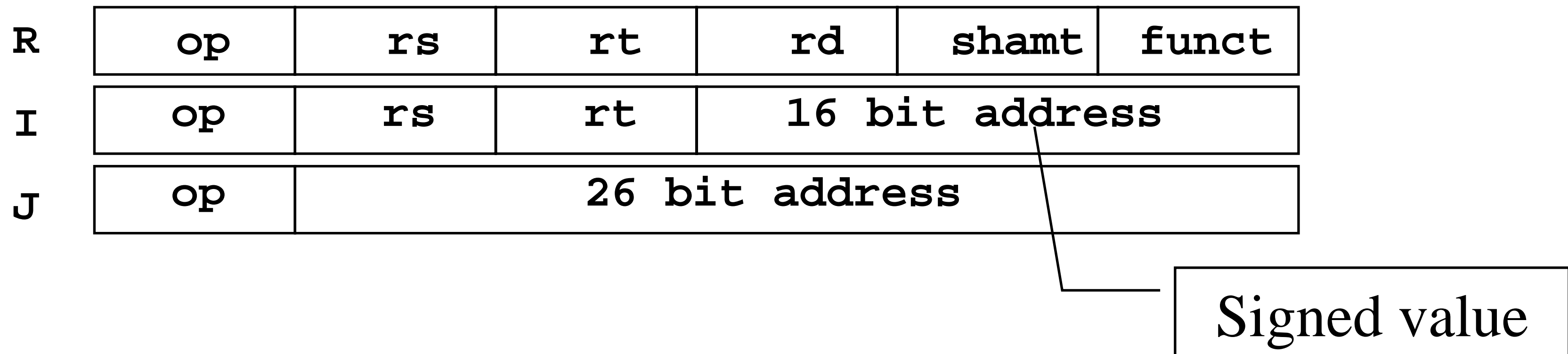
Assembly vs. machine language

- So far we've been using assembly language.
  - We assign names to operations (e.g., add) and operands (e.g., $t0).
  - Branches and jumps use labels instead of actual addresses.
  - Assemblers support many pseudo-instructions.
- Programs must eventually be translated into machine language, a binary format that can be stored in memory and decoded by the CPU.
- MIPS machine language is designed to be easy to decode.
  - Each MIPS instruction is the same length, 32 bits.
  - There are only three different instruction formats, which are very similar to each other.
- Studying MIPS machine language will also reveal some restrictions in the instruction set architecture, and how they can be overcome.

©1998 Morgan Kaufmann Publishers

Three MIPS formats

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats



©1998 Morgan Kaufmann Publishers

Constants

- Small constants are used quite frequently (50% of operands)

  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;

- MIPS Instructions:

```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
```


©1998 Morgan Kaufmann Publishers

Larger constants

- Larger constants can be loaded into a register 16 bits at a time.
  - The load upper immediate instruction lui loads the highest 16 bits of a register with a constant, and clears the lowest 16 bits to 0s.
  - An immediate logical OR, ori, then sets the lower 16 bits.
- To load the 32-bit value 0000 0000 0011 1101 0000 1001 0000 0000:

```
lui  $s0, 0x003D          # $s0 = 003D 0000 (in hex)
ori  $s0, $s0, 0x0900     # $s0 = 003D 0900
```

- This illustrates the principle of making the common case fast.
  - Most of the time, 16-bit constants are enough.
  - It's still possible to load 32-bit constants, but at the cost of two instructions and one temporary register.
- Pseudo-instructions may contain large constants. Assemblers will translate such instructions correctly.
- We used a lw instruction before. Later we will see the differences between the two approaches.

©1998 Morgan Kaufmann Publishers

Loads and stores

- The limited 16-bit constant can present difficulties for accesses to global data.
  - Let's assume the assembler puts a variable at address 0x10010004.
  - 0x10010004 is bigger than 32,767
- In these situations, the assembler breaks the immediate into two pieces.

```
lui   $t0, 0x1001            # 0x1001 0000
lw    $t1, 0x0004($t0)       # Read from Mem[0x1001 0004]
```

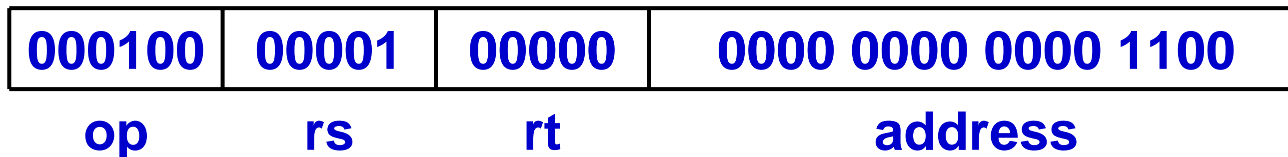

©1998 Morgan Kaufmann Publishers

Branches

- For branch instructions, the constant field is not an address, but an *offset* from the *next* program counter (PC+4) to the target address.

```
        beq   $at, $0, L
        add   $v1, $v0, $0
        add   $v1, $v1, $v1
        j     Somewhere
L:      add   $v1, $v0, $v0
```

