CSCI516 - Fundamental Concepts of Computing / Machine Organization

Song Huang, Ph.D.
Song.Huang@tamuc.edu

This Course

- **Computer Organization and Design: The Hardware Software Interface: ARM Edition** (by Patterson and Hennessy, Morgan Kaufmann, ISBN-13: 978-0128017333)
- Textbook is for you to read. **Not for me to read it for you**.

Tentative Breakdown of Course Grade

Tentative Breakdown of Course Grade	
Homework Assignments	30%
Programming Assignments	20%
Quizzes	10%
Paper Review	10%
Exams	30%

This Course

No late submission will be accepted. If you are not able to submit your work before the deadline for some reasons, you need to send out an email to the Instructor 24 hours beforehand.

- **Homework**. Students are required to answer questions in the homework.
- **Programming Assignment**. Students need to use DS-5 environment for your programming assignment.
- **Quiz**. Some quizzes will be given, and students will know the quiz time beforehand.
- **Paper Review**. Students are required to read technical paper and summarize the key points of the paper.
- **Exam**. There will be 2 exams in the semester.

Ethics

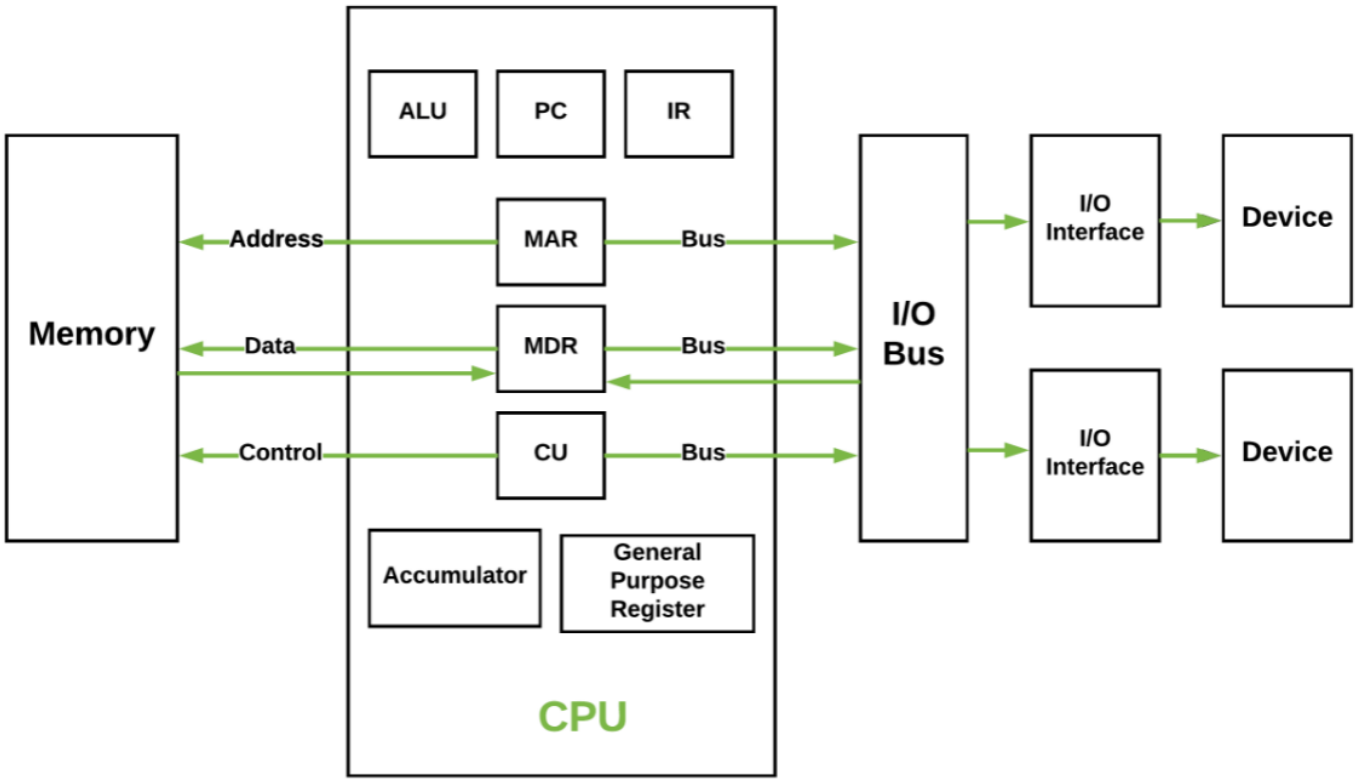
There is no group project / work in this class. All of the work is independent work. Students can communicate the thoughts of solving problems, but there shouldn't be any duplicate solutions.

- All students will be trusted to pursue their academic careers with honesty and integrity.
- Academic dishonesty includes, but not limited to, cheating on a test or other course work, plagiarism, unauthorized collaboration with other persons.
- Students found guilty of dishonesty will be subject to penalties.

Machine Organization

Machine Organization is one type of architecture. An abstraction that defines a boundary between different levels as viewed by users.

- It defines how the various hardware components of a processor are connected together, and information flows between them.
- Central Processing Unit (CPU), Cache, Memory, I/O, etc and components within a CPU (the various pipe stages, registers etc)



Instruction Set Architecture(ISA)

The abstraction that describes the interface between hardware and software.

```
main:
        ADRP X0, v
        ADD X0, X0, :lo12:v
        LDUR X1, [X0, #0]
        LDUR X2, [X0, #8]
        ADD X1, X1, X2
        LDUR X2, [X0, #16]
        ADD X1, X1, X2
Exit:
```

The Computer Revolution

Progress in computer technology - Underpinned by **Moore's Law**

- Circuit resources double every 18 months (1.5 years)
- Twice as many transistors every 18 months (1.5 years) - can be used to add more cores, more cache memory
- This is made possible by "shrinking" the size of transistors

There is another law called **Dennard's law** (or scaling)

- Until now, Shrinking size also meant "shrinking" power consumption also We were able to achieve better performance without increasing power needs
- **This is no longer true.**
- If we reduce transistor size, we are not reducing power requirements
- Even worse, energy consumed my increase

The Computer Revolution

Technology revolution is making more applications possible:

- Computers in automobiles
- Airplanes
- Cell phones
- Human genome project
- World Wide Web
- Search Engines

Classes of Computers

Classes of Computers

- Personal computers (desktops laptops)
  - 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 (https://www.top500.org)
  - High-end scientific and engineering calculations
  - Highest capability but represent a small fraction of the overall computer market
  - Exa-scale computers (1018 FLOPs - floating point operations per second)

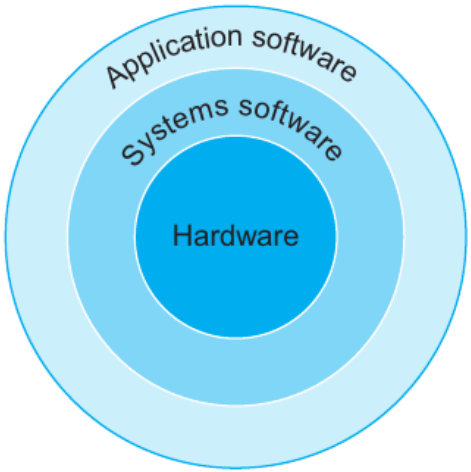
Classes of Computers

Classes of Computers

- Embedded computers
  - Hidden as components of systems
  - Computers in cars, airplanes
  - Somewhat related - Cyber physical systems (CPS)
    - Smart "things"
    - Smart homes, smart grids, smart transportation
- Cloud computing
  - Warehouse Scale Computers (WSC)
  - Software as a Service (SaaS)
  - Portion of software run on a personal devices and a portion run in the Cloud
  - Amazon and Google
  - Throughput is more important

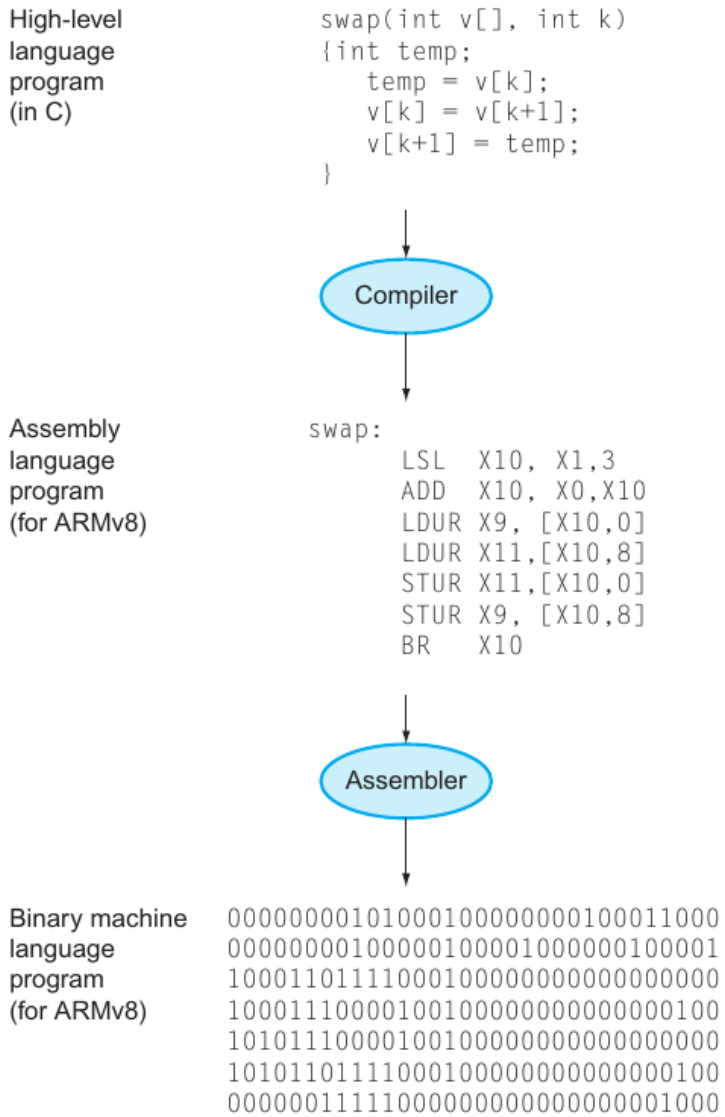
Below your Program

- Application software: Written in high-level language (like C)
- System software
  - Compiler: translates high level language code to machine code
  - Operating System (and runtime): service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources
- Hardware
  - Processor, memory, I/O controllers
  - Actually run your program



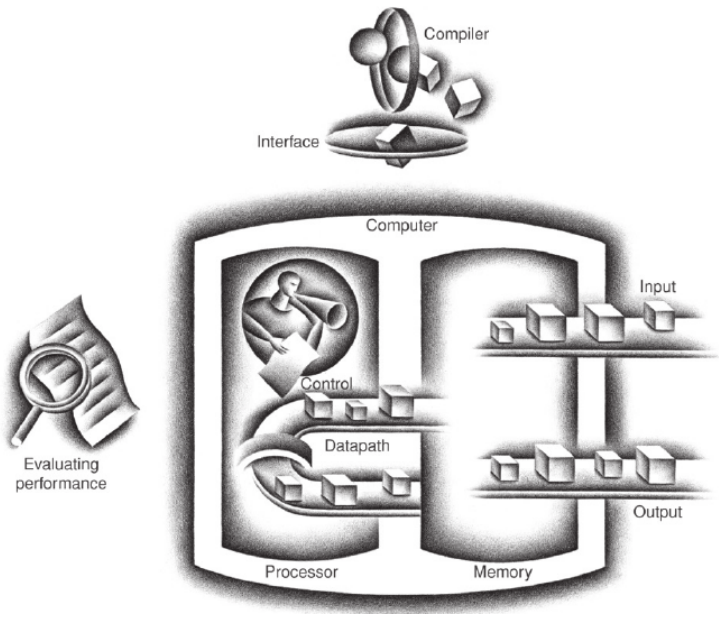
Levels of Program Code

- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - Textual representation of instructions
- Hardware representation (machine language)
  - Binary digits (bits)
  - Encoded instructions and data



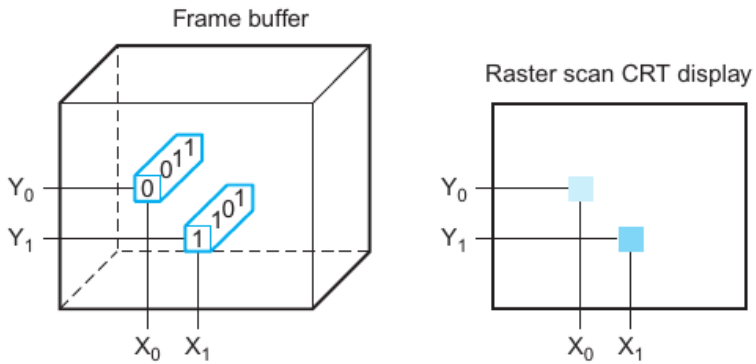
Components of a Computer

- Five Components:
  - Input - Ex: keyboard, mouse, touch screen
  - Output - Ex: Display, Speaker, files
  - Memory (DRAM)
  - Control
  - Datapath
- Storage devices: Hard disk, CD/DVD, flash, Xpoint
- Network adapters: For communicating with other computers
- Same components may be used by all kinds of computers: Desktop, server, and embedded; But the speed, cost, capacities differ

