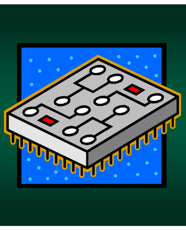
Part 9

Design Principles

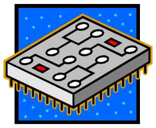


Technological Trends

The Constant Evolution

Technological Trends

- Since the design of the integrated circuit, computers have advanced *dramatically*
- Home computer's today have more power than mainframes did 30 years ago
- A hand calculator has more power than the computer that took us to the Moon



Integrated Circuits Improved In…

- *Density* – number transistors and wires can be placed in a fixed area on a silicon chip
- *Speed* – how quickly basic logic gates and memory devices operate
- *Area* – the physical size of the largest integrated circuit that can be fabricated

---

Rate of Improvement

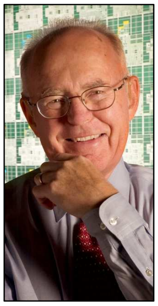
- The increase in performance does not increase at a linear rate
- Speed and Density improves *exponentially*
  - from one year to the next… it has been a relatively constant fraction of the previous year's performance
  - …rather than constant absolute value

Rate of Improvement

- On average…
  - number of transistors that can be fabricated on a silicon chip increases by about **50%** per year
  - transistor speed increases for basic logic gates (AND, OR, etc.) by **13%** per year

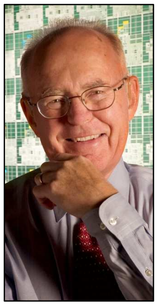
Moore's Law

- Gordon Moore is one of the co-founders of Intel
- He first observed (and predicted) computer performance improves *exponentially*, not linearly



Moore's Law

- *Moore's Law* states the performance doubles every 18 months
- This law has held for nearly 50 years

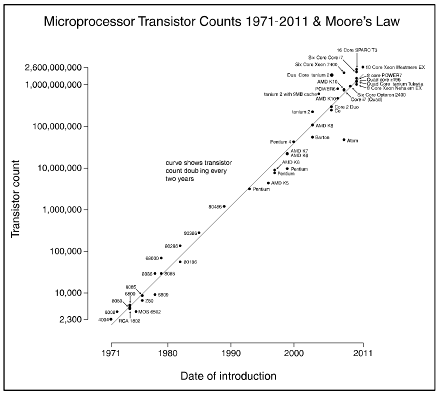


Intel Processors Over Time

Processor	Year	Speed (KHz)	Transistors	Technology
4004	1971	108	2,300	10
8008	1972	800	3,500	10
8080	1974	2,000	4,500	6
8086	1978	5,000	29,000	3
8088	1979	5,000	29,000	3
80286	1982	6,000	134,000	1.5
80386	1985	16,000	275,000	1.5
80486	1989	25,000	1,200,000	1

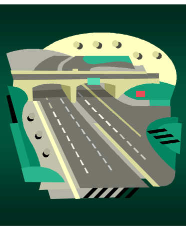
Intel Processors Over Time

Processor	Year	Speed (KHz)	Transistors	Technology
Pentium	1993	66,000	3,100,000	0.8
Pentium Pro	1995	200,000	5,500,000	0.6
Pentium II	1997	300,000	7,500,000	0.25
Pentium III	1999	500,000	9,500,000	0.18
Pentium 4	2000	1,500,000	42,000,000	0.18
Pentium M	2002	1,700,000	55,000,000	0.13
Pentium D	2005	3,200,000	291,000,000	0.065



The Late 1990's

- Processor performance (per unit energy dissipation) has also improved *exponentially* rather than linearly
- This has made feasible
  - smart phones
  - tablets
  - handheld consoles



von Neumann Architecture

The Information Super Highway

---

von Neumann Machine Architecture

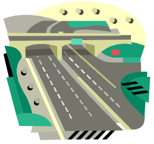
- Modern computers are based on the design of John von Neumann
- His design greatly simplified the construction of (and use) computers

