



# Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

## MIPS Assembly

# Lab Schedule

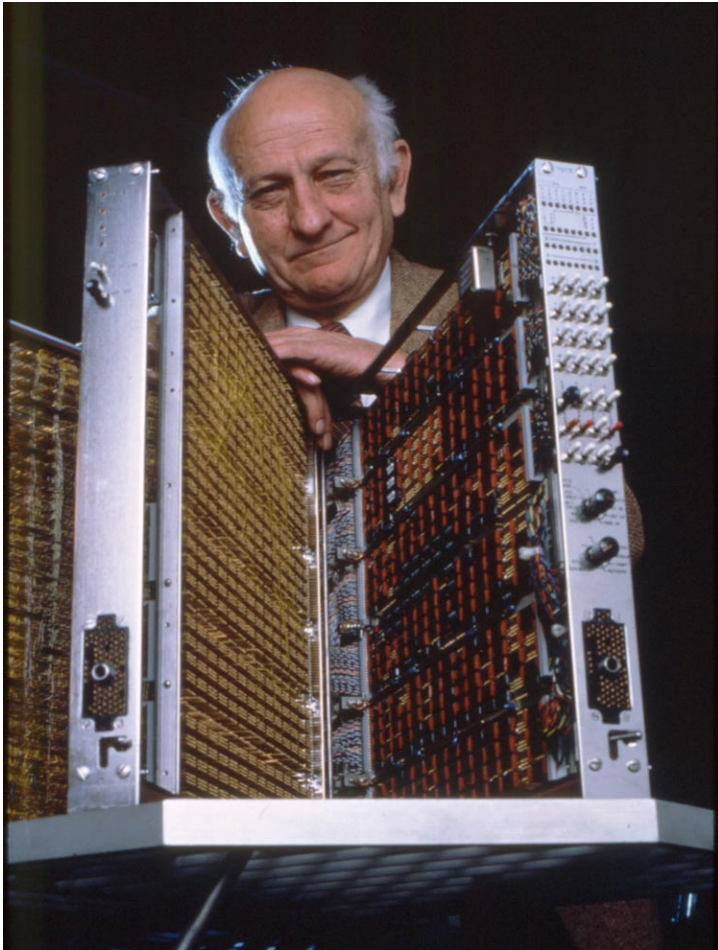
## Activities

- **Today**
  - MIPS discussion
  - Practice problems (paper & pencil)
- **Next Tuesday / Thursday**
  - **Lab 8 – MIPS**
  - Using the QtSPIM simulator

## Assignments Due

- **Sunday, Oct 28<sup>th</sup>**
  - **Lab 6 due by midnight**

# Person of the Day – John Cocke



- Computer architecture pioneer
  - “Father of RISC Architecture”
  - Developed IBM 801 processor, 1975-1980
- Winner, *ACM Turing Award*, 1987