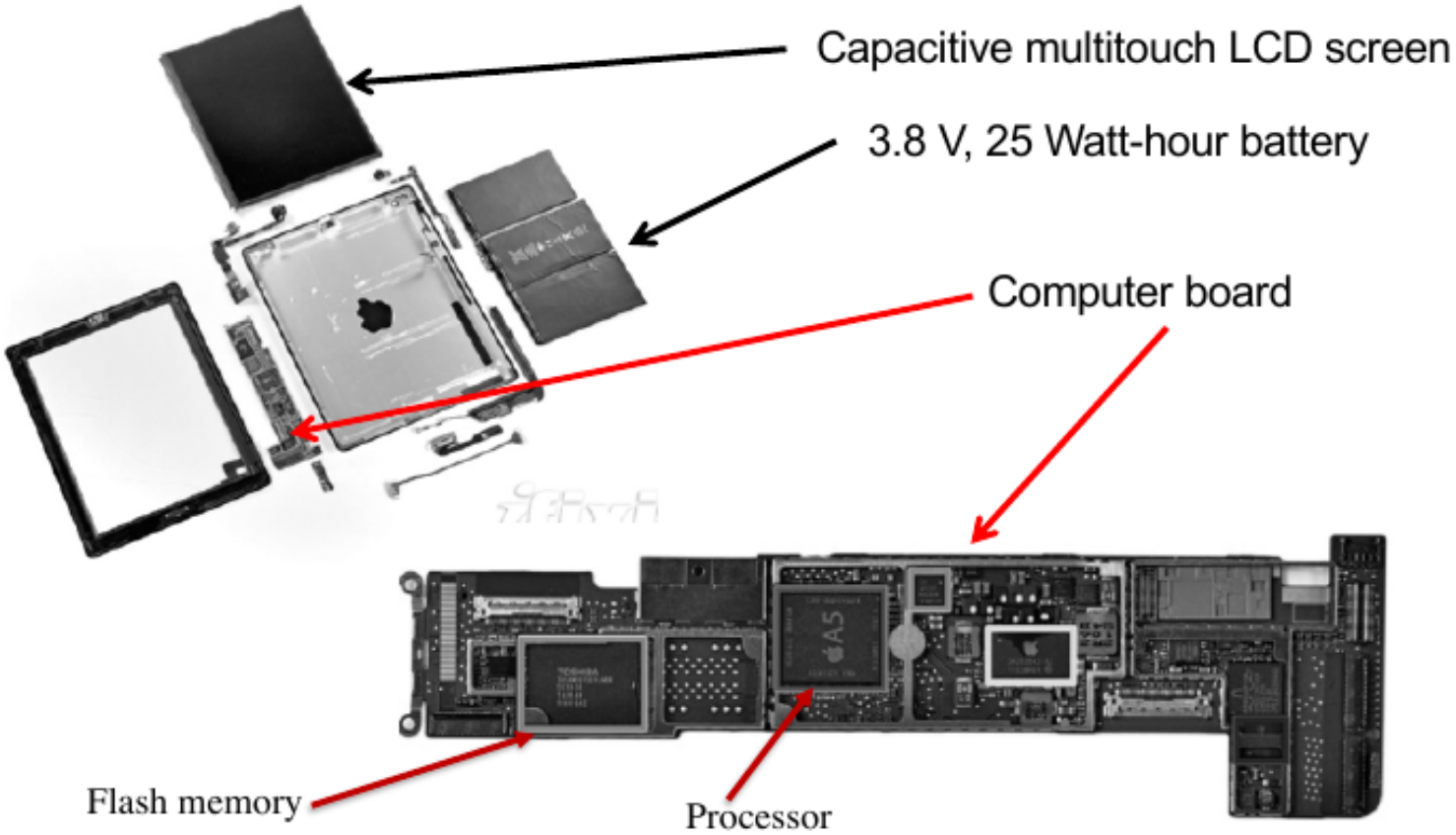


Touchscreen

- PostPC I/O devices: Handheld devices, tablets
- Supersedes keyboard and mouse
- Resistive and Capacitive types (resistive reacts to pressure Capacitive to conducting properties)
  - Most tablets, smart phones use capacitive
  - Capacitive allows multiple touches simultaneously
  - Resistive are more durable less accurate (used in grocery stores, hospitals, ATMs etc)
- LCD screen: picture elements (pixels) : Mirrors content of frame buffer memory



Opening the Box (iPad)



Inside the Processor (CPU)

- Apple A5

GPU/graphics units

2 Cores

Inside the Processor (CPU)

- Data path: Defines all the hardware units needed and how they are connected
  - These units perform actual computations and communicate with each other
  - Examples include: Arithmetic Logic Unit, Registers, Comparators
- Control path: decides when a data path is enabled
  - When a particular hardware unit becomes active, which communication paths are used at a particular time
  - For example, if we are dealing with floating point addition, enable that unit and not integer add or multiply unit
- Cache memory: Small fast SRAM memory for immediate access to data

A Safe Place for Data

- Volatile main memory (SRAM and DRAM)
  - Loses instructions and data when power off
- Non-volatile secondary memory
  - Magnetic disk
  - Flash memory (and PCM)
  - New technologies (intel 3D Xpoint)
  - Optical disk (CDROM, DVD)

Network

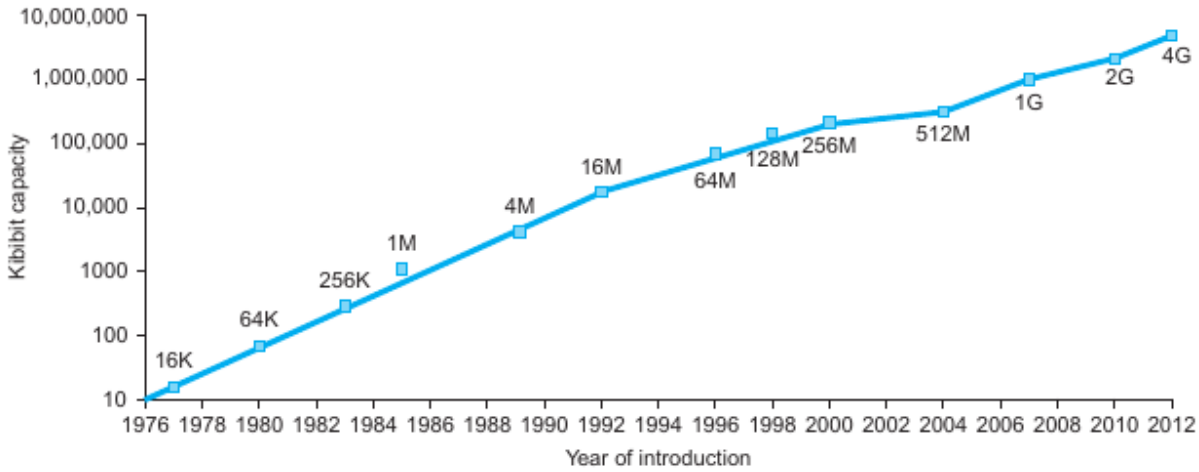
- Communication and resource sharing
- Local area network (LAN): Ethernet: Within a building
- Wide area network (WAN: the Internet)
- Wireless network: Wi-Fi, Bluetooth

Technology Trends

Electronics technology continues to evolve

- Increased capacity and performance
- Reduced cost

DRAM Capacity



Relative performance per unit cost of technologies used in computers.

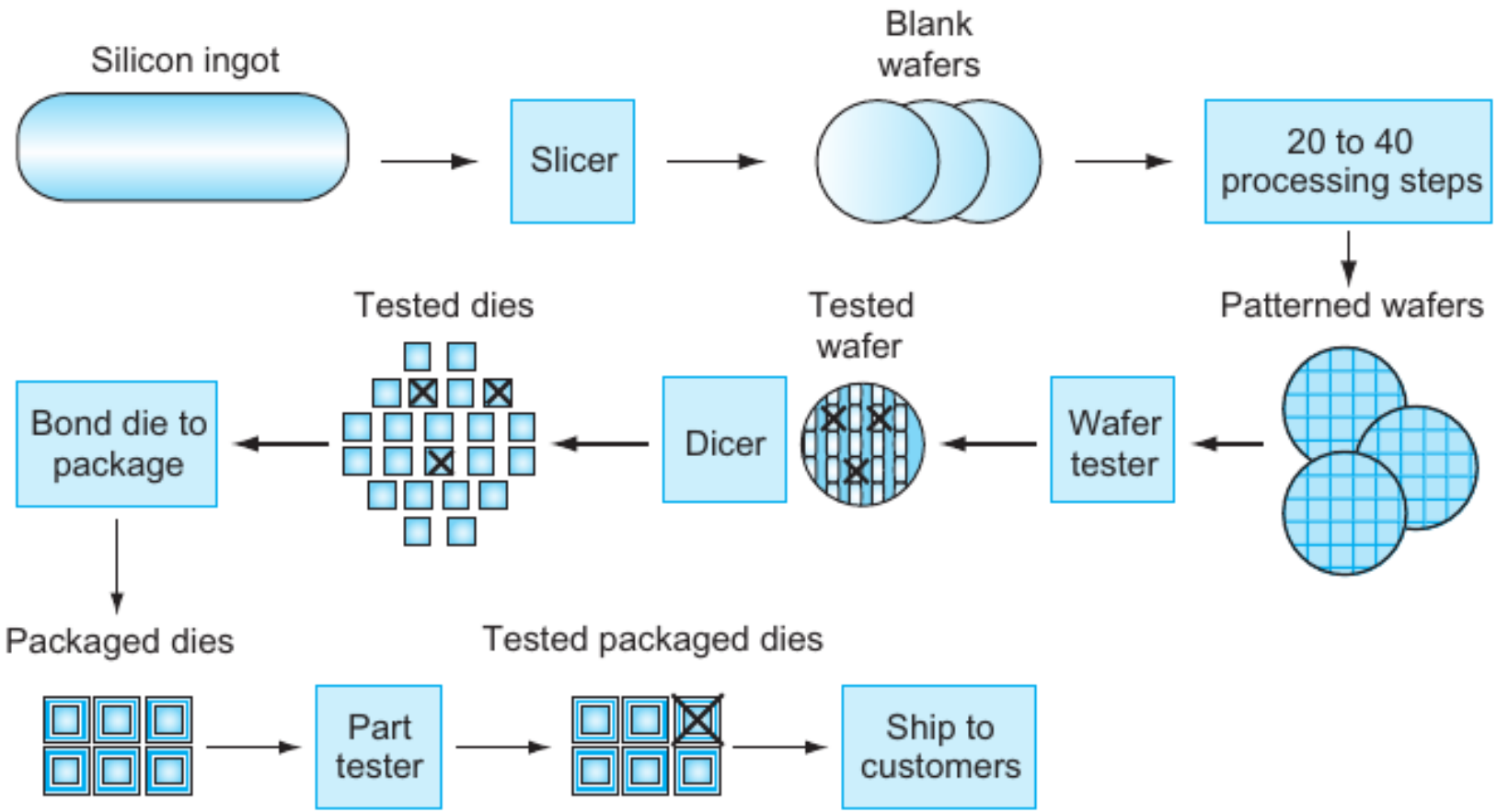
Year	Technology used in computers	Relative performance/unit cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit	900
1995	Very large-scale integrated circuit	2,400,000
2013	Ultra large-scale integrated circuit	250,000,000,000

Semiconductor Technology

- Silicon: semiconductor
- Add materials to transform properties:
  - Conductors
  - Insulators
  - Switch (actual device that can switch between zero and one): Typically a transistor
- Two types of semiconductor designs
  - CMOS (Complementary Metal Oxide Semiconductor): Used to build logic devices (ALUs, control logic)
  - NMOS (n channel Metal Oxide Semiconductor): Used to build memories (DRAM)

Manufacturing ICs

The process:



Yield: How many good chips from each wafer

Integrated Circuit Cost

$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{Yield}}$$

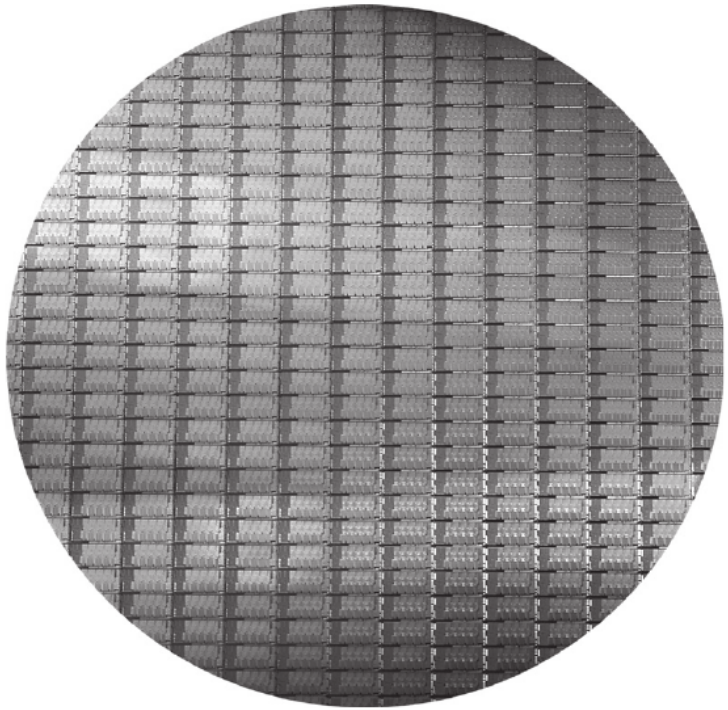
$$\text{Dies per wafer} \approx \text{Wafer area}/\text{Die area}$$

