

# Lecture 3: MIPS Instruction Set

---

- Today's topic:
  - Wrap-up of performance equations
  - MIPS instructions
- HW1 is due on Thursday
- TA office hours posted

# A Primer on Clocks and Cycles

---

# Performance Equation - I

---

CPU execution time = CPU clock cycles x Clock cycle time  
Clock cycle time =  $1 / \text{Clock speed}$

If a processor has a frequency of 3 GHz, the clock ticks 3 billion times in a second – as we'll soon see, with each clock tick, one or more/less instructions may complete

If a program runs for 10 seconds on a 3 GHz processor, how many clock cycles did it run for?

If a program runs for 2 billion clock cycles on a 1.5 GHz processor, what is the execution time in seconds?

# Performance Equation - II

---

CPU clock cycles = number of instrs x avg clock cycles  
per instruction (CPI)

Substituting in previous equation,

Execution time = clock cycle time x number of instrs x avg CPI

If a 2 GHz processor graduates an instruction every third cycle,  
how many instructions are there in a program that runs for  
10 seconds?

# Factors Influencing Performance

---

Execution time = clock cycle time x number of instrs x avg CPI

- Clock cycle time: manufacturing process (how fast is each transistor), how much work gets done in each pipeline stage (more on this later)
- Number of instrs: the quality of the compiler and the instruction set architecture
- CPI: the nature of each instruction and the quality of the architecture implementation

# Example

---

Execution time = clock cycle time x number of instrs x avg CPI

Which of the following two systems is better?

- A program is converted into 4 billion MIPS instructions by a compiler ; the MIPS processor is implemented such that each instruction completes in an average of 1.5 cycles and the clock speed is 1 GHz
- The same program is converted into 2 billion x86 instructions; the x86 processor is implemented such that each instruction completes in an average of 6 cycles and the clock speed is 1.5 GHz

# Power and Energy

---

- Total power = dynamic power + leakage power
- Dynamic power  $\propto$  activity  $\times$  capacitance  $\times$  voltage<sup>2</sup>  $\times$  frequency
- Leakage power  $\propto$  voltage
- Energy = power  $\times$  time  
(joules)      (watts)      (sec)

# Example Problem

---

- A 1 GHz processor takes 100 seconds to execute a program, while consuming 70 W of dynamic power and 30 W of leakage power. Does the program consume less energy in Turbo boost mode when the frequency is increased to 1.2 GHz?



# Example Problem

---

- A 1 GHz processor takes 100 seconds to execute a program, while consuming 70 W of dynamic power and 30 W of leakage power. Does the program consume less energy in Turbo boost mode when the frequency is increased to 1.2 GHz?

Normal mode energy =  $100 \text{ W} \times 100 \text{ s} = 10,000 \text{ J}$

Turbo mode energy =  $(70 \times 1.2 + 30) \times 100/1.2 = 9,500 \text{ J}$

Note:

Frequency only impacts dynamic power, not leakage power. We assume that the program's CPI is unchanged when frequency is changed, i.e., exec time varies linearly with cycle time.

# Benchmark Suites

---

- Each vendor announces a SPEC rating for their system
  - a measure of execution time for a fixed collection of programs
  - is a function of a specific CPU, memory system, IO system, operating system, compiler
  - enables easy comparison of different systems

The key is coming up with a collection of relevant programs

# SPEC CPU

---

- SPEC: System Performance Evaluation Corporation, an industry consortium that creates a collection of relevant programs
- The 2006 version includes 12 integer and 17 floating-point applications
- The SPEC rating specifies how much faster a system is, compared to a baseline machine – a system with SPEC rating 600 is 1.5 times faster than a system with SPEC rating 400
- Note that this rating incorporates the behavior of all 29 programs – this may not necessarily predict performance for your favorite program!
- SPEC 2017 was released recently

# Deriving a Single Performance Number

---

How is the performance of 29 different apps compressed into a single performance number?