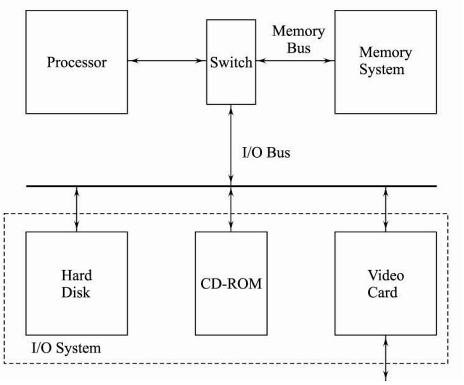
Some von Neumann Attributes

- Programs are stored and executed in memory
- Separation of processing from storage
- Different system components communicate over shared buses

The Bus

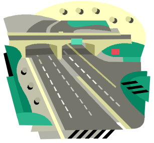
- Electronic pathway that transports data between components
- Think of it as a "highway"
  - data moves on shared paths
  - otherwise, the computer would be very complex





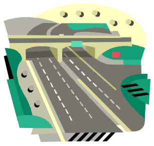
System Bus

- Interconnects the processor with the memory
- Also called the "system bus" since it interconnects the subsystems)



System Bus

- The information sent on the memory bus falls into 3 categories
- Three sets of signals
  - address bus
  - data bus
  - control bus



Address Bus

- Used by the processor to access a specific piece of data
- This "address" can be
  - a specific byte in memory
  - unique IO port
  - etc…

Address Bus Characteristics

- Total number of bits used in the address limits the total number of bytes that can be accessed
- For an address-size of *n* bits, you have 2^n memory addresses

Address Bus Size Examples

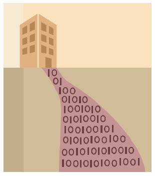
- 8-bit → 256 bytes
- 16-bit → 64 KB (65,536 bytes)
- 32-bit → 4 GB (4,294,967,296 bytes)
- 64-bit → 18 EB (18,446,744,073,709,551,616)

Historic Address Sizes

- Intel 8086
  - original 1982 IBM PC
  - 20-bit address bus (1 MB)
  - only 640 KB usable for programs
- MOS 6502 computers
  - Commodore 64, Apple II, Nintendo, etc…
  - 16-bit address bus (64 KB)

Data Bus

- The actual data travels over the *data bus*
- An integer that has the same number of bits as the system is called a *word*



Data Bus

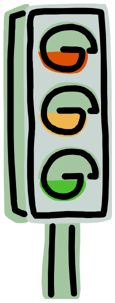
- Different processors use a different amount of bytes to store and manipulate data
- Example:
  - 8-bit system uses 8 bit integers for data
  - 16-bit system uses 16 bits (2 bytes) for data
  - 32-bit system uses 32 bits (4 bytes) for data
  - etc…

Data Bus

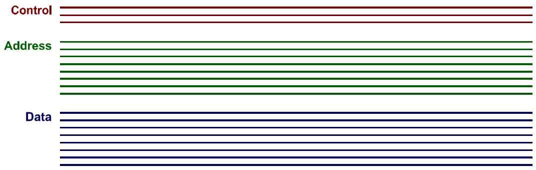
- Often the processor's address bus and data bus use different bit counts
- Examples:
  - MOS 6502 – 8 bit data, 16 bit address bus
  - Intel 8086 – 16 bit data, 20 bit bus (well 16, but expanded to 20 using a trick)

Control Bus

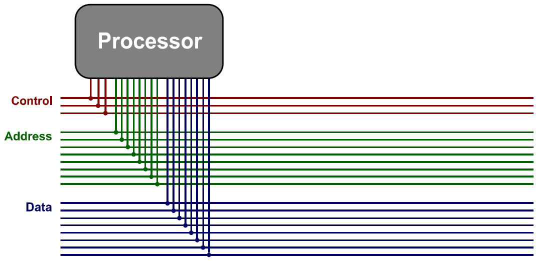
- The *control bus* controls the timing and synchronizes the subsystems
- Specifies what is happening
  - read data
  - write data
  - reset
  - etc…