**RISC = Reduced Instruction Set Computing**

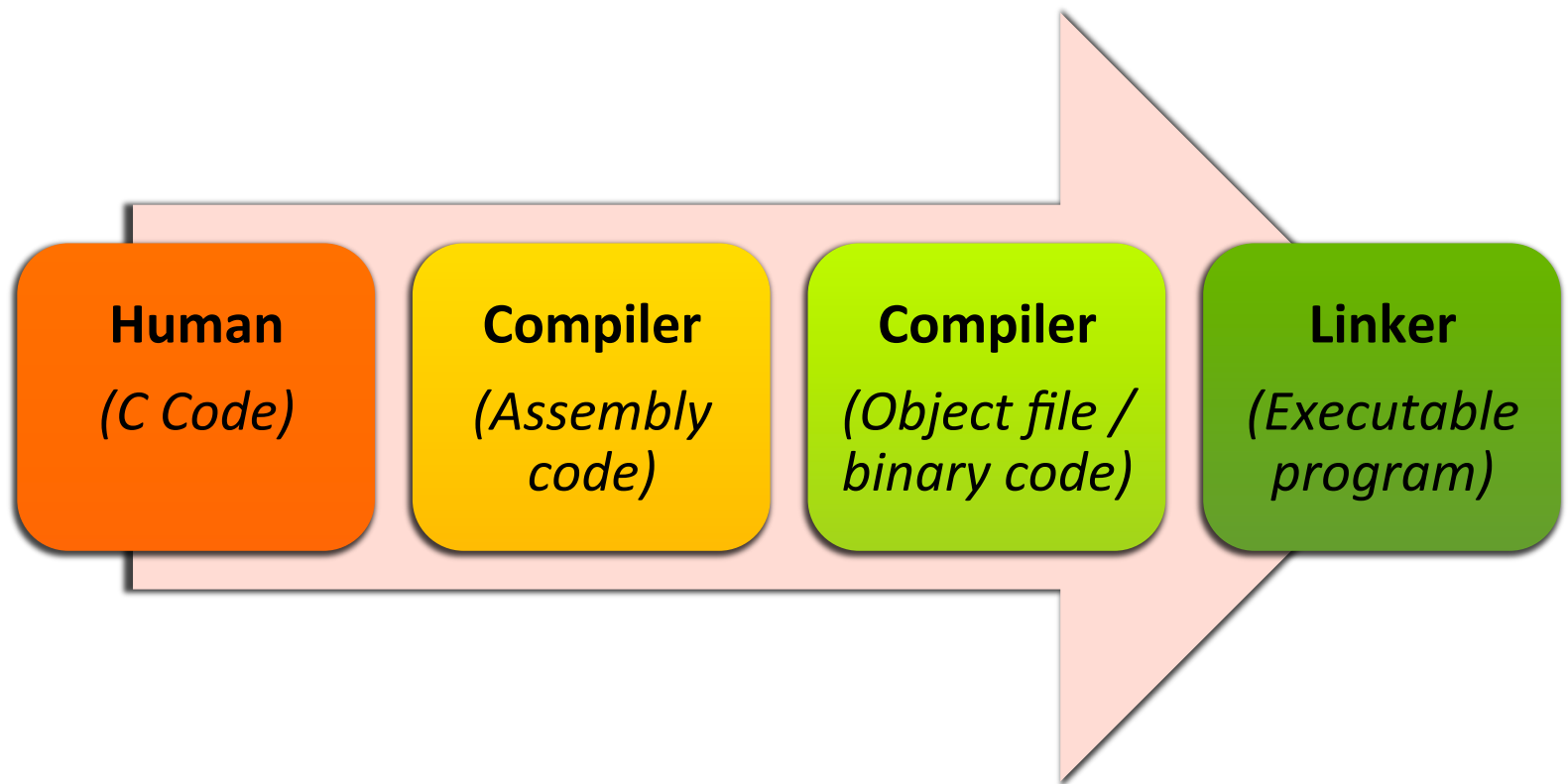
Achieve higher performance with simple instructions that execute faster

