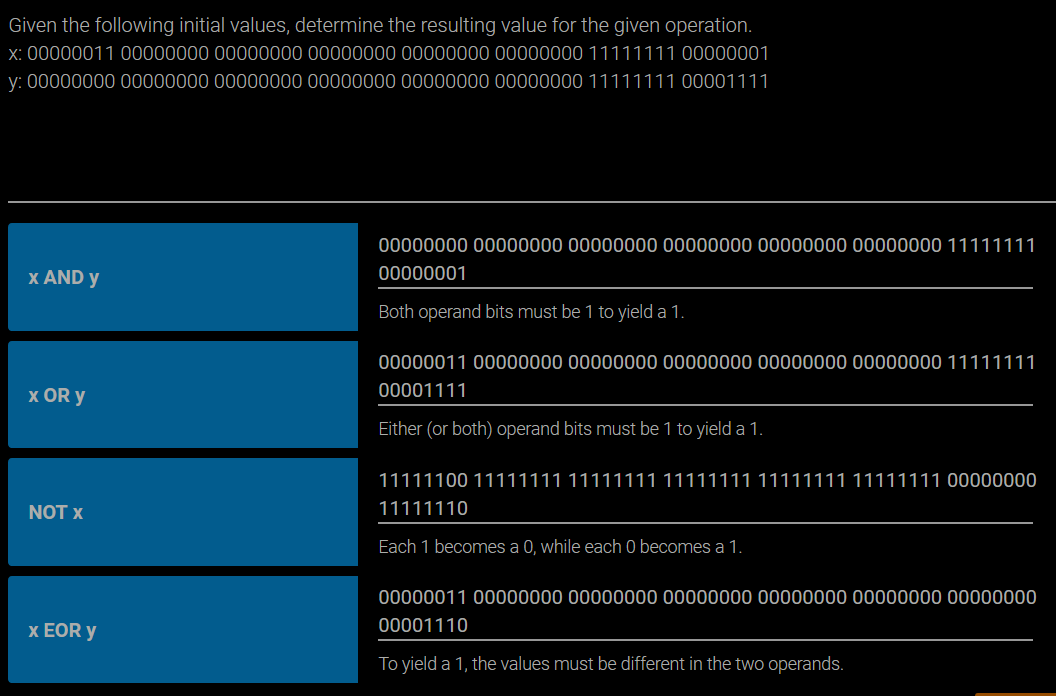
### Summary of Instruction Format

## 2.7 – Logical Operations



Shift allows you to divide/multiply by multiples of 2n.

LSL X11, X19, #4



Unlike almost all other computer architectures, ARMv8 (and ARMv7) allows a register to be shifted as part of an arithmetic or logical instruction: add an optionally shifted register, subtract an optionally shifted register, AND an optionally shifted register, and so on. Since this combination is unusual in computer architectures and not frequently generated by compilers--and since supporting it would make the data path in COD Chapter 4 (The Processor) much more complicated and unlike of other computer datapaths--we decided to treat shifts as separate instructions, as it is in virtually every other computer architecture.

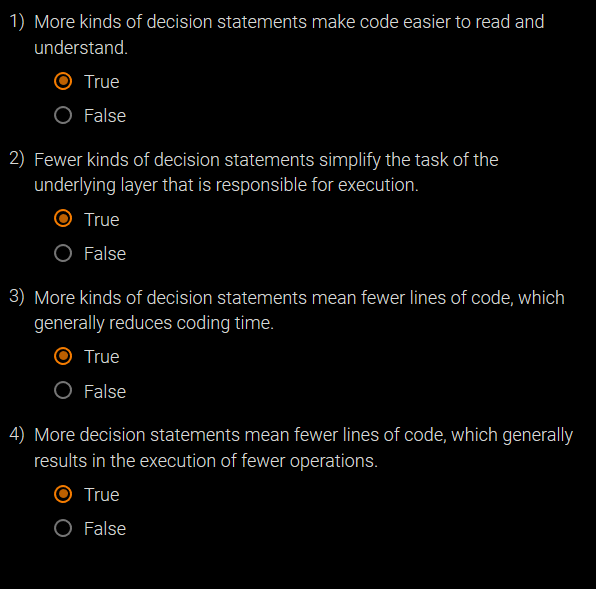
While you can synthesize a shift using either ADD with XZR or OR with XZR, it would be confusing to use the same opcode for shifts as ADD or OR. Thus, we follow the ARMv8 recommendation of using UBFM (unsigned bitfield move) instruction and its opcode.

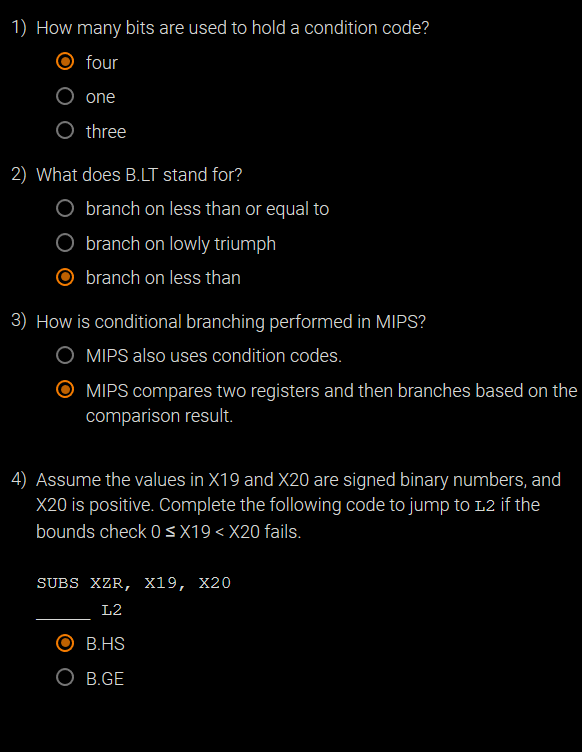
## 2.8 – Instructions for Making Decision

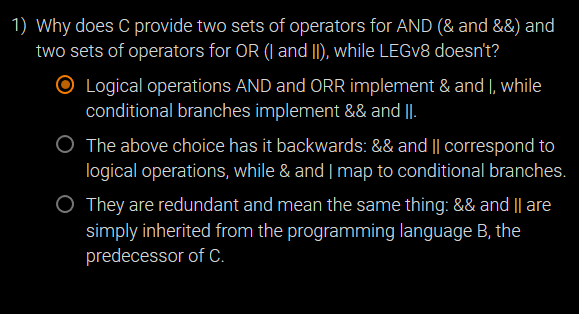
Architects long ago figured out how to handle all these cases by keeping just four extra bits that record what occurred during an instruction. These four added bits, called *condition codes* or *flags* are named:

* *negative (N)* - the result that set the condition code had a 1 in the most significant bit;
* *zero (Z)* - the result that set the condition code was 0;
* *overflow (V)* - the result that set the condition code overflowed; and
* *carry (C)* - the result that set the condition code had a carry out of the most significant bit or a borrow into the most significant bit.