- Since the branch target L is three *instructions* past the first add, the address field would contain 3×4=12. The whole beq instruction would be stored as:



000100	00001	00000	0000 0000 0000 1100
op	rs	rt	address

©1998 Morgan Kaufmann Publishers

Larger branch constants

- Empirical studies of real programs show that most branches go to targets less than 32,767 instructions away—branches are mostly used in loops and conditionals, and programmers are taught to make code bodies short.
- If you do need to branch further, you can use a jump with a branch. For example, if "Far" is very far away, then the effect of:

```
          beq   $s0, $s1, Far
          ...
```

can be simulated with the following actual code.

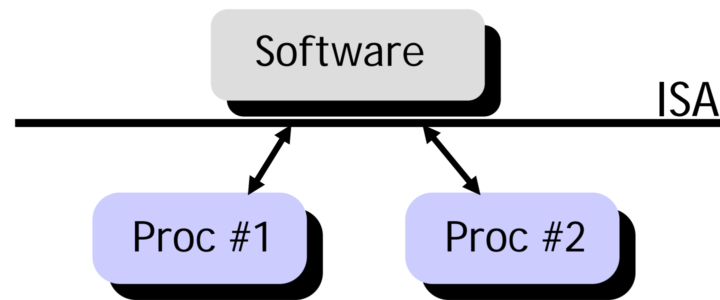
```
          bne   $s0, $s1, Next
          j     Far
Next:     ...
```

- Again, the MIPS designers have taken care of the common case first.

©1998 Morgan Kaufmann Publishers

Summary Instruction Set Architecture (ISA)

- The ISA is the interface between hardware and software.
- The ISA serves as an abstraction layer between the HW and SW
  - Software doesn't need to know how the processor is implemented
  - Any processor that implements the ISA appears equivalent



- An ISA enables processor innovation without changing software
  - This is how Intel has made billions of dollars.
- Before ISAs, software was re-written for each new machine.

©1998 Morgan Kaufmann Publishers

RISC vs. CISC

- MIPS was one of the first RISC architectures. It was started about 20 years ago by John Hennessy, one of the authors of our textbook.
- The architecture is similar to that of other RISC architectures, including Sun's SPARC, IBM and Motorola's PowerPC, and ARM-based processors.
- Older processors used complex instruction sets, or CISC architectures.
  - Many powerful instructions were supported, making the assembly language programmer's job much easier.
  - But this meant that the processor was more complex, which made the hardware designer's life harder.
- Many new processors use reduced instruction sets, or RISC architectures.
  - Only relatively simple instructions are available. But with high-level languages and compilers, the impact on programmers is minimal.
  - On the other hand, the hardware is much easier to design, optimize, and teach in classes.
- Even most current CISC processors, such as Intel 8086-based chips, are now implemented using a lot of RISC techniques.

©1998 Morgan Kaufmann Publishers

RISC vs. CISC

- Characteristics of ISAs

CISC	RISC
Variable length instruction	Single word instruction
Variable format	Fixed-field decoding
Memory operands	Load/store architecture
Complex operations	Simple operations

©1998 Morgan Kaufmann Publishers

A little ISA history

- 1964: IBM System/360, the first computer family
  - IBM wanted to sell a range of machines that ran the same software
- 1960's, 1970's: Complex Instruction Set Computer (CISC) era
  - Much assembly programming, compiler technology immature
  - Simple machine implementations
  - Complex instructions simplified programming, little impact on design
- 1980's: Reduced Instruction Set Computer (RISC) era
  - Most programming in high-level languages, mature compilers
  - Aggressive machine implementations
  - Simpler, cleaner ISA's facilitated pipelining, high clock frequencies
- 1990's: Post-RISC era
  - ISA complexity largely relegated to non-issue
  - CISC and RISC chips use same techniques (pipelining, superscalar, ..)
  - ISA compatibility outweighs any RISC advantage in general purpose
  - Embedded processors prefer RISC for lower power, cost
- 2000's: ??? EPIC? Dynamic Translation?

©1998 Morgan Kaufmann Publishers

Chapter 4 – Assessing and Understanding Performance

©1998 Morgan Kaufmann Publishers

Why know about performance

- Purchasing Perspective:
  - Given a collection of machines, which has the
    - Best Performance?
    - Lowest Price?
    - Best Performance/Price?
- Design Perspective:
  - Faced with design options, which has the
    - Best Performance Improvement?
    - Lowest Cost?
    - Best Performance/Cost ?
- Both require
  - Metric for evaluation
  - Basis for comparison

©1998 Morgan Kaufmann Publishers

Computer Performance: TIME, TIME, TIME

- **Response Time (latency)**
  - How long does it take for my job to run?
  - How long does it take to execute a job?
  - How long must I wait for the database query?
- **Throughput**
  - How many jobs can the machine run at once?
  - What is the average execution rate?
  - How much work is getting done?
- ***If we upgrade a machine with a new processor what do we increase?***

If we add a new machine to the lab what do we increase?