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

Nonlinear relation to area and defect rate

- Wafer cost and area are fixed
- Defect rate determined by manufacturing process
- Die area determined by architecture and circuit design

Intel Core i7



- 300mm wafer, 280 chips, 32nm technology
- Each chip is 20.7 x 10.5 mm

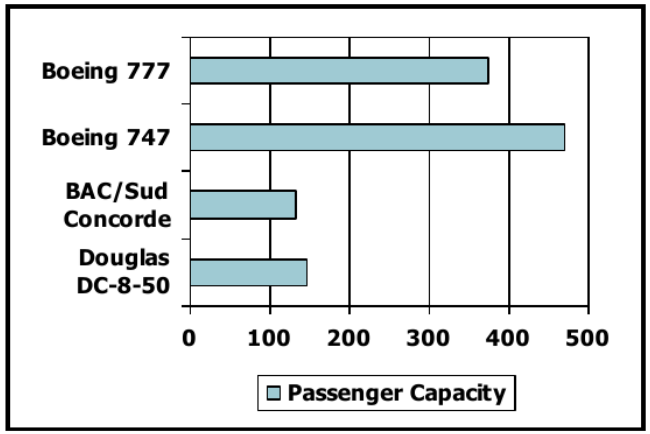
Understanding Performance

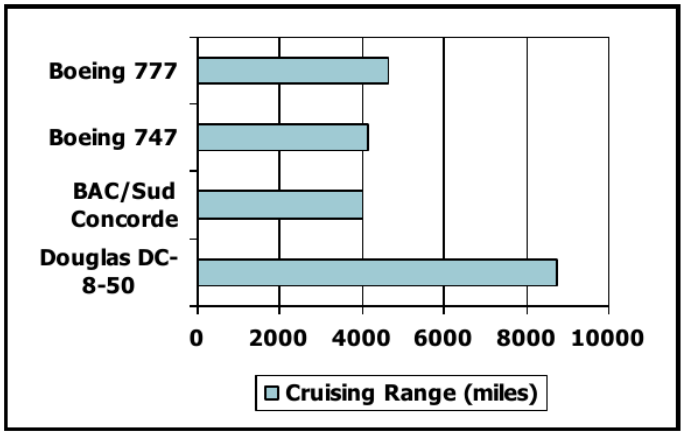
What impacts performance?

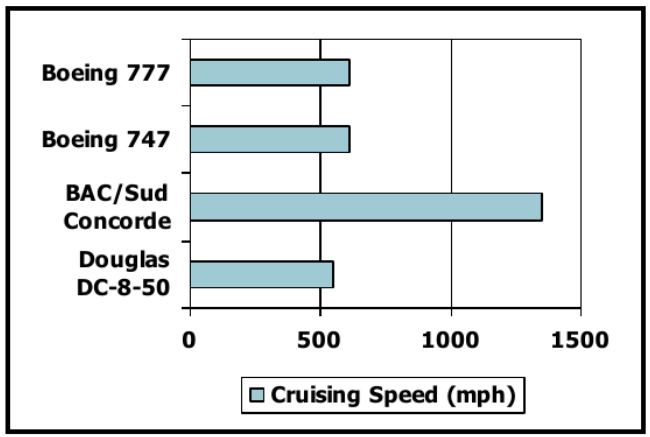
- Algorithm
  - Determines number of operations executed
  - For example quick sort is the best in terms of execution complexity
- Programming language, compiler, and Instruction Set architecture
  - For example C is more efficient than C++ or Java
  - Compiler optimizations may also differ
  - Different instruction result in different number of instructions
- Processor and memory system
  - Determine how fast instructions are executed
  - This is what we will explore in this class
- I/O system (including OS): Determines how fast I/O operations are executed

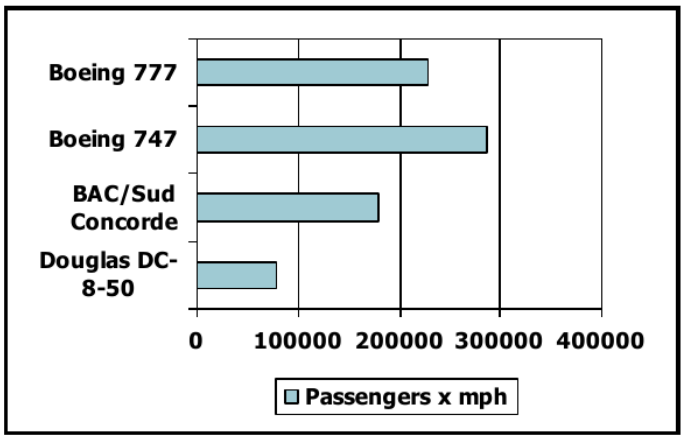
Defining Performance

- Performance may mean different things: Different ways of measuring how good a system is
- Which airplane has the best performance?









Response Time and Throughput

- Response time: How long it takes to do a task
- Throughput
  - Total work done per unit time
  - e.g., tasks/transactions/... per hour
- How are response time and throughput affected by
  - Replacing the processor with a faster version?
  - Adding more processors?
- We' ll focus on response time for now...
- Other measures
  - Power and Energy consumed
  - Reliability, Availability
  - Cost

CSCI516 - Fundamental Concepts of Computing / Machine Organization

Song Huang, Ph.D.
Song.Huang@tamuc.edu

Relative Performance

- Performance = 1/Execution Time
- "X is n time faster than Y"

$$\text{Performance}_X / \text{Performance}_Y = \text{Execution time}_Y / \text{Execution time}_X = n$$

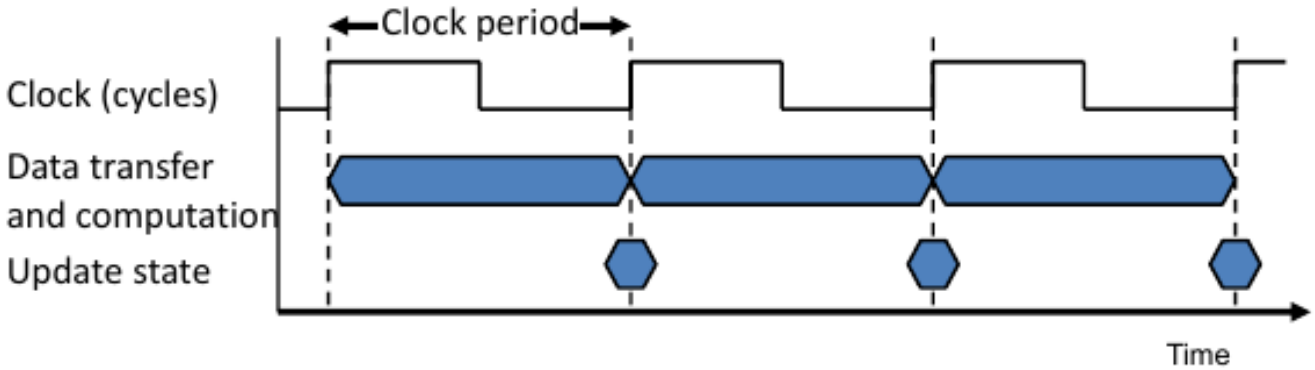
- Example: time taken to run a program
  - 10s on A, 15s on B
  - Execution Time B / Execution Time A = 15s / 10s = 1.5
  - So A is 1.5 times faster than B

Measuring Execution Time

- Elapsed time
  - Total response time, including all aspects: Processing, I/O, OS overhead, idle time
  - Determines system performance
- CPU time
  - Time spent processing a given job Discounts I/O time, other jobs' shares
  - Comprises user CPU time and system CPU time
  - Different programs are affected differently by CPU and system performance
  - In this class we primarily focus on USER CPU time only

CPU Clocking

- Operation of digital hardware governed by a constant-rate clock



- Clock period: duration of a clock cycle
  e.g., 250ps = 0.25ns = $250x10^{-12}$s
- Clock frequency (rate): cycles per second
  e.g., 4.0GHz = 4000MHz = $4.0x10^{9}$Hz
- Clock_time = 1/(Clock_frequency)

CPU Time

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

CPU Clock Cycles = total number of clock cycles needed to complete execution

- Performance can be improved by
  - Reducing number of clock cycles
  - Increasing clock rate

  Hardware designer must often trade off clock rate against cycle count
- Example:
  A program takes 1000 clock cycles to run on a processor running at 2 GHz. What is the time spent on the CPU by the program?
  CPU time = cycles /clock rate = $1000/(2\text{x}10^9) = 0.5 \text{ x } 10^{-6} = 0.5\mu\text{s}$

CPU Time Example 2

textbook page 34-35

- Computer A: 2 GHz clock, Execution time (CPU Time) = 10s
- **Designing Computer B**
  - Aim for execution time (CPU time) to 6s CPU
  - Can use faster clock, but causes 1.2 x clock cycles - need more cycles
- How fast must Computer B clock (clock rate) be?

1.9 Assume for arithmetic, load/store, and branch instructions, a processor has CPIs of 1, 12, and 5, respectively. Also assume that on a single processor a program requires the execution of 2.56E9 arithmetic instructions, 1.28E9 load/store instructions, and 256 million branch instructions. Assume that each processor has a 2 GHz clock frequency.

Assume that, as the program is parallelized to run over multiple cores, the number of arithmetic and load/store instructions per processor is divided by $0.7 \times p$ (where p is the number of processors) but the number of branch instructions per processor remains the same.

Instruction Count and CPI

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- Instruction Count for a program
  - Determined by program, ISA and compiler
- Average cycles per instruction
  - Determined by CPU hardware
  - If different instructions have different CPI
    - Average CPI affected by instruction mix
    - CPI means average number of cycles per instructions
    - Note different instructions take different number of cycles

CPI Example 1

- Computer A:
  Clock Cycle Time = 250ps (clock rate = 4GHz), CPI = 2.0
- Computer B:
  Clock Cycle Time = 500ps (clock rate = 2GHz), CPI = 1.2
- Same Program and ISA (same number of instructions)

Which is faster, and by how much?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$
$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}$$

A is faster...

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$
$$= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2$$

...by this much
20% faster

Faster clock = smaller clock cycle.

CPI in More Detail

- Different instruction classes take different numbers of cycles

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

- Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}} \right)$$

CPI Example 2

- Alternative compiled code sequences using instructions in classes A, B, C **For example Arithmetic, Load/Store and Branch instructions**

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

Sequence 1: IC = 5

- Clock Cycles
  = 2×1 + 1×2 + 2×3
  = 10
- Avg. CPI = 10/5 = 2.0

- Sequence 2: IC = 6
  - Clock Cycles
    = 4×1 + 1×2 + 1×3
    = 9
  - Avg. CPI = 9/6 = 1.5

- Do we have different CPI's for different programs?
- Do we need to compute CPI across many programs?

Performance Summary

$$\text{CPU Time} = \frac{\textit{Instruction Count}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

$$\text{CPU Time} = (\text{Instruction Count}) * \text{CPI} * 1/(\text{Clock Rate})$$

Performance depends on

- Algorithm: affects Instruction count (IC), possibly CPI
- Programming language: affects IC, CPI
- Compiler: affects IC, CPI
- Instruction set architecture: affects IC, CPI, Clock rate

CPI Example 3

The average frequencies for various instruction types.

- ALU operations occur 43% of time and take 1 clock cycle to execute
- Load instructions occur 21% of the time and need 2 cycles
- Store instructions occur 12% of the time and need 2 cycles
- Branch instructions occur 24% of the time and need 2 cycles

How do we get CPI = average cycles per instruction?

The average number of cycles per instruction =
0.43*1+0.21*2+0.12*2+0.24*2 = 1.57 Cycles per instruction

MIPS: Millions of instruction per second

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6} \qquad \text{MIPS} = \frac{\text{Instruction count}}{\frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

Once we have the CPI, and clock speed, we can find the MIPS ratings of a processor

- If we are using 1Ghz processor (10^9 cycles per second)
- the MIPS rating is given by $10^9/\text{CPI} = 10^9 /(1.57\text{x}10^6) = 637$ Million Instruction Per Second
- Execution time = (instruction_count)/ MIPS = IC*(1/637 MIPS) = (instruction_count)*6.37*10^{-9} Sec

MIPS: Meaningless Information Per Second

MIPS rating can be misleading.

- Suppose we have a compiler that can optimize the program
- The optimized compiler can eliminate 50% of Arithmetic instructions.

What is the new CPI?

- ALU operations occur 21.5% of time and take 1 clock cycle to execute
- Load instructions occur 21% of the time and need 2 cycles
- Store instructions occur 12% of the time and need 2 cycles
- Branch instructions occur 24% of the time and need 2 cycles

MIPS: Meaningless Information Per Second

But we need to scale these fractions since the total is only 78.5%

- So CPI = [(21.5%)*1 + (21%)*2+(12%)*2+(24%)*2]/(78.5%) = 1.73 CPI — higher CPI?
- MIPS = (1 Ghz)/[1.73*10^6] = 578 MIPS

So the computer with an optimized compiler has lower MIPS rating!