Bounds Check Shortcut

Treating signed numbers as if they were unsigned gives us a low-cost way of checking if 0 ≤ x < y, which matches the index out-of-bounds check for arrays. The key is that negative integers in two's complement notation look like large numbers in unsigned notation; that is, the most significant bit is a sign bit in the former notation but a large part of the number in the latter. Thus, an unsigned comparison of x < y checks if x is negative as well as if x is less than y.

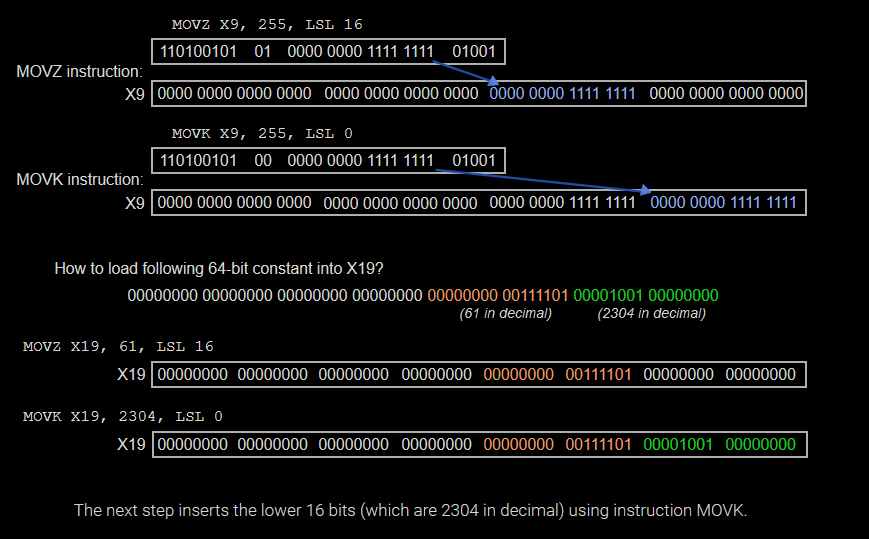




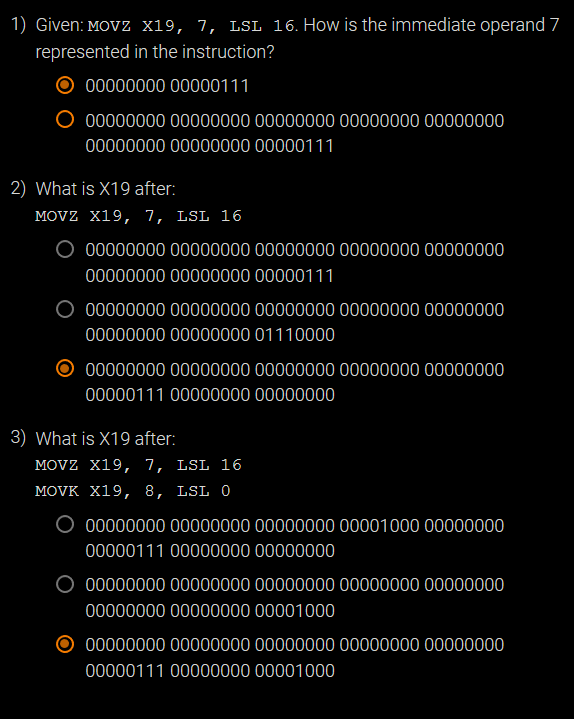
## 2.9 – Addressing for Wide Immediates and Addresses

### Wide Immediate Operations

Allow a 32b constant to be created from 2 32b instructions.

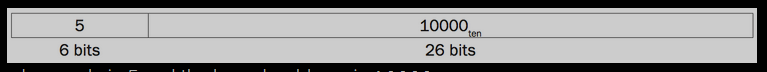


Either the compiler or the assembler must break large constants into pieces and then reassemble them into a register. As you might expect, the immediate field's size restriction may be a problem for memory addresses in loads and stores as well as for constants in immediate instructions.



### Addressing in Branches

B-Type Instruction: can contain addresses from 0 -> 226 – 1.



CB-Type Instruction: only 19 b for address. Presents a problem that limits the size of a program to 219 words.

Problem: absolute addressing has a limited range.

Solution: relative addressing.

Conditional branches are found in loops and if statements.

Program counter = register + constant (branch offest)

Since PC contains the address of the current isntruction, we can branch within ± 218 words.

PC-relative addressing: An addressing regime in which the address is the sum of the program counter (PC) and a constant in the instruction.

Bc all instructions are 4 words long, LEGv8 stretches a branch by having PC relative addressing refer to number of words to the next instruction, not bytes.

#### LEGv8 uses PCRA for all conditional branches.

On the other hand, branch-and-link instructions invoke procedures that have no reason to be near the call, so they normally use other forms of addressing.

### Conditional Branching to far Away

Although most conditional branches are to a nearby location, sometimes they branch far away, farther than what can be represented in the conditional branch instruction.

In the same way the assembler triumphed with large addresses or constants: it inserts an unconditional branch to the branch target, and inverst the condition.

CBZ X19, L1

*changes to…*

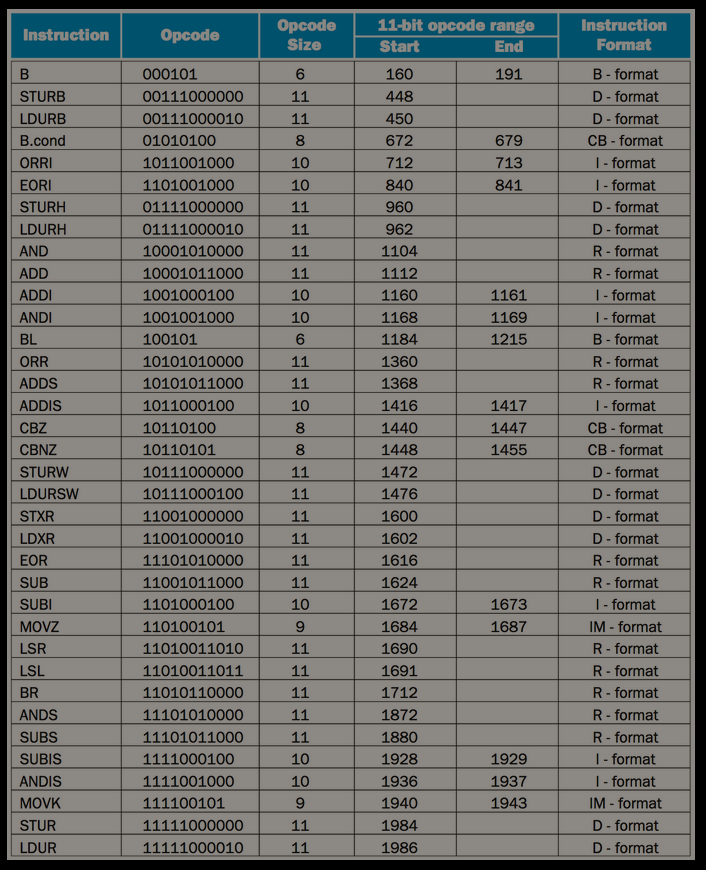
CBNZ X19, L2

B L1

L2:

### LEGv8 Addressing Mode Summary

1. Immediate addressing: The operand is a constant within the instruction itself
2. Register addressing: The operand is a register
3. Base addressing / displacement addressing: The operand is at the memory location whose address is the sum of a register and a constant in the instruction
4. PC-relative addressing: The branch address is the sum of the PC and a constant in the instruction .



### Decoding Machine Code

1) Take first 11 bits.

2) Based on their range or specific value, find the opcode.

3) Use that opcode to lookup operation.

Translate the following to assembley language:

8b0f00130hex

First 11 b:

10001011000 or 1112 or ADD

Therefore:

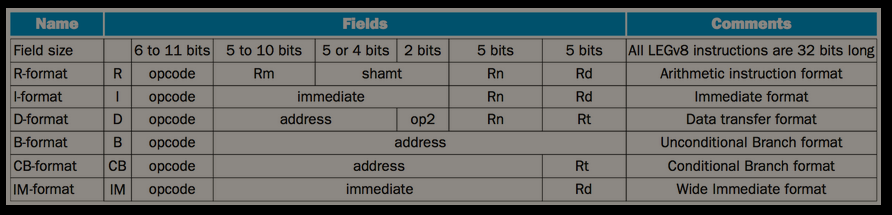
op Rm shamt Rn Rd

10001011000 00101 000000 01111 10000

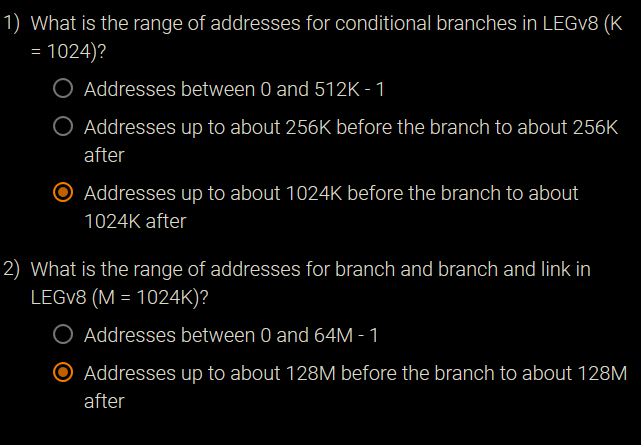
Finally:

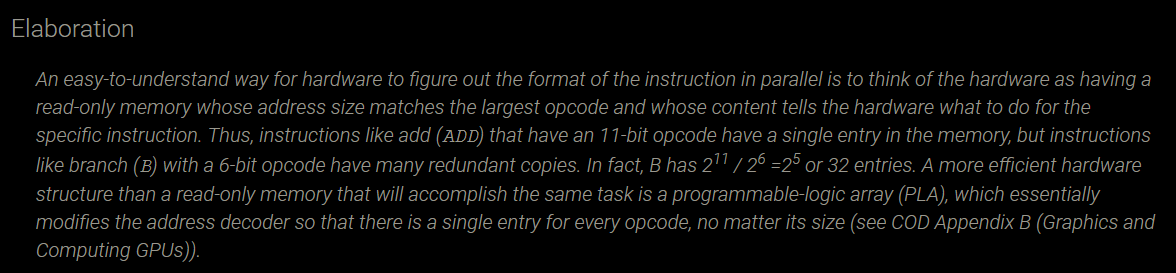
ADD X16, X15, X5

### LEGv8 Instruction Formats via Type



### HELP





## 2.9 - Parallelism and Instructions: Synchronization

Task cooperation: some tasks are writing values that other tasks must read.

Data race: Two memory accesses form a data race if they are from different threads to same location, at least one is a write, and they occur one after another.

In computing, synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions.

Ability required to implement synchronization: atomically read and modify a memory location. Nothing else can insert itself between the read and write of a memory location.

Architects don’t expect users to employ hardware primitives, but instead expect system programmers to use them to build a synchronization library.

### Hardware Primitive Example: Used to Build Basic Synchronization Primitive.

Atomic Exchange or Atomic Swap: interchanges a value in a register with a value in memory. This is a tool. Below is one method of how the tool can be used.

Imagine a lock where 0 is free, and 1 is locked.

A processor attempts to set the lock by exchanging 1 (in a register) with the memory address corresponding to the lock.

If value returned by exchange is 1, some other processor is using it. If 0, then its unused. If the latter, then the value in memory is changed to 1, locking out other processors.

The key here is the operation is that it is atomic – it is indivisible. It is impossible for two or more processors to set the synchronization variable (held in memory) such that they both think they’ve set the variable.

Atomic operations are tricky in that they require both a memory read and a write in a **single uninterruptible** instruction.

Alternative: a pair of instructions wherein the second instruction returns a value confirming whether or not the execution was atomic.

In LEGv8 this pair of instructions includes a special load called:

Store exclusive register (STXR)

Register 1: Stores the value of a (presumably different) register to memory.

Register 2: Changes another register to 0 for success, 1 to failure.

Register 3: Address of memory location.

Load exclusive register (LDXR)

Return values: 0 for success, 1 for failure.

These instructions are used in sequence: if the contents of the memory location specified by the load exclusive are changed before the store exclusive to the same address occurs, then the store exclusive fails and does not write the value to memory.

again: LDXR X10,[X20,#0] // load exclusive

STXR X23, X9, [X20] // store exclusive

CBNZ X9, again // branch if store fails

ADD X23, XZR, X10 // put loaded value in X23

Any time a processor intervenes and modifies the value in memory between the LDXR and STXR instructions, the STXR returns 1 in X9, causing the code sequence to try again. At the end of this sequence, the contents of X23 and the memory location specified by X20 have been atomically exchanged.