- SPEC uses geometric mean (GM) – the execution time of each program is multiplied and the  $N^{\text{th}}$  root is derived
- Another popular metric is arithmetic mean (AM) – the average of each program's execution time
- Weighted arithmetic mean – the execution times of some programs are weighted to balance priorities

# Amdahl's Law

---

- Architecture design is very bottleneck-driven – make the common case fast, do not waste resources on a component that has little impact on overall performance/power
- Amdahl's Law: performance improvements through an enhancement is limited by the fraction of time the enhancement comes into play
- Example: a web server spends 40% of time in the CPU and 60% of time doing I/O – a new processor that is ten times faster results in a 36% reduction in execution time (speedup of 1.56) – Amdahl's Law states that maximum execution time reduction is 40% (max speedup of 1.66)

# Common Principles

---

- Amdahl's Law
- Energy: performance improvements typically also result in energy improvements – less leakage
- 90-10 rule: 10% of the program accounts for 90% of execution time
- Principle of locality: the same data/code will be used again (temporal locality), nearby data/code will be touched next (spatial locality)

# Recap

---

- Knowledge of hardware improves software quality: compilers, OS, threaded programs, memory management
- Important trends: growing transistors, move to multi-core and accelerators, slowing rate of performance improvement, power/thermal constraints, long memory/disk latencies
- Reasoning about performance: clock speeds, CPI, benchmark suites, performance equations
- Next: assembly instructions

# Instruction Set

---

- Understanding the language of the hardware is key to understanding the hardware/software interface
- A program (in say, C) is compiled into an executable that is composed of machine instructions – this executable must also run on future machines – for example, each Intel processor reads in the same x86 instructions, but each processor handles instructions differently
- Java programs are converted into portable bytecode that is converted into machine instructions during execution (just-in-time compilation)
- What are important design principles when defining the instruction set architecture (ISA)?



# Instruction Set

---

- Important design principles when defining the instruction set architecture (ISA):
  - keep the hardware simple – the chip must only implement basic primitives and run fast
  - keep the instructions regular – simplifies the decoding/scheduling of instructions

We will later discuss RISC vs CISC

# A Basic MIPS Instruction

---

C code: `a = b + c ;`

Assembly code: (human-friendly machine instructions)

`add a, b, c     # a is the sum of b and c`

Machine code: (hardware-friendly machine instructions)

`00000010001100100100000000100000`

Translate the following C code into assembly code:

`a = b + c + d + e;`

# Example

---

C code     $a = b + c + d + e$ ;  
translates into the following assembly code:

add a, b, c		add a, b, c
add a, a, d	or	add f, d, e
add a, a, e		add a, a, f

- Instructions are simple: fixed number of operands (unlike C)
- A single line of C code is converted into multiple lines of assembly code
- Some sequences are better than others... the second sequence needs one more (temporary) variable f

# Subtract Example

---

C code     $f = (g + h) - (i + j);$

Assembly code translation with only add and sub instructions:

# Subtract Example

---

C code  $f = (g + h) - (i + j);$   
translates into the following assembly code:

add t0, g, h		add f, g, h
add t1, i, j	or	sub f, f, i
sub f, t0, t1		sub f, f, j

- Each version may produce a different result because floating-point operations are not necessarily associative and commutative... more on this later

# Operands

---

- In C, each “variable” is a location in memory
- In hardware, each memory access is expensive – if variable *a* is accessed repeatedly, it helps to bring the variable into an on-chip scratchpad and operate on the scratchpad (registers)
- To simplify the instructions, we require that each instruction (add, sub) only operate on registers
- Note: the number of operands (variables) in a C program is very large; the number of operands in assembly is fixed... there can be only so many scratchpad registers

# Registers

---

- The MIPS ISA has 32 registers (x86 has 8 registers) – Why not more? Why not less?
- Each register is 32-bit wide (modern 64-bit architectures have 64-bit wide registers)
- A 32-bit entity (4 bytes) is referred to as a word
- To make the code more readable, registers are partitioned as \$s0-\$s7 (C/Java variables), \$t0-\$t9 (temporary variables)...