©1998 Morgan Kaufmann Publishers

Execution Time

- Elapsed Time
  - counts everything (disk, I/O , etc.)
  - a useful number, but often not good for comparison purposes
  - can be broken up into system time, and user time

- CPU time
  - doesn't count I/O or time spent running other programs
  - Include memory accesses

- Our focus: user CPU time
  - time spent executing the lines of code that are "in" our program

©1998 Morgan Kaufmann Publishers

Book's Definition of Performance

- For some program running on machine X,

  $Performance_X = 1 / Execution\ time_X$

- "X is n times faster than Y"

  $Performance_X \ / \ Performance_Y = n$

- Problem:
  - machine A runs a program in 20 seconds
  - machine B runs the same program in 25 seconds

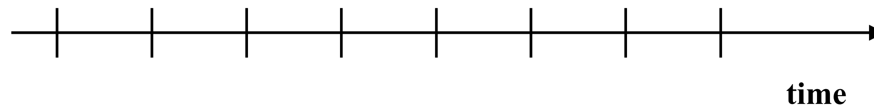
©1998 Morgan Kaufmann Publishers

Clock Cycles

- Instead of reporting execution time in seconds, we often use cycles

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

- Clock "ticks" indicate when to start activities (one abstraction):



- cycle time = time between ticks = seconds per cycle
- clock rate (frequency) = cycles per second (1 Hz. = 1 cycle/sec)

A 200 Mhz. clock has a $\frac{1}{200 \times 10^{6}} \times 10^{9} = 5 \text{ nanoseconds}$ cycle time

©1998 Morgan Kaufmann Publishers

How to Improve Performance

- $$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

- So, to improve performance (everything else being equal) you can either

  ______↓______ the # of required cycles for a program, or

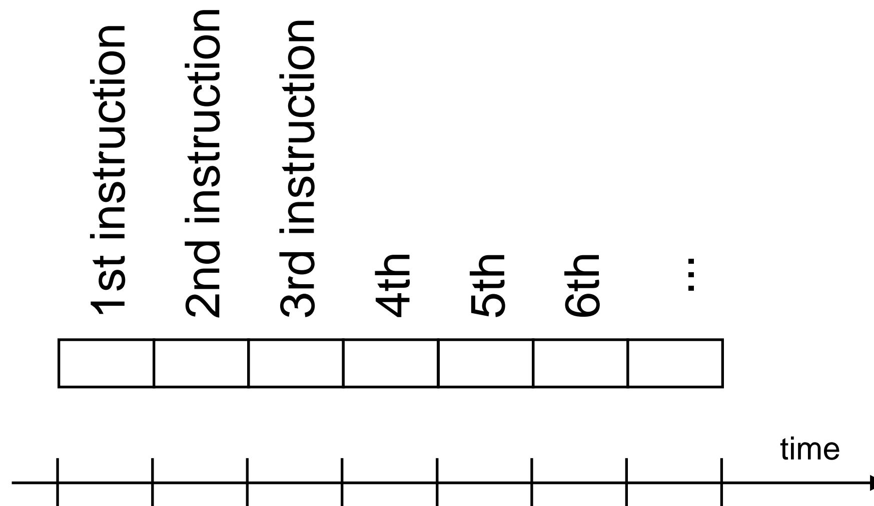
  ______↓______ the clock cycle time or, said another way,

  ______↑______ the clock rate.

©1998 Morgan Kaufmann Publishers

How many cycles are for a program?

- Could assume that # of cycles = # of instructions



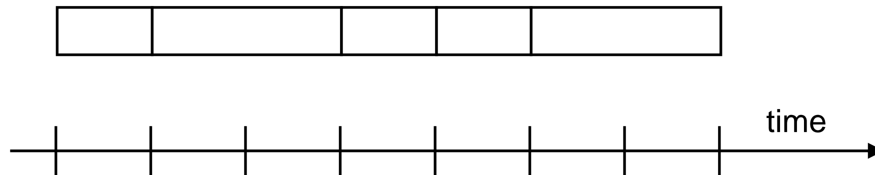
This assumption is incorrect,

different instructions take different amounts of time on different machines.

Why? hint: remember that these are machine instructions, not lines of C code

©1998 Morgan Kaufmann Publishers

Different numbers of cycles for different instructions



- Multiplication takes more time than addition
- Floating point operations take longer than integer ones
- Accessing memory takes more time than accessing registers

- Important point: changing the cycle time often changes the number of cycles required for various instructions (more later)

©1998 Morgan Kaufmann Publishers