- Execution time before optimization = (IC)/(637*10^6)
- After optimization = 0.785*(IC)/(578*10^6)
- optimized version is 1.16 times faster

One more Example

1.5 [4] <§1.6> Consider three different processors P1, P2, and P3 executing the same instruction set. P1 has a 3 GHz clock rate and a CPI of 1.5. P2 has a 2.5 GHz clock rate and a CPI of 1.0. P3 has a 4.0 GHz clock rate and has a CPI of 2.2.

a. Which processor has the highest performance expressed in instructions per second?

- Execution time = (Instruction_count)*(CPI)/(Clock_rate)
- Instruction per Second = $\frac{(Instruction\_count)}{Executiontime} = \frac{Clock\_rate}{CPI}$

Then P2 is faster:

- P1: 3 GHz / 1.5 = 2 x 10^9 instruction/second
- P2: 2.5 GHz / 1.0 = 2.5 x 10^9 instruction/second
- P3: 4 GHz / 2.2 = 1.8 x 10^9 instruction/second

One more Example - Continued

b. If the processors each execute a program in 10 seconds, find the number of cycles and the number of instructions.

Total cycles = Clock_rate * Execution time;

- Cycles on P1 = $3*10^9 *10 = 3*10^{10}$;
- Cycles on P2 = $2.5*10^9 *10 = 2.5*10^{10}$
- Cycles on P3 = $4*110^9 *10 = 4*10^{10}$

Instruction Count = Total_cycles/CPI

- Instruction count on P1 = $3*10^{10}/1.5 = 2*10^{10}$
- Instruction count on P2 = $2.5*10^{10}/1 = 2.5*10^{10}$
- Instruction count on P3 = $4*10^{10}/2.2 = 1.8*10^{10}$

One more Example - Continued

c. We are trying to reduce the execution time by 30%, but this leads to an increase of 20% in the CPI. What clock rate should we have to get this time reduction?

Execution time = (Instruction_count)*(CPI)/(Clock_rate)

- New execution time = 0.7*(original execution time)
  = (IC)*1.2*(original CPI)/(new clock rate)
- New clock rate should be 1.71 times the original clock rate
- So P1 should run at: 3*1.71 =5.14 GHz
- P2 should run at: 1.71*2.5 = 4.286 GHz
- P3 should run at: 4.0*1.71 = 6.86 GHz

Reporting Performance Data

How to report performance data?

- Execution time for one program
- Execution times for MANY programs
- Average execution time across all programs
- Arithmetic Mean (Assuming n programs) $= \frac{1}{n}\sum_{i=1}^{n}(Time)_i$
- Weighted Arithmetic Mean (weigh program by its frequency or importance) $= \sum_{i=1}^{n}(Weight)_i * (Time)_i$

We can also consider using "normalized" execution times (or relative execution times)

- That is, use execution times on some standard machine
- Then take ratio of execution times of your machine with the standard machine

Reporting Performance Data

We can use arithmetic mean (average) of the normalized ratios. Authors recommend using Geometric Means (what is this?)

- $\sqrt[n]{\prod_{i=1}^{n}(Relative\_execution\_time)_i}$
- Execute a wide variety of "benchmark" programs
- Find Geometric mean (need a reference system to compare)

For example, use SPEC CPU2006 benchmarks (CPUInt and CPUFloat)

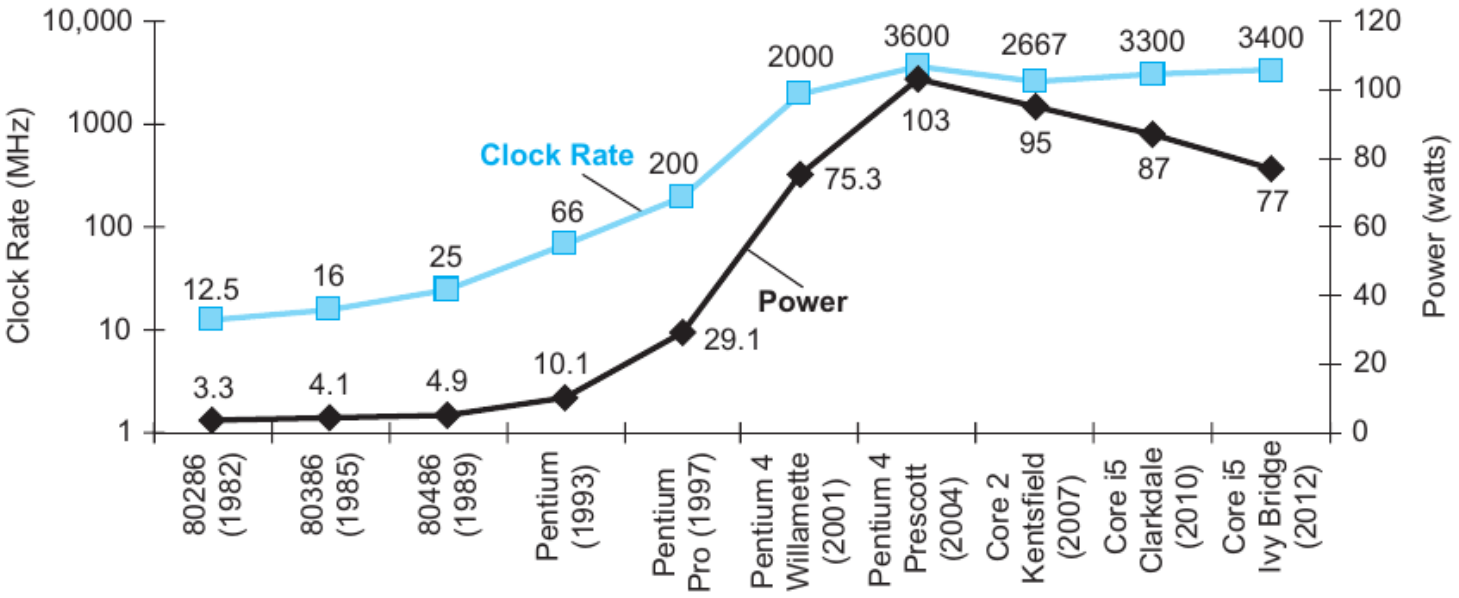
- Focus on CPU performance (almost no I/O)
- Created by Standard Performance Evaluation Corp (SPEC)
- Develops benchmarks for CPU, I/O, Web, ...

CINT2006 for Intel Core i7 920

Reference time → some known CPU (VAX)

Description	Name	Instruction Count x 10^9	CPI	Clock cycle time (seconds x 10^{-9})	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2252	0.60	0.376	508	9770	19.2
Block-sorting compression	bzip2	2390	0.70	0.376	629	9650	15.4
GNU C compiler	gcc	794	1.20	0.376	358	8050	22.5
Combinatorial optimization	mcf	221	2.66	0.376	221	9120	41.2
Go game (AI)	go	1274	1.10	0.376	527	10490	19.9
Search gene sequence	hmmer	2616	0.60	0.376	590	9330	15.8
Chess game (AI)	sjeng	1948	0.80	0.376	586	12100	20.7
Quantum computer simulation	libquantum	659	0.44	0.376	109	20720	190.0
Video compression	h264avc	3793	0.50	0.376	713	22130	31.0
Discrete event simulation library	omnetpp	367	2.10	0.376	290	6250	21.5
Games/path finding	astar	1250	1.00	0.376	470	7020	14.9
XML parsing	xalancbmk	1045	0.70	0.376	275	6900	25.1
Geometric mean	–	–	–	–	–	–	25.7

Power Trends



$$Power \propto 1/2 \times Capacitive\,load \times Voltage^2 \times Frequency\,switched$$

Reducing power: reduce voltage, reduce frequency or capacitive load

Reducing Power

Suppose a new CPU has:

- 85% of capacitive load of old CPU
- 15% voltage reduction and 15% frequency reduction
- A reduction in voltage normally requires proportional reduction in clock frequency
- $$\frac{P_{new}}{P_{old}} = \frac{C_{old} * 0.85 * (V_{old})^2 * F_{old} * 0.85}{C_{old} * (V_{old})^2 * F_{old}} = 0.85^4 = 0.52$$
- New CPU requires only 52% as much power as the old CPU

The Power Wall

- We can't reduce voltage further
- We can't remove more heat

How else can we improve performance?

An Example of Reducing Power

1.8 The Pentium 4 Prescott processor, released in 2004, had a clock rate of 3.6 GHz and voltage of 1.25 V. Assume that, on average, it consumed 10 W of static power and 90 W of dynamic power.

The Core i5 Ivy Bridge, released in 2012, has a clock rate of 3.4 GHz and voltage of 0.9 V. Assume that, on average, it consumed 30 W of static power and 40 W of dynamic power.

What is the capacitive load:
$(Dynamic)Power = (1/2) * (capacitiveload) * (Voltage)^2 * (clockfrequency)$

- Pentium 4 Prescott
  $90 = (1/2) * C * 1.25^2 * 3.6 * 10^9 \Rightarrow \text{load} = 32 * 10^{-9}\text{farads}$
- Core i5 Ivy Bridge
  $40 = (1/2) * C * 0.9^2 * 3.4 * 10^9 \Rightarrow \text{load} = 29 * 10^{-9}\text{farads}$

Notice that Ivy Bridge reduced voltage and capacitive load to achieve lower DYNAMIC power

An Example of Reducing Power

Find the percentage of the total dissipated power comprised by static power and the ratio of static power to dynamic power for each technology.

- Pentium: total power = 90+10; Static power % = 10/100 = 10%
- Ivy bridge: total power= 30+40; Static power % =30/70 = 42.8%
- static to dynamic power ratio:
  - Pentium = 10/90 =11%
  - Ivy bridge = 30/40 = 75%

  Newer systems have higher static power!

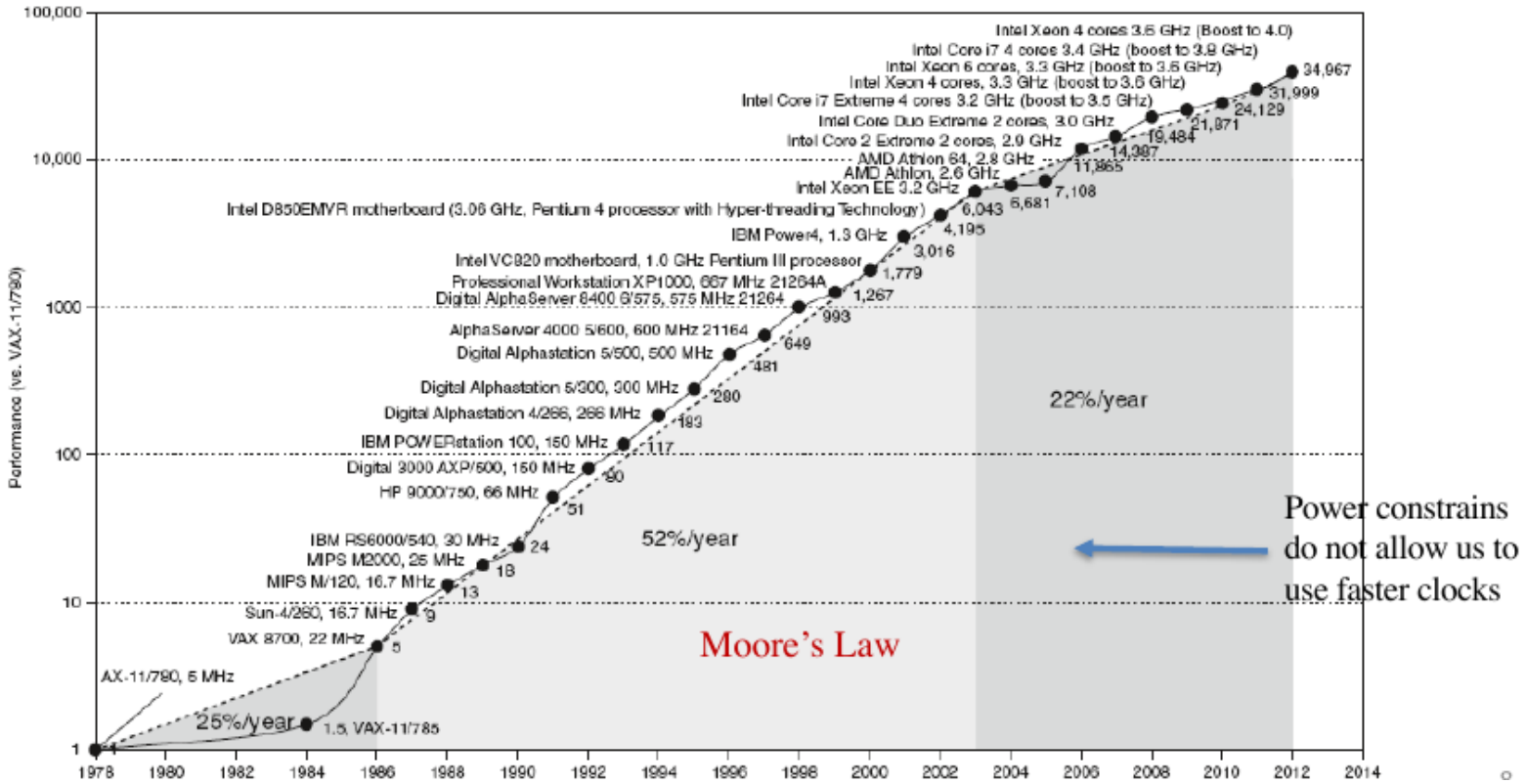
SPEC Power rating: Power consumption of server at different workload levels

- Performance: ssj_ops/sec (ssj_ops =server side java operations)
- Power: Watts (Joules/sec)

Execute 10 different workloads, collect ssj_ops and power for each workload

Why we need multiprocessors?