# Person of the Day – John Hennessy

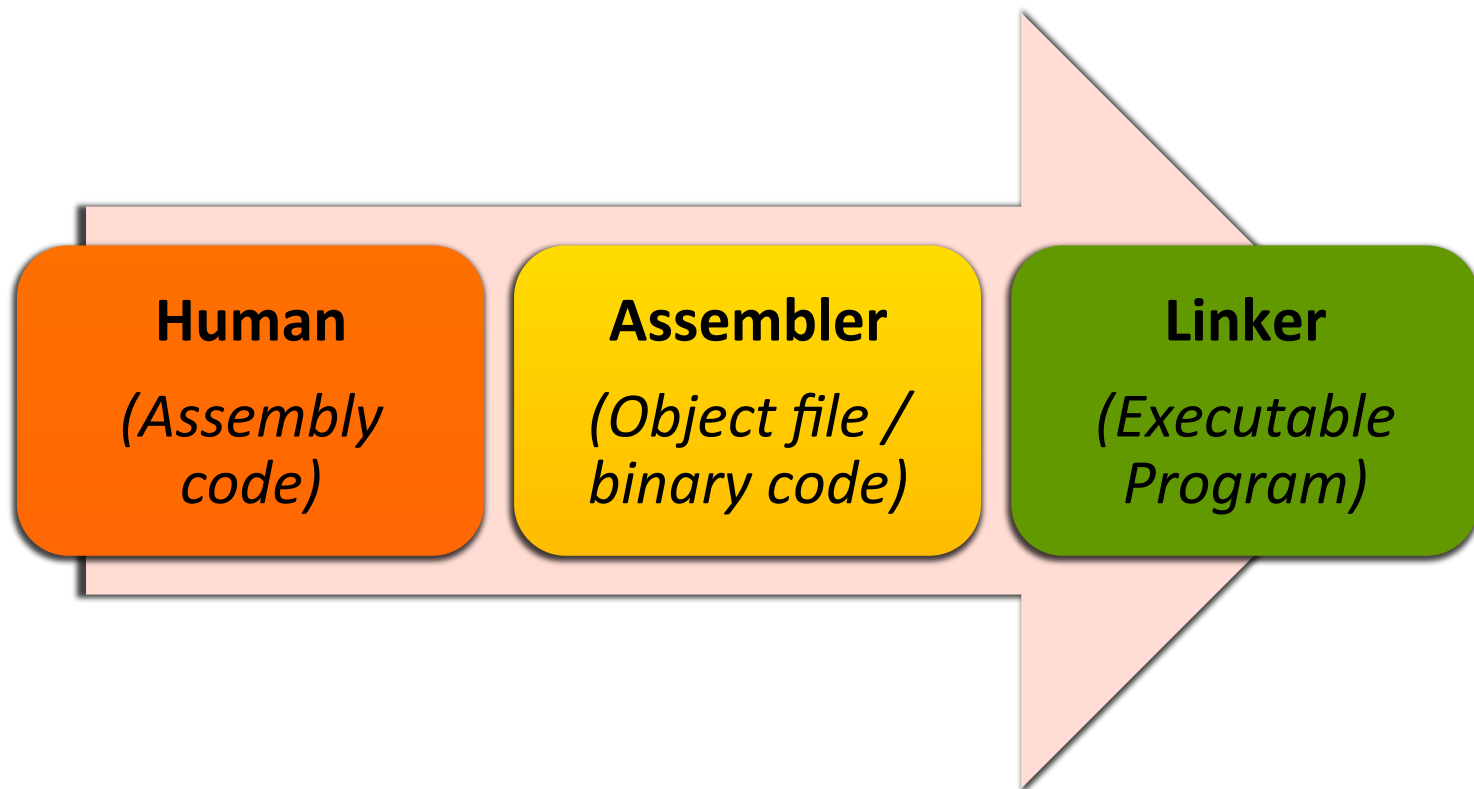


- Computer architecture pioneer
- Popularized RISC architecture in early 1980's
- Founder of MIPS Computer Systems in 1984
- Currently president of an obscure school: *Stanford University*

# Class to Date



# Class Now



# MIPS



# MIPS Overview

- Family of computer processors first introduced in 1981
- **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages
  - Original acronym
  - Now MIPS stands for nothing at all...



# MIPS Products

- **Embedded devices**
  - Cisco/Linksys routers
  - Cable boxes
  - MIPS processor is buried inside *System-on-a-Chip (SOC)*
  
- **Gaming / entertainment**
  - Nintendo 64
  - Playstation 2, PSP
  
- **Computers?**
  - Not so much anymore...
  - SGI / DEC / NEC workstations back in 1990's

# MIPS Design

- RISC – **What does this mean?**
  - Reduced Instruction Set Computing
  - Simplified design for instructions
  - Use more instructions to accomplish same task
    - But each instruction runs much faster!
  
- 32 bits (originally) – **What does this mean?**
  - 1 “word”= 32 bits
  - Size of data processed by an integer add instruction
  - New(er) MIPS64 design is 64 bits, but we won’t focus on that

# MIPS Assembly Programming



# Quotes – Donald Knuth



“People who are more than casually interested in computers should have at least **some idea of what the underlying hardware is like.** Otherwise the programs they write will be pretty weird.”  
– Donald Knuth

Remember this when we’re learning MIPS assembly in Labs 8 and 9!

# Why Learn Assembly Programming?

- **Computer Science** track
  - Understand capabilities (and limitations) of physical machine
  - Ability to optimize program performance (or functionality) at the assembly level *if necessary*
  
- **Computer Engineer** track
  - Future courses (e.g. ECPE 173) will focus on processor design
  - Start at the assembly programming level and move into hardware
    - *How* does the processor implement the `add` instruction?
    - *How* does the processor know what data to process?