## An advantage of the load/store exclusive mechanism is that it can be used to build other synchronization primitives, such as atomic compare and swap or atomic fetch-and-increment, which are used in some parallel programming models.2.11 – Translating and Starting a Program

This section describes the four steps in transforming a C program in a file from storage (disk or flash memory) into a program running on a computer.

**1 ) Compile** – High-level program compiled into assembley language

**2 ) Assemble** – The code is assembled into machine code as an object module.

**3 ) Link** – Linker combines modules with library routines to resolve all references.

**4 ) Load** – Loader slaps machine code into proper memory locations for the processor to execute.

### Compiler

In the 70s operating systems and assemblers were written in machine language because memory was small and compilers were inefficient. The million-fold increase in DRAM has reduced program size concerns.

Assembly language: A symbolic language that can be translated into binary machine language

### Assembler

Primary purpose: assemble assembley language into machine code.

Object file: what the assembler creates. It is a combination of machine language instructions, data, and information needed to place instructions properly in memory.

In UNIX, the object file contains 6 distinct pieces:

Header: describes the size and position of the other pieces of the object file.

Text segment: contains the machine language code.

Static data segment: contains data allocated for the life of the program.

UNIX allows programs to use both static data, which is allocated throughout the program, and dynamic data, which can grow or shrink as needed by the program.

Relocation information: identifies instructions and data words that depend on absolute addresses when the program is loaded into memory.

Symbol table: contains the remaining labels that are not defined, such as external references.

Symbol table: A table that matches names of labels to the addresses of the memory words that instructions occupy.

To produce the binary version of each instruction, the assembler must determine the addresses corresponding to all labels.

Pseudoinstruction: A common variation of assembly language instructions often treated as if it were an instruction in its own right.

Assembley language is an interface to higher-level software. It can therefore treat common variations of machine language instructions as if they were instructions in their own right.

MOV X9, X10 → ORR X9, XZR, X10

CMP X9, X10 → SUBS XZR, X9, X10

[use X9 - X10 to set condition codes]

### Linker

It would be super inefficient to recompile and resassemble code every single time a single line was changed in the high-level portion.

Alternative: compile and assemble each procedure independently, and then have a linker or link editor stitch them together. Thus, changes to code only demands recompiling and reassembliing for one section, rather than the whole shebang.

**Linker**: a systems program that combines independently assembled machine language programs and resolves all undefined labels into an executable file.

1) Place code and data modules symbolically in memory.

2) Determine the addresses of data and instruction labels.

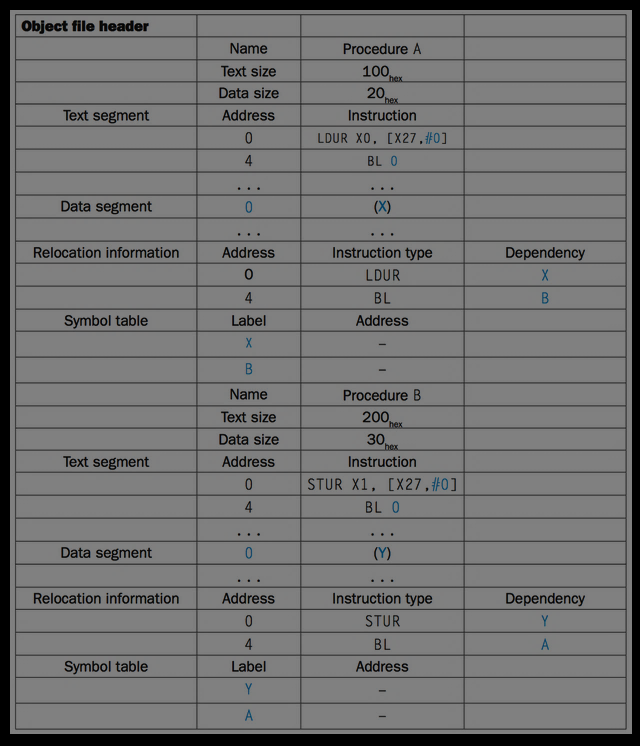
3) Patch both the internal and external references.

The job of this program is much like that of an editor: it finds the old addresses and replaces them with the new addresses

If all external references are resolved, the linker next determines the memory locations each module will occupy.

Since the files were assembled in isolation, the assembler could not know where a module's instructions and data would be placed relative to other modules. When the linker places a module in memory, all absolute references, that is, memory addresses that are not relative to a register, must be relocated to reflect its true location.

**Executable file**: A functional program in the format of an object file that contains no unresolved references. It can contain symbol tables and debugging information. A "stripped executable" does not contain that information. Relocation information may be included for the loader.



Note the 2 types of instruction formats:

The branch and link instructions use PC-relative addressing.

The load and store addresses are harder because they are relative to a base register. This example uses X27 as the base register, assuming it is initialized to 0000 0000 1000 0000hex.

### Loader

Now that the executable file is on disk, the operating system reads it to memory and starts it. The loader follows these steps in UNIX systems:

Reads the executable file header to determine size of the text and data segments.

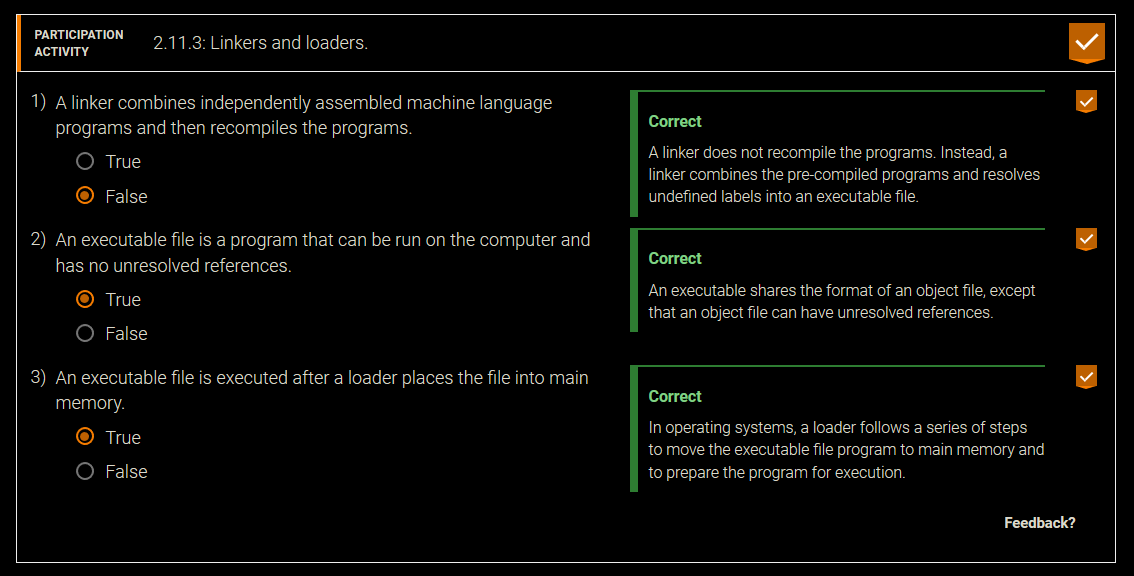
Creates an address space large enough for the text and data.