- The power wall: We can't reduce voltage further and We can't remove more heat.
- How else can we improve performance? We may need to use "parallelism" - use multiple cores and multi-threading



Multiprocessors

- Multicore microprocessors: More than one processor per chip
- Requires explicit parallel programming (divide program across multiple threads, running on different cores)
  - Hard to program multiple cores
    - Programming for performance (multithreaded implementations)
    - Load balancing
    - Optimizing communication and synchronization
  - Compare with instruction level parallelism (out of order execution)
    - Hardware executes multiple instructions at once
    - Hidden from the programmer

More shocking is: multi-threaded implementations execute MORE instructions than single threaded implementations - parallelism overhead

- Each thread may need to replicate some initialization
- Need synchronization instructions (not needed in single threaded implementations)

An Example of Multi-processing

1.9 Assume for arithmetic, load/store, and branch instructions, a processor has CPIs of 1, 12, and 5, respectively. Also assume that on a single processor a program requires the execution of 2.56E9 arithmetic instructions, 1.28E9 load/store instructions, and 256 million branch instructions. Assume that each processor has a 2 GHz clock frequency.

Assume that, as the program is parallelized to run over multiple cores, the number of arithmetic and load/store instructions per processor is divided by $0.7 \times p$ (where p is the number of processors) but the number of branch instructions per processor remains the same.

- Ideally, we expect that if we have p processors, each processor executes $1/p$ times as many instructions as that of a single processor
- Here we are told that each processor executes $1/0.7 * p$ arithmetic and load instruction as a single processor and the same number of branch instructions no matter how many processors

An Example of Multi-processing

Find the total execution time for this program on 1, 2, 4, and 8 processors, and show the relative speedup of the 2, 4, and 8 processors result relative to the single processor result.
Let us start with single processor case

- $2.56 * 10^9$ arithmetic instructions and each takes 1 cycle
- $1.28 * 10^9$ Load/Store instructions and each takes 12 cycles
- $256 * 10^6$ Branch instructions and each takes 5 cycles

- Total number of instructions executed $= 4,096 * 10^6$
- $CPI = [2.56 * 10^9 * 1 + 1.28 * 10^9 * 12 + 256 * 10^6 * 5]/4096 * 10^6 = 4.6875$
- Execution time = (instruction count)* CPI / (clock rate) = $(4096 * 10^6 * 4.6875)/2 * 10^9 = 9.6 sec$

An Example of Multi-processing

Now consider the case with 8 processors (I will leave it to compute for 2 and 4 processors)

- Number of instructions executed by each processor
  - $(2.56 * 10^9)/(0.7 * 8) = 457 * 10^6$ arithmetic instructions and each takes 1 cycle
  - $(1.28 * 10^9)/(0.7 * 8) = 228.6 * 10^6$ load/store instructions and each takes 12 cycles
  - $256 * 10^6$ branch instructions and each takes 5 cycle
  - Total instructions executed by EACH of the 8 processors $= 941.6 * 10^6$
- $CPI = [457 * 10^6 * 1 + 228.6 * 10^6 * 12 + 256 * 10^6 * 5]/941.6 * 10^6 = 4.758$
- Execution time = (instruction count)* CPI / (clock rate)
  $= (941.6 * 10^6 * 4.758)/2 * 10^9 = 2.24 sec$

An Example of Multi-processing

Ideally: if we have 8 processors, the program should complete in 1/8 the time it took on 1 processor:

- Speed up = (execution time on 1 processor)/(execution time on p processors)
- Speed up should be 8 with 8 processors

But we got: speed up = (9.6 sec)/(2.24 sec) = 4.29
The rest of the question will be your homework.

Amdahl's Law

What is Amdahl's law?

- Improving an aspect of a computer and expecting a proportional improvement in overall performance
- $T_{improved} = \frac{T_{affected}}{Improvement\ Factor} + T_{unaffected}$
- $Speedup = (T_{original}/T_{improved})$

An Example: We want to improve the performance for web browsing. We can improve the performance for this functionality by a factor of 10. We are told that the actual computation for web browsing is 40% of all the total computations – that is we are only improving 40% of the execution time by a factor of 10.

- New execution time $= 0.6 + 0.4/10 = 0.64$
- Speed up $= 1/[0.64] = 1.5625$

Only 56% speed up

Amdahl's Law

Another Example: Suppose 80% of all instructions are multiply instructions, How fast should we make multiply so that we can get an overall speedup of 5?

- $T_{improved} = \frac{T_{affected}}{(improvementfactor)} + T_{unaffected} = \frac{0.8}{n} + 0.2$
- Speedup $= 5 = \frac{T_{original}}{T_{improved}} = \frac{1}{(0.8/n) + 0.2}$
  becomes $0.2n = 0.8 + 0.2n$
- **This cannot be done** (5 times speed up cannot be achieved no matter how fast you make the multiply)

Amdahl's Law

Another Example: 25% of all operations are Floating Point; 2% are FPSQR All other operations (integer arithmetic, Load/Store, Branch) represent 75%

- FPSQR takes 20 cycles (CPI)
- Other FP operations take 4 cycles (CPI)
- Remaining operations take 1.33 (CPI)

Two improvements proposed:

- (a) Reduce FPSQR to 2 cycles - An improvement of 10 times
- (b) Reduce all FP operations (except FPSQR) to 2 cycles - improvement of 2 times

Which one is better (reduces execution time)?

Amdahl's Law

Since we do not know the total number instructions executed or clock frequency, we compare CPI

- Original CPI = 0.23*4+0.02*20+0.75*1.33 = 2.3175
- (a) = 0.23*4+0.02*2+0.75*1.33 = 1.9575
  improvement ratio = 1.184
- (b) = 0.23*2 +0.02*20+0.75*1.33 =1.8575
  improvement ratio = 1.248

So option b is better

What does all this mean?

MAKE COMMON CASE FASTER: Improve the performance of something that is heavily used

e.g., Divide is not that common, SqRt is not common

CSCI516 - Fundamental Concepts of Computing / Machine Organization
Chapter 2: Instructions: Language of the Computer

Song Huang, Ph.D.
Song.Huang@tamuc.edu

Instruction Sets

- We need to understand the operations and operands (data containers)
- We need to understand the data types permitted

Different types of operations:

- Arithmetic: Add, Subtract, Multiply, Divide
  - Integer: Possibly byte, 16-bit, 32-bit and 64-bit versions
  - Floating point: possibly single (32-bit) and double (64-bit) precision
- Logic:
  - Shift (left or right) — two types: arithmetic and logical shift
  - And, Or, Exclusive-Or, Not
- Compare
  - Can be implied by Add or Subtract
  - Equal, Not Equal, Less Than, Greater Than, LE, GE

Instruction Sets

Different types of operations: (continued)

- Branch Instructions (jump to some other point in the program if condition is true)
  - Unconditional Branch (like go to)
  - Conditional Branch: Branch if equal, Branch if Negative
    Needed to implement if...then...else; case (or switch); loops
  - Procedural Call and Return
- Moving Data
  - From/To Memory: Load and Store
  - Between Registers
  - From/To Program Counter and Stack Pointer
- Operands: data containers not the actual values to be operated on
  - We have data in memory (such as program variable)
  - We have data in registers
  - We have constants (literals or immediate values)
  - the instruction itself is the data container

The ARMv8 Instruction Set

- A subset, called LEGv8 (Lessen Extrinsic Garrulity), used throughout the book
- There are some differences between LEGv8 and ARMv8 - only ARMv8 works on DS5
- ARMV8 has many more instructions than LEGv8
- Commercialized by ARM Holdings (www.arm.com)
- Typical of many modern ISAs - See ARM Reference Data tear-out card

How programs executes:

- Stored program computers: Your program (machine language) must be first stored in memory (or DRAM). The code will migrate to instruction cache
- The program will be executed one instruction at a time
  - Program Counter indicates which instruction to execute
  - (PC contains the address of the instruction to execute)

The ARMv8 Instruction Set

- Instructions operate on data values: Data is stored in memory at specified memory addresses
- We need to know the address of data to access those values of variables

ARM only performs operations on registers: two source registers and one destination register

- We need to move the data from a memory address to a register, We will use load (LDUR) instruction for this

```
LDUR X1, [X2, #8]   // memory address is given by (X2) + 8
    Read that memory location, and place the data from memory into register X1
```

- Likewise we can move data from a register to memory using store instruction

```
STUR X1, [X2, #8]   // store X1 at memory address (X2) + 8
```

If we want to translate A = A+1;

```
LDUR X1, [X2, #0]      // assume address of A is in X2
ADD X1, X1, #1         // Add 1 to X1, result in X1
STUR X1, [X2, #0]      //store X1 back to variable A
```

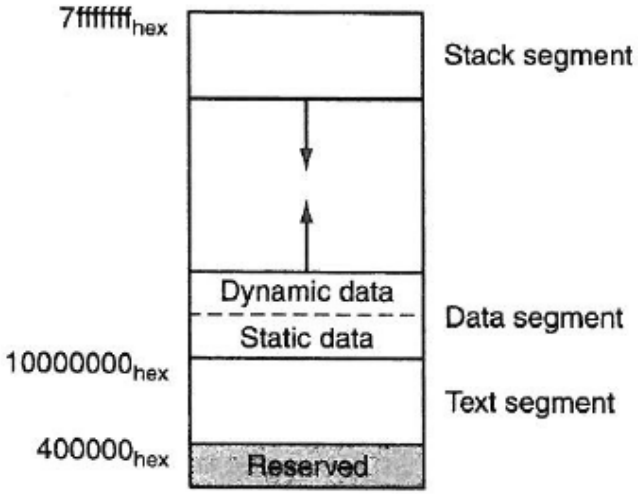
The ARMv8 Instruction Set

The program in test.S file to be used in help lab

```
        .data                       // data section of your program for static variables
        .type  v, %object           // declare v as a data object
        .size  v, 3                 // v is an array with 3 elements
v:      .xword 1, 2, 3              // initialize array elements to 1, 2, 3
        .text                       //  code starts here  (code normally called text)
        .global main                // declaring main as a globally visible function
        .arch armv8-a+fp+simd       // declare which instruction set we are using
        .type main, %function       // declaring main as a function – not a data item
main:                               //start of code for main
        ADRP X0, v                  // get the address of variable v and store it in X0
        ADD X0, X0, :lo12:v         // too complicated to explain now
        LDUR X1, [X0, #0]           // access data at memory location (X0)+#0 or v[0] and store inX1
        LDUR X2, [X0, #8]           //  access data at memory location (X0)+#8 or v[1] and store inX2
        ADD X1, X1, X2              // X1 = X1+X2 = v[0]+v[1]
        LDUR X2, [X0, #16]          // access data at memory location (X0)+#16 or v[2] and store inX2
        ADD X1, X1, X2              //X1 = X1+X2 = X1 + v[2] = v[1]+v[2]+v[3]
Exit:
```

Memory Layout - Physical / Virtual Memory

- Program instructions must be stored in memory
- actually the same memory contains both instructions and data



Dynamic data relates to malloc, new, etc

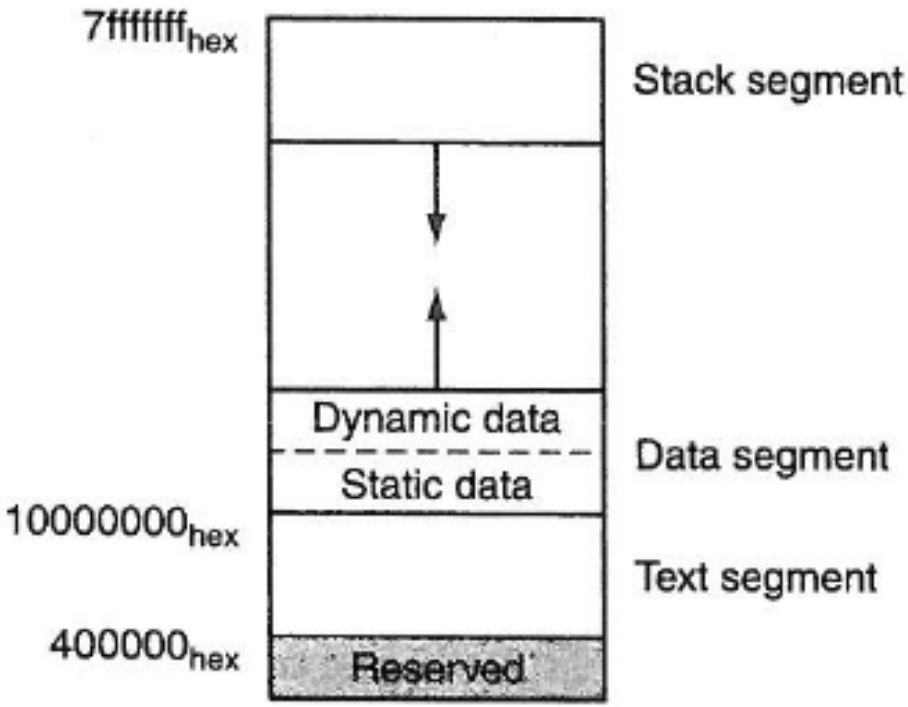
Text segment = your code

All these addresses are "virtual"
At runtime, physical pages (or DRAM addresses) are allocated for the virtual pages