# Instruction Set Architecture

- **Instruction Set Architecture (ISA)** is the interface between hardware and software
  - Specifies the format of processor instructions
  - Specifies the format of memory addresses (and addressing modes)
  - Specifies the primitive operations the processor can perform

# Instruction Set Architecture

- ISA is the “contract” between the *hardware designer* and the assembly-level *programmer*
- Documented in a manual that can be hundreds or thousands of pages long
  - Example: Intel 64 and IA-32 Architectures Software Developers Manual
  - <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>
  - No joke – the manual PDF from August 2012 is **3020 pages long!**

# Instruction Set Architecture

➤ Processor families share the same ISA

➤ Example ISAs:

- Intel x86
- Intel / AMD x86-64
- Intel Itanium
- ARM
- IBM PowerPC
- MIPS

All completely different,  
in the way that C++, Java,  
Perl, and PHP are all  
different...

... and yet learning one  
language makes learning  
the next one much easier



# Why MIPS?

- Why choose MIPS?
  - The MIPS ISA manual (volume 1, at least) is a svelte **108 pages!**
  - Extremely common ISA in textbooks
  - Freely available simulator
  - Common embedded processor
  - Good building-block for other RISC-style processors
  - Aligns with ECPE 173 course

# Arithmetic Instructions

## ➤ Addition

```
add <result>, <input1>, <input2>
```

## ➤ Subtraction

```
sub <result>, <input1>, <input2>
```

Operation / "Op code"

Operands

# Task : Write Code

➤ **Write MIPS assembly for**

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

```
add temp0, g, h
add temp1, i, j
sub f, temp0, temp1
```

**Congratulations!**

You're now an assembly  
programming expert!

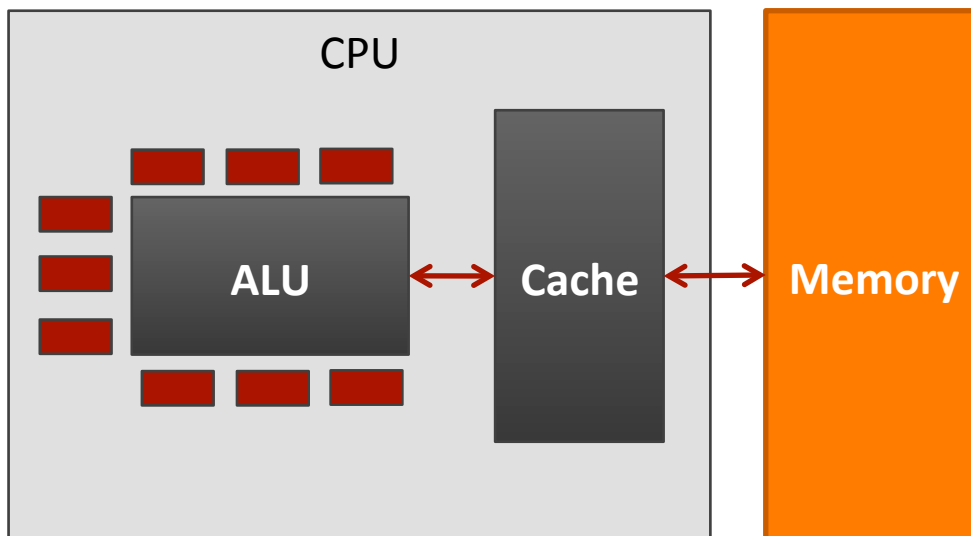
# Data Sources

- Previous example was *(just a little bit)* fake...
  - We made up some variables:  
temp0, temp1, f, g, h, i, and j
  - This is what you do when programming in C++  
(or any high level language)

**Problem: You can't make up  
variables in assembly!**  
*(as least, not in this fashion)*

# Data Sources

Where can we explicitly place data in assembly programming?



