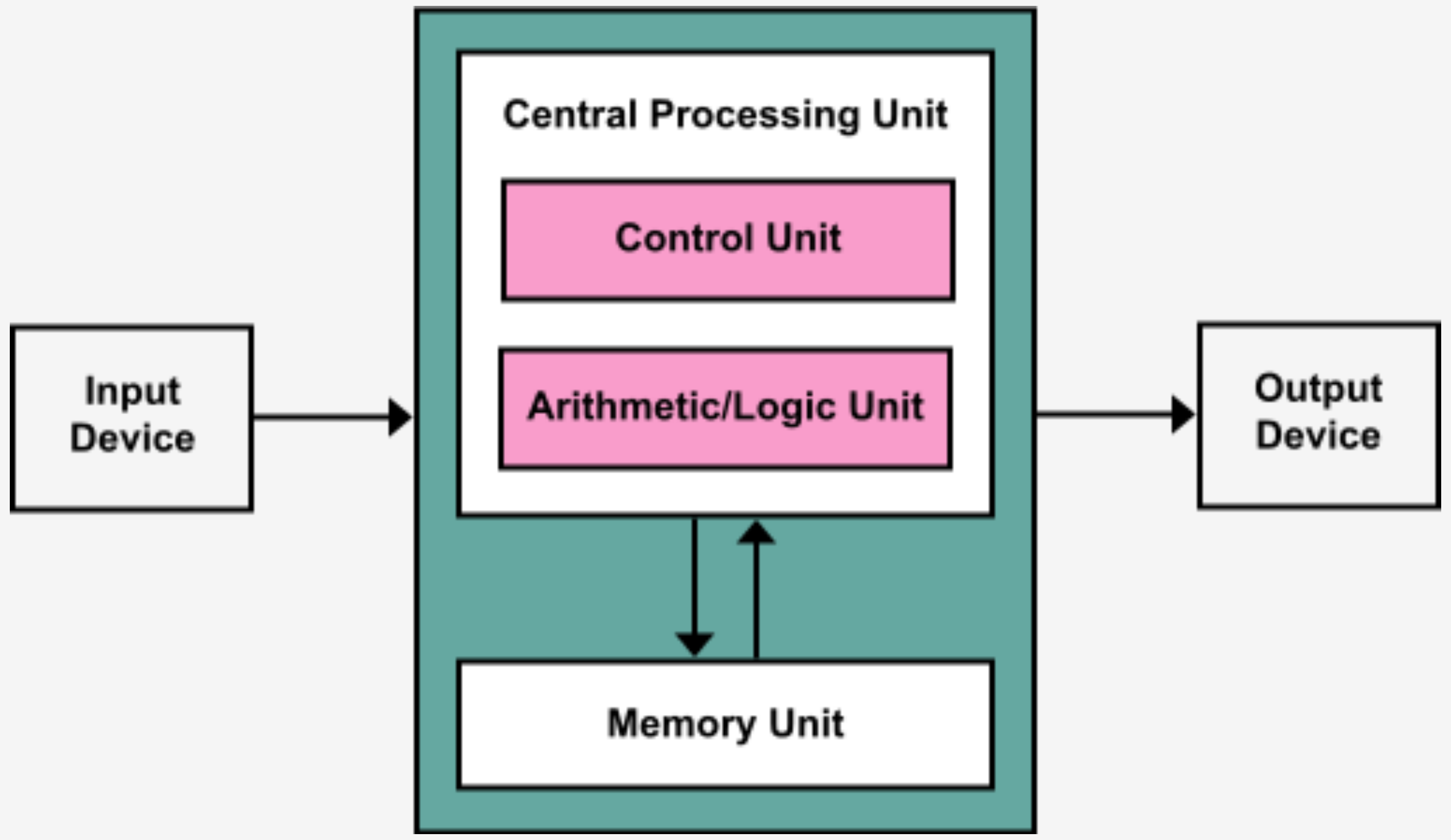
ECE260: Fundamentals of Computer Engineering