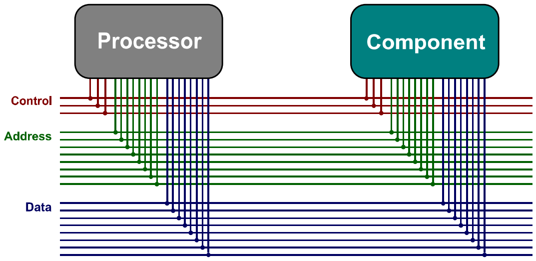
The Bus



Processor



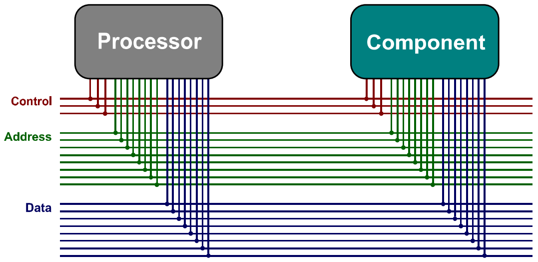
Components Are On the Bus



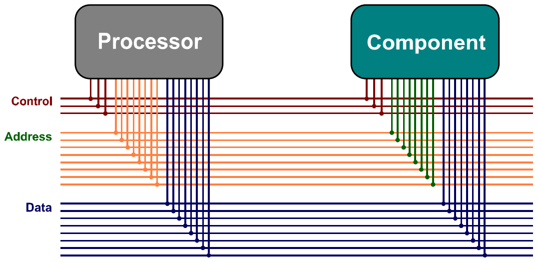
Reading Data

- Processor has the address, but needs data
- Actions:
  1. processor puts the address on the bus
  2. signals the component to read
  3. component reads the address
  4. component puts the data on the bus
  5. processor stores the data