## 1. Registers

- On the CPU itself
- Very close to ALU
- Tiny
- Access time: 1 cycle

## 2. Memory

- Off-chip
- Large
- Access time: 100+ cycles

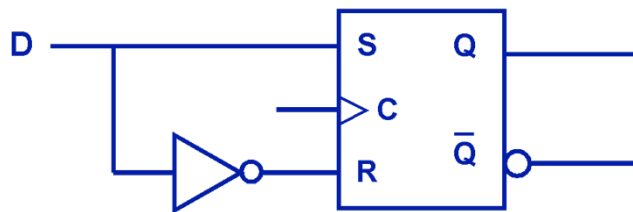
# Aside – Cache

- **Review: Does the programmer explicitly manage the cache?**
- **Ans: No!**
  - The assembly programmer just reads/writes memory addresses
  - Cache is managed automatically in hardware
  - Result: Memory *appears* to be faster than it really is

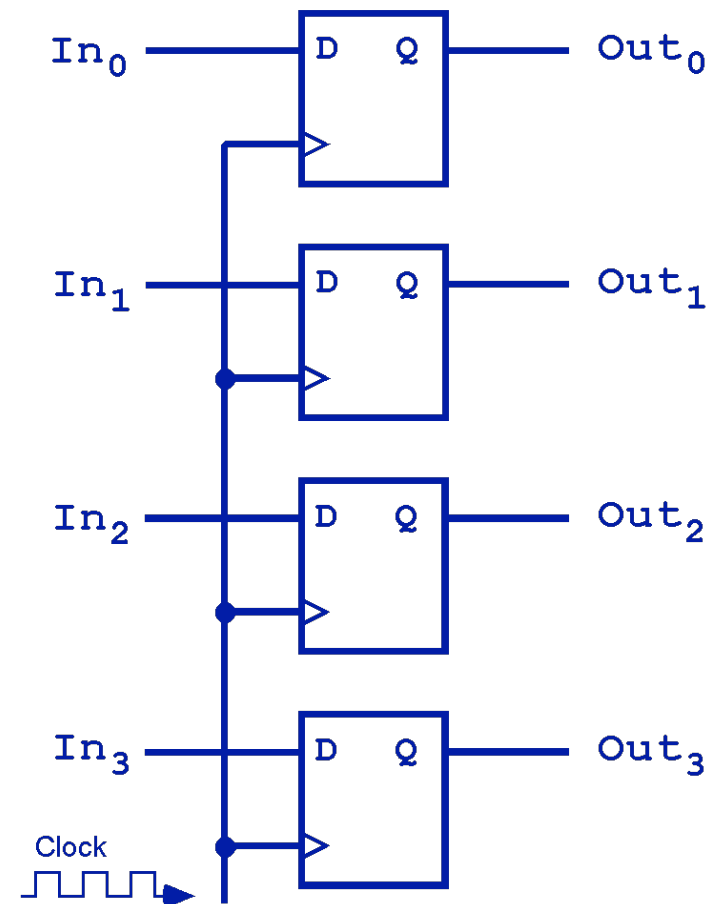
# ECPE 71

➔ From your knowledge of ECPE 71 (Digital Design), how would you construct a register?

Flip Flops! (*D Flip Flop shown*)