Copies the instructions and data from the executable file into memory.

Copies the parameters (if any) to the main program onto the stack.

Initializes the processor registers and sets the stack pointer to the first free location.

Branches to a start-up routine that copies the parameters into the argument registers and calls the main routine of the program. When the main routine returns, the start-up routine terminates the program with an exit system call.



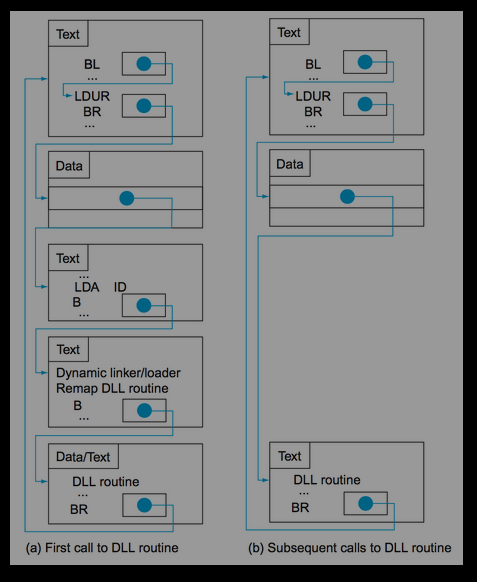
### Dynamically Linked Libraries

Although statically linking libraries is faster, there are disadvantages:

Library routines are a part of the code. New updates to library cannot be used.

Loads all routines called anywhere, even if those routines are never executed.

Dynamically linked libraries (DLLs): Library routines that are linked to a program during execution.

This trick relies on some misdirection.

First, it starts with nonlocal routines calling a set of dummy routines at the end of a program – with one entry per nonlocal routine. Each dummy entry contains an indirect branch.

Image to right:

a) The first time a library routine is called, the program calls the dummy entry and follows the indirect branch.

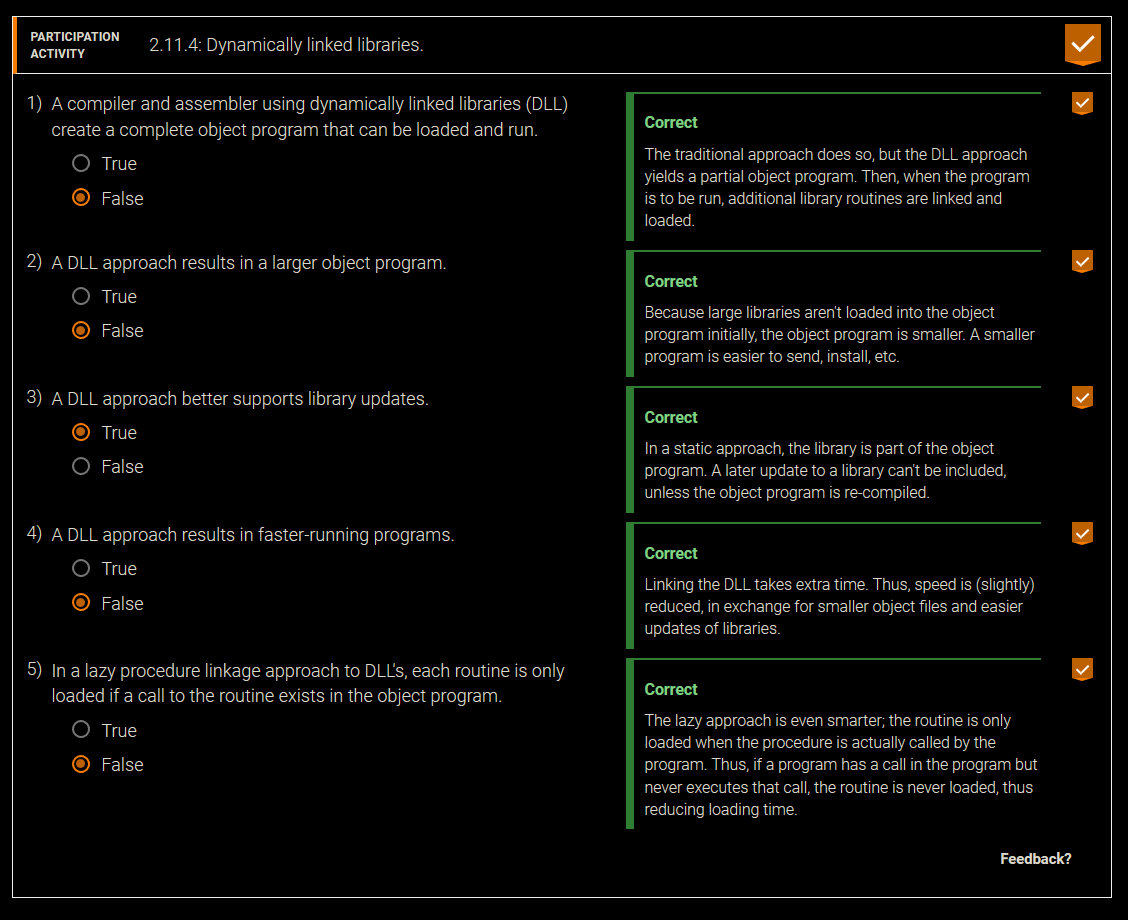
It points to code that puts a number in a register to identify the desired library routine and then branches to the dynamic linker/loader.

The linker/loader finds the wanted routine, remaps it, and changes the address in the indirect branch location to point to that routine. It then branches to it.

When the routine completes, it returns to the original calling site.

b) Thereafter, the call to the library routine branches indirectly to the routine without the extra hops.

In summary, DLLs require additional space for the information needed for dynamic linking, but do not require that whole libraries be copied or linked. They pay a good deal of overhead the first time a routine is called, but only a single indirect branch thereafter.



### Java

Java was invented with a different set of goals, however. One was to run safely on any computer, even if it might slow execution time. Virtually no optimizations are performed. Like the C compiler, the Java compiler checks the types of data and produces the proper operation for each type. Java programs are distributed in the binary version of these bytecodes.

Java Virtual Machine (JVM) : The program that interprets Java bytecodes.

Just In Time compiler (JIT) : The name commonly given to a compiler that operates at runtime, translating the interpreted code segments into the native code of the computer.