Instructions & Instruction Sets

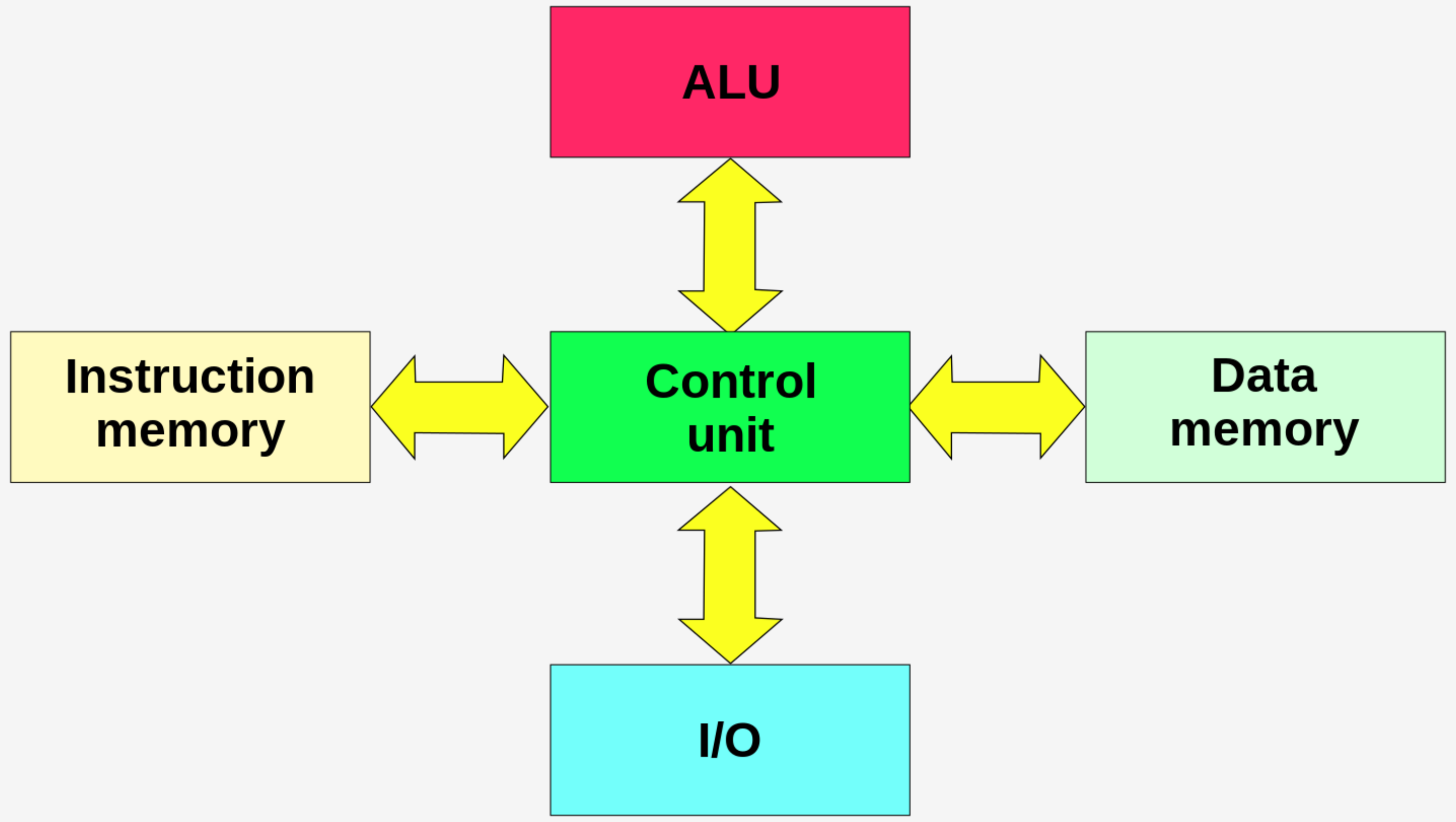
James Moscola
Dept. of Engineering & Computer Science
York College of Pennsylvania

General-Purpose Computers

- Many models for a general-purpose computer have been explored
  - Von Neumann architecture
    - Includes a CPU, I/O devices, and a ***single*** memory unit that stores ***BOTH*** instructions and data
    - Single bus between CPU and memory; cannot read instructions and data at the same time
  - Harvard architecture
    - Includes CPU, I/O devices, and ***separate*** memory units for instructions and data
    - Separate buses for communicating with memory units
  - Modified Harvard architecture
    - Definition varies, depends who you ask … 😕



von Neumann architecture



Harvard architecture

Stored Program Concept

- The idea that instructions and data of many types can be stored in memory as numbers, leading to the stored program computer
  - Distinct from Application-Specific Integrated Circuits (ASICs),
    - ASICS are hardwired to perform some fixed-functions on inputs
- Enables ***programmable*** computers
  - Can be programmed and re-programmed to perform different tasks
- Stored program computers utilize an ***instruction set architecture (ISA)*** as a specification for all stored programs
  - Applications must conform to ISA specification for compatibility