- So we need to understand the difference between a virtual and physical address
- Addresses generated by your program (either instruction address, or address generated by a Load or Store) are virtual addresses

Memory Layout – Virtual Memory

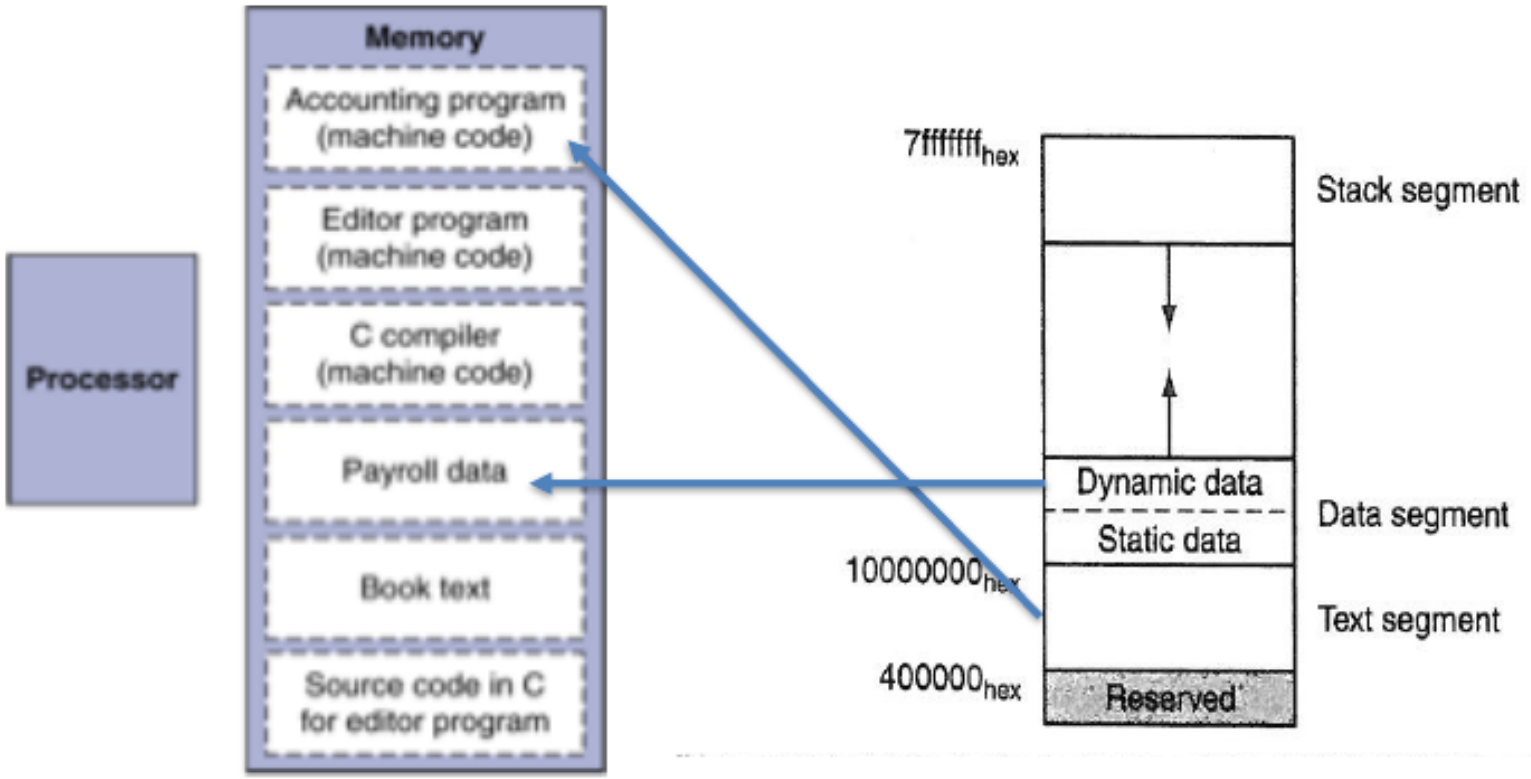
Program instructions must be stored in memory - actually the same memory contains both instructions and data



- Reserved: store kernel code.
- Text: binary image of the process. your code.
- Static data: stores global variables, static variables, constant arrays and strings.
- Dynamic data (Heap): stores dynamic variables allocated by the programmers.
- Stack: it is program stack. It stores automatic variables (local to a function's scope), caller's return address, etc.

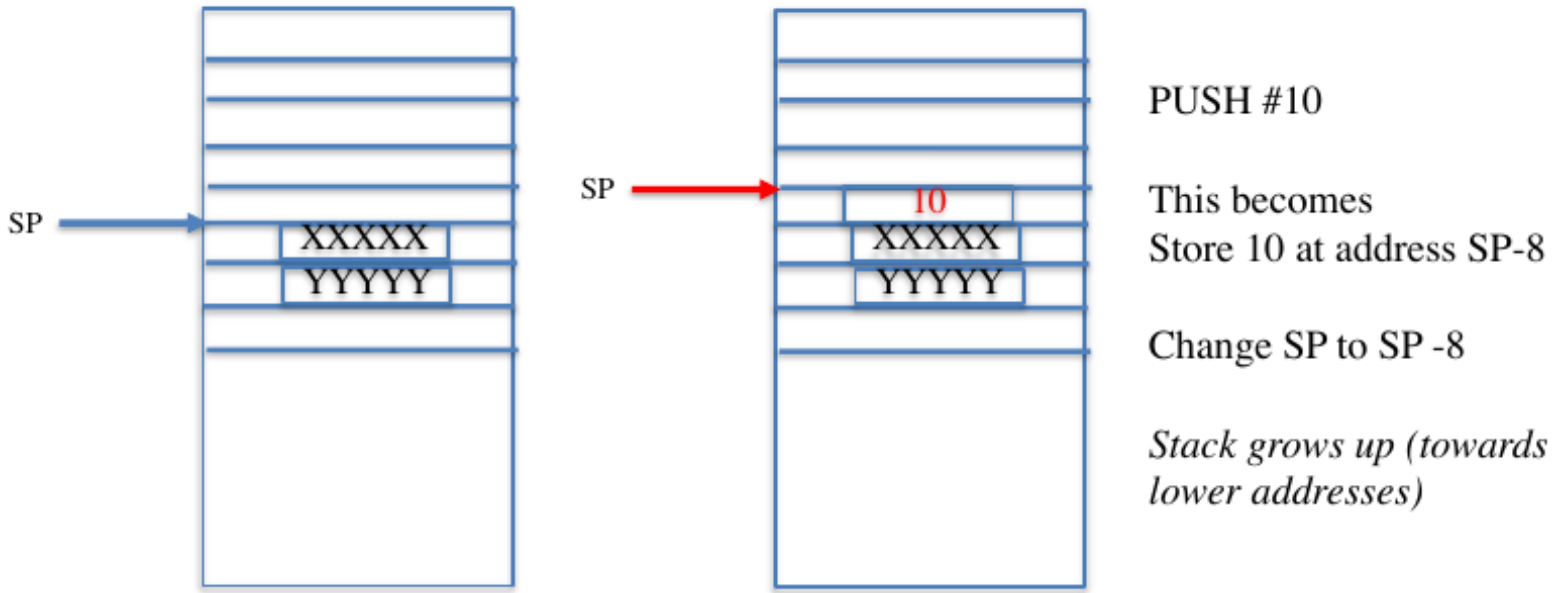
Memory Layout

DRAM is shared b many users. So your text, data, stack, heap must be mapped to DRAM along with others.



Stack and Operations

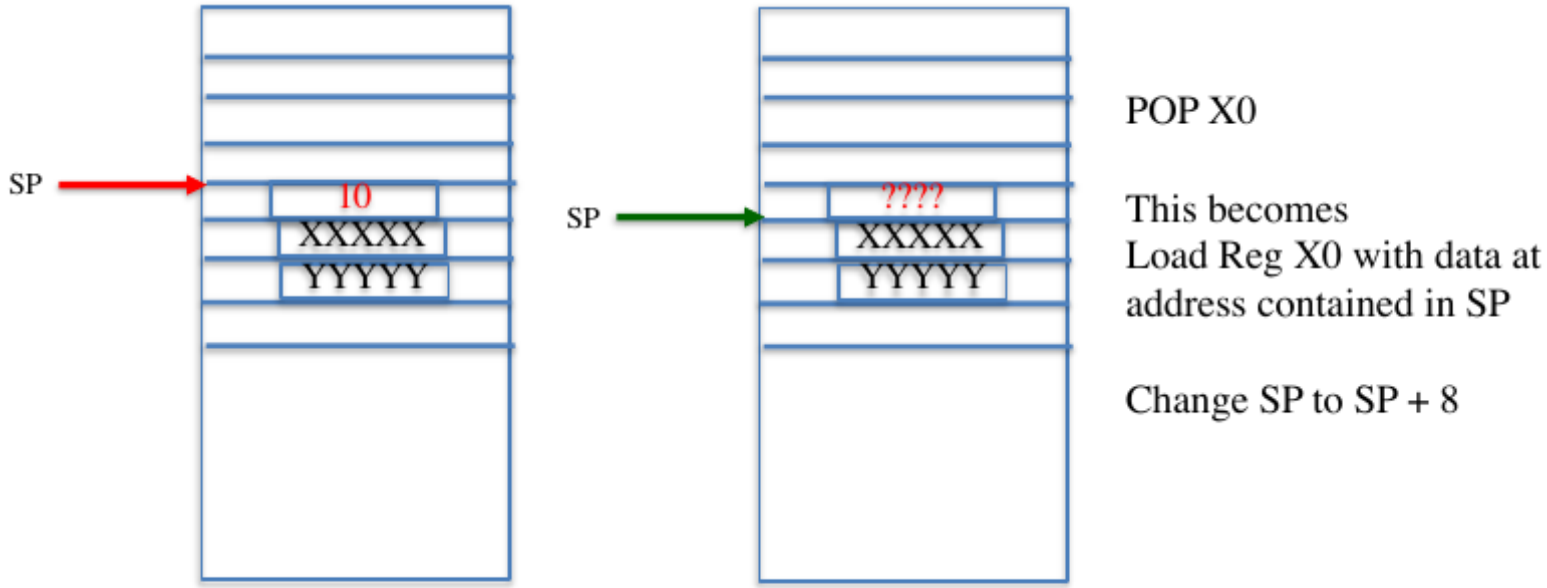
In the following Diagram, address zero is at the top. Stack Pointer - to keep track of the top of the stack
Limited operations: increment or decrement corresponding to Pop and Push



Stack grows towards lower address, so we need to subtract 8 from stack to push a new item.

Stack and Operations

Stack Pointer - to keep track of the top of the stack
Limited operations: increment or decrement corresponding to Pop and Push



In ARMv8 (and LEGv8): One of the general purpose register (X28) is designated as the Stack Pointer. Some machines may have a separate and dedicated Stack Pointer

Stack grows towards lower address, so we need to add 8 to Stack to pop a element.

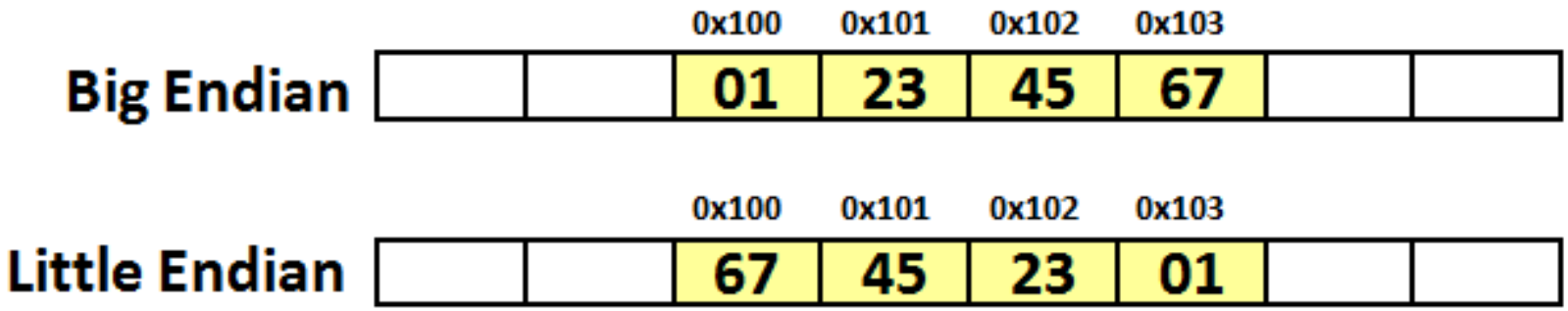
How does memory look like?