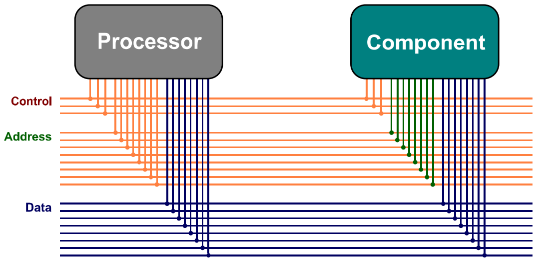
Reading Data



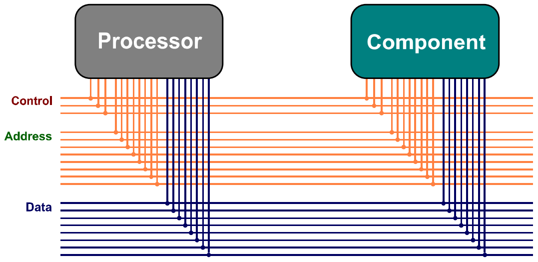
Read: Put Address on Bus



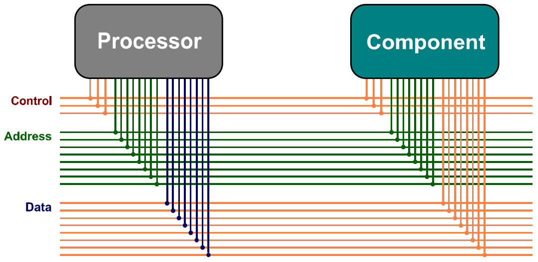
Read: Signal Component



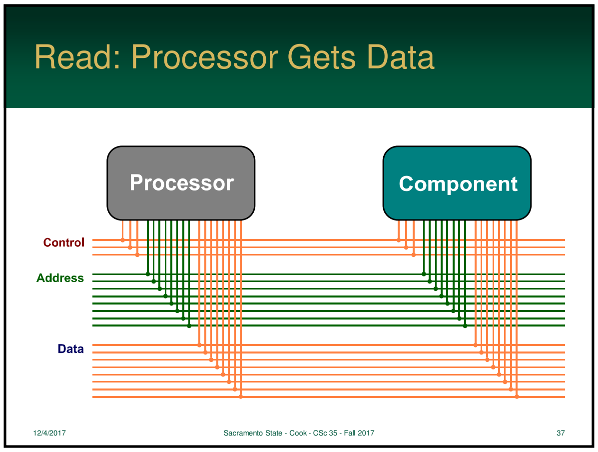
Read: Component Gets Address



Read: Component Returns Data



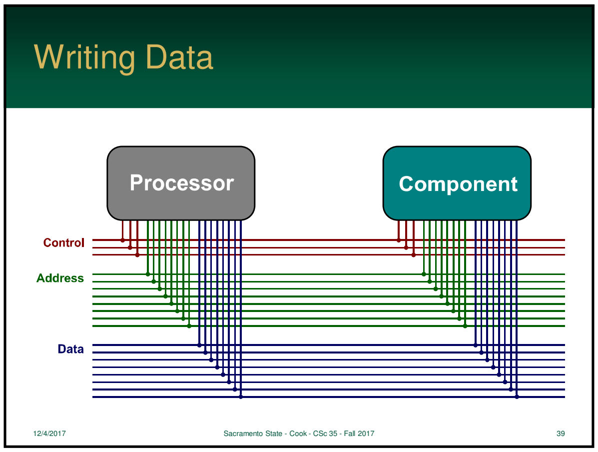
Read: Processor Gets Data



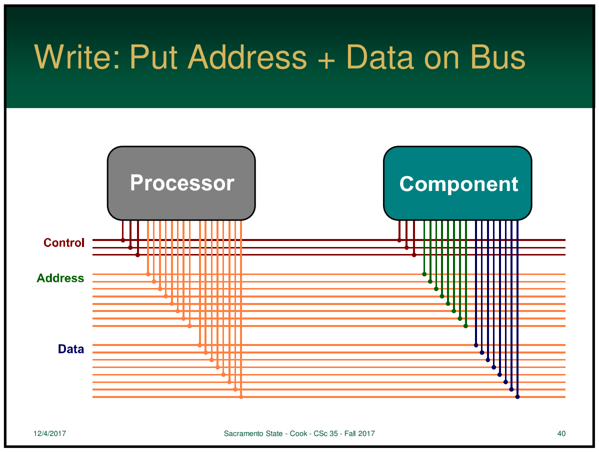
Writing Data

- Processor has the address and data
- Actions:
  1. processor puts the address and data on bus
  2. signals the component to write
  3. component takes the data and then stores it