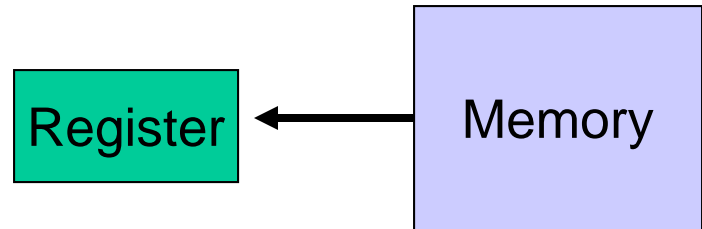
# Memory Operands

---

- Values must be fetched from memory before (add and sub) instructions can operate on them

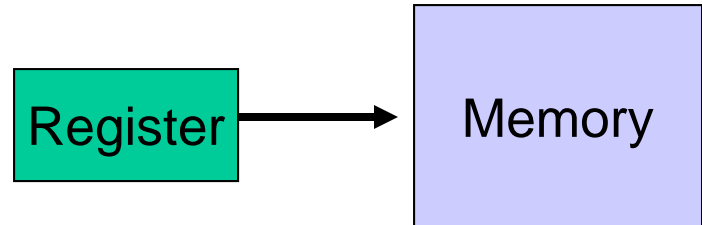
Load word

lw \$t0, memory-address



Store word

sw \$t0, memory-address



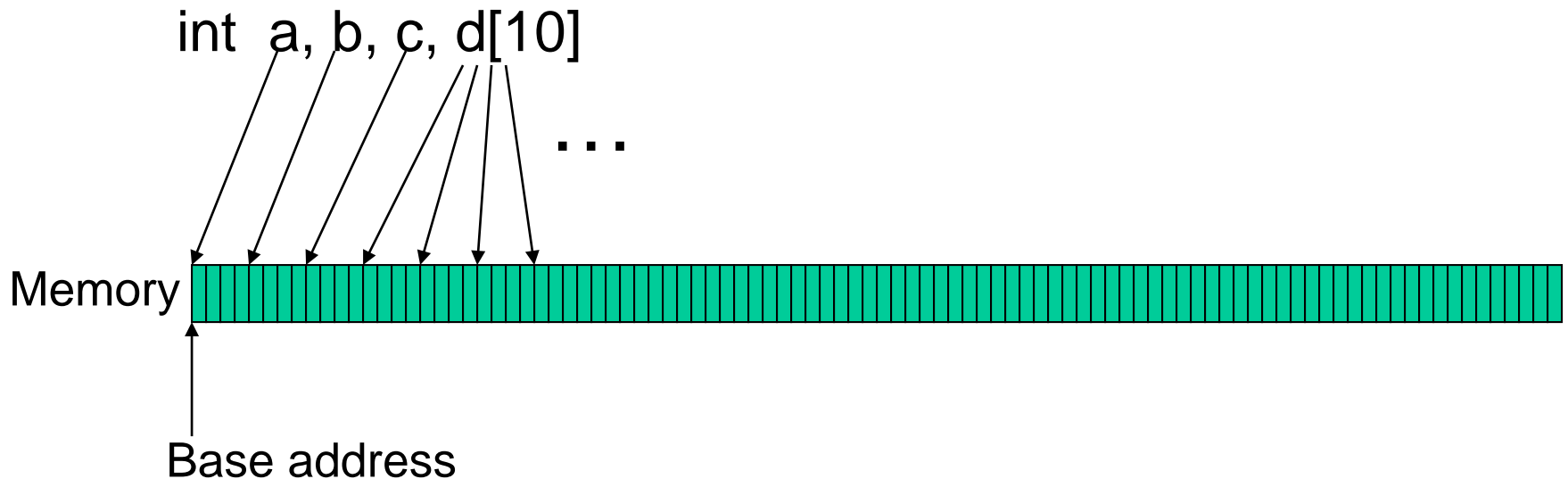
How is memory-address determined?



# Memory Address

---

- The compiler organizes data in memory... it knows the location of every variable (saved in a table)... it can fill in the appropriate mem-address for load-store instructions



# Title

---

- Bullet