An Array:

- If we have a 64 bit architecture, each memory location address is incremented by 8 (8 bytes)
- This is important for “data alignment“
- Architectures differ in how bytes in a memory location are numbered
  Big Endian vs Little Endian.

Little-endian and Big-endian

if you have a number to store: 0x01234567, there are two orders: little-endian, and big-endian.
(Look at the memory address and offset, the order to store.)



How data is stored in memory

Conside an Array: long long int b[8]; 64 bit data items

- Each element of the array is 8 bytes.
- Total size of the object b is 64 bytes.
- In general, the address of b[j] is given by address of b[0] + (j)* 8.
- what if the array contains long int's (32 bit per element)? address of b[j] = b[0] + (j) * 4

Arithmetic Operations

- C code:

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

- Compiled LEGv8 code (Not the actual code):

```
ADD t0, g, h      # temp t0 = g + h
ADD t1, i, j      # temp t1 = i + j
SUB f, t0, t1     # f = t0 - t1
```

We can't use variable names like f, g, h, i in instructions

- We can only use Registers with these (arithmetic) instructions
- So, we need to first bring data from variables into registers (using Load instructions)
- Registers are fast than DRAM and thus it is better to perform arithmetic with registers

Register Operand Example

C code example:

- f = (g + h) -(i+ j);
- f, g, h, i, and j in X19, X20, X21, X22, and X23

Compiled LEGv8 code:

- ADD X9, X20, X21
- ADD X10, X22, X23
- SUB X19, X9, X10

Operations

- Arithmetic: Add, Subtract, Multiply, Divide
  - Integer: Possibly byte, 16-bit, 32-bit and 64-bit versions
  - Floating point: possibly single (32-bit) and double (64-bit) precision
- Logic: Shift (left or right)–two types: arithmetic and logical shift, And, Or, Exclusive-Or, Not
- Compare: Can be implied by Add or Subtract, Equal, Not Equal, Less Than, Greater Than, LE, GE
- Branch Instructions (jump to some other point in the program if condition is true)
  - Unconditional Branch (like go to)
  - Conditional Branch: Branch if equal, Branch if Negative,if then else, case (or switch), loops, Procedural Call and Return
- Moving Data
  - From/To Memory: Load and Store
  - Between Registers
  - From/To Program Counter and Stack Pointer

Operands

- Operands: data containers not the actual values to be operated on.
  - We have data in memory (such as program variable)
  - We have data in registers
  - We have constants (literals or immediate values)
- 64-bit data is called a "doubleword", 31 x 64-bit general purpose registers X0 to X30.
- 32-bit data called a "word". 31 x 32-bit general purpose sub-registers W0 to W30.

Registers

- General purpose: can be used by users to hold data or temporary values
  - Integer Register: can hold both integer data (positive and negative) and memory addresses. (32 bit - W registers, 64 bit - X registers)
  - Floating point: single precision(32 bit - F registers) or double precision (64 bit - D registers)
  - Vector (SIMD) registers: each register can hold up to 4 array elements (64*4=256 bits)
- Program Counter: Contain the address of the next instruction to execute.
  - Limited operations allowed by user program.
  - Can save the current value from PC to some other register or memory (stack)
  - Can load PC with a value from another register or from memory (return address)

Register vs. Memory

Register vs. Memory

- Register are faster to access than memory
- Operating on memory data require loads and stores.
- Compiler must use registers for variables as much as, only spill to memory for less frequently used variables.

Memory Operands:

- Memory used for composite data: Array, structures, dynamic data.
- To apply arithmetic operations:
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed, each address identifies an 8-bit byte

Memory Operand Example

You need to tell the register stores Value or Address

Example 1:

C code: A[2] = h + A[1]

- h in X21, means X21 stores the value of h
- base address of A in X22, means X22 stores the base address of A.

Compiled LEGv8 code: index of 1 requires offset of 8.

- LDUR X9, [X22, #8] // load value from memory to register
- ADD X9, X21, X9 // add values (h + A[1])
- STUR X9, [X22[1], #16[2]] // store value (h + A[1]) to memory

[1] X22 - Address (base address of the array)

[2] #16 - immediate (value for the offset)

Memory Operand Example

Example 2:
C code: A[12] = h + A[8]

- h in X21, means X21 stores the value of h
- base address of A in X22, means X22 stores the base address of A.

Compiled LEGv8 code: index of 1 requires offset of 8.

- LDUR X9, [X22, #64[3]] // load value from memory to register
- ADD X9, X21, X9 // add values (h + A[1])
- STUR X9, [X22, #96[4]] // store value (h + A[1]) to memory

[1] #64 - immediate (value for the offset: 8 * 8)
[2] #16 - immediate (value for the offset: 12 * 8)

Immediate Operands

Example: add the constant 4 to register X22

Code:

LDUR X9, [X20, Address of Constant 4] // X9 = constant 4
ADD X22, X22, X9 // X22 = X22 + X9 (X9 stores 4)

Alternative solution:

ADDI X22, X22, #4 // X22 = X22 + 4 (not for ARMv8)

Make the common case fast

- Small constants are common.
- Immediate operand avoids a load instruction.

Program Example

Assembly code for $\sum_{n=0}^{a} n!$: The input a is in X19 and the input is 64-bits, nonnegative integer less than 10. Store the result y in X20.

```
        .text
        .global main
        .arch armv8-a+fp+simd
        .type main, %function

main:

        MOV X20, #1             // X20 stores 1 as an initialized value (for sum)
        MOV X10, #1             // set X10 as a temp variable (for temp)
        MOV X11, #1             // X11 stores the value of n

loop:
        CMP X11, X19            // compare n with a, X19 stores the value of a
        BGT Exit                // if n is greater than a, then just out of loop.

        MUL X10, X10, X11       // temp = temp * n
        ADD X20, X20, X10       // sum = sum + temp

        ADD X11, X11, #1        // n = n + 1
        B loop                  // go back to loop

Exit:
```

CSCI516 - Fundamental Concepts of Computing / Machine Organization

Song Huang, Ph.D.
Song.Huang@tamuc.edu

Unsigned Binary Integers

All the bits represent number, there is no sign bit.

- Given an n-bit number:
  $X = X_{n-1}2^{n-1} + X_{n-2}2^{n-2} + ... + X_1 2^1 + X_0 2^0$
- Range: 0 to $+2^n - 1$
- Example: 0000 0000 0000 0000...0000 0000 0000 $1011_2 =$
  $0 + ... + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 0 + ... + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits: 0 to 4,294,967,295
- Using 64 bits: 0 to 8,446,744,073,709,551,615

2s-Complement Signed Binary Integers

There is a sign bit at the most significant bit.

- Given an n-bit number:
  $X = -X_{n-1}2^{n-1} + X_{n-2}2^{n-2} + ... + X_1 2^1 + X_0 2^0$
- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example:
  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2 = -1 * 2^{31} + 1 * 2^{30} + ... + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 = -2,147,483,648 + 2,147,483,644 = -4_{10}$
- Using 32 bits: -2,147,483,648 to 2,147,483,647
- Using 64 bits: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

2s-Complement Signed Binary Integers

- Bit 31 is sign bit:
  - 1 for negative numbers
  - 0 for non-negative numbers
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some Specific numbers
  - 0: 0000 0000...0000
  - -1: 1111 1111...1111
  - Most-negative: 1000 0000...0000
  - Most-positve: 0111 1111...1111

Signed Negation

- Complement and add 1
  - complement means $1 \rightarrow 0$, $0 \rightarrow 1$
  - $X + \bar{X} = 1111...111_2 = -1$
  - $\bar{X} + 1 = -X$
- Example: negate + 2
  - $+2 = 0000\ 0000\ ...\ 0010_2$
  - $-2 = 1111\ 1111\ ...\ 1101_2 + 1 = 1111\ 1111\ ...\ 1110_2$
  - means $1 \rightarrow 0, 0 \rightarrow 1$

Signed Extension

- Representing a number using more bits: preserve the numeric value
- Replicate the sign bit to the left
  unsigned value: extend with 0s
- Examples: 8-bit to 16-bit
  - $+2 : 0000\ 0010 \rightarrow 0000\ 0000\ 0000\ 0010$
  - $-2 : 1111\ 1110 \rightarrow 1111\ 1111\ 1111\ 1110$
- in LEGv8 instruction set
  - LDURSB: sign-extend loaded byte
  - LDURB: zero-extend loaded byte

Hexadecimal

- Compact representation of bit strings
- 4 bits per hex digit

0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

- Example: ECA8 6420:
  1110 1100 1010 1000 0110 0100 0010 0000

Representing Instructions

- Instructions are encoded in binary (machine code)
- LEGv8 instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code(opcode), register numbers...

Different instruction formats:

- R-format: ADD X9, X20, X21
- D-format: LDUR X9, [X6, #8]
- I-format: ADDI X9, X9, #1
- B-format: B L1
- CB-format: CBNZ X19, exit

LEGv8 R-format Instructions

opcode (11 bits)	Rm (5 bits)	Shamt (6 bits)	Rn (5 bits)	Rd (5 bits)

Instruction fields:

- opcode: operation code
- Rm: the second register source operand
- shamt: shift amount (00000 for now)
- Rn: the first register source operand
- Rd: the register destination

Example: ADD X9, X20, X21

1112_{ten}	21_{ten}	0_{ten}	20_{ten}	9_{ten}

$1000\ 1011\ 0001\ 0101\ 0000\ 0010\ 1000\ 1001_{two} = 8B150289_{16}$

LEGv8 D-format Instructions

opcode (11 bits)	address (9 bits)	op2 (2 bits)	Rn (5 bits)	Rt (5 bits)

Load/Store instructions:

- Rn: base register
- address: constant offset from contents of base register
- Rt: destination (load) or source (store) register number

LEGv8 I-format Instructions

opcode (10 bits)	immediate (12 bits)	Rn (5 bits)	Rd (5 bits)

Immediate instructions:

- Rn: source register
- Rd: destination register

Example

```
A[30] = h + A[30] + 1;
```

is compiled into

```
LDUR X9, [X10,#240] // Temporary reg X9 gets A[30]
ADD  X9,X21,X9      // Temporary reg X9 gets h+A[30]
ADDI X9,X9,#1       // Temporary reg X9 gets h+A[30]+1
STUR X9, [X10,#240] // Stores h+A[30]+1 back into A[30]
```

opcode	Rm/address	shamt/op2	Rn	Rd/Rt
1986	240	0	10	9
1112	9	0	21	9
580	1		9	9
1984	240	0	10	9

Binary format of the instructions:

11111000010	011110000	00	01010	01001
10001011000	01001	000000	10101	01001
1001000100	000000000001		01001	01001
11111000000	011110000	00	01010	01001

Logical Operations

Logical operations	C operators	Java operators	LEGv8 instructions
Shift left	<<	<<	LSL
Shift right	>>	>>>	LSR
Bit-by-bit AND	&	&	AND,ANDI
Bit-by-bit OR	|	|	OR,ORI
Bit-by-bit NOT	~	~	EOR,EORI

Example:

opcode (11 bits)	Rm (5 bits)	Shamt (6 bits)	Rn (5 bits)	Rd (5 bits)

- shamt: how many positions to shift
- shift left logical: multiplies by 2^i
- shift right logical: divided by 2^i

AND Operation

Useful to mask bits in a word: select some bits, clear others to 0
Example: AND X9, X10, X11

register X11 contains

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00001101\ 11000000_{two}$

and register X10 contains

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00111100\ 00000000_{two}$

then, after executing the LEGv8 instruction

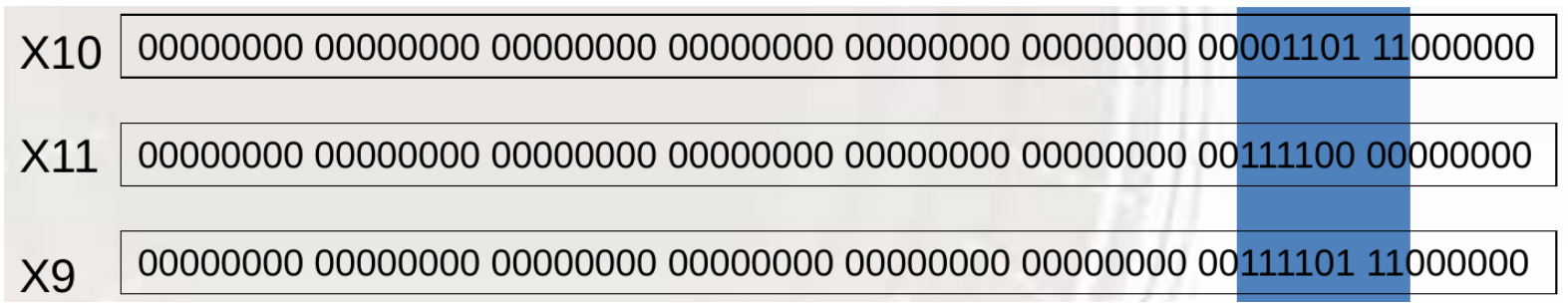
```
AND X9,X10,X11      // reg X9 = reg X10 & reg X11
```

the value of register X9 would be

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00001100\ 00000000_{two}$

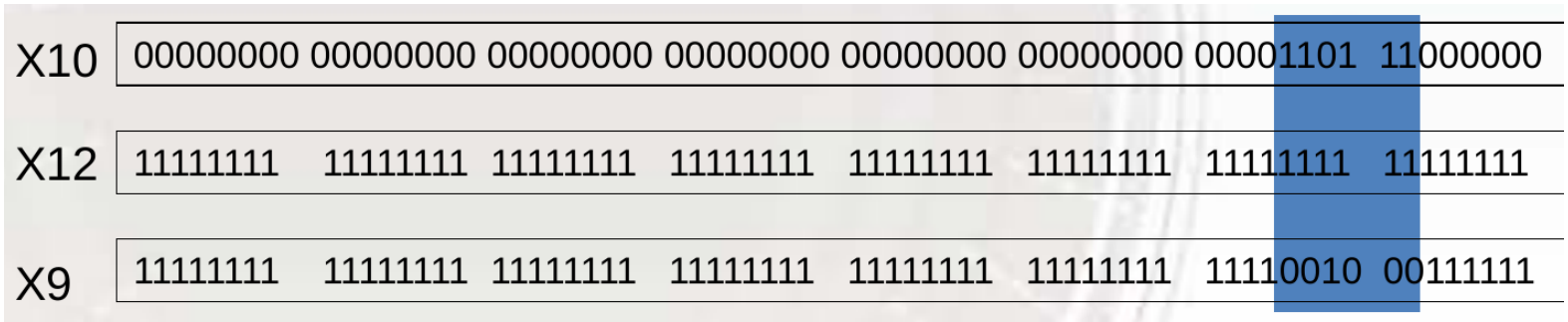
OR Operation

Useful to include bits in a word: set some bits to 1, leave others unchanged
Example: OR X9, X10, X11