## 2.12 – Supporting Procedures in Computer Hardware

Procedure: A stored subroutine that performs a specific task based on the parameters with which it is provided.

As mentioned above, registers are the fastest place to hold data in a computer, so we want to use them as much as possible. LEGv8 software follows the following convention for procedure calling in allocating its 32 registers:

X0-X7: eight parameter registers in which to pass parameters or return values.

LR (X30): one return address register to return to the point of origin.

Branch-and-link instruction: An instruction that branches to an address and simultaneously saves the address of the following instruction in a register (LR or X30 in LEGv8).

BL *ProcedureAddress*

Return address: A link to the calling site that allows a procedure to return to the proper address; in LEGv8 it is stored in register LR (X30)

To return back to the next instruction of the program (calling address + 4):

BR LR

### Using More Registers

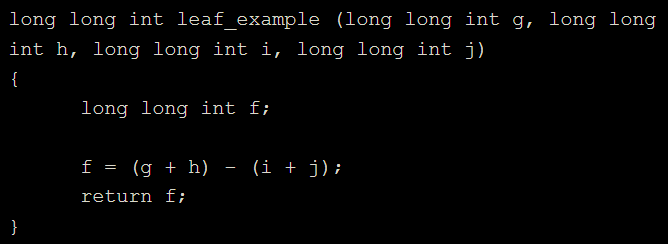
Sometimes we need more than 8 argument registers to get shit done son. To use others, we must cover our tracks after completion.

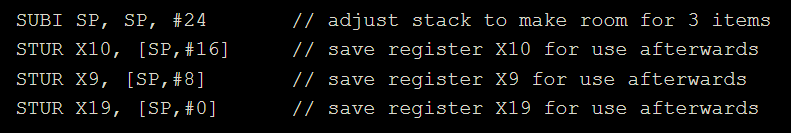
Stack: A data structure for spilling registers organized as a last-in- first-out queue.

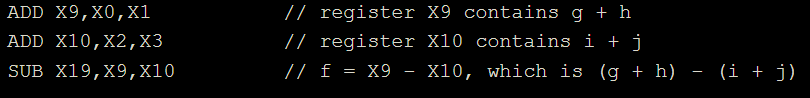
Stack pointer: A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found. In LEGv8, it is register SP.

By historical precedent, stacks "grow" from higher addresses to lower addresses. This convention means that you push values onto the stack by subtracting from the stack pointer. Adding to the stack pointer shrinks the stack, thereby popping values off the stack.

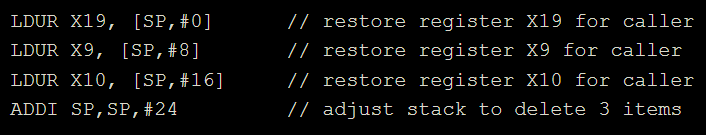
### Example – Compiling a C Procedure



1) The procedure leaf\_example uses the registers 9, 10, and 19. We must therefore preserve these values by pushing them onto the stack, because the program may have been using them until this point.

2) Now the stack has preserved the values of the register by storing them in memory, we can go ahead and do whatevs with 9, 10, and 19.

3) Once done computing, we put the return value in 0.

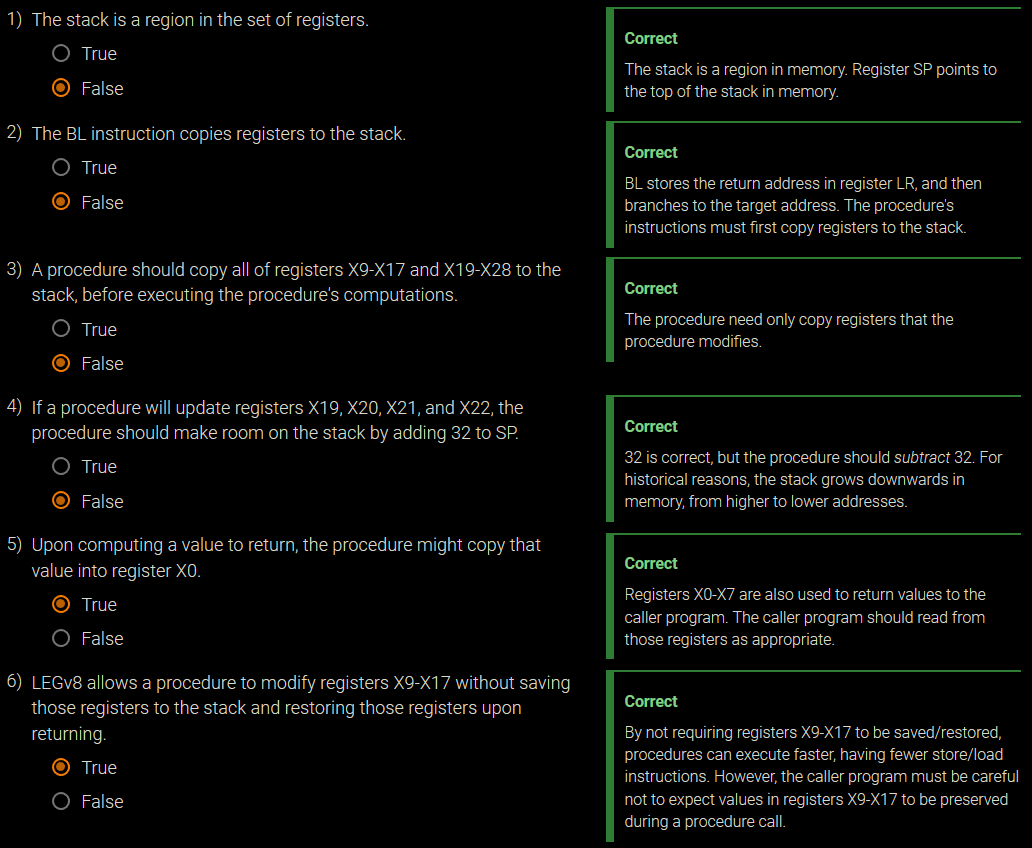
4) Before returning to previous program point, we restore the values of the register.

5) Finally we return.

In the previous example, we used temporary registers and assumed their old values must be saved and restored. To avoid saving and restoring a register whose value is never used, which might happen with a temporary register, LEGv8 software separates 19 of the registers into two groups:

X9 - X17: temporary registers that are not preserved by the callee.