Instruction Sets

- ***Instructions*** are the “words” of a computer’s language
  - ADD, SUB, SHIFT, etc.
- ***Instruction set*** consists of a computer’s complete vocabulary
- Different computers have different instruction sets (i.e. they speak different languages)
  - But with many aspects in common (like Spanish and Portuguese)

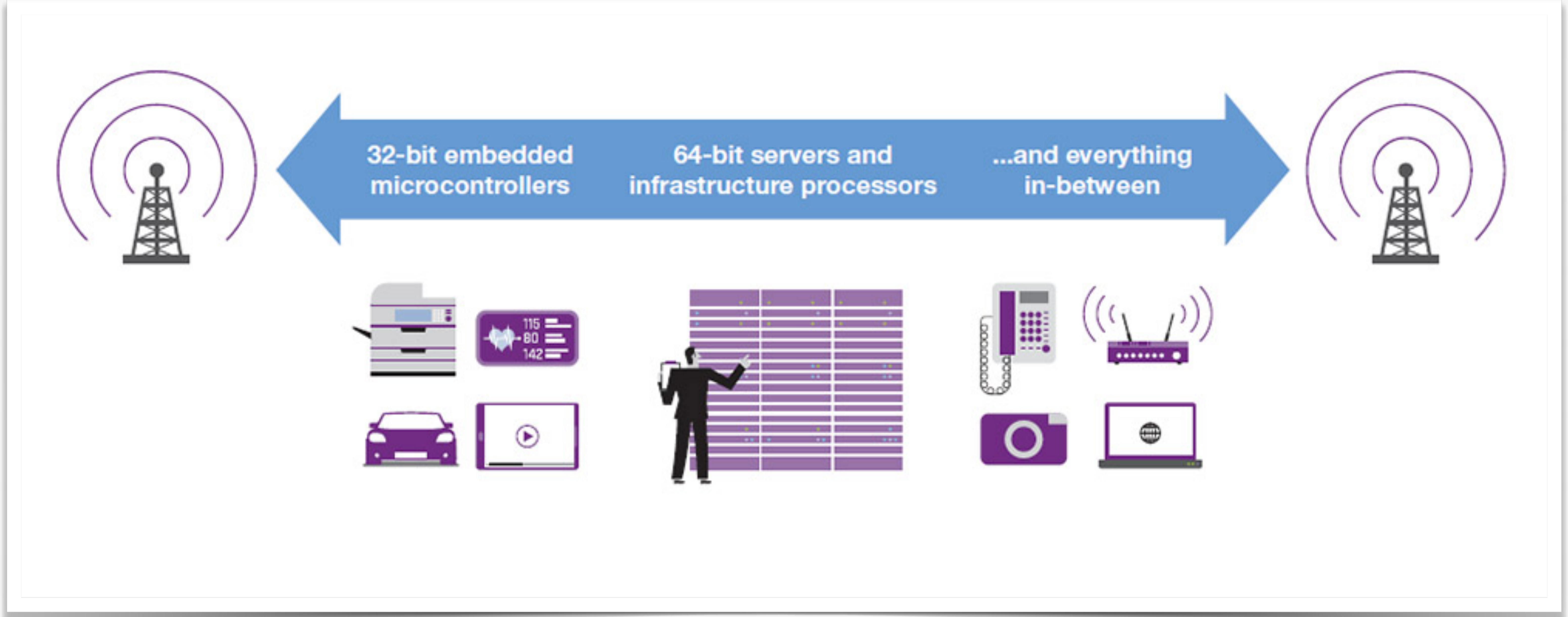
Instruction Sets (Continued…)

- Some modern computers have <u>simple</u> instruction sets
  - RISC processors (RISC = Reduced Instruction Set Computing)
    - **Examples**: ARM, MIPS, RISC-V
  - More instructions, but fewer cycles per instruction