EOR Operation

Differencing operation: set some bits to 1, leave others unchanged
Example: EOR X9, X10, X12

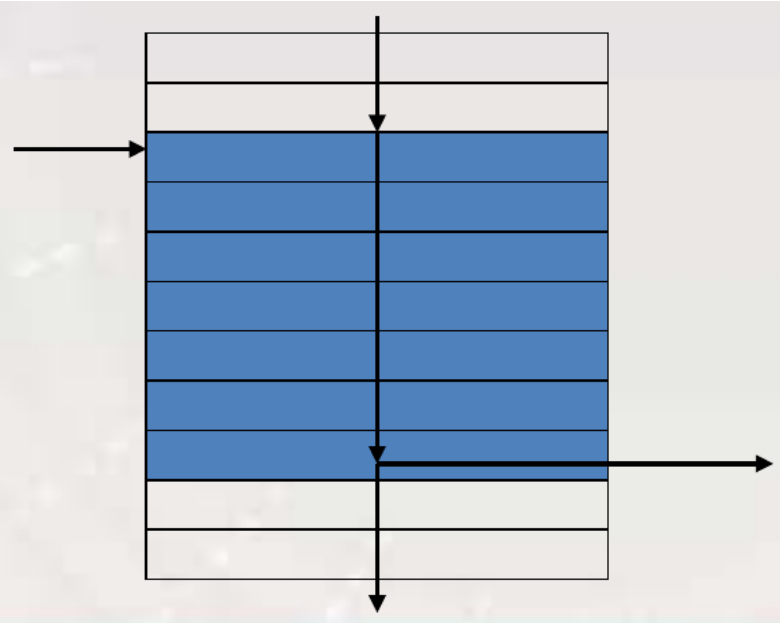


LEGv8 Register

- X0 - X7: procedure arguments/results
- X8: indirect result location register
- X9 - X15: temporaries
- X16 - X17 (IP0 - IP1): may be used by linker as a scratch
- register, other times as temporary register
- X18: platform register for platform independent code;
- otherwise a temporary register
- X19 - X27: saved
- X28 (SP): stack pointer
- X29 (FP): frame pointer
- X30 (LR): link register (return address)
- XZR (register 31): the constant value 0

Basic Blocks

- A basic block is a sequence of instructions with:
  - No embedded branches (except at end)
  - No branch targets (except at the beginning)



- A compiler identifies basic blocks for optimization.
- An advanced processor can accelerate execution of basic blocks

Branch Operations

Branch to a labeled instruction if a condition is true, otherwise, continue sequentially

- CBZ register, L1:
  if (register == 0) branch to instruction labeled L1;
- CBNZ register, L1:
  if (register != 0) branch to instruction labeled L1;
- B L1:
  branch unconditionally to instruction labeled L1;

B and CB Format

- B-type:
  B L1 // go to a specific location of L1

opcode (6 bits)	address (26 bits)

- CB-type:
  CBNZ X19, Exit // go to a specific location of Exit.

opcode (8 bits)	address (19 bits)	Rt (5 bits)

- Both addresses are PC-relative
  Address = PC + offset(from instruction)

Compiling if Statements

C code:

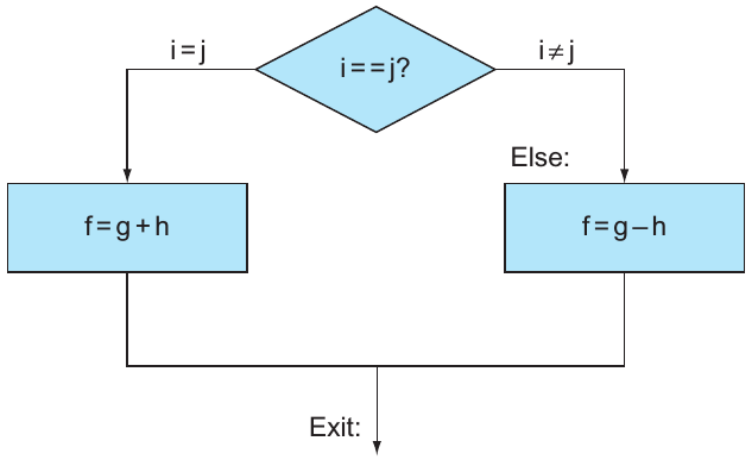
if(i == j) f = g + h;

else f = g -h;

—f, g, h, i, and j, in X19, X20, X21, X22, and X23

Compiled LEGv8 code:

```
    SUB X9, X22, X23
    CBNZ X9, Else
    ADD X19, X20, X21
    B Exit
Else: SUB X19, X20, X21
Exit:
```



Compiling for Statements

C code:

while(save[i] == k) i+= 1

—i in X22, k in X24, address of save in X25, save is 64-bit array

Compiled LEGv8 code:

```
Loop: LSL X10, X22, #3      // Multiply i*8
      ADD X10, X10, X25     // address of save[i]
      LDUR X9, [X10, #0]    // Load save[i]
      SUB X11, X9, X24      // check save[i] == k
      CBNZ X11, Exit        // If false, exit
      ADDI X22, X22, #1     // else i++
      B Loop                // Loop back
Exit: ...
```

More Conditional Operations

Condition codes, set from arithmetic instruction with S-suffix:
—(ADDS, ADDIS, ANDS, ANDIS, SUBS, SUBIS)

- negative(N): result had 1 in MSB
- zero(Z): result was 0
- overflow(V): result overflowed
- carry(C): result had carryout from MSB

Use subtract to set flags, then conditionally branch:

- B.EQ; B.NE
- B.LT (less than, siqned); B.LO(less than, unsigned)
- B.LE (less than or equal, signed); B.LS (less than or equal, unsigned)
- B.GT(greater than, signed); B.HI (greater than, unsigned)
- B.GE (greater than or equal, signed); B.HS (greater than or equal, unsigned)

Branch Instruction Design

C code:

$if(a > b) \quad a += 1$

—a in X22, b in X23

LEGv8 code:

```
        SUBS X9, X22, X23    // use usb to make comparison
        B.LE Exit            // conditional branch
        ADDI X22, X22, #1    // a++
Exit:   ...
```

For Loop Design

C code:

$for(i = 0; i < a; ++i)\ \ b[i] = a + i$

—a in X22, base address of array b in X23

LEGv8 code:

```
        ADDI X9, XZR, #0     // assuming i is in temp register X9
Loop:   SUBS X10, X9, X22    // Check for i ¡ a
        B.GE Exit            // If false exit else continue
        ADD X10, X22, X9     // Compute a + i
        LSL X11, X9, #3      // Multiply i*8 for 64-bit
        ADD X11, X23, X11    //compute address for b[i]
        STUR X10, [X11, #0]  // store the a + i in b[i]
        ADDI X9, X9, #1      // i++
        B Loop               // Loop back
Exit:   ...
```

Program Example

Assembly code for $\sum_{n=0}^{a} n!$: The input a is in X19 and the input is 64-bits, nonnegative integer less than 10. Store the result y in X20.
C++ code:

```
int result = 1;
int temp = 1;
for(int i = 1; i < a; ++i){
        temp = temp * i;
        result = result + temp;
}
```

Program Example

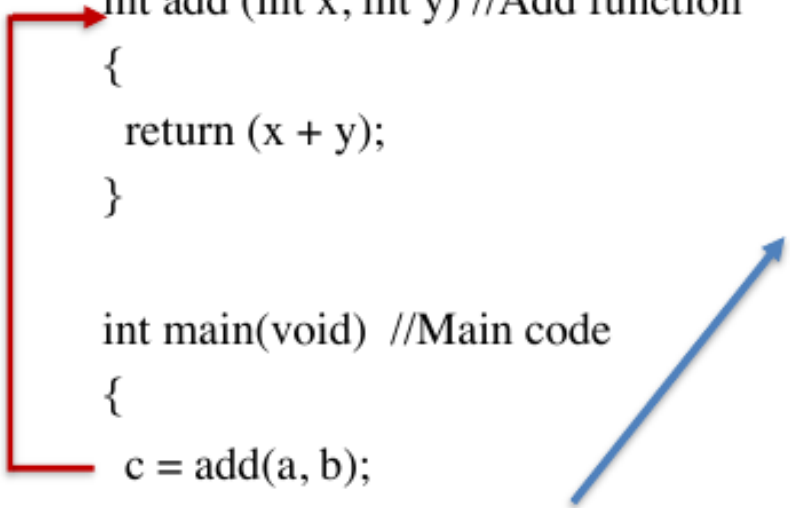
Assembly code for $\sum_{n=0}^{a} n!$: The input a is in X19 and the input is 64-bits, nonnegative integer less than 10. Store the result y in X20.

```
        .text
        .global main
        .arch armv8-a+fp+simd
        .type main, %function

main:

        MOV X20, #1             // X20 stores 1 as an initialized value (for sum)
        MOV X10, #1             // set X10 as a temp variable (for temp)
        MOV X11, #1             // X11 stores the value of n

loop:
        CMP X11, X19            // compare n with a, X19 stores the value of a
        BGT Exit                // if n is greater than a, then just out of loop.

        MUL X10, X10, X11       // temp = temp * n
        ADD X20, X20, X10       // sum = sum + temp

        ADD X11, X11, #1        // n = n + 1
        B loop                  // go back to loop

Exit:
```

Procedure / Function

C code:

```
#include "stdio.h"
int a=5, b=4, c;
int add (int x, int y) //Add function
{
  return (x + y);
}

int main(void)  //Main code
{
  c = add(a, b);
  printf("Result: %d\n", c);
  return 0;
}
```



Printf itself is a function managed by Operating System
In this class we will not deal with OS functions
So, when we write procedures in assembly language we will skip the printf or scanf type procedures

Procedure Calling

Steps required

1. Place parameters in registers X0 to X7
2. Transfer control to procedure
3. Acquire storage for procedure
4. Perform procedure's operations
5. Place result in register for caller
6. Return to place of call (address in X30)

Procedure Call Instructions

- Procedure call: **BL ProcedureLabel** (Branch and Link)
  - Address of following instruction put in X30
  - Jumps to target address
- Procedure return: **BL LR** (Branch Register)
  - Copies LR to program counter
  - Can also be used for computed jumps
    e.g. for case/switch statements

Procedure Example

```
        .data
        .type f, %object
        .size 1
        .type g, %object
        .size 1
        .type h, %object
        .size 1
        .type i, %object
        .size 1
        .type j, %object
        .size 1

f:      .xword

g:      .xword  5
h:      .xword 7
i:      .xword 9
j:      .xword 15
        .text
        .global main
        .global leaf
        .arch armv8-a+fp+simd
        .type main, %function
        .type leaf, %function
```

```
leaf:
        SUB SP, SP,#24
        STUR X10,[SP,#16]
        STUR X9,[SP,#8]
        STUR X19,[SP,#0]
        ADD X9,X0,X1
        ADD X10,X2,X3
        SUB X19,X9,X10
        ADD X0,X19,XZR
        LDUR X10,[SP,#16]
        LDUR X9,[SP,#8]
        LDUR X19,[SP,#0]
        ADD SP,SP,#24
        BR LR
```

```
main:
        ADRP X0, g
        ADD X0, X0, :lo12:g
        LDUR X0, [X0,#0]
        ADRP X1, h
        ADD X1, X1, :lo12:h
        LDUR X1, [X1,#0]
        ADRP X2, i
        ADD X2, X2, :lo12:i
        LDUR X2, [X2,#0]
        ADRP X3, j
        ADD X3, X3, :lo12:j
        LDUR X3, [X3,#0]
        BL leaf
        ADRP X1, f
        ADD X1,X1, :lo12:f
        STUR X0, [X1,#0]

Exit:
```

Note: the code for functions appear before the code for main