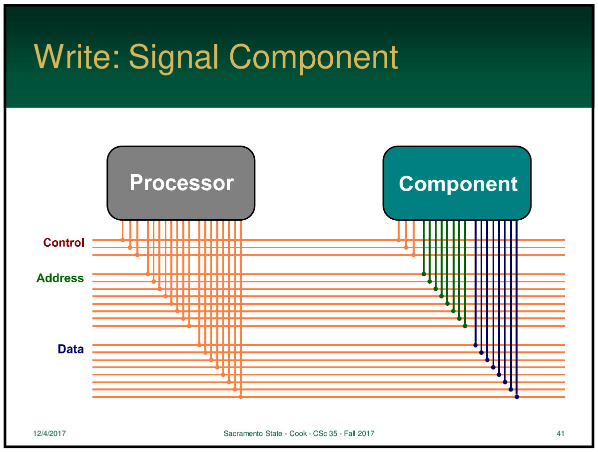
Writing Data



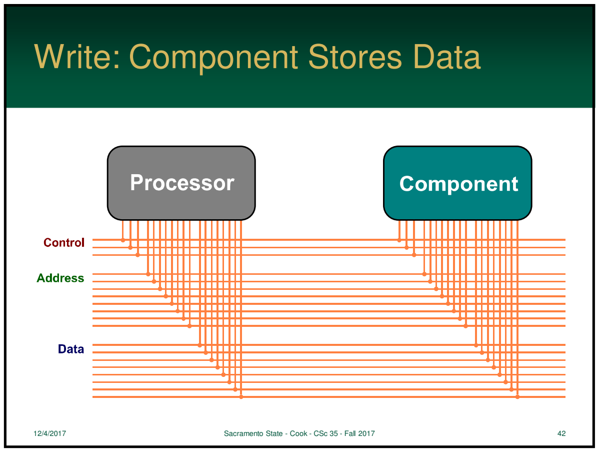
Write: Put Address + Data on Bus

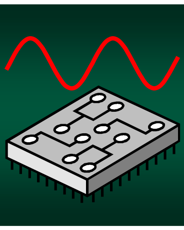


Write: Signal Component



Write: Component Stores Data

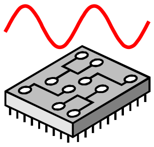




Processor Speed

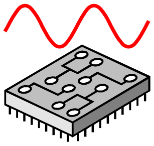
Let's rock around the clock tonight

The Clock



- The rate in which instructions are executed is controlled by the CPU clock
- The faster the clock rate, the faster instructions will be executed
- Measured in Hertz – number of oscillations per second

The Clock



- Computers are typically (and generically) labeled on the processor clock rate
- In the early 80's it was about 1 Megahertz – million clocks per second
- Now, it is terms of Gigahertz – billion clocks per second

Clock and Instructions

- Not all instructions are "equal"
- Some require multiple clock cycles to execute
- For example:
  - a simple add can take a single clock
  - but floating-point math could require a dozen
- Some processors can also execute several instructions at a time

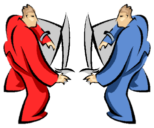
---

CISC vs. RISC

Highlander: "there only can be one"

(well, not really)

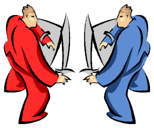
CISC vs. RISC



- There is, an often contentious, debate on how to design a processor
- For instance:
  - how is memory going to be accessed
  - what instructions are needed
  - how to encode/structure them

CISC vs. RISC

- Typically the debate comes down to CISC vs. RISC
- Processors are typically put into these two categories
- Rarely is a processor "pure" RISC or CISC
- It is a design philosophy with a large "gray" area between extremes



CISC

- Complex Instruction Set Computer (CISC) emphasizes flexibility in instructions
- Hardware should contain the complexity rather than the software

The Semantic Gap

- Pre-1980's focused on reducing the *"semantic gap"* between languages and the processor
- *So, can we make instructions more like high-level languages?*

The Semantic Gap

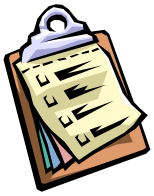
- In high level languages…
  - blocks are common, but there are no "blocks" in assembly
  - if statements are common, but there is no "if" instruction
  - while statements are common, but there is no "while" instruction

CISC Reasoning

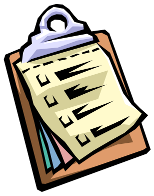
1. Results in better performance
   - each instruction does more
   - reduces the number of instructions required to implement a program
2. Easier to compile high-level languages
   - statements can be mapped directly into instructions
   - compilers will be simpler and result in more consistent machine code

CISC Characteristics

- Very few general purpose registers – memory access is emphasized
- Some special-purpose registers
- Instructions can take multiple cycles – depending on how complex

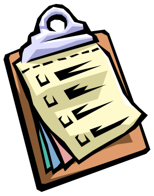


CISC Characteristics



- Operands are *generalized*
  - each can access different resources – memory, immediates or registers
  - one typically is the destination
- This allows combinations like:
  - register to register
  - register to memory
  - memory to register

CISC Advantages



- Generally requires fewer instructions than RISC to perform the same computation
- Programs written for CISC architectures tend to take less space in memory

CISC Today

- CISC architectures became increasingly complex (some even having case blocks)
- After the 1980's…
  - CISC architectures attempted to have a middle ground between flexibility and complexity
  - dropped instructions that were not used often
  - complex instructions had to justify their implementation (inclusion in the instruction set)