- Some modern computers have <u>complex</u> instruction sets
  - CISC processors (CISC = Complex Instruction Set Computing)
    - **Examples**: x86, IA64, x86_64
  - Fewer instructions, but more cycles per instruction

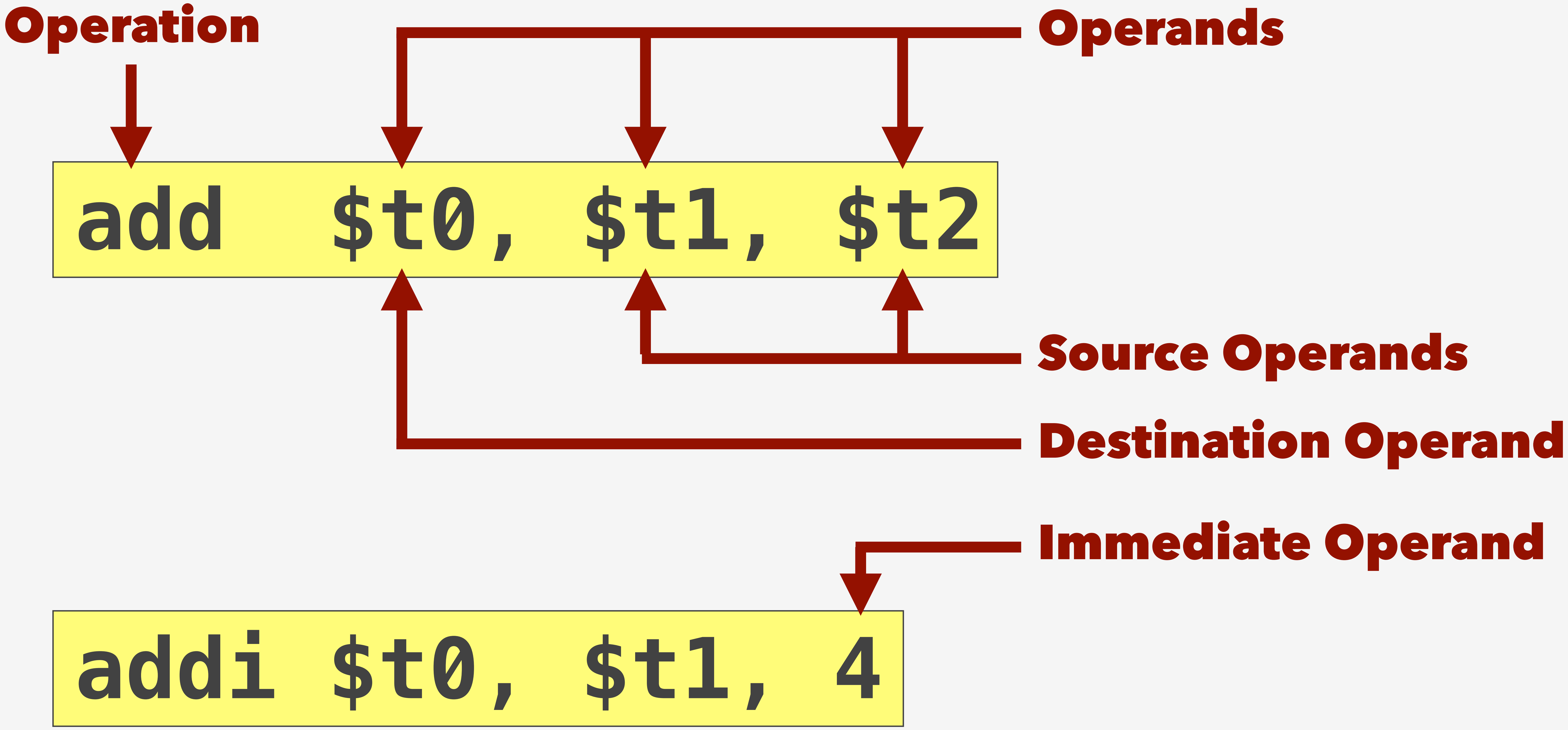
The MIPS Instruction Set

- Used as the example throughout your textbook – a RISC ISA
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, etc.