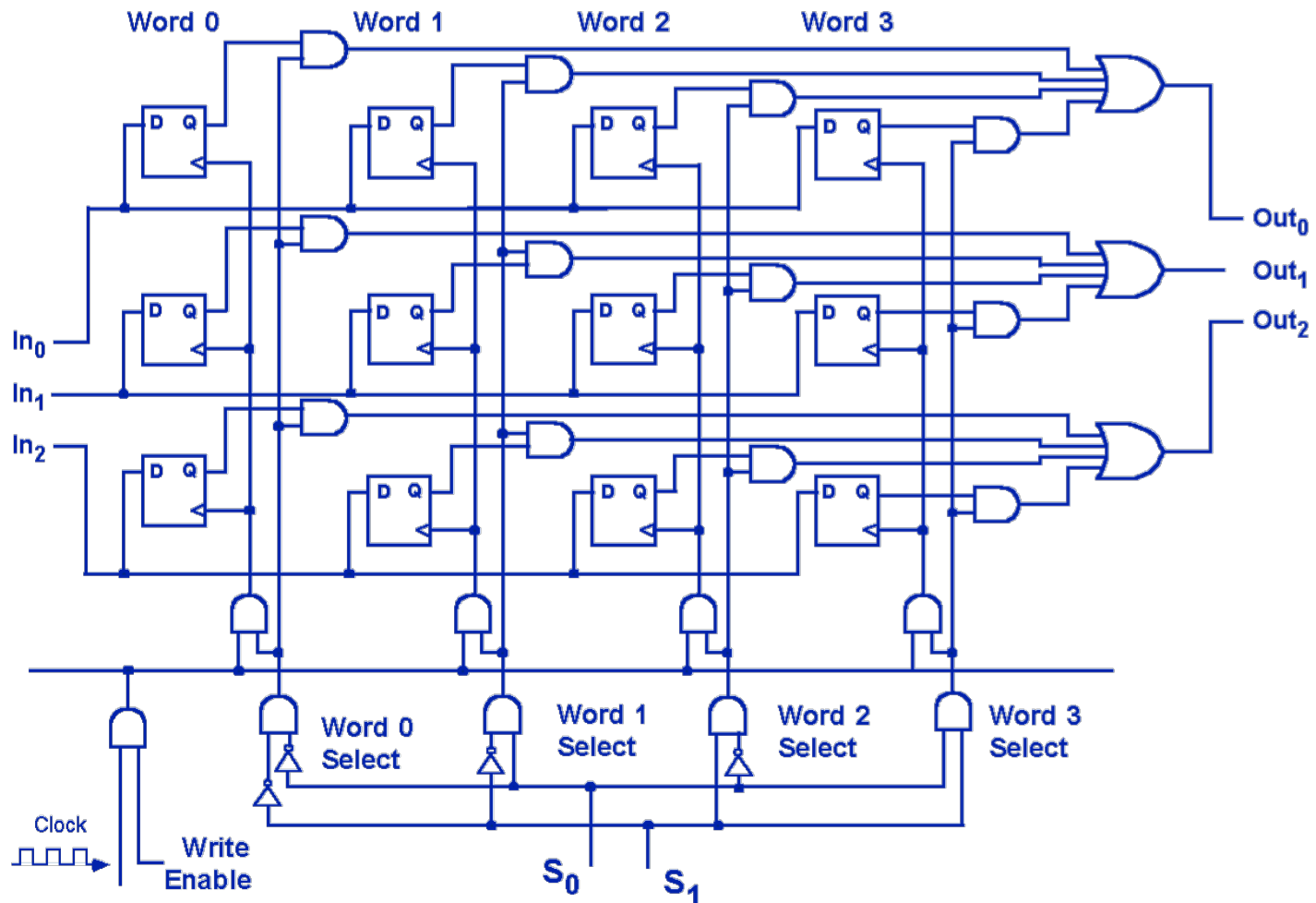


D	Q (t+1)
0	0
1	1





# ECPE 71 – Group of Registers



# Registers

- MIPS design: **32 integer registers**, each holding **32 bits**
  - “Word size” = 32 bits

Name	Use
\$zero	Constant value: ZERO
\$s0-\$s7	Local variables
\$t0-\$t9	Temporary results

- **This is only 19 – where are the rest of the 32?**
  - Reserved *by convention* for other uses
  - We’ll learn a few more later...

# Task : Write Code

➔ Write MIPS assembly using registers for:

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

## Map:

\$s0 = g  
\$s1 = h  
\$s2 = i  
\$s3 = j  
\$s4 = f

## Code:

```
add $t0, $s0, $s1  
add $t1, $s2, $s3  
sub $s4, $t0, $t1
```

# More Arithmetic Instructions

## ➤ Add Immediate

```
addi <result>, <input1>, <constant>
```

Register

Register

Can be a positive or  
negative number!

# Task : Write Code

➤ Write MIPS assembly using registers for:

$$f = g + 20$$

**Map:**

$\$s0 = f$

$\$s1 = g$

**Code:**

```
addi $s0, $s1, 20
```

# Memory

- Challenge: **Limited supply of registers**
  - Physical limitation: We can't put more on the processor chip, and maintain their current speed
  - *Many elements compete for space in the CPU...*
- Solution: **Store data in memory**
- MIPS provides instructions that transfer data between memory and registers

# Memory Fundamentals

MIPS cannot directly manipulate data in memory!

Data must be moved to a register first! (And results must be saved to a register when finished)

Common design in *RISC-style* machines...

# Memory Fundamentals

Yes, it's a **pain** to keep moving data between registers and memory.