Example CISC Processors

- Intel x86
  - evolved from the 8088 processor and contains 8-bit, 16-bit, and 32-bit instructions
  - dominant processor for PCs
- Motorola 68000
  - used in many 80's computers
  - …including the first Macintosh

Example CISC Processors

- VAX
  - contained even more addressing modes than we will cover
  - specialized instructions – even case blocks!
  - supported data types beyond float and int: variable-length strings, variable-length bit fields, etc…

Moore's Law and CISC

- Computer speed through the 1980's grew exponentially
- However…
  - rate of increase in processor speed has been far greater than that of memory
  - so, memory *relative to the processor's speed* has gotten much slower

Memory is the Bottleneck

- CISC can access memory with nearly every instruction
- But, memory is slow compared to register-to-register operations
- It is far more efficient (now) to do all work on the processor and use memory only when absolutely necessary

RISC

- Reduced Instruction Set Computer (RISC) emphasizes simplicity
- Software should contain the complexity rather than hardware

RISC

- So, RISC contains fewer instructions than CISC – only the minimum needed to work
- Minimalize memory accesses
  - only a few instructions can access memory
  - usually limited to register load and store instructions

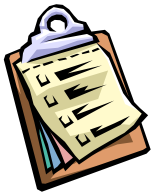
RISC Reasoning

1. Results in higher performance
   - simple instructions can execute at higher clock rates than CISC
   - memory access is limited, ending the bottleneck
2. Easy to compile high-level languages
   - compiler only needs to understand a few instructions
   - compilers can create blocks of instructions fairly robotically

---

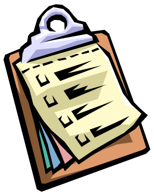
RISC Characteristics

- Access to memory is restricted to load/store instructions – that only can be used with a register
- All other instructions only work with registers



RISC Characteristics

- Since registers are used to hold more data, RISC processors typically have many
- Instructions typically take one clock cycle each



RISC Advantages

- Simpler instructions make it easier to implement on different processors – and make them more efficient
- Easier to program and master by programmers – less to learn
- Memory access is minimalized

Example RISC Processors

- ARM
  - dominant processor used by smartphones - iPhone and Droid
  - designed to reduce transistors
  - which reduces cost, creates less heat, and uses less power

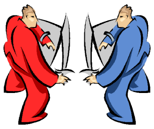
---

Example RISC Processors

- IBM PowerPC 601
  - developed in by IBM, Apple, and Motorola (AIM)
  - used by 1990's Macintosh computers

Addressing: RISC vs. CISC

- There are a large number of possible addressing modes
- RISC tends to limit the number of addressing modes – to about 4 or 5
- CISC tends to have more – sometimes exceeding a dozen or more



RISC vs. CISC Comparison

CISC	RISC
Emphasis on hardware complexity	Emphasis on software complexity
Operands are generalized	Load/Store instructions
Low number of registers	Higher number of registers
Instructions tend towards multiple clock cycles	Instructions tend towards one per clock cycle

---

Latest Approach

- After the 1990s, RISC architectures have incorporated some of most useful complex instructions from CISC architectures
- Rely on their micro-architecture to implement these instructions with little impact on the clock cycle

CISC Example

```
# n  =  a * (b + c) - 5

mov    b, %R1
add    c, %R1      # b + c

mul    a, %R1      # a * (b + c)
sub    $5, %R1     # a * (b + c) - 5

mov    %R1, n
```

RISC Example

```
# n  =  a * (b + c) - d

load   b, %R1
load   c, %R2
add    %R1, %R2     # b + c

load   a, %R3
mul    %R3, %R2     # a * (b + c)
load   $5, %R4
sub    %R4, %R2     # a * (b + c) - 5

store  %R2, n
```

Instruction Operands

How much data does each need?