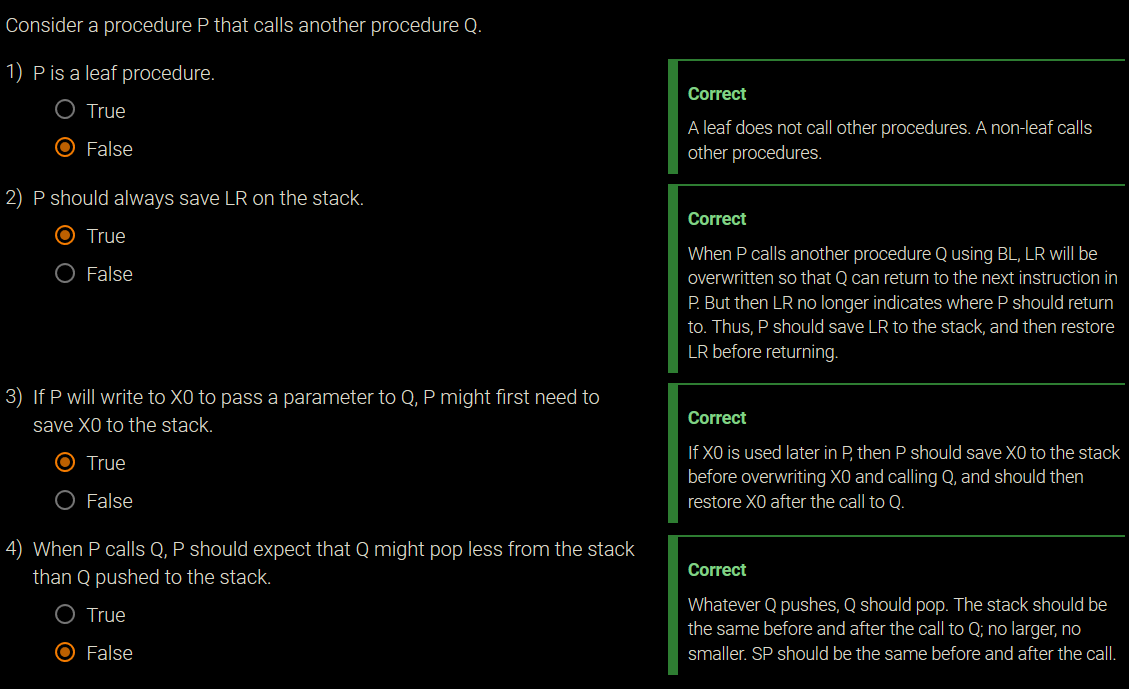
X19 - X28: saved registers that must be preserved on a procedure call.



### Nested Procedures

Leaf Procedure: one that does not call another.

Dead Brain: Example 2.12.2 all the way to the exercise below.



### Allocating Space for New Data on the Stack

Procedure frame: Also called activation record. The segment of the stack containing a procedure's saved registers and local variables.

Dead Brain: Fucking all of 2.12.

### Allocating Space for Data on the Heap

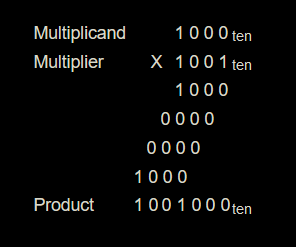
GIVE ME YOUR TEARS GYPSY

# 3 – More Arithmetic for Computers

## 3.1 – Introduction

stuff

## 3.2 – Multiplication

The first operand is called the multiplicand and the second the multiplier. The final result is called the product.

Dead Brain: Activity 2.3.2

stuff

## 3.3 – Division

Process of Hardware Division

The Divisor register, ALU, and Remainder register are all 128 bits wide, with only the Quotient register being 64 bits.

The divisor is placed in the left half of Divisor register, the dividend is placed in the Remainder, and the Quotient is set to 0.

The Divisor is subtracted from the Remainder, then the result is placed into the Remainder.

If the result is negative, then the Remainder value is restored by adding the divisor to the Remainder.

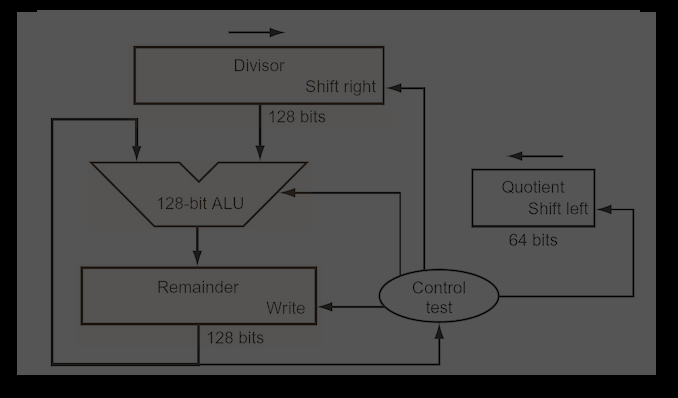
A negative Remainder indicates the divisor did not go into the dividend. Control shifts Quotient left 1 bit and places a 0 in the least significant bit.

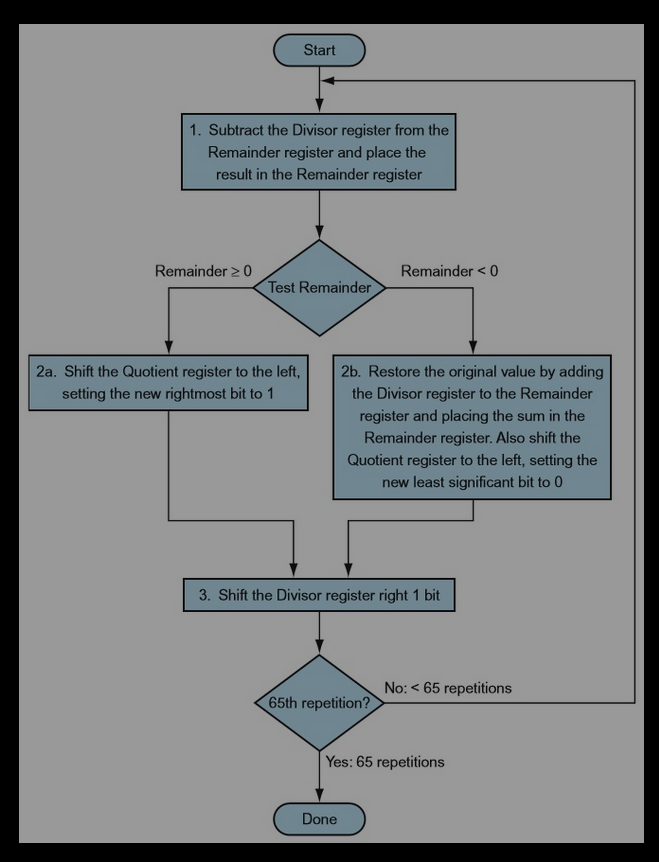
Divisor is shifted right 1 bit and the divide operation repeats.

The Divisor is subtracted from the Remainder, then the result is placed into the Remainder.