But consider it your *motivation* to reduce the number of memory accesses. That will **improve program performance!**



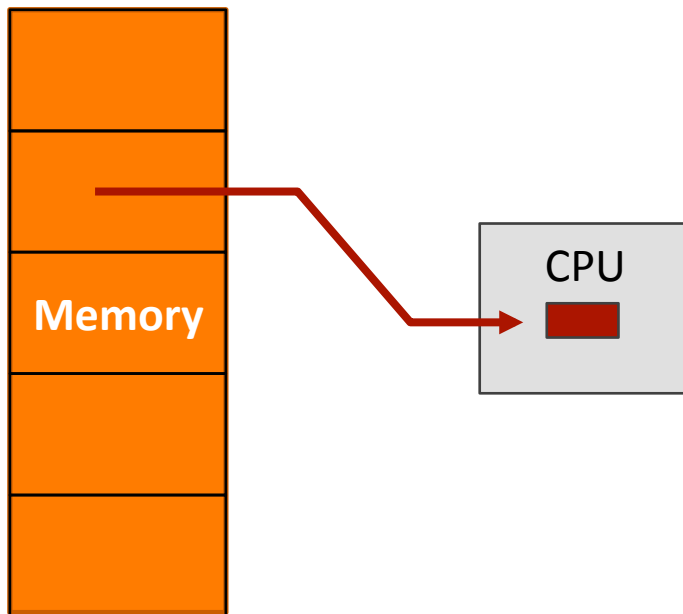
# Memory Fundamentals

- Four questions to ask when accessing memory:
1. What **direction** do I want to copy data?  
(i.e. to memory, or from memory?)
  2. What is the specific **memory address**?
  3. What is the specific **register name**? (or number)
  4. How **much data** do I want to move?

# Memory – Fundamental Operations

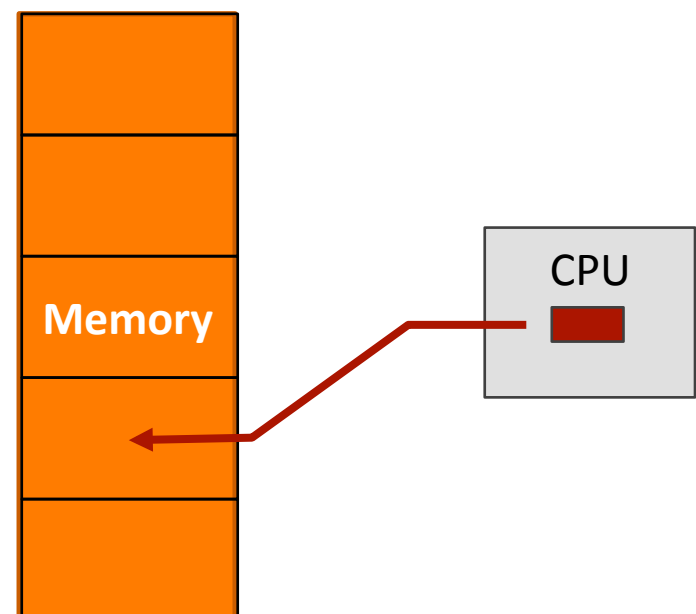
## Load

- Copy data from memory to register



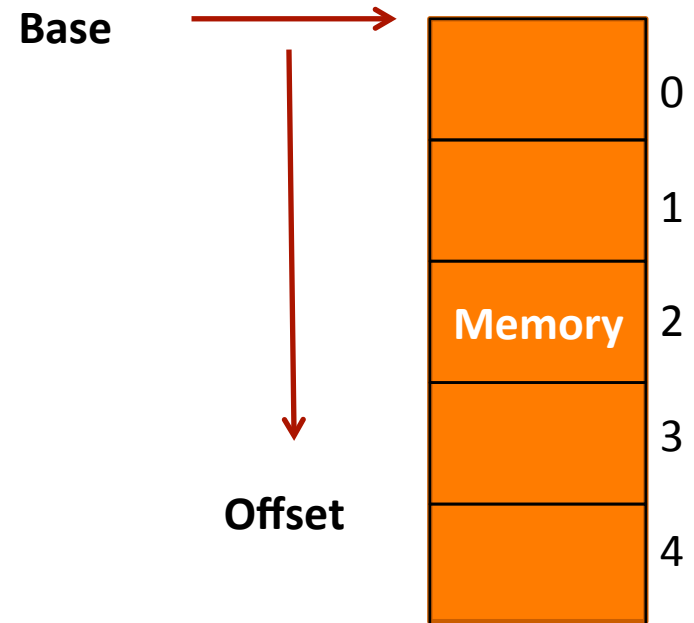
## Store

- Copy data from register to memory



# Memory – Determining Address

- There are many ways to calculate the desired memory address
  - These are called *addressing modes*
  - We'll just learn one mode now:  
**base + offset**
- The base address could be HUGE! (32 bits)
  - We'll place it in a **register**
- The offset is typically small
  - We'll directly include it in the instruction as an "immediate"