Anatomy of an Instruction

- An instruction is a primitive operation
  - Specifies an operation and its operands (e.g. the variables on which to perform the operation)
  - Types of operands include: ***immediate***, ***source***, and ***destination*** operands



Arithmetic Operations on MIPS

- Each instruction performs only a single operation
  - Arithmetic instructions must always have *3 operands*
    - Two sources operands and one destination operand
- All arithmetic operations have the same form with 3 operands
  - Simplifies hardware design

```
add  $t0, $t1, $t2
```

Example: $t0 = $t1 + $t2
The values stored in registers $t1 and $t2 are added together
The result is stored in register $t0

Register Operands

- Arithmetic instructions use register operands
- MIPS architecture has a 32 × 32-bit register file (i.e. it has 32 32-bit registers)
  - Use for frequently accessed data
  - Registers are numbered 0 to 31
  - 32-bit architecture (32-bit data value called a “word”)
    - *Side Note*: a byte is ALWAYS 8-bits; a “word” size depends on the architecture
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables

Register Operand Example

- C code:

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

- Compiled MIPS code:
  - Assume:

f is stored in register	$s0
g is stored in register	$s1
h is stored in register	$s2
i is stored in register	$s3
j is stored in register	$s4

```
add $t0, $s1, $s2   # register $t0 contains g + h
add $t1, $s3, $s4   # register $t1 contains i + j
sub $s0, $t0, $t1   # f contains $t0 - $t1
```

Memory Operands and Addressing

- Main memory used for larger data structures
  - Arrays, structs, dynamic data, etc.
  - Not enough registers to store all program data
- To apply arithmetic operations
  - Must bring data values into CPU registers, cannot operate on values while they are in memory
    - Load values from memory into registers
    - Store result from register to memory

- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Addresses must be a multiple of 4
    - If you want a byte that is stored, read the entire word and extract the byte you want

Byte Address (decimal)	Data (decimal)
⋮	⋮
12	100
8	10
4	101
0	1

Byte Address (hexadecimal)	Data (hexadecimal)
⋮	⋮
0x0C	00000064
0x08	0000000A
0x04	00000065
0x00	00000001

Memory Operand Example #1

- C code:

```
g = h + A[8];
```

A[8] indicates an *offset* of 8 words from *base* address of A
Given 4 bytes per word, offset is 8 * 4 = 32 bytes

- Compiled MIPS code:
  - Assume:

g is stored in register	$s1
h is stored in register	$s2
base address of A (i.e. A[0])	$s3

Offset from base register
Base register

```
lw  $t0, 32($s3)    # load word 32 bytes from A[0]
add $s1, $s2, $t0   # register $s1 contains h + A[8]
```

Memory Operand Example #2

- C code:

```
A[12] = h + A[8];
```

Read operand from memory and store result in memory

- Compiled MIPS code:
  - Assume:

h is stored in register	$s2
base address of A (i.e. A[0])	$s3

Offset from base register

Base register

```
lw  $t0, 32($s3)    # load word 32 bytes from A[0]
add $t0, $s2, $t0   # register $t0 contains h + A[8]
sw  $t0, 48($s3)    # store word 48 bytes from A[0]
```

Source Data

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler prefers to use registers for variables as much as possible
- If program has more variables than registers, variables ***spill*** into main memory
  - Only spill to memory for less frequently used variables
  - Spilled variables must be loaded back into CPU register when needed
  - Register optimization is important

Immediate Operands

- Constant data can be specified directly in an instruction (MIPS permits 16-bit immediate values)
  - Immediate operand avoids a load instruction, saves a trip to memory
  - Incrementing by small constants is very common (e.g. loop variables)
  - Think … increment operations: x = x + 4

```
addi $s3, $s3, 4
```

Add immediate: increment the value stored in register $s3 by 4.

- No subtract immediate instruction in most ISAs
  - Just add a negative constant (hooray for signed numbers!)

```
addi $s2, $s1, -1
```

Add immediate: decrement the value stored in register $s1 by 1. Store the result in register $s2

MIPS Constant Zero Register

- MIPS has a dedicated register that represents the constant 0
  - Register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Always available, useful for common operations
  - For example, moving data between registers since MIPS has no “move” instruction
  - Adding $zero to a register and storing the result in another registers behaves like a “move”

```
add $t2, $s1, $zero
```

Move data from $s1 to $t2