Instruction Operand

- The number of operands used in an instruction varies greatly by processor
- More operands give greater functionality, but require more bits to store in memory
- Typically processors contain 1, 2 or 3 operands

Single Operand Processors

- Single operand processors are also known as *accumulators*
- Operates similar to your hand calculator
- The accumulator register
  - used for all mathematical computations
  - other registers simply are used to compare and hold temporary data
- Examples: MOS 6502

Single Operand Instruction

```
#  z  =  50 - (x + y)

load   x
add    y           # x + y
store  temp

load   $50
sub    temp       # 50 - temp

store  z
```

Two Operand Processors

- Allows two operands to be specified
- For computations, both operands are typically treated as input and one is used to store the result
- Examples:
  - x86 processors
  - PowerPC

Two Operand Instruction

```
#   z  =  50 - (x + y)

mov    x, %R1
add    y, %R1      # x + y

mov    $50, %R2
sub    %R1, %R2    # 50 - R1

mov    %R2, z
```

Three Operand Processors

- Allows two input values like before, but also can specify a third output operand
- The third operand can also be used as a index for simple addressing
- Examples:
  - ARM
  - Intel Itanium

Three Operand Instruction

```
#   z  =  50 - (x + y)

add    x, y, %R1      # x + y
sub    $50, %R1, z
```

Let's Make a Processor

Appreciate the encoding by making one

Let's Make a Processor

- When a processor is designed, the architects needs to balance features and simplicity
- Generally, architects want to find an encoding that is both compact and simple

Design Considerations

- A compact encoding:
  - takes as little space, as possible, to store instructions
  - programs require less storage
- A simple encoding:
  - requires little logic to decode
  - minimizes the circuitry on the processor and reduces cost

Let's Make a Processor

- Assume we are creating a simple processor
- It will have a total of 4 general purpose registers: R0, R1, R2, R3
- The processor will support two operand instructions

Assigning Bits

- Since there are 4 registers, we need enough bits store a code for each
- $2^2 = 4$, so each register code can be stored with just two bits
- So, to store both operands, we will need: 2 operands × 2 bits → 4 bits

---

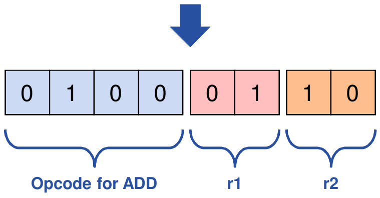
Our Opcodes

- If we want to store each instruction in a single byte
  - we have 8 - 4 → 4 bits left
  - so, the opcode will be 4 bits
  - this will allow a total of 2^4 → 16 instructions
- Now, we assign opcode bit values for each operation we want to perform

---

Sample Encoding

```
ADD %r1, %r2
```



Limits of this encoding

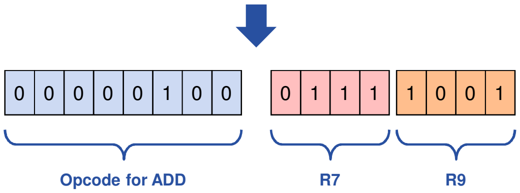
- With only 4 bits for an opcode, it is limited to 16 distinct processor instructions
- This is simply not enough for all the features that we would like – addressing etc…
- Essentially, the more bits used in the opcode, the more features we can add to the processor

Two Bytes

- If the instruction set is expanded to 2 bytes, we have far more complexity
- An entire byte can be used for the opcode – giving 256 unique instructions
- The second byte could contain two 4-bit fields – allowing 16 registers!

Sample Encoding: 2 Byte

```
ADD %R7, %R9
```



0	0	0	0	0	1	0	0	0	1	1	1	1	0	0	1

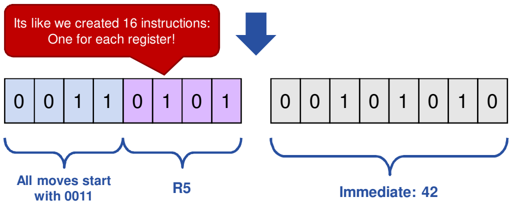
Opcode for ADD

R7

R9

Bits Can Be Used Differently

```
mov $42, %R5
```



Its like we created 16 instructions: One for each register!

0	0	1	1	0	1	0	1	0	0	1	0	1	0	1	0

All moves start with 0011

R5

Immediate: 42