**MIPS notation: `offset(base)`**

# Memory – Register Name

- What is the name of the register to use as either the data destination (for a *load*) or a data source (for a *store*)?
- Use the same register names previously learned

# Memory - Data Transfer Size

- How much data do I want to load or store?
  - A full word? **(32 bits)**
  - A “half word”? **(16 bits)**
  - A byte? **(8 bits)**
  
- **We’ll have a different instruction for each quantity of data**
  
- No option to load an entire array!

# Memory – Data Transfer Instructions

## ➤ Load (copy from memory to register)

Word: `lw <reg>, <offset>(<base addr reg>)`

Byte: `lb <reg>, <offset>(<base addr reg>)`

## ➤ Store (copy from register to memory)

Word: `sw <reg>, <offset>(<base addr reg>)`

Byte: `sb <reg>, <offset>(<base addr reg>)`

Register

Memory Location

# Example

## ➤ What will this instruction do?

```
lw $s1, 20($s2)
```

- Load word copies from memory to register:
  - Base address: stored in register \$s2
  - Offset: 20 bytes
  - Destination register: \$s1
  - Amount of data transferred: 1 word (32 bits)

# Task : Write Code

➔ **Write MIPS assembly for:**

**$g = h + \text{array}[16]$**

*(Array of words. Can leave  $g$  and  $h$  in registers)*

**Map:**

$\$s1 = g$   
 $\$s2 = h$   
 $\$s3 = \text{base}$   
address of  
array

**Code:**

```
# Assume $s3 is already set
lw $t0, 16($s3)
add $s1, $s2, $t0
```



# Memory Address

- Slight flaw in previous solution
  - The programmer intended to load the 16<sup>th</sup> array element
  - Each element is 4 bytes (1 word)
  - The offset is in bytes
  - $16 * 4 = 64$

## Correct Code:

```
# Assume $s3 is already set
lw $t0, 64($s3)
add $s1, $s2, $t0
```

# Task : Write Code

➔ **Write MIPS assembly for:**

**array[12] = h + array[8]**

*(Array of words. Assume h is in register)*

**Map:**

\$s2 = h  
\$s3 = base  
address of  
array  
\$t1 = temp

**Code:**

```
# Assume $s3 is already set  
lw $t0, 32($s3)  
add $t1, $s2, $t0  
sw $t1, 48($s3)
```

# Task : Write Code

➔ **Write MIPS assembly for:**

$$g = h + \text{array}[i]$$

*(Array of words. Assume  $g$ ,  $h$ , and  $i$  are in registers)*

## Map:

```
$s1 = g  
$s2 = h  
$s3 = base  
address of  
array  
$s4 = i
```

## Code:

```
# "Multiply" i by 4  
add $t1, $s4, $s4    # x2  
add $t1, $t1, $t1    # x2 again  
# Get addr of array[i]  
add $t1, $t1, $s3  
# Load array[i]  
lw $t0, 0($t1)  
# Compute add  
add $s1, $s2, $t0
```

# Aside – Compiler

- **When programming in C / C++, are your variables (int, float, char, ...) stored in memory or in registers?**
- **Ans: It depends**
- **Compiler will choose** where to place variables
  - Registers: Loop counters, frequently accessed scalar values
  - Memory: Arrays, infrequently accessed data values

# Resources

- Resources on Website – view “Resources” page
  - **MIPS Instruction Set** (partial guide)
  
- Resources available in Sakai site (under ECPE 170)
  - **HP\_AppA.pdf**
    - Appendix A from famous Hennessy & Patterson *Computer Organization* textbook
    - Assemblers, Linkers, and the SPIM simulator
    - Starting on page 51 is an overview of the MIPS assembly commands!
  - **MIPS\_Green\_Sheet.pdf**
    - “Cheat sheet” for expert programmers
    - MIPS commands, registers, memory conventions, ...