If the Remainder is positive, then the divisor was smaller or equal to the dividend. Control shifts Quotient left 1 bit and places a 1 in the least significant bit.

Divisor is shifted right 1 bit and the divide operation repeats. The steps repeat a total of 65 times.





stuff

## 3.4 – Floating Point

stuff

## 3.5 – Parallelism and Computer Arithmetic

stuff

# 4 – The Processor

## 4.1 – Introduction

Chapter 1 – Performance determined by instruction count, clock cycle time, and clock cycles per instruction.

Chapter 2 – Copmiler and instruction set architecture determine the instruction count. The implimentation deterimines the clock cycle and number of cycles per instruction.

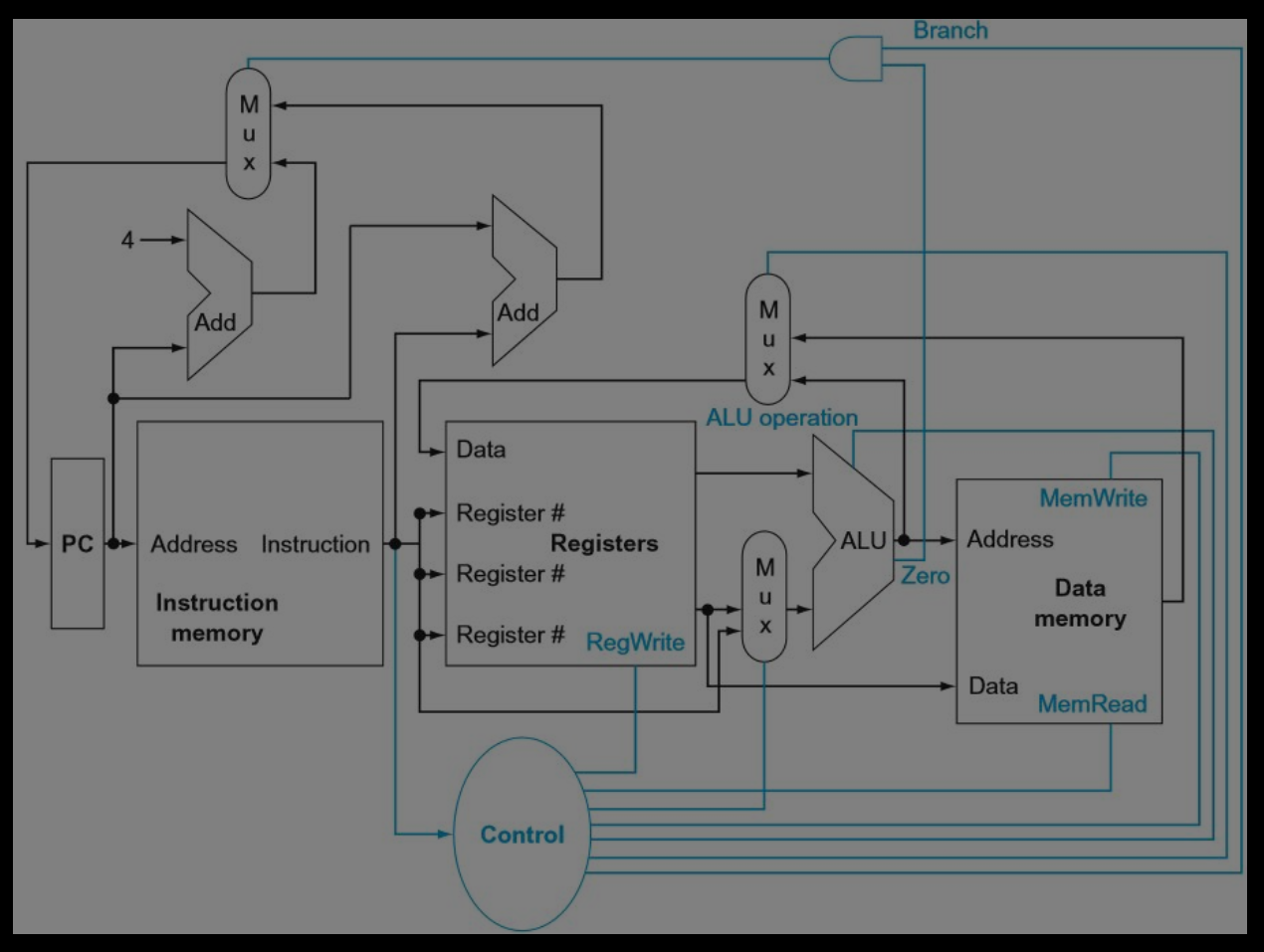
For every instruction, the first two steps are identical:

Send the program counter (PC) to the memory that contains the code and fetch the instruction from that memory.

Read one or two registers, using fields of the instruction to select the registers to read.

After these two steps, the actions required to complete the instruction depend on the instruction class. Fortunately, for each of the three instruction classes (memory-reference, arithmetic-logical, and branches), the actions are largely the same, independent of the exact instruction.

### Basic Implimentation of LEGv8 Subset



Top mux replaces PC with (PC + 4) or branch address.

Middle mux is used to:

Steer the output of the ALU, in event of a arithmetical logical instruction, to the register.

Steer the output of data memory, in event of a load, to the register.

Bottom mux determines whether 2nd ALU input is from registers (arithmetic/branch) or from the offset field of the instruction (loadstore).

Control lines determine:

The operation performed at the ALU.

Whether data memory should read or write.

Whether registers should perform a write operation.

While easier to understand, this approach is not practical, since the clock cycle must be severely stretched to accommodate the longest instruction.

## 4.2 – Logic Design Conventions

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. Has 2 inputs and 1 output.

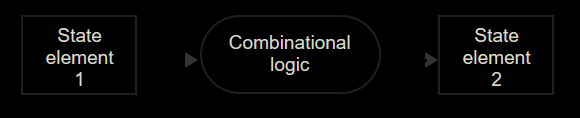
Inputs: clock and data to be written.

Output: value written at earlier clock cycle.

### Clocking Methedology

Clocking methodology : The approach used to determine when data is valid and stable relative to the clock.

Because only state elements can store a data value, any collection of combinational logic must have its inputs come from a set of state elements and its outputs written into a set of state elements.



All signals must propagate from state element 1, through the combinational logic, and to state element 2 in the time of one clock cycle. The time necessary for the signals to reach state element 2 defines the length of the clock cycle.

The state element is changed only when the write control signal is asserted and a clock edge occurs.

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.

It doesn't matter whether we assume that all writes take place on the rising clock edge (from low to high) or on the falling clock edge (from high to low), since the inputs to the combinational logic block cannot change except on the chosen clock edge.



## 4.3 – Building a Datapath

## 4.4 – Implimentation Scheme

ststuff